Mesure locale des temps d'exécution : application au contrôle d'intégrité et au fingerprinting

Gaël Delalleau*

ZenCom Secure
Directeur technique
gael.delalleau+sstic@m4x.org

Mots Clés : timing, analyse temporelle, détection, modules, noyau, rootkit, backdoor, sécurité

Résumé L'environnement système lors de l'exécution d'un programme n'est jamais exactement le même. Certaines variations sont anodines (place libre sur le disque dur, type et charge du processeur, nombre de connexions réseau ouvertes...). D'autres peuvent être révélatrices d'une activité hostile d'espionnage ou de camouflage sur le système (comportement du noyau et des bibliothèques dynamiques, "traçage" du processus, lancement dans une machine virtuelle...). Je propose d'utiliser l'analyse temporelle pour prendre l'empreinte de certaines parties critiques de l'environnement d'exécution d'un processus. Des critères de comparaison sont définis pour différencier les variations acceptables de celles qui déclencheront une alerte de sécurité. Nous verrons en particulier comment cette technique permet de détecter différents types de *rootkits* depuis un processus utilisateur non privilégié.

1 Les différents types de "timing attacks"

Plusieurs grands types d'attaques par mesure du temps ("timing attacks") ont été rendus publics. Il convient de les rappeler brièvement, avant de nous intéresser à l'application du timing au contrôle d'intégrité et à la prise d'empreinte locale.

1.1 Détection du chemin d'exécution

Lorsque l'on effectue une requête, la mesure du temps de réponse de l'application concernée peut permettre de déduire la valeur de certaines données confidentielles. Il s'agit de données connues uniquement par l'application, qui les utilise dans son fonctionnement interne mais ne les communique jamais directement à l'utilisateur. Selon les paramètres fournis par l'attaquant, qu'elle compare à ces données secrètes, l'exécution de l'application prendra des chemins

différents avant de délivrer une réponse. Ces chemins prennent vraisemblablement un temps d'exécution différent. Si l'attaquant connaît le fonctionnement interne de l'application, il peut donc mettre au point un protocole d'attaque qui consistera à envoyer certains paramètres à l'application puis à mesurer le temps mis pour recevoir une réponse.

Prenons l'exemple d'un serveur d'authentification. Ce logiciel, accessible par le réseau, prend en paramètre un nom d'utilisateur et un mot de passe, et retourne toujours le message "authentification refusée" si l'une de ces deux données est incorrecte. L'application commence par vérifier si le nom d'utilisateur est connu. Si ce n'est pas le cas, le message d'erreur est renvoyé. Sinon, l'application vérifie le mot de passe, et renvoie ce même message d'erreur si le mot de passe n'est pas valable. La vulnérabilité provient du fait que la réponse sera renvoyée plus tard si l'attaquant a fourni un nom d'utilisateur correct, puisque la vérification du mot de passe n'est effectuée que dans ce cas. Les attaques par dictionnaire grâce au timing, permettant de découvrir des noms d'utilisateurs valides, sont ainsi possibles sur un grand nombre de services Unix [3].

Le seul exemple, à ma connaissance, de mise en œuvre d'une attaque de ce type contre le noyau du système d'exploitation a été apporté par Andrew Griffiths dans un message sur la liste de discussion Bugtraq [4]. En mesurant le temps mis par l'appel système `open()` pour retourner, il parvenait à déterminer l'existence ou non d'un fichier arbitraire sur le disque.

Les contre-mesures sont délicates à mettre en œuvre. En effet, il est inévitable que des chemins d'exécution différents soient empruntés en fonction des paramètres fournis. On ne peut qu'essayer de rapprocher au maximum leur temps d'exécution.De plus, les délais aléatoires qui peuvent être introduits (que ce soit volontairement ou du fait des aléas de la transmission réseau) peuvent être éliminés par une moyenne statistique sur un grand nombre d'essais.

1.2 Effets de cache et vie privée

Les caches ont pour objet d'accélérer la transmission de certaines données, en les stockant temporairement à un emplacement proche de leur destinataire potentiel. En demandant l'accès à une ressource, et en mesurant le temps effectivement mis pour y avoir accès, on peut savoir si cette ressource était, ou non, dans un cache.

Cette technique peut en particulier permettre des attaques sur la vie privée. Dans le cadre d'un environnement réseau pluri-utilisateurs accédant à Internet par un même serveur proxy, il serait possible de demander l'accès à différentes pages Internet et, par mesure du temps d'accès, de savoir si ces pages ont été consultées récemment ou non. Un exemple du même ordre, étudié dans [1], montre qu'un script Javascript exécuté au sein d'une page web peut savoir quels sites ont été visités par l'utilisateur. Le script demande l'accès à différents sites et mesure leur temps de chargement, détectant ainsi leur présence éventuelle dans le cache local (disque ou mémoire).

1.3 Analyse du trafic réseau

Un espion placé sur un brin de réseau a accès au contenu des communications. Si celles-ci sont chiffrées, les informations obtenues se résument à :

- le type de protocole utilisé
- l'adresse de l'émetteur et du destinataire sur le réseau
- la taille des données
- et le moment auquel ces données sont transmises

L'analyse temporelle du trafic permet dans certains cas de déduire des informations confidentielles sur le contenu de la communication. Il s'agit, en fonction des connaissances que l'on a sur le protocole utilisé, de faire des corrélation avec la fréquence et la taille des données observées. Le travail effectué sur le protocole SSH 1 (voir [2]) a prouvé la faisabilité de cette attaque pour deviner la longueur du mot de passe et certaines commandes tapées lors de sessions interactives.

Des informations sur le contenu du mot de passe peuvent même être obtenues par ce moyen. Par exemple, l'utilisateur enchaînera deux lettres placées côte à côte sur le clavier plus rapidement que s'il avait à utiliser la touche SHIFT pour entrer un chiffre dans sa deuxième frappe.

1.4 Cryptanalyse

Les attaques par timing sont très utilisées en cryptanalyse. Elles exploitent les déficiences de l'implémentation informatique des algorithmes. Selon les données fournies en entrée par l'attaquant (texte à déchiffrer, par exemple), les calculs mathématiques prennent plus ou moins de temps, à cause des optimisations ou de bugs d'implémentation. Une attaque intelligente peut permettre de déduire tout ou partie de la clé privée, qui n'est normalement connue que par l'application de cryptographie.

La cryptanalyse de certaines implémentations de RSA était ainsi possible [5]. Ces attaques sont possibles aussi bien en local qu'à distance, comme l'atteste la démonstration de David Brumley et Dan Boneh sur un serveur OpenSSL [6].

