Cryptographie : attaques tous azimuts - Comment attaquer la cryptographie sans cryptanalyse intensive ?

Jean-Baptiste Bedrune[1], Éric Filiol[2], and Frédéric Raynal[1,3]

[1] Sogeti ESEC
[2] ESAT
[3] MISC Magazine

Résumé Cet article porte sur les attaques opérationnelles menées contre des outils cryptographiques. Le problème est abordé sous plusieurs angles, l'objectif étant toujours d'obtenir un maximum d'informations sans recourir aux cryptanalyses intensives. Pour cela, nous nous appuyons sur des erreurs, volontaires ou non, dans l'implémentation ou dans l'utilisation de ces outils, ou encore sur des fuites d'informations.

Tout d'abord, nous examinons les attaques directes sur les clés de chiffrement. Nous les recherchons dans les fichiers binaires, dans la mémoire, ou dans des fichiers mémoire (comme le fichier d'hibernation). Nous montrons également comment la mauvaise initialisation de générateurs aléatoires réduit fortement l'entropie de la clé, et comment réduire à néant cette entropie en insérant des portes dérobées.

Ensuite, nous nous mettons dans la position d'un attaquant confronté à des outils cryptographiques. Il doit commencer par détecter que de tels algorithmes sont employés, puis les identifier. Nous présentons des solutions à ces problèmes, aussi bien pour l'analyse de fichiers binaires que pour celle sur des flux de communication.

Parfois, un attaquant n'a accès qu'aux flux chiffrés, sans disposer des outils capables de générer ce flux, et est dans l'incapacité de casser le chiffrement utilisé. Dans ces situations, nous constatons qu'il reste souvent des fuites d'information qui se révèlent largement intéressantes. Nous montrons comment les méthodes classiques de *supervision de réseau*, de *forensics*, ou encore de *sociologie* au travers de l'étude des réseaux sociaux permettent de récupérer des informations pertinentes. Nous construisons par exemple des sociogrammes susceptibles de révéler les éléments clés d'organisation, de déterminer le type d'organisation, etc.

La dernière partie met en application l'ensemble des résultats obtenus précédemment au travers de l'analyse d'un protocole de communication réseau fermé. L'identification du format des paquets s'appuie sur l'analyse du comportement du programme après identification de tous les éléments cryptographiques.

Mots clés : rétro-conception, protocole fermé, chiffrement, cryptanalyse, fuite d'information

1 Introduction et problématiques

Les failles qu'un attaquant peut exploiter quand il se trouve confronté à des outils cryptographiques sont parfois liées aux propriétés mêmes de ces outils ou primitives. En général, les attaques ne portent pas sur les primitives elles-même, mais plutôt sur leur utilisation ou leur implémentation. Tout en cryptographie repose sur la propriété d'aléa des éléments secrets (suite chiffrante des systèmes par flot, clés, qu'elles relèvent de la cryptographie symétrique ou asymétrique) et des quantités produites par les algorithmes de chiffrement, en premier lieu le texte chiffré. Tout est contenu dans la notion même de secret parfait, définie par Shannon [1]. Le canal ne doit pas révéler d'information – il est dit hermétique – et la capture du cryptogramme ne doit apporter qu'une information nulle[4].

Or cette herméticité du canal repose sur le fait que les messages clairs doivent être équiprobables entre eux, ainsi que les clés et les cryptogrammes ; et donc sur la notion d'aléa, ou du moins sur l'idée que nous nous en faisons. Car ce qui semble aléatoire ne l'est peut-être pas vraiment, et ce qui ne semble pas l'être, l'est finalement plus qu'on ne le croit. Tout simplement parce que l'aléa est une notion que nous ne sommes capables de définir que relativement à une batterie de tests, et que ces tests peuvent être manipulés [2] : simulabilité faible et simulabilité forte [3] sont les deux mamelles de tout art de la dissimulation dans l'attaque ou la défense, ces deux notions étant duales. Autrement dit, dès lors que l'attaquant connaît les tests utilisés pour évaluer et valider un système ou un protocole cryptographique – grosso modo, dans l'industrie tout système validé par les tests américains du FIPS 140 [2,4,5] est déclaré bon pour le service –, il est capable de les manipuler et d'introduire une trappe dans le système qui sera indétectable par ces mêmes tests. C'est la raison pour laquelle les pays en pointe dans le domaine de la cryptographie conservent secrets certains des tests utilisés, d'autant plus que cela permet, de manière duale, éventuellement de détecter des faiblesses cryptographiques non détectées par l'adversaire pour ses propres systèmes[5]

Nous nous plaçons donc dans la position d'un attaquant face à un logiciel s'appuyant sur de la cryptographie. La structure de cet article suit la méthodologie de cet attaquant pour arriver à ses fins, sans recourir à une cryptanalyse intensive, trop exigeante en calculs pour être rapidement rentable.

Tout d'abord, il s'agit d'examiner le logiciel afin de vérifier les erreurs (classiques ou volontaires) liées à une mauvaise gestion des clés. Pour cela, on regarde autant le fichier binaire que l'exécution en mémoire, au cas où traîneraient des choses qui ne devraient pas.

La deuxième section aborde le problème de la détection et de l'identification de la cryptographie. Si rien de notable n'est relevé, il faut alors identifier, dans le programme, le code relatif à l'emploi de la cryptographie. Nous proposons une nouvelle approche pour automatiser cette analyse. Parfois, l'attaquant n'a pas accès au logiciel et se contente d'analyser des captures *a posteriori*, hors de tout

[4] Il est important de mentionner que toutes ces notions s'appliquent également au domaine de la cryptographie asymétrique, laquelle repose pourtant sur des propriétés mathématiques, comme la primalité, plus faciles à définir que la notion d'aléa. La seule différence avec la cryptologie symétrique réside dans le fait qu'il est nécessaire de considérer des ensembles beaucoup plus grands pour qu'une fois les éléments aux propriétés mathématiques indésirables – les nombres composites par exemple – identifiés et écartés, les autres éléments éligibles soient encore très nombreux et gouvernés par la loi de l'équiprobabilité.

[5] Dans cet article, nous supposerons qu'aucune technique de simulabilité, forte ou faible, n'est appliquée et que l'attaqué joue le jeu. Mais cette situation a-t-elle des chances de durer encore longtemps ? L'avenir le dira et il sera alors bien plus difficile de gérer les choses, si tant est que cela soit encore possible, du moins en pratique.

contexte qui pourrait lui donner des indices. Nous voyons alors comment reconnaître un flux chiffré, ainsi que le chiffrement mis en œuvre.

Dans la troisième section, nous traitons les risques liés au canal de communication. En effet, dans certains cas, nul besoin d'accéder au contenu en clair, l'étude du canal suffit pour comprendre ce qui se passe. L'existence même des communications – même chiffrées – constitue une fuite d'information qui peut s'avérer payante pour un attaquant. Pour cela, il suffit de regarder le canal de communication. Les méthodes classiques de supervision ou d'analyse post-mortem en révèlent alors beaucoup sur le réseau, et l'analyse de réseaux sociaux sur les personnes.

Enfin, la dernière section est un exemple servant à illustrer les résultats précédents : à partir du logiciel *OneBridge*, une solution de synchronisation développée par Sybase. Nous montrons comment nous avons construit un client à partir de l'analyse du protocole.

2 Attaquer les clés

Les clés, ou plus généralement la notion de *secret*, sont largement utilisées dans de nombreuses primitives cryptographiques. Réussir à y accéder permet à un attaquant d'usurper parfaitement l'identité du propriétaire. En conséquence, la protection des clés devra être un aspect essentiel de la sécurité quand la cryptographie intervient. Néanmoins, dans de nombreux cas, les clés ou plus généralement le secret, sont (volontairement ou non) mal gérées.

2.1 Les faiblesses triviales (mais néanmoins communes)

Lors de l'analyse d'un système, on distingue deux cas pour accéder aux secrets :

1. ceux présents dans les fichiers ;
2. ceux présents dans la mémoire du système à un instant donné.

Dans la suite, nous fournissons quelques exemples associés à ces 2 situations, mais aussi au cas des *fichiers mémoire* (typiquement, le swap ou le fichier d'hibernation).

Il faut bien comprendre que ces erreurs suivent une construction logique. Tout d'abord, un fichier est stocké sur le disque dur. Puis, à n'importe quel moment, le système peut choisir de le mettre en mémoire, par exemple pour l'exécuter. Si un secret (ex. : un mot de passe) est demandé, il se trouvera également en mémoire. Il suffit donc de lire cette mémoire au bon moment pour trouver le secret. En outre, une fois le secret utilisé, rares sont les programmes qui "nettoient" la mémoire. Où que soit stocké le secret, il faut le remettre à 0 avant de libérer la mémoire qu'il occupe dès que nous avons fini de nous en servir. Si tel n'est pas le cas, le secret reste tant que la zone mémoire n'est pas utilisée pour autre chose. Dès lors, si le processus est swappé ou l'ordinateur mis en hibernation, il est normal de retrouver ces éléments de secret dans les fichiers correspondants.