1.5 Contrôle d'intégrité et prise d'empreinte locale

Nous allons nous intéresser ici à un autre usage du "timing". En mesurant des temps d'exécution en local, au sein du système d'exploitation, nous allons tenter d'implémenter un dispositif de sécurité permettant de surveiller l'intégrité du noyau et des processus.

Un *contrôle d'intégrité* consiste à faire une photographie d'un système à un instant donné, puis à comparer régulièrement l'état du système avec l'image initiale. Les différences observées sont susceptibles d'avoir été introduites par un individu malveillant. Cette surveillance peut être réalisée à différents niveaux, du plus bas au plus élevé : mémoire morte, mémoire CMOS, microcode du microprocesseur, secteurs de démarrage du disque dur, image du noyau sur le disque, système de fichiers (zones de contrôle, fichiers exécutables, fichiers de données), code et données du noyau en mémoire, code et données des différents processus

en mémoire, etc. Nous allons nous intéresser ici à la problématique de la modification en mémoire de code, au sein du noyau comme au sein des processus. En effet, c'est la partie la plus critique et la plus délicate, sur laquelle se focalisent les efforts des pirates comme ceux des défenseurs, puisque c'est celle qui permet de contrôler entièrement le fonctionnement du système.

Un pirate va souvent ajouter ou modifier du code en mémoire, afin de modifier le comportement du système. Ses objectifs sont au minimum de camoufler ses activités et d'ajouter un accès caché, par exemple un mot de passe universel dans le processus serveur SSH. Nous allons voir que les modifications ainsi introduites peuvent être détectées par l'analyse temporelle.

Une *prise d'empreinte* consiste à effectuer un certain nombre d'expériences sur un système pour le caractériser à partir des résultats obtenus. Ici, notre objectif pourra être par exemple de définir les caractéristiques de *rootkits* du noyau connus, par rapport à un ensemble de tests temporels sur les appels systèmes. A travers l'exemple de sebek, module de surveillance des pots à miels, nous verrons qu'il est aussi possible de prendre l'empreinte des outils de sécurité intervenant au niveau du noyau, et donc de les détecter.

2 Méthodes pour la mesure et l'analyse des temps d'exécution locaux

2.1 Travaux précurseurs

J.K. Rutkowski propose dans [7] (pour Linux) et [8] (pour Windows) une technique de mesure du nombre d'instructions exécutées par le microprocesseur au cours d'un appel système. Toute modification significative de ce nombre, pour un même appel système exécuté avec des paramètres d'entrée identiques, indique qu'une altération du noyau a eu lieu. Cela permet de détecter les *rootkits* qui insèrent leur propre code dans le noyau. En effet, afin de camoufler les activités d'un pirate, ces *rootkits* détournent certains appels systèmes, modifiant ainsi le nombre d'instructions exécutées. L'auteur reconnaît lui-même plusieurs inconvénients à sa méthode de mesure, qui consiste à "tracer" le code pas à pas en incrémentant un compteur :

- elle nécessite une modification du noyau. Il faut donc posséder les droits administrateur sur le système, et recompiler et installer son propre noyau, ou bien insérer un module (si cette fonctionnalité est activée dans le noyau, ce qui a le défaut de rendre plus simples les attaques par *rootkits* noyaux). Sous Windows, il faut installer un driver.
- elle modifie l'environnement d'exécution des appels systèmes de façon détectable par l'attaquant (activation du bit TF dans le registre EFLAGS, mise en place d'un gestionnaire d'exception spécifique...). La modification du noyau, concrétisée par l'ajout de code au sein d'un appel système de contrôle et par la mise en place d'un gestionnaire d'exception spécifique pour l'interruption INT1, peut aussi être détectée.

– le code noyau peut être attaqué par le *rootkit* pour renvoyer des résultats faussés. L'auteur explique qu'il faudrait utiliser des techniques de polymorphisme inspirées des virus pour éviter la détection et la modification du code noyau. Le programme *Patchfinder 2*, implémentation sous Windows détaillée dans [9], intègre d'ailleurs une protection de l'IDT pour empêcher la désactivation du gestionnaire d'interruption INT 1. Mais toute protection peut être contournée puisque le *rootkit* opère au même niveau de privilège que l'outil de sécurité. Une technique de désactivation de *Patchfinder 2* a d'ailleurs été publiée récemment [11].

Marcin Szymanek avait initialement proposé à J.K. Rutkowski de mesurer le temps d'exécution d'un appel système depuis le mode utilisateur, en utilisant l'instruction `rdtsc` des processeurs Intel x86. Mais l'auteur explique que cette technique, bien que potentiellement plus intéressante, n'est pas fiable. D'après lui, la faute en revient aux optimisations réalisées par les processeurs modernes, qui génèrent des écarts importants entre les résultats de deux mesures du temps d'exécution d'un même appel système. C'est cependant dans cette direction que nous allons aller.

2.2 Cahier des charges

Notre objectif est de savoir mesurer et comparer de manière fiable les temps d'exécution de certaines parties de code. Afin que le champ d'application de cette technique soit le plus large possible, nous nous fixons les contraintes suivantes :

– la mesure du temps doit se faire depuis le mode utilisateur (surnommé "ring 3" sur les processeurs Intel). Aucune modification du noyau ne doit être nécessaire.
– elle ne doit pas nécessiter des droits élevés sur le système. Un compte utilisateur non privilégié (uid>0 sur les systèmes Unix) doit suffire.
– le code dont on veut mesurer le temps d'exécution doit pouvoir être situé n'importe où, du moment que l'on a les droits en exécution suffisants : espace utilisateur (ring 3 Intel), et espace noyau (ring 0 Intel).
– la technique de mesure du temps utilisée doit permettre d'établir des critères de comparaison efficaces, c'est-à-dire capables de détecter des modifications suffisamment fines du code testé sans pour autant déclencher de fausses alertes (pas de faux positifs, le moins possible de faux négatifs).

À propos de ce dernier point, il faut noter que le temps réel d'exécution - exprimable en microsecondes - importe peu. Ce qui est significatif, c'est la comparaison d'une mesure par rapport à une autre. Pour cela, on pourra par exemple quantifier les écarts entre deux mesures et définir un seuil d'alerte. Il faudra donc choisir une ou plusieurs mesures de référence, établir une définition calculable de la "distance" existant entre deux mesures (ou de leur corrélation), et enfin définir un seuil numérique au-dessus duquel on déclenchera une alerte. Une façon de faciliter les comparaisons entre des systèmes informatiques différents sera de normer les résultats obtenus sur chaque système en les divisant par une mesure de référence caractéristique du système (par exemple le temps mis pour exécuter une boucle vide).