Les clés dans les fichiers Commençons par le cas des fichiers. Il s'agit ici de programmes qui contiennent eux-mêmes les clés employées pour les opérations cryptographiques. Naturellement, nous pensons aux codes malicieux polymorphes dont une première partie est une boucle de déchiffrement (souvent un simple `xor` ou `add` avec un mot machine).

Nous trouvons également un tel comportement dans des logiciels grand public et professionnels. Citons l'exemple d'une application destinée à gérer des salaires, fiches de paie, etc. Elle est composée d'un serveur web qui fournit les formulaires que l'utilisateur doit remplir pour enregistrer les opérations. Quand nous nous connectons sur le serveur, une applet Java est chargée côté client. En la récupérant puis en la décompilant, nous trouvons le code suivant dans le fichier `RSACoder.java` (voir listing 1.1) :

```
public static final byte h[] = {
    48, -127, -97, 48, 13, 6, 9, 42, -122, 72, -122, -9, 13, 1, 1, 1, 5, 0, 3, -127,
    ... };

public static final byte f[] = {
    48, -126, 2, 117, 2, 1, 0, 48, 13, 6, 9, 42, -122, 72, -122, -9, 13, 1, 1, 1,
    ...};

public static final byte j[] = {
    48, 92, 48, 13, 6, 9, 42, -122, 72, -122, -9, 13, 1, 1, 1, 5, 0, 3, 75, 0,
    ...};
```

```
public static final byte a[] = {
    48, -126, 1, 83, 2, 1, 0, 48, 13, 6,9, 42, -122, 72, -122, -9, 13, 1, 1, 1,
    ...};
```

Listing 1.1: Clés RSA dans une classe Java

Cette application est capable de fonctionner en RSA 512 et 1024. Ces tableaux de nombres représentent en fait les clés publiques et privées pour chaque mode aux formats PKCS#1 et PKCS#8.

Nous voyons ici une erreur de programmation flagrante. Dans un souci d'optimisation, les développeurs ont rassemblé les primitives liées à RSA dans la seule classe `RSACoder`, avant de la dériver respectivement en `RSAEncoder` et `RSADecoder`. Or la communication étant à sens unique entre le client et le serveur, le client n'a besoin que de `RSAEncoder`, et certainement pas des clés privées.

Il devient trivial pour cette application de déchiffrer les jetons d'authentification, d'autant que les autres classes de l'applet en donnent la syntaxe.

```
#!/usr/bin/env python
import base64

p=int("DF...", 16)
q=int("B5...", 16)
n=p*q
d=int("E0...", 16)
token="k4...=="

msg=base64.b64decode(token)
c = int("".join(["%02x" % ord(i) for i in msg]), 16)

enc = hex(pow(c, d, n)) # chiffrement RSA
dec = ""
for i in range(2, len(enc)-1, 2):
    dec += chr(int(enc[i:i+2], 16))

pos = 1
l = int(dec[0]);
print "login=[%s]" % (dec[pos:pos+l])
pos+=l
l = int(dec[pos]);
print "pwd=[%s]" % (dec[pos:pos+l])
pos+=l
l = 7
print "sessionID=[%s]" % (dec[pos:pos+l])
pos+=l
print "tstamp=[%s]" % (dec[pos:])
```

Listing 1.2: Décodage de jetons d'authentification

À l'aide du programme donné en listing 1.2, la compréhension devient triviale :

```
>> ./decodetoken.py
login=[FRED_O]
pwd=[SECRET]
sessionID=[RAU26S03]
tstamp=[2007-12-17-15.54.14]
```

Cependant, les choses sont rarement aussi simples. Déjà, il n'est pas toujours possible d'accéder aux sources, ce qui ne signifie pas pour autant que des clés ne soient pas présentes dans des binaires.

Autant il est assez aisé de parcourir les fichiers source à la recherche de noms explicites relatifs à l'usage de cryptographie, autant cela peut s'avérer bien plus complexe dans des binaires.

A. Shamir, N van Someren dans [6] et E. Carrera dans [7,8] proposent une approche fondée sur l'entropie pour détecter les clés dans les binaires. Ce dernier fait l'hypothèse que, lorsque les clés sont stockées sous forme binaire, leur entropie est plus élevée que celle du reste du fichier. Comme souligné dans [7,8], ce n'est plus le cas quand elles sont représentées sous forme de nombre ASCII ou en ASN.1. En plus, cela ne fonctionne que pour les clés de grande taille[6].

Les clés dans la mémoire Rechercher des clés dans la mémoire d'un système peut s'apparenter à rechercher une aiguille dans une botte de foin. Nous pouvons citer, pour le lecteur qui douterait encore, les résultats fournis par A. Bordes [9] ou plus récemment les *cold boot attacks* [10].

Dans cette section, nous prenons comme exemple l'agent ssh. Il sert à stocker en mémoire des clés privées. En effet, il est recommandé de protéger ses clés privées avec une *passphrase* demandée à chaque utilisation. Dans le cas d'un emploi intensif des clés privées, il faut re-saisir maintes et maintes fois la passphrase. Afin d'éviter cela, l'agent conserve les clés en mémoire. Pour cela, nous activons d'abord l'agent :

```
>> eval `ssh-agent `
>>  env|grep -i ssh
SSH_AGENT_PID=22157
SSH_AUTH_SOCK=/tmp/ssh-ULWEQ22156/agent.22156
```

Ensuite, quand nous voulons ajouter une clé, nous utilisons `ssh-add` :

```
>> ssh-add
Enter passphrase for /home/jean-kevin/.ssh/id_dsa:
Identity added: /home/jean-kevin/.ssh/id_dsa (/home/jean-kevin/.ssh/id_dsa)
```

Ne disposant que d'une clé privée, il la prend immédiatement en compte. Nous pouvons vérifier les clés présentes en mémoire :

```
>> ssh-add -l
1024 b2:d4:19:92:c8:7e:00:1a:2b:06:63:02:21:10:45:35 /home/jean-kevin/.ssh/id_dsa (DSA)
```

Cette commande affiche l'empreinte de la clé publique. En examinant les sources, nous constatons que l'agent fonctionne avec une socket UNIX et dispose d'un protocole très rudimentaire. Par exemple, il peut calculer un challenge ou renvoyer des clés publiques. Rien n'est prévu – et heureusement – pour accéder directement aux clés privées.

Ainsi, quand le client `ssh` a besoin de s'authentifier auprès d'un serveur, il se transforme alors en relais entre le serveur et l'agent : le serveur envoie le challenge, le client le transmet à l'agent qui calcule la bonne réponse en fonction de la clé privée en sa possession, puis repasse cette réponse au client qui la retourne au serveur.

Étant donné qu'aucun moyen n'est prévu pour accéder directement aux clés privées, nous allons devoir aller les chercher en mémoire. Signalons d'emblée que la protection de la mémoire mise en

[6] Son travail portait sur les clés RSA, ce qui explique cette hypothèse sur les tailles de clés.

place par l'agent est efficace. En l'état, l'utilisateur lui-même ne peut pas accéder à la mémoire du processus, seul root le peut. En effet, par défaut, sous Linux, la génération de fichier `core` est interdite à cause de l'appel `prctl(PR_SET_DUMPABLE, 0);`. Cela a également comme effet de bord d'interdire à l'utilisateur le `ptrace()` du processus.

Il nous faut maintenant regarder comment sont organisées les structures en mémoire. L'agent conserve toutes ses informations dans un tableau `idtable` qui contient 2 listes, l'une pour les clés de type RSA1, l'autre pour celles en RSA ou DSA. Pour chaque clé, une structure `identity` contient, outre un pointeur vers la clé elle-même, des informations diverses comme la durée de vie en mémoire. Par défaut, cette durée est illimitée. Les clés sont déclarées dans une structure interne à OpenSSH (cf. listing 1.3, dans `$OPENSSH/Key.h`)

```
struct Key {
    int  type;
    int  flags;
    RSA *rsa;
    DSA *dsa;
};
```

Listing 1.3: Structure des clés dans OpenSSH

Ainsi de suite, nous remontons les structures tant qu'il y a des pointeurs pour tout reconstruire. Les types `RSA` et `DSA` qui contiennent les clés recherchées sont propres à OpenSSL. Au final, la figure 1 page suivante décrit l'organisation de la mémoire. L'analyse des structures doit également comprendre la portée de ces structures. Ainsi, le programme réalisé pour récupérer les clés pourra fonctionner plus largement. Nous notons ainsi que le même programme pourra agir sur OpenSSH dès qu'il s'agira de récupérer une structure de type `Key`, ou avec tout programme manipulant des objets `DSA` d'OpenSSL (par exemple des certificats pour Apache).

Pour bien comprendre l'organisation de la mémoire et les structures impliquées, nous débuggons un processus. D'après les sources, les clés privées (et quelques autres informations) sont pointées par des variables globales, que nous retrouvons en mémoire dans les zones `.bss` et `.data` selon qu'elles sont initialisées ou non.

À titre illustratif, nous récupérons les clés DSA placées dans l'agent. Le point de départ est `idtable[2]` dont nous récupérons l'adresse en examinant la zone `.data`.

```
>> objdump -h `which ssh-agent`
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  23 .data         00000034  00013868  00013868  00012868  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 24 .bss          000024ec  000138a0  000138a0  0001289c  2**5
                  ALLOC

>> sudo gdb -q -p $SSH_AGENT_PID
(gdb) dump memory data.mem 0x08059000 0x0805a000
                idtable[1]                idtable[2]
          [nentries ] [ idlist  ]  [nentries ] [ idlist  ]
08059c50  00 00 00 00 50 9c 05 08  01 00 00 00 40 44 06 08  |....P.......@D..|
```

La structure `identity` se trouve donc en 0x08064440 et ne contient qu'une clé :

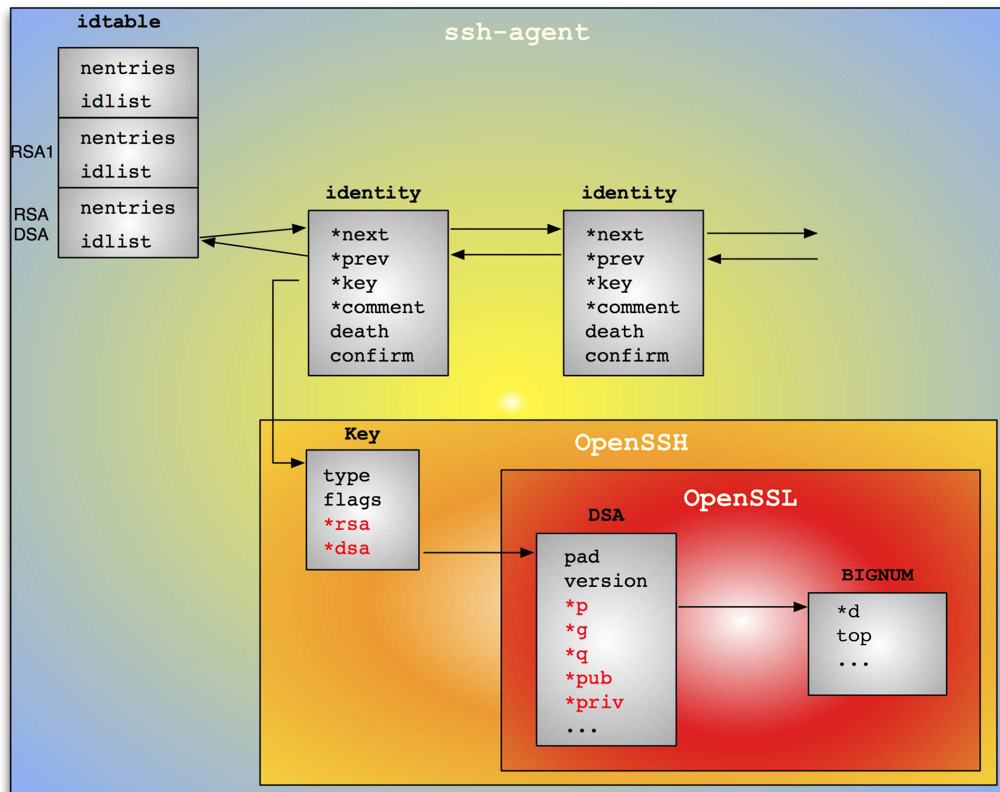


Fig. 1: Organisation des structures liées aux clés dans l'agent ssh

```
(gdb) x/8x 0x08064440
[next  ] [**prev  ] [*key     ]       [*comment]
0x8064440: 0x00000000 0x08059c5c 0x08064218 0x080643e0
(gdb) x/s 0x080643e0
0x80643e0:  "/home/jean-kevin/.ssh/id_dsa"
```

Le pointeur sur `comment` contient en réalité le nom du fichier clé : nous sommes sur la bonne piste car il s'agit bien d'une clé privée DSA. Examinons la clé `Key` située en 0x08064218

```
(gdb) x/4x 0x08064218
type flags *rsa *dsa
0x8064218: 0x00000002 0x00000000 0x00000000 0x08069748
KEY_DSA
```

La véritable structure décrivant la clé est de type `DSA` en 0x08069748 :

```
          [   pad    ] [ version ]  [  param  ] [    *p   ]
08069748: 00 00 00 00 00 00 00 00  01 00 00 00 90 97 06 08  ................
          [    *q    ] [    *g   ]  [   *pub  ] [ *priv   ]
08069758: a8 97 06 08 c0 97 06 08  d8 97 06 08 80 96 06 08  ................
          [  *kinv   ] [    *r   ]  [  flags  ] [  *meth  ]
08069768: 00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  ................
          [   ref    ] [                      ] [         ]
08069778: 01 00 00 00 00 00 00 00  72 9b dd 5c 00 ba f6 b7  ........r.._..
          [ *engine ]
08069788: 00 00 00 00                                       ....
```

Il nous reste à récupérer les `BIGNUM` pour `p, q, g, pub, priv` pour avoir la connaissance de la clé privée.

Une fois cette analyse réalisée, il reste à écrire un programme (cf. listing 1.4) et lui donner un point de départ pour lire la mémoire : soit nous donnons une adresse connue, comme nous l'avons fait ici à titre d'exemple avec `idtable`, soit nous nous appuyons sur des heuristiques simples pour retrouver les éléments en mémoire. Les structures employées par OpenSSL ont des signatures caractéristiques qui permettent de les retrouver assez facilement en mémoire.

```c
int  dump_bignum(BIGNUM *num, void *addr)
{
  BIGNUM *agent_num;

  memread((long)addr, sizeof(*num));
  memcpy(num, fmembuf, sizeof(*num));
  agent_num = (BIGNUM *)fmembuf;

  if (bn_wexpand(num, agent_num->top) == NULL)
    return 0;

  memread((long)agent_num->d, sizeof(agent_num->d[0])*agent_num->top);
  memcpy(num->d, fmembuf,sizeof(num->d[0])*num->top);
  return 1;
}

int dump_private_key_dsa(DSA *dsa, void *addr)
{
  ...
  if ((dsa->p = BN_new()) == NULL)
    fatal("dump_private_key_dsa: BN_new failed (p)");
  if (!dump_bignum(dsa->p, (void*)agent_dsa.p))
    return 0;
  ...
}

int dump_private_key(Key *key, void *addr) {
  ...
  switch (key->type) {

    case KEY_RSA1:
    case KEY_RSA:
      return dump_private_key_rsa(key->rsa, ((Key*)(fmembuf))->rsa);
    case KEY_DSA:
      key->dsa = DSA_new();
      return dump_private_key_dsa(key->dsa, ((Key*)(fmembuf))->dsa);
    default:
      break;
  }
}
```

```
int  dump_idtable(void *addr) {

  Idtab agent_idtable[3];

  memread((void*)addr, sizeof idtable);

  for (i=0; i<3; i++) {

    for (nentry = 0; nentry<agent_idtable[i].nentries; nentry++) {
      memread((void*)id, sizeof(Identity));
      if (dump_private_key(key, ((Identity*)fmembuf)->key)) {

    snprintf(buf, sizeof buf, "/tmp/key-%d-%d", i, nentry);
    if (!key_save_private(key, buf, "", "ssh-dump powered"))
      fprintf(stderr, "  unable to save key\n");
    else
      fprintf(stderr, "  key saved in %s\n", buf);
    }
  }
}
```

Listing 1.4: Pseudo code pour dumper les clés ssh

La réalisation de ce programme étant à vocation pédagogique, nous nous appuyons en outre sur de multiples fonctions propres à OpenSSH, comme `key_save_private()` qui sauvegarde la clé récupérée directement dans un format compréhensible par OpenSSH.

```
>> nm ./ssh-agent|grep idtable
0804a980 t idtab_lookup
08059c20 B idtable
>> eval `./ssh-agent`
Agent pid 2757
>> export SSH_AGENT_PID=2757 ./ssh-dump -i 0x08059c20
Dumping pid=2757
dump_idtable(): read 1 item(s)
08059c20: 00 00 00 00 00 00 00 00  24 9c 05 08 01 00 00 00  ................
08059c30: b8 4d 06 08 b8 4d 06 08  01 00 00 00 40 44 06 08  .M...M......@D..
08059c40: 40 44 06 08                                       @D..

*** idtab[0]=0 ***

*** idtab[1]=0 ***

*** idtab[2]=1 ***
Dumping identity 0
dump_private_key_dsa(): read 1 item(s)
08064230: 00 00 00 00 00 00 00 00  01 00 00 00 68 43 06 08  ............hC..
08064240: 80 43 06 08 98 43 06 08  b0 43 06 08 c8 43 06 08  .C...C...C...C..
08064250: 00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  ................
08064260: 01 00 00 00 00 00 00 00  00 00 00 00 00 3a f4 b7  .............:..
08064270: 00 00 00 00                                       ....
  key saved in /tmp/key-1-0
Bye bye agent=2757
```