Remarque : si le code de notre outil de détection s'exécute en mode utilisateur, il possède moins de privilèges que le *rootkit*. Ce dernier peut donc l'attaquer comme bon lui semble. Paradoxalement, c'est pourtant la meilleure solution. En effet, installer notre outil dans le noyau lui donnera des droits égaux à ceux du *rootkit*, mais pas plus. Ca ne résoud donc pas le problème. De plus, toute modification du noyau sera facilement détectable par le pirate, alors que l'activation occasionnelle d'un processus utilisateur non privilégié (pourquoi pas camouflé au sein d'un service standard tournant en utilisateur *nobody*) sera beaucoup plus discrète.

D'autre part, il est intéressant pour un utilisateur de disposer d'un moyen de vérifier l'intégrité noyau des machines sur lesquelles il a un accès, sans avoir besoin de posséder les droits d'administrateur. Nous verrons aussi que les pirates peuvent utiliser ces techniques pour détecter des outils de surveillance comme *sebek* ainsi que les pots à miels situés dans des machines virtuelles.

2.3 Outils de mesure d'un temps d'exécution

- alarme et boucle
  La fonction POSIX `alarm(unsigned int seconds)` et la fonction BSD `setitimer(...)` programment la délivrance d'un signal au processus au bout d'un temps donné. On peut définir une alarme et rentrer dans une boucle sans fin faisant enchaînant des appels à la fonction à chronométrer. Lorsque l'alarme interrompt le processus, au bout de `n` secondes, on regarde le nombre `N` d'itérations effectuées dans la boucle, et on en déduit le temps d'exécution moyen `n/N` de la fonction. Les inconvénients de cette méthode sont qu'elle ne peut obtenir qu'une moyenne du temps d'exécution, et qu'elle peut être sensible à la charge du système (présence d'autres processus utilisant le temps du processeur).
- appel système
  Des appels systèmes dédiés fournissent le temps écoulé depuis une référence, avec une précision d'une seconde (ex : fonction POSIX `time()`) ou d'une microseconde (ex : fonction BSD `gettimeofday()` ; fonctions ANSI `times()` et POSIX `clock()` qui ne comptent que le temps écoulé dans le processus courant). Ces fonctions du noyau prennent un certain temps lors de leur exécution, ce qui génère un biais à prendre en compte. D'autre part, si le noyau a été compromis, le résultat renvoyé par ces fonctions peut être contrôlé par l'attaquant.
- instruction ou registre dédié(e) du microprocesseur
  Certains microprocesseurs possèdent une instruction spécifique non privilégiée qui renvoie le nombre de battements d'horloge (ticks). Sur les processeurs Intel x86, il s'agit de l'instruction `rdtsc`. D'autres microprocesseurs disposent d'un registre accessible en lecture contenant le nombre de ticks, comme le registre `%tick` sur les processeurs Sparc. L'utilisation d'une instruction ou d'un registre dédié(e) pour mesurer le temps possède de nombreux avantages : temps d'exécution constant (et très faible), impossible à intercepter ou à détecter puisque cela ne nécessite pas d'appel au

noyau. C'est la solution que nous retiendrons dans la suite de cette étude, même si les appels système de mesure du temps pourraient parfois suffire.

2.4 Normalisation par rapport à un temps de référence

Il sera intéressant de fixer un temps de référence pour normer les temps que l'on va mesurer. Cela permettra de rendre les résultats comparables d'un système à un autre. Nous choisissons de prendre le temps d'exécution de `N` itérations d'une boucle vide comme référence, où `N` est choisi suffisamment petit pour qu'il n'y ait pas de changement de contexte ou d'interruption du système qui fausserait la mesure. Sur processeur x86 cela peut être implémenté comme ceci :

```
__asm__ ("rdtsc\n movl %%eax, %0" : "=a" (t1));
for (j=0; j<10; j++)   // N = 10
          ;            // boucle dix fois sans autre action
__asm__ ("rdtsc\n movl %%eax, %0" : "=a" (t2));
ref = t2 - t1;
```

Remarquons cependant que certaines variations des résultats, dépendant de la version du microprocesseur, ne seront pas éliminées. En effet, selon le type de microprocesseur, une même instruction pourra mettre plus ou moins longtemps à s'exécuter. Les mesures dépendront aussi des performances et du comportement du cache mémoire. Les comparaisons entre deux systèmes différents seront donc surtout qualitatives. Le code dont on veut mesurer le temps d'exécution peut aussi diffèrer entre les systèmes (différentes version du noyau, par exemple).

Il sera donc nécessaire, pour le contrôle d'intégrité, de procéder à une première série de mesures sur le système sain, que l'on prendra ensuite comme base de comparaison.

Dans d'autres cas, on cherchera néanmoins à effectuer une détection par prise d'empreinte sans disposer d'une telle référence (détection de *sebek* par exemple). Pour obtenir tout de même des résultats valides, on devra choisir arbitrairement une référence. Au moins trois solutions sont envisageables, selon ce que l'on veut mesurer :

- la première possibilité consiste à mesurer le temps d'exécution d'un code similaire, en fonction duquel on sait que l'on peut calculer une référence fiable. Par exemple, si on souhaire mesurer le temps d'exécution de l'appel système `setuid()` pour détecter un éventuel *rootkit*, on peut essayer de prendre comme référence le temps d'exécution d'un appel système similaire, comme `setgid()`, affecté d'un coefficient que l'on déterminera auparavant sur un système de test.
- une possibilité similaire est de faire des comparaisons entre les temps d'exécution de différents chemins au sein de la même fonction. Cela fonctionne si la modification que l'on souhaite détecter n'affecte pas tous les

chemins. Donnons-en deux exemples. Le module `sebek` en mode de surveillance interactive n'active l'envoi d'un paquet udp que sur les appels `read()` ayant une taille de 1. Les autres appels à `read()` n'étant pas affectés, on peut les prendre comme base de référence. Autre exemple, les *rootkits* noyau interceptent l'appel système `readdir()` pour camoufler certains fichiers et processus. Lorsque cette interception utilise le détournement de pointeurs VFS, l'utilisation de `readdir()` sur certains systèmes de fichiers ne sera pas affectée par le *rootkit*.
- une autre option serait de générer une base de donnée de mesures effectuées sur différents systèmes sains (différents processeurs, différents noyaux). On pourrait alors choisir la référence qui devrait se rapprocher le plus du système étudié. Sur architectures x86, le type de processeur s'obtient avec la commande assembleur non privilégiée `cpuid`.
- enfin, la solution la plus simple est de choisir une mesure de référence arbitraire déterminée comme une moyenne grossière des résultats obtenus lors des tests précédents. Cela fonctionne bien dans de nombreux de cas, étant donné que les modifications de temps d'exécution que l'on souhaite détecter sont généralement d'amplitude bien plus grande que les variations que l'on peut observer d'un système sain à un autre.