La clé est directement utilisable par OpenSSL :

```
>> sudo openssl dsa -in /tmp/key-1-0 -text
read DSA key
Private-Key: (1024 bit)
priv:
    00:9a:80:14:08:4e:38:a3:44:77:dd:cf:59:bc:12:
    d1:d3:46:78:a2:e9
pub:
    00:91:e7:27:3b:61:94:e3:9a:ec:d6:60:2b:95:f5:
    9b:95:0a:fc:fb:e1:e0:b4:c9:b3:8f:ec:6c:2f:f7:
    ...
P:
    00:c6:df:b9:2a:25:c0:f2:28:41:0e:91:6a:b5:4c:
    9e:06:3e:ac:fd:ce:8d:96:f6:8b:c7:d5:93:af:7c:
    ...
Q:
    00:e5:3a:ac:db:8b:6a:27:42:a9:ed:a4:40:a0:01:
    48:a9:61:33:03:29
G:
    00:bd:e7:9a:d7:38:3d:f0:94:de:a3:b2:07:de:fb:
    4f:c0:ee:da:f6:f6:fa:f4:93:c3:22:1a:8c:59:8c:
    ...
>>
```

Ou encore en tant qu'identité pour OpenSSH :

```
>> ssh -i /tmp/key-1-0 batman
You have mail.
Last login: Wed Apr  2 17:46:45 2008 from gotham
batman>>
```

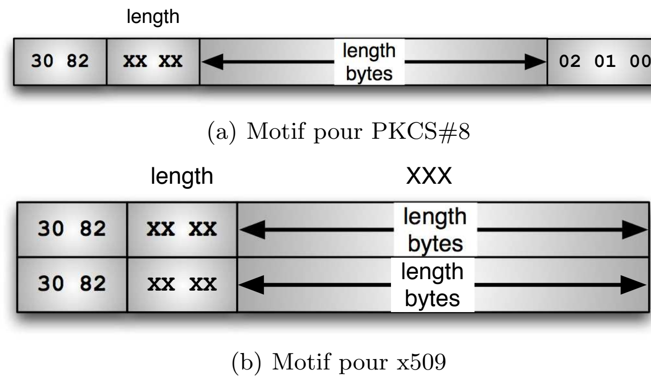
Ainsi, nous récupérons les clés stockées en mémoire. Dans le cas de l'agent OpenSSH, il est normal que les clés se trouvent toujours en mémoire puisque c'est son rôle de les conserver.

Plus généralement, il est recommandé de distinguer les opérations d'accès à la mémoire, de la fouille. Si nous prenons le cas de certificats SSL, ils sont stockés au format PKCS#8 pour les clés privées, et x509 v3 pour les certificats. La figure 2 page suivante représente les motifs à rechercher en mémoire pour retrouver ces clés. Dans ce cas, nous *dumpons* toute la mémoire d'un processus cible (Apache par exemple), et nous recherchons ensuite les informations voulues.

Dans toutes ces recherches, il faut garder à l'esprit que ce n'est pas parce que le secret n'est pas en cours d'utilisation qu'il n'est plus présent en mémoire ...

Les clés en fichier de mémoire Comme expliqué en introduction, une fois un secret en mémoire, il est susceptible de se retrouver ensuite sur le disque dur si des mesures ne sont pas prises. Il existe deux types de fichiers reflétant l'état de la mémoire à un instant donné : le swap et le fichier d'hibernation.

Le swap a rapidement été considéré comme une source potentielle de problème. En particulier, des mesures durent être prises pour éviter que des données stockées sur une partition chiffrée ne



(a) Motif pour PKCS#8

(b) Motif pour x509

Fig. 2: Motifs à rechercher en mémoire

se retrouvent en clair dans le swap[7]. La démarche suivie par OpenBSD suit d'ailleurs cette logique [11]. Depuis, la plupart des systèmes d'exploitation proposent une option similaire, pas toujours par défaut comme nous le verrons.

Le fichier d'hibernation a fait l'objet d'une attention bien plus récente [12]. Ce fichier est utilisé quand le système se met en sommeil. Le noyau copie alors l'intégralité de la mémoire sur le disque dur, dans un fichier dédié, et règle quelques paramètres pour que le réveil se passe comme un reboot, mais en utilisant ce fichier d'hibernation en guise de noyau : tout se retrouve ainsi dans l'état antérieur à la mise en veille. Là encore, si des informations précieuses se trouvent en mémoire, elles sont alors placées sur le disque dur.

Avec le bruit autour des *cold boot attacks*, l'exploration de la mémoire redevient à la mode, et de nombreux articles présentent cela en détail, au travers d'approches variées. En conséquence, nous ne donnerons qu'un bref historique de ce qui se fait sous Mac OS X. Les tests ont été menés sous Tiger et Leopard, avec des résultats identiques.

En 2004, une première alerte a été donnée [13] indiquant que le swap contenait le mot de passe de session (entre autres) en clair. Cela a depuis été fixé, d'autant que les Unix disposent d'un appel système spécifique, `mlock()` pour empêcher certaines zones mémoire de se retrouver en swap. Dès lors, il est évident que si ce mot de passe est dans le swap, il vient d'une application, en particulier celle qui gère les sessions, `Loginwindow.app`. Effectivement, les auteurs de la *cold boot attack* le retrouvent 4 ans plus tard [14]. Quand nous savons comment fonctionne un système d'exploitation, il n'est alors pas étonnant de retrouver aussi le mot de passe dans le fichier d'hibernation[8]. En effet, `mlock` empêche la mise en swap, mais pas la mise en hibernation.

À titre illustratif, nous montrons comment récupérer cette information dans les deux cas. Commençons en examinant la mémoire. Le processus `Loginwindow` sert à un utilisateur pour démarrer sa session. [14] rappelle que le mot de passe se trouve en clair dans la mémoire. Nous écrivons un script python (cf. listing 1.5 page ci-contre) qui créer des *dumps* mémoire à l'aide de `vmmap`[9] et `gdb`.

[7] D'autres mesures, sur la gestion des pages mémoire sont également proposées par ailleurs, mais sortent du cadre de cette étude.

[8] Étrangement, il n'y a aucune mention à cela nul part ...

[9] Il s'agit d'un outil propre à Mac OS X, affichant les zones mémoire d'un processus.

Nous constatons que le mot de passe se trouve dans deux zones mémoire, en clair (le mot de passe a été remplacé par `XXXXXXXXXXXXXXX`) :

```
>> sudo ./mdosx.py 103 stack malloc
Password:
['stack', 'malloc']
Dumping PID 103 ... done
>> ls *raw
ls *raw
103-malloc-00065000-00070000.raw 103-malloc-007a9000-007aa000.raw
103-malloc-01827000-01829000.raw 103-stack-b009e000-b009f000.raw
103-malloc-00080000-00084000.raw 103-malloc-007e7000-007f2000.raw
103-malloc-01829000-0182d000.raw 103-stack-b009f000-b00b7000.raw
...
>> grep -mc "XXXXXXXXXXXXXXX" *raw
Binary file 103-malloc-00300000-00405000.raw matches
Binary file 103-malloc-01800000-01826000.raw matches
```

Dans son *advisory* sur le swap [13], l'auteur signale que le mot de passe est très souvent proche du mot `longname`, ce qui se vérifie ici dans le second fichier :

```
>>  strings -a 103-malloc-01800000-01826000.raw|grep -A3 longname
longname
Jean-Kevin LeBoulet
password
XXXXXXXXXXXXXXX
```

Le mot `longname` constitue effectivement un bon point de repère puisque nous trouvons bien l'information recherchée.

```
#!/usr/bin/env python

import os, sys, re

pid = sys.argv[1]
reg = sys.argv[2:]

f = os.popen("vmmap -w "+pid)
lines = f.readlines()
f.close()
gdb = open("dump.gdb", "w")
gdb.write("attach %s\n" % pid)

for l in lines:
    for p in reg:
        prg = re.compile(p, flags=re.IGNORECASE)
        res = prg.match(l)
        if res:
            l = l[len(p):].strip()
            prg = re.compile("[\da-f]+-[\da-f]+", flags=re.IGNORECASE)
            res = prg.search(l)
            if not res: continue
            b, e = res.span()
            mem = l[b:e]
```

```
            start, end =  mem.split('-')
            f = "%s-%s-%s.raw" % (pid, p.lower(), mem)
            cmd = "dump mem %s 0x%s 0x%s" % (f, start, end)
            gdb.write(cmd+"\n")

gdb.write("quit\n")
gdb.close()

os.system("gdb -q -x dump.gdb")
```

Listing 1.5: Dump mémoire sous Mac OS X

Passons maintenant au cas du fichier d'hibernation. Recherchons comme indiqué dans [13] à partir de `longname` :

```
>> sudo strings -8 /var/vm/sleepimage |grep -A 4 -i longname
longname
JeanKevLeB
password
XXXXXXXXXXXXXXX
shel/bin/basshouldunmoun
```

En fouillant, le mot de passe se trouve **10** fois dans le fichier d'hibernation :

```
>> sudo grep -c XXXXXXXXXXXXXXX /var/vm/sleepimage
10
```

Précisons que les informations (utilisateurs / mots de passe) dans le fichier d'hibernation sont celles liées au dernier utilisateur pour qui la machine est entrée en hibernation. Une extinction ou un reboot n'efface pas ce fichier.