2.5 Le problème des écarts de mesure

Entre deux expériences identiques de mesure du temps, sur un même système, on observe des variations importantes pouvant paraître aléatoires au premier abord. Le nombre de coups d'horloge (ticks) utilisés pour exécuter un même morceau de code n'est pas toujours le même ! Pour mieux comprendre ce phénomène, nous avons réalisé un grand nombre de mesures sur une même portion de code, en affichant les résultats sous la forme d'un graphe des fréquences d'apparition des différentes valeurs.

Fig. 1. Graphe du temps d'exécution de l'appel système `setuid()` sur Linux/x86.

Les mesures sont réalisées sous Linux 2.4 à l'aide d'un code de ce type :

```
/* Nombre de mesures */
#define HOWMANY 200000
/* Définition de la fonction _setuid() */
#define __NR__setuid __NR_setuid
/* comme étant l'appel système setuid() */
_syscall1(int, _setuid, int, uid);
[...]
```

```
for (i=0; i<HOWMANY; i++) {
  __asm__ ("rdtsc\n movl %%eax, %0" : "=a" (t1));
 /* Appel à la fonction dont on */
 /* mesure le temps d'exécution */
  _setuid(mon_uid);

  __asm__ ("rdtsc\n movl %%eax, %0" : "=a" (t2));
  temps = t2 - t1;
  [...]
}
```

Lors d'un premier essai de 200000 mesures successives, on obtient les résultats montrés sur les figures 1 et 2.

Fig. 2. Graphe du temps d'exécution de l'appel système `read("/proc/net/tcp")` sur Linux/x86.

On observe que les valeurs obtenues pour le temps d'exécution sont des pics discrets, bien définis. Toutes les valeurs intermédiaires possibles entre deux pics ne sont pas nécessairement obtenues. Plus le chemin d'exécution est long, plus on tend vers un étalement et un lissage des valeurs. L'hypothèse d'une distribution purement aléatoire des valeurs autour d'un centre s'effondre à la vue de ces graphes, qui ne peuvent être assimilés à des courbes gaussiennes. On ne peut donc pas procéder par moyenne statistique pour éliminer un bruit blanc. Au contraire, ces distributions de valeurs peuvent être considérées comme des caractéristiques du système mesuré, qu'il pourra être intéressant de prendre complètement en compte plutôt que de tenter de définir une moyenne.

Quelle est la cause de ces variations de temps d'exécution ? L'hypothèse des changements de contexte entre processus est invalidée par la théorie et l'expérience, du moins sous Linux 2.4 lors de la mesure d'un temps d'exécution en mode noyau. Les résultats ne sont pas notablement différents suivant que l'on lance le processus effectuant la mesure avec une priorité maximale ou minimale, et ils ne dépendent pas non plus de la charge du microprocesseur. Dans certains cas, sur des mesures longues, il est possible qu'un changement de contexte ait lieu : cela peut facilement être détecté, car le résultat de la mesure est aberrant. Il suffit alors de ne pas en tenir compte. La véritable cause est à chercher ailleurs, probablement dans les interruptions matérielles, les optimisations du processeur, et surtout les différents caches de mémoire comme expliqué dans [12]. Les accès en exécution, lecture ou écriture prennent un nombre de battements d'horloge différent selon que les zones accédées sont présentes ou non dans un des caches.

Une difficulté supplémentaire apparaît lorsque l'on réitère ces expériences. L'allure du graphe obtenu n'est pas toujours identique selon les moments, sur un même système ! Pour les mesures de codes courts, comme l'appel système `setuid()`, cela se traduit par un déplacement des pics de quelques unités, une redistribution des poids relatifs de chaque pic (n'allant toutefois pas jusqu'à inverser les grandes tendances), et l'apparition éventuelle d'un petit nombre d'autres pics. Pour les mesures de codes longs, comme un appel `read()` sur le fichier `/proc/net/tcp`, on peut observer de grands changements dans le détail des pics, mais globalement les valeurs minimale, moyenne et maximale changent relativement peu. On peut cependant observer parfois l'apparition de nouvelles zones de forte concentration de pics.

*Remarque :*Pour déterminer un temps de référence de boucle vide qui soit toujours le même sur un même système, on doit donc effectuer cette mesure un grand nombre de fois et choisir la plus petite valeur obtenue.

2.6 Analyse des écarts et définition de critères d'alerte

Les problèmes posés par les variations du résultat des mesures ne sont pas insolubles, même s'ils rendent cette technique de détection plus difficile à mettre en œuvre et potentiellement moins précise que celle de J. Rutkowska [10]. Il nous faut arriver à distinguer les variations normales des variations introduites par un élément intrusif.

A cet effet, nous réalisons d'abord un grand nombre de séries de mesures sur le système sain, qui seront les mesures de référence. Une fois cette phase d'apprentissage terminée, les informations sont stockées pour servir de base de comparaison future. Ensuite, lors des tests réels, les mesures seront comparées aux mesures de référence selon certains critères. Si ces derniers ne sont pas respectés, une alerte de sécurité est déclenchée. Plusieurs critères de comparaison peuvent être définis.

Le critère le plus simple consiste à comparer le temps minimum mesuré pour l'exécution de la fonction considérée. En effet, la quasi-totalité des modifications de code rajoutent des instructions lors de l'exécution. Le cas typique est le détournement de la fonction au tout début de son exécution, le passage par une séquence de code rajouté effectuant une action (qui peut dépendre des arguments passés à la fonction), puis l'appel de la fonction initiale (technique décrite par exemple dans [19]). On constate donc que le temps minimum d'exécution mesuré ne pourra qu'être supérieur à celui pris en référence. Ce critère pourrait permettre de détecter de petits ajouts de code au sein de grandes fonctions, mais cela n'a pas encore été vérifié.