Quels sont les risques, sachant que ces fichiers ne sont accessibles uniquement quand nous disposons déjà d'un maximum de privilèges ? Déjà, toutes les attaques *offline* fonctionnent, c'est-à-dire que la simple copie du disque dur, puis son analyse *a posteriori* suffisent pour révéler les secrets. À ce titre, un boot sur une clé usb suffit. Le lecteur pourra regarder [15] pour voir comment construire une clé qui modifie au minimum la mémoire du système. Cependant, des accès directs à la mémoire, par exemple via FireWire (cf. dans cette même édition de SSTIC [16]), donnent suffisamment de droits pour réussir.

Mais la recherche du mot de passe n'est pas si triviale qu'il paraît. Comme avec les outils présentés dans [15], la recherche est ici facilitée par à la connaissance du mot de passe : nous vérifions sa présence, plus que nous ne cherchons véritablement à l'extraire. Pour réussir cela, il faut regarder un peu plus précisément comment sont construits ces fichiers ou l'espace mémoire des processus (comme nous l'avons montré précédemment pour ssh).

2.2 Dérivation de clés et PRNG

Une des difficultés majeures dans les solutions de chiffrement, après l'implémentation des primitives, est la génération d'aléa, utilisé notamment lors de la génération des clés de chiffrement. Certaines solutions génèrent des aléas en fonction de paramètres trop prédictibles, ou dont la taille de l'espace est trop faible. D'autres n'utilisent que les fonctions de génération de nombres aléatoires des bibliothèques standards, loin d'être sûres d'un point de vue cryptographique.

Générateur Delphi Delphi est un langage souvent utilisé pour des programmes destinés au grand public. Il dispose d'un générateur pseudo-aléatoire qui n'est pas reconnu comme cryptographiquement sûr. Il est donc préférable *a priori* de ne pas l'utiliser. Pourtant, la majorité des programmes Delphi[10] s'appuie dessus.

Le générateur est initialisé avec la fonction `Randomize`, et retourne des valeurs de 32 bits via la fonction `Random` (voir la fonction identique `_RandInt` ci-après). C'est l'équivalent des fonctions `rand` et `srand` en C, qui elles ne sont quasiment jamais utilisées dans les applications de chiffrement[11], contrairement aux primitives équivalentes Delphi.

Dans les versions 5 et antérieures, la fonction d'initialisation calcule une graine `RandSeed` uniquement à l'aide de l'heure système, correspondant au nombre de millisecondes écoulées depuis le début de la journée :

```
procedure Randomize;
var
  st: _SYSTEMTIME;
begin
  GetSystemTime(st);
  RandSeed := ((st.wHour * 60 + st.wMinute) * 60 + st.wSecond) * 1000
    + st.wMilliseconds;
end;
```

Il y a donc 86400000 initialisations possibles, soit une entropie d'un peu plus de 26 bits. Cette entropie, déjà faible, peut être considérablement réduite si on a une connaissance approximative de l'heure à laquelle le générateur a été initialisé.

Les versions ultérieures initialisent le générateur avec une graine légèrement moins prédictible, reposant non pas sur l'heure système mais sur l'*uptime* de la machine. Sa taille est toujours de 32 bits :

```
procedure        Randomize;
var
  Counter: Int64;
begin
  if QueryPerformanceCounter(Counter) then
    RandSeed := Counter
  else
    RandSeed := GetTickCount;
end;
```

Les valeurs aléatoires sont ensuite générées ainsi :

```
procedure        _RandInt; //"alias" de Random
asm
{     ->EAX      Range    }
{     <-EAX      Result   }
        PUSH     EBX
        XOR      EBX, EBX
        IMUL     EDX,[EBX].RandSeed,08088405H
        INC      EDX
        MOV      [EBX].RandSeed,EDX
        MUL      EDX
        MOV      EAX,EDX
        POP      EBX
end;
```

---

[10] Si ce n'est tous . . .

[11] Le seul exemple que nous connaissions est `PasswordSafe 3.0` de Bruce Schneier

La mise à jour de la graine à chaque appel de `RandInt` est donc :

```
RandSeed := RandSeed * $8088405 + 1;
```

Quel que soit l'état du générateur, nous pouvons revenir à son état précédent en décrémentant la graine et en la multipliant par l'inverse de `$8088405` :

```
RandSeed := (RandSeed - 1) * $d94fa8cd; // 0xd94fa8cd = 1 / 0x8088405 Mod 2**32
```

Supposons que des clés de chiffrement soient générées avec `Random`. Nous retrouvons ces clés par recherche exhaustive : nous calculons toutes les clés avec chaque valeur possible de la graine, et nous testons chacune d'entre elles. Il y aura au maximum 2^{32} opérations en ne faisant aucune hypothèse sur la valeur de la graine, ce qui est très raisonnable.

Voici le cas (fictif) d'un programme chiffrant des données dont nous connaissons les premiers octets. La clé a été générée ainsi :

```
for i := 0 to 15 do result := result + IntToHex(Random($100), 2);
```

La recherche de la clé se fait en quelques étapes :

1. on génère une graine ;
2. on génère une nouvelle clé à partir de cette graine ;
3. on chiffre un clair connu ;
4. si le chiffré correspond au chiffré de référence, c'est qu'on a trouvé la bonne clé, sinon, on recommence au début.

Le code suivant met cela en œuvre :

```
unsigned int RandSeed;

/* RandInt rippe du code original, legerement modifie */
unsigned int __declspec(naked) __fastcall RandInt(unsigned int LimitPlusOne)
{
    __asm{
        mov eax, ecx
        imul edx, RandSeed, 8088405h
        inc edx
        mov RandSeed, edx
        mul edx
        mov eax, edx
        ret
    }
}

int main()
{
    const unsigned char encrypted[16];
    const char plaintext[] = "-----BEGIN BLOCK-----"; /* Le texte en clair */
    unsigned char block[] = {                         /* Le texte chiffre */
        0x54, 0xF6, 0x0C, 0xB8, 0x78, 0x99, 0x55, 0xF1,
        0x46, 0x83, 0xB3, 0x96, 0x7F, 0x79, 0xCD, 0x80
    };
    unsigned char k[16];
    unsigned int current_seed = 0;
    int i = 0;

  /* Bruteforce de la graine pour obtenir la cle */
    do
```

```
    {
        current_seed++;
        RandSeed = current_seed;
        for(i = 0; i < 16; i++)
            k[i] = RandInt(0x100);
        aes128_setkey(&ctx, k);
        aes128_encrypt(&ctx, plaintext, encrypted);
    } while(memcmp(encrypted, block, 16));
    printf("%x\n", current_seed);
    for(i = 0; i < 16; i++)
        printf("%02x ", k[i]);
    return 0;
}
```

Une faiblesse dans un RNG est souvent longue à expliquer, c'est pourquoi nous nous sommes limités à un cas simple. Un exemple un peu plus consistant a été traité dans [17], et dans une édition précédente du SSTIC [18].