Nous allons aussi utiliser un critère plus avancé, qui prend en compte l'ensemble des pics observés sur une série de mesures. L'objectif est ici de pouvoir quantifier mathématiquement la ressemblance entre deux séries de mesures. Par exemple, deux graphes montrant des pics d'amplitudes similaires à peu près aux mêmes endroits auraient un fort coefficient de ressemblance, alors que des graphes montrant des pics éloignés ou de formes très différentes auraient un faible coefficient. On peut utiliser pour cela le produit scalaire des transformées de Fourier des deux graphes (sans prendre en compte les coefficients correspondants aux hautes fréquences). Deux graphes possédant des pics similaires auront des coefficients de Fourier semblables dans les basses fréquences, leur produit scalaire normé tendra donc vers 1. A l'opposée, des graphes trop disparates verront leur produit scalaire tendre vers zéro. On pourrait bien sûr trouver d'autres moyens de mesurer cette ressemblance ; en particulier, cette formule est peu adaptée à la détection des petites modifications de codes courts.

$$\text{ressemblance} = \frac{\mathcal{F}(g_1).\mathcal{F}(g_2)}{|\mathcal{F}(g_1)|.|\mathcal{F}(g_2)|}$$

D'autres critères seraient intéressants à mettre en œuvre, en particulier pour la détection des modifications de codes courts (mesures présentant un faible nombre de pics discrets). La comparaison du nombre de pics et des valeurs exactes qu'ils prennent avec les valeurs autorisées obtenues lors de la phase d'apprentissage pourrait permettre, là aussi, de détecter de petites modifications du code.

Le critère grossier qui consisterait à prendre la moyenne des temps d'exécution sur un certain nombre de mesures ne sera pas utilisé ici. Cependant, il est simple et devrait fonctionner dans un certain nombre de cas. On remarquera en effet que les modifications de temps d'exécution engendrés par les *rootkits* et autres sont souvent très importantes, donc facilement détectables.

3 Détection des "rootkits noyau"

3.1 Rappels sur les attaques du noyau

Les attaques par insertion de code dans le noyau du système permettent d'en modifier le comportement et d'obtenir un total contrôle sur le système d'exploitation. Elles se sont fortement développées à partir de l'année 1997, d'abord sur Linux [13], puis sur les systèmes BSD, Solaris [14] et Windows [15]. Trois objectifs peuvent être atteints par l'insertion d'un *rootkit* dans le noyau : camouflage de certaines activités (fichiers, processus, connexions réseau), accès privilégié caché local et/ou distant, espionnage (interception des communications sur les tty et les sockets, par exemple).

Plusieurs techniques sont utilisées pour écrire du code dans le noyau. La plus simple est l'insertion de modules du noyau, ou Loadable Kernel Modules (LKM). L'infection de modules légitimes du noyau permet de faire survivre l'attaque à un redémarrage [20]. Une autre possibilité est d'injecter le code dans l'image du noyau stockée sur le disque dur [18]. Enfin, il est possible d'écrire directement dans la mémoire vive pour modifier le noyau en temps réel, en accédant au périphérique `/dev/kmem` sous Linux [16] ou `\Device\PhysicalMemory` sous Windows [17].

3.2 Principe de la détection

Des outils de détection des *rootkits* noyau existent, comme chkrootkit, rkscan, carbonite, kstat, ou saint michael. Cependant, ils sont régulièrement rendus caducs par la découverte de nouvelles techniques. Ils ne peuvent donc, en règle générale, que détecter les *rootkits* connus ou utilisant des techniques connues pour s'insérer dans le noyau et intercepter les appels système.

Au contraire, les modifications du noyau opérées par les *rootkits* peuvent permettre de les détecter par timing, même s'ils sont inconnus. En effet, les actions qu'ils effectuent (en particulier le camouflage et l'espionnage) doivent, par définition, interférer même indirectement avec l'exécution de certains appels système. Le principe de la détection sera donc de mesurer le temps d'exécution de ces appels système pour détecter les éventuelles modifications.

3.3 Quels appels systèmes, et avec quels paramètres ?

De grandes constantes existent parmi les *rootkits*. Ils peuvent modifier en particulier, parmi d'autres, le fonctionnement des appels suivants auxquels nous allons nous intéresser :

- `readdir()` dans le système de fichiers racine, pour camoufler l'existence de certains fichiers.
- `readdir()` dans le système de fichiers `/proc`, pour camoufler certains processus.
- `read()` sur certains fichiers, en particulier `/proc/net/tcp` pour camoufler certaines connexions réseau.

En complément de ces tests génériques, la mise en place de tests de détection spécifiques à certains *rootkits* connus est envisageable.

3.4 Résultats expérimentaux

L'outil de détection fonctionne en deux phases. Tout d'abord, il effectue une collection de mesures durant la phase d'apprentissage. Plus cette dernière est longue, plus les détections futures pourront être précises, et les éventuelles fausses alertes pourront être évitées. Une "mesure" est en fait un ensemble de 200000 valeurs mesurées du temps d'exécution de l'appel système à tester. Durant cette phase, le programme effectue une première mesure, qu'il prend comme référence. Il effectue ensuite une deuxième mesure, et calcule le facteur de ressemblance avec la première mesure effectuée. Si ce facteur est inférieur à un seuil déterminé (0,5), la nouvelle mesure est ajoutée dans le pool des références. D'autres mesures sont ensuite effectuées successivement, et ajoutées de la même façon en tant que références si elles ne ressemblent à aucune mesure effectuée précédemment. Il est important que le système ait une activité réseau, CPU et disque variée durant la phase d'apprentissage, afin qu'un maximum de mesures de référence valides soient enregistrées. On constate que le nombre de comportements différents reste limité. C'est ce qui permet à cette méthode de détection d'être utilisable en pratique.

Dans la deuxième phase, le programme continue à faire régulièrement des mesures, et les compare à chacune des mesures de référence. Si la mesure actuelle ne ressemble à aucune des mesures connues avec un facteur de ressemblance supérieur à un certain seuil (0,2), une alerte de sécurité est déclenchée. Voyons un exemple d'utilisation de ce programme de démonstration du concept :

```
$ ./ktime -v
[ Contrôleur d'intégrité Ktime (c) 2004 Gaël Delalleau ]
config: measure readdir() on the /proc filesystem
reference ticks count : 126
Collecting the reference measures (learning phase)
Taking a maximum of 20 measures during 60 seconds
Found new behavior !
( MIN = 1067 | produit scalaire : 0.978689 )
( MIN = 1065 | produit scalaire : 0.998739 )
( MIN = 1072 | produit scalaire : 0.637619 )
  [...]
( MIN = 1065 | produit scalaire : 0.063830 )
Found new behavior !
  [...]
Recorded 2 different behaviors during learning phase
Now watching for suspicious modification of the kernel
    environnement
( MIN = 1065 | produit scalaire : 0.893860 )
```

```
( MIN = 1062 | produit scalaire : 0.638144 )
```

[A ce point, on insère le rootkit noyau adore-ng : `insmod adore-ng.o` *]*

```
( MIN = 14190 | produit scalaire : 0.000925 )
>> ALERT ! Potential kernel attack detected <<
```

L'insertion du rootkit a été largement détectée, à la fois par le calcul de la ressemblance avec les mesures de référence et par le calcul du temps minimum d'exécution, qui a été multiplié par un facteur 10. Les différences ne sont pas toujours aussi claires, cela dépend de la quantité de code rajoutée par le *rootkit*.