2.3 Des clés et des trappes

Les *backdoors* qu'on peut trouver "dans la vraie vie" sont principalement de deux types :

- un affaiblissement volontaire de l'entropie, afin de retrouver les clés utilisées en un temps raisonnable.
  Prenons un exemple connu. En 1997, le gouvernement suédois audite Lotus Notes, la suite d'IBM pour gérer messagerie et conférences. Grand bien lui en prend car il découvre une faiblesse importante dans le chiffrement des messages.
  À l'époque, les États-Unis interdisaient l'exportation de produits de cryptographie proposant des clés trop grandes. Ce fut la position prise par IBM pour se défendre, comme en atteste Eileen Rudden, vice-président de Lotus : *The difference between the American Notes version and the export version lies in degrees of encryption. We deliver 64 bit keys to all customers, but 24 bits of those in the version that we deliver outside of the United States are deposited with the American government. That's how it works today.*
  Ces 24 bits constituent une différence critique. Pour qui ne les possède pas, casser une clé de 64 bits est pratiquement impossible. En revanche, casser une clé de 64 bits quand il n'en reste que 40 à deviner prend quelques secondes sur un ordinateur rapide.
  Signalons de plus qu'à la même époque, ce logiciel équipait aussi le ministère de la défense allemand, le ministère de l'éducation français ... et bien d'autres.
- un séquestre des clés : un fichier est chiffré. Les clés de chiffrement sont ensuite chiffrées avec un système à clé publique, dont la clé secrète est détenue par une entité externe. Le résultat est ensuite ajouté au fichier chiffré.

Un exemple de séquestre. Le programme présenté ici est un logiciel de chiffrement commercial assez répandu. Il existe deux versions d'évaluation, très différentes quand à leur fonctionnement interne : algorithmes de chiffrement, structure des fichiers chiffrés, etc.). Les deux versions contiennent une porte dérobée. Nous étudions ici la version la plus récente.

Le programme est écrit en C++ et fait un usage intensif de l'API Windows. Il compresse puis chiffre les données, après avoir créé une clé protégée avec le mot de passe utilisateur.

En traçant le programme pas à pas, nous rencontrons lors de la génération d'aléa une fonction étrange : elle utilise plusieurs chaînes d'octets, décodées durant l'exécution du programme. Il n'y a dans le reste du programme aucune protection visant à camoufler des données ou du code. De plus,

aucune API n'est appelée, et la structure du code n'a rien à voir avec le reste du programme : nous avons une forte impression que cette partie a été développée par une autre personne.

Elle semble prendre comme paramètre d'entrée une chaîne d'octets de 64 caractères, que nous noterons m. Un buffer local est initialisé avec des constantes arbitraires :

```
.text:1000B5C9  mov   edi, 0B8C0FEFFh
.text:1000B5CE  mov   ebx, 4A81EB01h
.text:1000B5D3  mov   esi, 723A8743h
.text:1000B5D8  mov   dword ptr [esp+12Ch+var_encoding], edi
.text:1000B5DC  mov   dword ptr [esp+12Ch+var_encoding+4], ebx
.text:1000B5E0  mov   dword ptr [esp+12Ch+var_encoding+8], esi
```

Il sert à décoder 5 chaînes d'octets différentes. La valeur initiale de la première chaîne d'octets est :

```
.rdata:1006272C bignum1     db 72h, 0C3h, 3Ch, 6, 8Bh, 0C5h, 0BFh, 42h, 84h, 76h, 86h
.rdata:1006272C             db 59h, 79h, 0EAh, 0D3h, 23h, 0A0h, 13h, 5, 0B1h, 28h
.rdata:1006272C             db 1Bh, 4Ch, 0B9h, 57h, 0F5h, 73h, 58h, 6Ah, 31h, 0E1h
.rdata:1006272C             db 4Dh
```

La boucle de décodage, très simple, est :

```
.text:1000B5E7 @@loop1:
.text:1000B5E7      xor    edx, edx
.text:1000B5E9      mov    eax, ecx
.text:1000B5EB      mov    ebp, 0Ch
.text:1000B5F0      div    ebp
.text:1000B5F2      mov    al, [esp+edx+130h+var_encoding]
.text:1000B5F6      xor    al, ds:bignum1[ecx]
.text:1000B5FC      mov    [esp+ecx+130h+var_20], al
.text:1000B603      inc    ecx
.text:1000B604      cmp    ecx, 20h
.text:1000B607      jnz    short @@loop1
```

Elle est répétée 5 fois pour décoder chaque buffer. Nous les décodons manuellement un par un :

```
>>> import binascii
>>> def decode_buffer(buf):
     data = binascii.unhexlify(buf)
     k = binascii.unhexlify("fffec0b801eb814a43873a72")
     return binascii.hexlify("".join([chr(ord(k[i % 12])^ord(data[i]))
       for i in range(len(data))]))

>>> decode_buffer(bignum1)
'8d3dfcbe8a2e3e08c7f1bc2b8614139ba1f884fb6b9c76cba80bb3e06bda6007'
>>> decode_buffer(bignum2)
'0000000000000000000000000000000000000000000000000000000000000078'
>>> decode_buffer(bignum3)
'0000000000000000000000000000000000000000000000000000000000000083'
>>> decode_buffer(bignum4)
'2455367da071c81b65cf4c63ba7db614aa6915c065a0c3bc046f50630b8c1872'
>>> decode_buffer(bignum5)
'42840dc8199d1620ca8000e82d2e04b011ae07095dd2d4f649cdce1086993b70'
```

`bignum1` est un nombre premier. Les deux valeurs suivantes ont une forme bien particulière : seul leur dernier octet n'est pas nul.

Deux autres buffers étranges (ils ne correspondent pas à des constantes utilisées dans un algorithme de chiffrement) se trouvent juste en dessous. Ils ne sont pas décodés.

```
.rdata:100627CC bignum6      db 0, 0BBh, 5Ah, 0Ch, 99h, 0F6h, 2 dup(0B9h), 0ACh, 9Ah
.rdata:100627CC              db 49h, 91h, 0A2h, 90h, 2Dh, 0A7h, 23h, 0E3h, 9Bh, 0BDh
.rdata:100627CC              db 3Ah, 6, 8Bh, 0E3h, 77h, 0BCh, 0BDh, 11h, 98h, 0E2h
.rdata:100627CC              db 23h, 0B8h
.rdata:100627EC bignum7      db 48h, 0F5h, 60h, 12h, 13h, 6, 9, 0DBh, 1Ch, 6Eh, 0C6h
.rdata:100627EC              db 38h, 0A5h, 0A7h, 0EFh, 14h, 0E1h, 3Ch, 0ACh, 0C2h, 0C8h
.rdata:100627EC              db 0BEh, 0FCh, 5, 18h, 0F6h, 4Fh, 49h, 7Dh, 74h, 38h, 45h
```

m et tous ces buffers sont passées en paramètre à une nouvelle fonction, qui prend au total 10 arguments. Nous en détaillons le déroulement, qui montre également le calcul de la clé de recouvrement.

Les paramètres `bignum` et m sont convertis en structures typiques de grands nombres :

```c
typedef struct _big
{
  unsigned long allocated;
  unsigned long size;
  unsigned char *data;
} *big;
```

Le champ `data` contient la valeur du nombre écrite de droite à gauche (afin de faciliter les calculs). Après une étude rapide de la fonction, les paramètres se déduisent facilement. La majeure partie du code se déduit par intuition, il n'y a pas besoin d'analyser la totalité des fonctions appelées.

- `bignum1`, `bignum2` et `bignum3` sont les paramètres d'une courbe elliptique dont les points sont définis sur $\mathbb{F}_{p^*}$. La courbe est définie par l'équation de Weierstrass :

  $$\mathcal{C} \ : y^2 = x^3 + 120x + 131 \bmod p$$

  où

  $$p = 0x8d3dfcbe8a2e3e08c7f1bc2b8614139ba1f884fb6b9c76cba80bb3e06bda6007$$

- `bignum4` et `bignum5` sont les coordonnées cartésiennes d'un point de la courbe, noté G ;
- les deux autres buffers `bignum6` et `bignum7` sont les coordonnées d'un autre point de la courbe, noté Q.

Les coordonnées cartésiennes de G et Q sont :

$$G = \begin{pmatrix} x_G \\ y_G \end{pmatrix} = \begin{pmatrix} 0x2455367DA071C81B65CF4C63BA7DB614AA6915C065A0C3BC046F50630B8C1872 \\ 0x42840DC8199D1620CA8000E82D2E04B011AE07095DD2D4F649CDCE1086993B70 \end{pmatrix}$$

et

$$Q = \begin{pmatrix} x_Q \\ y_Q \end{pmatrix} = \begin{pmatrix} 0x00BB5A0C99F6B9B9AC9A4991A2902DA723E39BBD3A068BE377BCBD1198E223B8 \\ 0x48F56012130609DB1C6EC638A5A7EF14E13CACC2C8BEFC0518F64F497D743845 \end{pmatrix}$$

Nous calculons l'ordre de la courbe. Il contient un très grand facteur premier :

$$\begin{aligned} NP &= 0x8D3DFCBE8A2E3E08C7F1BC2B8614139BB5A6AA2106774BCAD07A01B9513FF803 \\ &= 5 * 0x1C3F98F2E86FA601C196BF3BE79D9D858ABAEED367B1758EF67ECD25103FFE67 \end{aligned}$$

Le message m, de 64 octets *a priori*, est tronqué à 32 octets (car la taille de l'ordre de la courbe est 256 bits).

La fonction retourne deux clés :

- k_e, la clé de chiffrement utilisée pour le fichier, protégée ultérieurement par le mot de passe utilisateur, est égale aux 128 bits de poids faible de l'abscisse de mG ;
- b, la représentation compressée (abscisse et bit de poids faible de l'ordonnée) de mQ, stocké en clair dans le fichier chiffré.

mQ se retrouve à partir de b. En conséquence la clé de chiffrement du fichier k_e est calculable instantanément si on connaît la valeur (clé secrète) d telle que $Q = dG$: ce sont les 128 bits de poids faible de $d^{-1}mQ$. Etant donnée la taille du facteur premier présent dans l'ordre de la courbe, il y a peu de chance de calculer d.

Afin de vérifier que nous ne nous sommes pas trompés dans l'analyse, nous modifions la valeur de b avant l'écriture du fichier chiffré sur le disque (en fait, avant qu'un calcul d'intégrité ne soit appliqué sur l'ensemble du fichier). Le déchiffrement se fait toujours sans problème, b n'est donc *a priori* pas utilisé pour déchiffrer le fichier. Nous pouvons raisonnablement penser qu'il s'agit d'une porte dérobée. Peut-être s'agit-il d'un service offert par l'éditeur pour récupérer des fichiers si le mot de passe a été oublié.

Pour information, la version précédente contenait un séquestre des clés et du vecteur d'initialisation. Ces données étaient chiffrées avec un RSA-1024. La clé publique était présente en clair dans le programme. Le séquestre était ajouté vers la fin du fichier, précédé des premiers octets de la clé publique (peut-être variait-elle selon les sous-versions ?). Le remplacement par un système à courbes elliptiques fait gagner de la place.

Les *backdoors* sont en général loin d'être triviales à mettre en place et à détecter : elles doivent rendre simple le cassage d'un système tout en n'étant pas détectable lors d'une analyse un peu poussée. Elles sont donc en général bien cachées dans un binaire : on peut trouver pêle-mêle des code ou clés fixes se décodant en mémoire (pourquoi les protéger ?), des parties de code dans un binaire très différentes du reste du code, comme des routines C en plein milieu de code entièrement en C++ (le code a-t-il été rajouté ?), ou encore des fonctions de hachage écrites plusieurs fois dans le binaire, une d'entre elles présentant un "bug".

Le fait qu'elles soient dissimulées est à la fois un avantage et un inconvénient : si le code analysé est complexe, il sera laissé de côté de prime abord. En revanche, lors d'une analyse plus longue, c'est vers ces parties qu'il faudra se concentrer : un code qui n'a rien à cacher n'a pas de raison d'être complexe.

3 Détecter, identifier et comprendre la cryptographie

Dans cette partie, nous nous intéressons à la détection et l'identification des éléments de cryptographie, c'est-à-dire les primitives utilisées dans les logiciels et les communications. Dans un premier temps, nous nous focalisons sur l'étude des logiciels. Toutefois, ils ne sont pas toujours disponibles (par exemple dans le cas des interceptions) et d'autres moyens doivent être employés sur les communications.

3.1 Identifier les fonctions cryptographiques dans les binaires

Afin de protéger les données ou les communications, de plus en plus de logiciels embarquent nativement des fonctions de cryptographie. Une large part de l'analyse de ces logiciels porte sur l'identification de ces fonctions, qu'il s'agisse de bibliothèques publiques (et donc connues), ou de développements propres. Dans cette rubrique, nous proposons des solutions pour automatiser la détection de ces fonctions lors de la rétro-conception d'un logiciel.

De nombreuses informations sur les méthodes de chiffrement utilisées sont souvent présentes dans la documentation d'un logiciel. De plus, l'emploi de certaines bibliothèques publiques doit légalement s'assortir d'une mention obligatoire dans le fichier de licence. On y retrouve le nom, et parfois même la version des différentes bibliothèques utilisées.

Recherche par signatures La façon la plus simple d'analyser des binaires contenant de la cryptographie est de charger des signatures identifiant une bibliothèque connue. On a repéré précédemment la bibliothèque utilisée à partir des informations présentes dans le binaire (chaînes de caractères, patterns caractéristiques, etc.) ou dans la documentation du programme. Cette méthode est applicable dans les cas suivants :

- nous connaissons la bibliothèque, ses sources sont disponibles et recompilables avec le compilateur utilisé par le programme ;
- le binaire de la bibliothèque est disponible ;
- nous avons déjà analysé un programme utilisant cette bibliothèque précédemment ;
- nous disposons déjà d'un fichier de signatures.

Dans les autres cas, une analyse manuelle s'impose. Nous voyons qu'elle peut être grandement automatisée.

Recherche par patterns Certaines primitives utilisent des constantes bien spécifiques. C'est notamment le cas des fonctions de hachage et des fonctions de chiffrement par blocs.

Les fonctions de hachage dédiées ont toutes plus ou moins le même principe : la valeur du condensat est initialisée à une valeur fixe. Les données d'entrée sont accumulées dans un tampon, puis sont compressées dès que ce dernier est rempli. La valeur du condensat est ensuite mise à jour. Pour obtenir le condensat final, les données restantes sont ajoutées puis compressées.

L'initialisation et la routine de compression d'une fonction de hachage sont facilement détectables dans les binaires, en particulier celles reposant sur MD4 ou SHA-0 : la valeur d'initialisation est une valeur fixe de longueur constante, et les différentes fonctions de compression utilisent de nombreuses valeurs bien particulières.

Par exemple, l'initialisation d'un SHA-1 dans un binaire est de la forme :

```
8B 44 24 04             mov     eax, [esp+arg_0]
33 C9                   xor     ecx, ecx
C7 00 01 23 45 67       mov     dword ptr [eax], 67452301h
C7 40 04 89 AB CD EF    mov     dword ptr [eax+4], 0EFCDAB89h
C7 40 08 FE DC BA 98    mov     dword ptr [eax+8], 98BADCFEh
C7 40 0C 76 54 32 10    mov     dword ptr [eax+0Ch], 10325476h
C7 40 10 F0 E1 D2 C3    mov     dword ptr [eax+10h], 0C3D2E1F0h
89 48 14                mov     [eax+14h], ecx
89 48 18                mov     [eax+18h], ecx
89 48 5C                mov     [eax+5Ch], ecx
```

Il est également aisé de retrouver des fonctions de chiffrement par blocs (établissement de clé, chiffrement ou déchiffrement) dans les binaires, car ces fonctions utilisent, pour la majorité d'entre elles, des S-Boxes ou des tables de permutations de taille conséquente. Les tables utilisées pour l'établissement de clés sont parfois différentes de celles utilisées pour le chiffrement, elles mêmes différentes de celles utilisées pour le déchiffrement. Ces tables sont en général de taille importante (plus de 256 octets) : les faux positifs sont très peu probables.

Par exemple, le début de la table `Te0` utilisée par la fonction de chiffrement d'AES est :

```
Te0          dd 0C66363A5h, 0F87C7C84h, 0EE777799h, 0F67B7B8Dh, 0FFF2F20Dh
             dd 0D66B6BBDh, 0DE6F6FB1h, 91C5C554h, 60303050h, 2010103h
             dd 0CE6767A9h, 562B2B7Dh, 0E7FEFE19h, 0B5D7D762h, 4DABABE6h
             dd 0EC76769Ah, 8FCACA45h, 1F82829Dh, 89C9C940h, 0FA7D7D87h
             dd 0EFFAFA15h, 0B25959EBh, 8E4747C9h, 0FBF0F00Bh, 41ADADECh
             ...
```

La recherche par patterns se fait de deux manières :
- en recherchant des tables de données, comme pour `Te0` (ces tables se situent généralement dans les sections (`.data` ou `.rdata`) ;
- en cherchant des listes de valeurs, pouvant être séparées par des blocs de petite taille (voir l'initialisation du SHA-1).

Certains algorithmes, comme Blowfish, n'utilisent de tables que lors de l'établissement de clé. Les fonctions de chiffrement devront être identifiées d'une autre manière.

Quelques algorithmes, comme TEA, n'utilisent pas de tables. TEA utilise le nombre d'or (`0x9e3779b9`) et sera détecté ainsi. Des faux-positifs ne sont pas à exclure, d'autant plus que cette valeur est utilisée dans d'autres algorithmes, comme RC5 ou RC6.

Les algorithmes de chiffrement par flot sont à traiter de la même façon : nous ne reconnaîtrons que ceux utilisant des tables spécifiques. Il y en a malheureusement, en proportion, beaucoup moins que précédemment. Il faudra laisser de côté des algorithmes courants, comme RC4.