Détection du rootkit Linux adore-ng			
syscall	temps minimal	produit scalaire	
proc readdir	1062	0.63	avant
proc readdir	14190	0.00	après
/ readdir	1987	0.96	avant
/ readdir	2250	0.02	après
proc/net read	8409	0.74	avant
proc/net read	29999	0.00	après

3.5 Prise d'empreinte des modules du noyau connus

On peut mettre en place des tests et critères spécifiques à certains modules afin de pouvoir les détecter sans avoir forcément besoin d'une phase préliminaire d'apprentissage. Prenons l'exemple du module *sebek*, dont le rôle est d'espionner les communications sur le système pour les envoyer sur le réseau local au sein de paquets udp particuliers. Ce module, disponible à l'adresse Internet *http ://www.honeynet.org/tools/sebek/*, est développé par le Honeynet Project pour être implanté dans les *honeypots*. Il perd donc tout son intérêt si les pirates peuvent détecter sa présence dès leur entrée dans le système piégé... ce qui est effectivement le cas.

En effet, *sebek* déclenche l'envoi d'un paquet sur le réseau au sein de l'appel système `read()`. Cette action supplémentaire prend un temps notable, comme le montrent les tests suivants, réalisés sur un système Solaris sur architecture Sparc 64 bits. Le principe de la mesure reste exactement le même que sur architecture x86, en replaçant `__asm__ ("rdtsc\n movl %%eax, %0" : "=a" (t1));` par `__asm__ ("rd %%tick, %0" : "=r" (t1));`.

```
$ ./ktime
[ Contrôleur d'intégrité Ktime (c) 2004 Gaël Delalleau ]
config: measure read() on /bin/ls (1 byte)
Collecting the reference measures (learning phase)
( MIN = 8225 | produit scalaire : 0.776282 )
  [...]
Recorded 2 different behaviors during learning phase
Now watching for suspicious modification of the kernel environnement
  [...]
```

[Insertion de sebek dans le noyau : `modload sebek64` *]*

```
( MIN = 29999 | produit scalaire : 0.009930 )
>> ALERT ! Potential kernel attack detected <<
```

Le temps minimal d'exécution de l'appel système `read()` est passé de 8225 à 29999. Cette dernière valeur correspond en fait à la valeur maximale supportée par le programme de détection : le véritable temps d'exécution est encore plus grand ! On se rend bien compte que, même sans disposer d'une phase d'apprentissage préliminaire sur le même système, le module *sebek* peut être détecté facilement par mesure du temps d'exécution. La prise d'empreinte est ici très simple : il suffit de tester le temps minimal mis pour exécuter un appel à `read()`, de le normer par rapport à la référence "boucle vide" pour que la comparaison avec les autres systèmes ait un sens, et de le comparer à une valeur de référence que l'on choisira d'après les tests effectués préalablement sur un autre système (en se basant sur ce test, on pourrait choisir la valeur 20000, par exemple).

4 Détection des "rootkits en mode utilisateur"

4.1 Rappels sur les attaques ELF

Certains *rootkits* ou *backdoors* fonctionnent uniquement en mode utilisateur. Leur principe est d'injecter du code dans d'autres programmes, pour en modifier le fonctionnement. A peu près toutes les applications pratiques que l'on a prêtées aux *rootkits* noyau peuvent également être implémentées en mode utilisateur. En règle générale, on peut considèrer que les fonctionalités de camouflage et de furtivité sont plus simples à implémenter dans le noyau, alors que le mode utilisateur offre plus de possibilités pour les fonctions d'espionnage et d'accès caché.

Ces attaques sur les processus utilisateurs utilisent un principe d'interception de fonctions similaire à celui utilisé pour les attaques du noyau. Pour modifier le fonctionnement d'un programme, il faut non seulement injecter du code dans son espace mémoire, mais aussi rediriger vers ce code certaines fonctions du

programme ou des bibliothèques dynamiques auxquelles il fait appel. Plusieurs techniques existent pour réaliser l'injection de code et le détournement de fonctions.

L'injection peut être statique (infection de l'image ELF du programme sur le disque [23]) ou dynamique (injection dans l'espace mémoire d'un processus déjà existant sous Unix [24] comme sous Windows [25]). La technique traditionnelle d'interception des appels aux fonctions des bibliothèques est de faire charger préalablement au programme une bibliothèque dynamique hostile, grâce aux hooks systèmes sous Windows (lire [26]) ou à la modification des fichiers de configuration ou des variables d'environnement du linker dynamique sous Unix (lire [21]). Cette bibliothèque "pirate" peut intercepter les appels aux fonctions exportées par les autres bibliothèques. Des techniques plus évoluées, permettant de cibler spécifiquement un certain processus, sont également apparues. Elles permettent d'intercepter discrètement certains appels aux bibliothèques, par exemple en modifiant la Procedure Linkage Table (PLT) d'un fichier ou d'un processus au format ELF [22]. La technique équivalente sous Windows consiste à modifier l'Import Address Table (IAT) du processus [27].

Les attaques les plus efficaces sont celles qui visent spécifiquement un processus déjà chargé en mémoire pour y injecter un code malicieux. Il faut donc chercher à faire un contrôle d'intégrité des processus tournant en mémoire ; le contrôle des fichiers stockés sur le disque dur (comme le fait tripwire) n'est pas suffisant. Ce n'est pas une tâche facile. A l'heure actuelle, il existe bien l'outil *elfcmp* qui est censé détecter ce genre d'attaque en comparant l'image ELF des processus en mémoire à leur image sur le disque. Mais il ne détecte ni l'injection de code au sein de l'espace mémoire du processus par utilisation de `mmap`, ni les techniques modernes permettant de détourner les appels de fonctions vers ce code (modification de la PLT ou de la GOT).

Les techniques de contrôle d'intégrité par mesure temporelle exposées précédemment peuvent être appliquées avec profit à la détection des *rootkits* et *backdoors* en mode utilisateur.

4.2 Principe de la détection

Nous pouvons mesurer les temps d'exécution de fonctions choisies des bibliothèques dynamiques, et les comparer à une base de références, exactement de la même manière que pour le noyau. Dans le cas où l'infection est généralisée à tous les processus, il suffit d'appeler les fonctions voulues au sein de notre programme de vérification. C'est par exemple le cas si l'attaquant a modifié la bibliothèque C sur le disque dur. Mais dans la plupart des cas, l'attaque est localisée uniquement dans un ou quelques processus tournant en mémoire. Il faut alors stopper le processus choisi, y injecter notre code de vérification, exécuter ce dernier dans l'espace mémoire du processus visé, puis récupérer les résultats et relancer normalement l'exécution du programme.

Les techniques d'injection et d'exécution de code dans l'espace mémoire d'un autre processus que nous utiliserons sont les mêmes que celles décrites dans [24] et [25].

4.3 Résultats expérimentaux

Prenons l'exemple d'un serveur OpenSSH sous Linux, tournant en mémoire. Le code hostile injecté dans ce processus par le pirate effectue une interception de l'appel à la fonction `read()` de la bibliothèque C. La nouvelle fonction `read()` commence par appeler la vraie fonction, enregistre le résultat dans un fichier (espionnage des communications), et retourne.

On injecte dans le processus OpenSSH un code de détection qui effectue 1000 appels à `read()` en en mesurant les temps d'exécution. L'appel à `read()` est réalisé en faisant un saut sur l'entrée correspondante de la PLT du processus, comme le fait le programme. Voici les résultats obtenus :

Détection de l'interception de la fonction read()			
temps minimum	temps moyen	temps maximum	
800	822	6820	avant l'attaque
1332	1339	64108	après l'attaque

Les données significatives sont le temps minimum et le temps moyen. On observe une augmentation d'environ 65% du temps d'exécution. La répétition de cette expérience montre que ces mesures sont fiables et varient peu. Cette différence significative permet ainsi de détecter la modification de l'environnement d'exécution du serveur OpenSSH, et de déclencher une alerte.

Il est aussi possible d'appeler des fonctions du programme testé plutôt que des fonctions de la bibliothèque. Par exemple, la fonction gérant l'authentification par mot de passe. Cependant, il est probable que des techniques pourront être mises en œuvre par le code hostile injecté pour repérer ces tests et échapper à la détection, plus facilement que dans le cas des *rootkits* noyau. Mais ces méthodes, ainsi que les contre-mesures possibles, sortent du cadre de cet article.

5 Détection d'une machine virtuelle

5.1 Principe de la détection par timing local

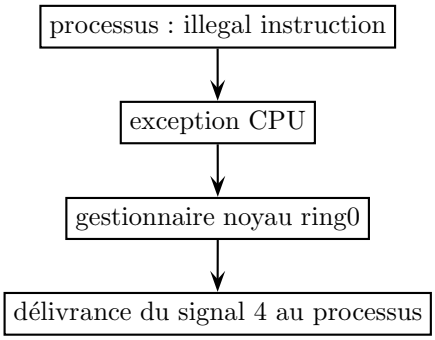
Les honeypots sont parfois implantés dans des machines virtuelles. Il est intéressant de savoir s'il est possible à un utilisateur non privilégié, éventuellement bloqué dans un `chroot`, de détecter qu'il se situe dans une telle machine virtuelle. En effet, les pirates risquent alors de faire retraite précipitament s'ils détectent le piège.

Il existe plusieurs types de machines virtuelles x86. L'émulation x86, utilisée par *bochs*, est beaucoup plus lente qu'un système normal et ne se comporte pas de la même manière, ce qui rend ces systèmes très facilement repérables. Il en est de même pour les systèmes sous *User Mode Linux (UML)*, au moins ceux qui utilisent le mode `ptrace` pour intercepter les appels systèmes des processus. En effet, cette méthode rajoute un délai énorme pour l'exécution des

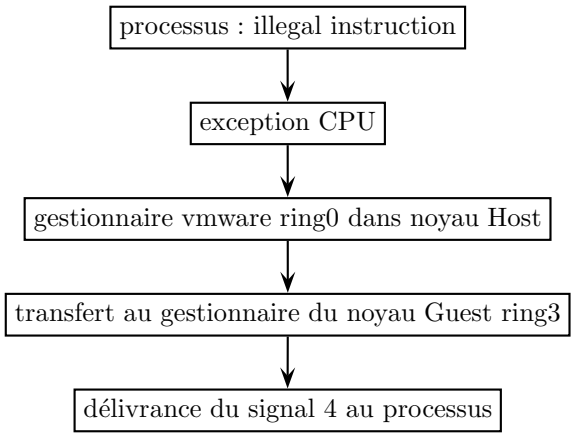
appels systèmes (facteur d'ordre 100), ce qui permet encore une fois de détecter facilement l'existence d'une machine virtuelle.

Enfin, la machine virtuelle la plus convaincante est *VMware*. Comme pour *UML*, les instructions en langage machine sont exécutées par le processeur réel, ce qui n'introduit donc pas de délai. Les appels systèmes sont gérés correctement, en modifiant l'IDT (ce qui est d'ailleurs un autre moyen de détecter vmware !). Même si on verra que des petites différences de temps peuvent apparaître sur des appels systèmes courts, le risque de fausse alerte reste élevé puisque l'on ne dispose a priori pas d'une mesure de référence exacte. Il faut donc trouver autre chose.

Un moyen efficace de détecter *VMware* consiste à mesurer la durée d'exécution d'instructions Intel privilégiées. Ces instructions peuvent être utilisées en mode noyau (ring 0), mais déclenchent une exception quand elles sont utilisées en mode utilisateur (ring 3). En temps normal, voici le cheminement suivi par le processus :



Sous *VMware* il se passe "a priori" quelque chose comme ceci :



Il est possible de mesurer la différence de temps d'exécution en installant un gestionnaire qui intercepte le signal 4.

5.2 Résultats expérimentaux

L'écriture d'un programme de test basé sur ce concept permet d'obtenir les résultats suivants :

```
SYSTEME LINUX NORMAL :
 Reference time... 21 mms  [1]
 Timing various instructions : 24 mms = 114 %  [1]
 Timing void syscall : 50 mms = 238 %  [2]
 Timing illegal instructions with signal
                 handlers : 776 mms = 3695 %  [36]
 Timing int 3 with signal handler : 386 mms = 1838 %  [18]

SYSTEME LINUX SOUS VMWARE :
 Reference time... 21 mms  [1]
 Timing various instructions : 24 mms = 114 %  [1]
 Timing void syscall : 106 mms = 504 %  [5]
 Timing illegal instructions with signal
                 handlers : 4978 mms = 23704 %  [237]
 Timing int 3 with signal handler : 2530 mms = 12047 %  [120]
 > special x86 instructions take a long time [VMWARE]
```

6 Détection d'un outil d'analyse du flot d'exécution du programme

L'analyse temporelle de l'exécution du processus courant par lui-même peut lui permettre d'obtenir des informations sur son environnement extérieur d'exécution. Un programme peut en effet trouver un intérêt à réagir différemment suivant qu'il est exécuté dans un environnement surveillé ou pas (présence d'un déboggueur, d'un outil de traçage des appels systèmes comme *strace, truss* ou *ktrace*, etc).

La mesure du temps mis pour exécuter un petit nombre d'instruction permet de détecter trivialement le traçage pas-à-pas du processus. De même, la mesure du temps écoulé pendant un appel à une fonction d'une bibliothèque dynamique permet de détecter les outils de type *ltrace*, et la mesure du temps passé pour exécuter un appel système révèlera immanquablement la présence de *strace*. En effet, l'interception de ces appels, ainsi que l'affichage correspondant sur le terminal, rajoutent un temps conséquent à l'exécution. Ce temps supplémentaire peut être détecté par les techniques exposées dans cet article.

Il est bon de s'en rendre compte afin de ne jamais faire confiance à un simple lancement d'un programme sous *strace* pour affirmer qu'il est inoffensif. Ce même programme, exécuté dans un environnement libre, pourrait révéler des fonctionnalités imprévues...

7 Conclusion

L'analyse temporelle des temps d'exécution est une méthode de fingerprinting et de contrôle d'intégrité "floue", par opposition aux tests habituels qui sont binaires. Il convient donc de définir avec soin des critères discriminants. Plus de travail est nécessaire pour créer un outil de sécurité efficace ; c'est le prix à payer pour avoir une plus grande liberté d'action. Nous avons vu en effet que cette technique prometteuse possède un champ d'application très large.

Il s'agit d'une mesure supplémentaire de défense en profondeur, que l'on peut mettre en place en pratique. Cette technique n'est évidemment pas infaillible, des contre-mesures seront probablement publiées, lesquelles pourront à leur tour être parées, et ainsi de suite. En effet, le défenseur évolue dans un système où il n'a pas plus de privilèges que le programme hostile qu'il veut détecter. Des interférences réciproques sont donc possibles.

Certains problèmes se poseront sans doute pour créer un véritable outil de sécurité utilisable en production à partir de cette technique : arriver à affiner les critères de détection au plus près pour éviter les faux positifs tout en détectant un maximum de choses ; rester discret (éviter que l'outil ne soit détecté par le rootkit afin de garantir son intégrité) ; décider de la méthode de stockage de la base de donnée des mesures de référence ; choisir à quelle fréquence lancer l'outil et évaluer l'impact sur les performances du système ; garantir l'efficacité de la phase d'apprentissage ; etc.

Les codes sources des démonstrations présentées dans cet article seront publiées sur Internet lors du SSTIC.

8 Références

Références

1. Edward W. Felten et Michael A. Schneider, Timing Attacks on Web Privacy, *ACM Conference on Computer and Communications Security*, 2000.
2. D. Song, D. Wagner et X. Tian, Timing Analysis of Keystrokes and SSH Timing Attacks, *10th USENIX Security Symposium*, 2001.
3. Sebastian Krahmer, *Execution Path Timing Analysis of Unix Daemons*, 2001.
4. Andrew Griffiths, *Syscall implementation could lead to whether or not a file exists*, Liste de discussion Bugtraq 2003. `http://www.securityfocus.com/archive/1/317425`
5. Paul C Kocher, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, *Advances in Cryptology - CRYPTO'96*, Lecture Notes in Computer Sciences, Springer, 1996.
6. David Brumley et Dan Boneh, Remote Timing Attacks Are Practical, *12th USENIX Security Symposium*, 2003.
7. Jan K. Rutkowski, Execution path analysis : finding kernel based rootkits, *Phrack* vol. 59, Phrack Staff, 2002. `http://www.phrack.org`
8. Jan K. Rutkowski, Advanced Windows 2000 Rootkits Detection, *Blackhat USA*, 2003.
9. Joanna Rutkowska, Detecting Windows Server Compromises with Patchfinder 2, 2004. `http://www.rootkit.com/vault/joanna/rootkits_detection_with_patchfinder2.pdf`
10. Joanna Rutkowska, Detecting Windows Server Compromises, *Hivercon*, 2003. `http://www.hivercon.com/conf/archive/hc03/Rutkowska_Win32RookitDetection_HC2003.ppt`
11. Edgar Barbosa, Avoiding Windows Rootkit Detection, 2004. `http://www.rootkit.com/vault/0pc0de/bypassEPA.pdf`
12. Intel, Using the RDTSC Instruction for Performance Monitoring, 1997.
13. Halflife, Abuse of the Linux Kernel for Fun and Profit, *Phrack*, vol. 50, 1997. `http://www.phrack.org`.
14. Plasmoid/THC, Attacking Solaris with loadable kernel modules, 1999. `http://www.thc.org/papers.php`
15. Greg Hoglund, A *REAL* NT Rootkit, patching the NT Kernel, *Phrack*, vol 55, 1999. `http://www.phrack.org`
16. Silvio Cesare, Run-time kernel patching, 1998. `http://www.u-e-b-i.com/silvio/runtime-kernel-kmem-patching.txt`
17. Crazylord, Playing with Windows /dev/(k)mem, *Phrack*, vol. 59, 2002. `http://www.phrack.org`
18. Jbtzhm, Static Kernel Patching, *Phrack*, vol. 60, 2002. `http://www.phrack.org`
19. Mayhem, Linux x86 kernel function hooking, *Phrack*, vol. 58, 2001. `http://www.phrack.org`
20. Truff, Infecting loadable kernel modules, *Phrack*, vol. 61, 2003. `http://www.phrack.org`

21. Halflife, Shared Library Redirection Techniques, *Phrack*, vol. 51, 1997. `http://www.phrack.org`
22. Silvio Cesare, Shared library call redirection using ELF PLT infection, 1997. `http://www.u-e-b-i.com/silvio/lib-redirection.txt`
23. Mayhem, The Cerberus ELF Interface, *Phrack*, vol. 60, 2002, `http://www.phrack.org`
24. Anonymous author, Runtime Process Infection, *Phrack*, vol. 59, 2002. `http://www.phrack.org`
25. Robert Kuster, Three Ways to Inject Your Code into Another Process, 2003. `http://www.codeproject.com/threads/winspy.asp`
26. Ivo Ivanov, API hooking revealed, 2002. `http://www.codeproject.com/system/hooksys.asp`
27. Yariv Kaplan, API Spying Techniques for Windows 9x, NT and 2000, 2000. `http://www.internals.com/articles/apispy/apispy.htm`
28. Silvio Cesare, Linux anti-debugging techniques (fooling the debugger), 1999. `http://www.u-e-b-i.com/silvio/linux-anti-debugging.txt`

Conférences invitées