Il est à noter qu'une recherche de détection par patterns échouera dans le cas de la *white-box cryptography* [19], qui vise à rendre complexe la détection des paramètres associés aux fonctions de chiffrement dans le cas où l'attaquant possède un accès complet à un système.

Des outils publics recherchent ces patterns dans les binaires. Nous pouvons citer le *plugin* `PEiD Krypto ANALyzer` [20], `FindCrypt` [21] pour IDA et, dans une moindre mesure, la commande `searchcrypt` d'Immunity Debugger [22].

Graphe d'appels des fonctions Les outils cités précédemment recherchent les patterns, indiquent leur adresse et parfois les références à ces adresses. Ils ne donnent toutefois pas d'informations sur les fonctions associées. La prise en compte des fonctions, en particulier du graphe d'appels autour de ces fonctions, apporte des informations supplémentaires.

Prenons l'exemple d'une fonction de hachage. Calculer le condensat de données quelconques requiert l'appel de quatre fonctions dans la plupart des implémentations :
- une fonction d'initialisation, qui remplit la valeur h_0 du condensat de départ ;
- une fonction de mise à jour, copiant les données à hacher dans le tampon interne de l'instance de la fonction de hachage ;
- une fonction de finalisation, retournant le condensat final ;
- une fonction de compression, appelée lorsque le tampon interne est rempli par la fonction de mise à jour, ou par la fonction de finalisation.

La recherche par signatures retrouve les fonctions d'initialisation et de compression. Les fonctions de mise à jour et de finalisation sont appelées peu après la fonction d'initialisation avec une forte probabilité, et appellent toutes les deux la fonction de compression.
- elles se situent dans le graphe engendré par les fonctions appelantes de la fonction de compression ;
- elles se situent dans un sous-graphe de faible profondeur contenant les 4 fonctions associées au condensat.
- la fonction de finalisation est appelée après la fonction de mise à jour.

Enfin, toutes les fonctions correspondant à une fonction de hachage se situent dans le même module ou la même classe. Surtout, elles se trouvent dans le même fichier source. On les retrouvera dans le fichier compilé dans le même ordre, donc à des offsets très proches. Ce qui donne un marqueur fort pour les repérer.

Toutes les fonctions associées au hachage de données sont retrouvées avec cette méthode. Le graphe d'appels des fonctions est également utile pour détecter les fonctions associées au chiffrement asymétrique.

Cas des fonctions de chiffrement asymétriques La détection des fonctions de chiffrement asymétrique est beaucoup plus complexe que les opérations précédentes. Dans cette partie seront détaillées des heuristiques pour les repérer avec une probabilité la plus forte possible, sans pour autant arriver à un résultat parfait.

Fonctions à identifier : Les principaux systèmes de chiffrement à clé publique rencontrés dans les logiciels de chiffrement reposent sur :
- le problème de factorisation : RSA ;
- le problème du logarithme discret dans $\mathbb{Z}_p^*$: ElGamal, DSS, Diffie-Hellman ;
- le problème du logarithme discret sur une courbe elliptique définie sur les corps finis $\mathbb{F}_q$ ou $\mathbb{F}_{2^m}$: ECDSA, ECDH.

Les opérations typiques utilisées pour ces types de chiffrement sont l'exponentiation modulaire pour les opérations dans $\mathbb{Z}_p^*$, et la multiplication par un scalaire d'un point d'une courbe elliptique. Ce sont ces fonctions que nous rechercherons en priorité.

Graphe d'appels des fonctions Les fonctions que nous cherchons à détecter sont de niveau assez haut, comparées à l'ensemble des fonctions présentes dans une bibliothèque de manipulation de grands nombres. Ces fonctions appellent des opérations arithmétiques plus basiques : une exponentiation modulaire devrait appeler les fonctions de multiplication et de réduction.

Cela est vrai pour des bibliothèques simples, mais ne sera pas le cas pour des bibliothèques plus évoluées (représentation de Montgomery [12], etc.). Les dépendances sont difficiles à interpréter autrement que manuellement.

Paramètres standards Les cryptosystèmes reposant sur le logarithme discret utilisent parfois des éléments de clés publiques génériques : modulus et générateur pour ElGamal et DSS, courbe et générateur (point de la courbe) pour ECDSA.

---

[12] Voir `http://en.wikipedia.org/wiki/Montgomery_reduction`

Ces paramètres sont codés dans les binaires. Les retrouver donne une bonne information sur le chiffrement qui sera utilisé. On note en particulier les courbes recommandées NIST pour les cryptosystèmes à courbes elliptiques, le format des clés publiques exportées par certaines bibliothèques, ou encore la valeur de certains paramètres générés par une bibliothèque donnée.

On déduit le rôle des fonctions faisant référence à ces informations :
`ascii_to_bignum`, `text_to_bignum`, `load_public_key`, etc.

Messages d'erreur Les messages d'erreur apportent également de l'information quant au rôle d'une fonction. Ceci n'est pas propre aux fonctions associées à la cryptographie. Si les messages sont faciles à interpréter pour un humain, faire interpréter ces messages pour identifier le rôle d'une fonction par un programme externe est beaucoup plus complexe.

3.2 Automatisation de l'analyse dans les binaires

Quel est l'apport de la reconnaissance des fonctions de chiffrement lorsqu'on étudie un programme ? Les parties d'un programme contenant des algorithmes de chiffrement sont, dans leur immense majorité, des empilements de primitives cryptographiques, avec très peu de code utile autour.

Prenons l'exemple du chiffrement des fichiers de Word 2003. Word utilise la CryptoAPI Microsoft pour ces opérations. Étudions le mécanisme de déchiffrement des fichiers : nous hookons toutes les fonctions de la CryptoAPI après avoir entré le mot de passe du fichier (ici `loremipsum`). Certaines parties ont été supprimées pour raccourcir le listing :

```
Address   Message
314A55E1  CryptCreateHash
             IN:
              hProv: hCrypt0
              AlgId: CALG_SHA1
              hKey: 0x0
              dwFlags: 0
             OUT:
              *phHash: hHash0
314A5601  CryptHashData
             IN:
              hCryptHash: hHash0
              pbData:
          0000   67 BE 94 49 F7 AD 88 0A A9 CE 49 CF A4 4D AB 8B    g..I......I..M..
              dwDataLen: 0x10
              dwFlags: 0
314A5620  CryptHashData
             IN:
              hCryptHash: hHash0
              pbData:
          0000   6C 00 6F 00 72 00 65 00 6D 00 69 00 70 00 73 00    l.o.r.e.m.i.p.s.
          0010   75 00 6D 00                                        u.m.
              dwDataLen: 0x14
              dwFlags: 0
314A5664  CryptGetHashParam
             IN:
              hHash: hHash0
              dwParam: HP_HASHVAL
```

```
              *pdwDataLen: 0x14
              dwFlags: 0x0
             OUT:
              pbData:
          0000   5C 78 88 FA 2F C3 24 00 62 55 07 26 DC 8D 1C 69    .x../.$.bU.&...i
          0010   32 EA 4A EF                                        2.J.
              *pdwDataLen: 0x14
```

Le SHA-1 du mot de passe converti en Unicode précédé d'un sel `salt` de 16 caractères est calculé. Nous l'appelons `h1`.

```
77DE8675   CryptAcquireContextA
             IN:
              pszContainer: NULL
              pszProvider: Microsoft Enhanced Cryptographic Provider v1.0
              dwProvType: PROV_RSA_FULL
              dwFlags: 0
             OUT:
              *phProv: hCrypt0
314A5477   CryptCreateHash
             IN:
              hProv: hCrypt0
              AlgId: CALG_SHA1
              hKey: 0x0
              dwFlags: 0
             OUT:
              *phHash: hHash0
314A549B   CryptHashData
             IN:
              hCryptHash: hHash0
              pbData:
          0000   5C 78 88 FA 2F C3 24 00 62 55 07 26 DC 8D 1C 69    .x../.$.bU.&...i
          0010   32 EA 4A EF                                        2.J.
              dwDataLen: 0x14
              dwFlags: 0
314A54AB   CryptHashData
             IN:
              hCryptHash: hHash0
              pbData:
          0000   00 00 00 00                                        ....
              dwDataLen: 0x4
              dwFlags: 0
314A54E1   CryptDeriveKey
             IN:
              hProv: hCrypt0
              AlgId: CALG_RC4
              hBaseData: hHash0
              dwFlags: 0
              *phKey: hKey0
```

`h1` est haché, ainsi qu'un buffer de 4 octets nuls `salt2`. Le condensat résultant est dérivé pour obtenir une clé de chiffrement `k`.

```
314A5580    CryptDecrypt
               IN:
                hKey: hKey0
                hHash: hHash0
                Final: FALSE
                dwFlags: 0
                pbData:
            0000   4D 9E 68 11 06 7E DD B4 51 2F FC 05 07 AF 9C 59    M.h..~..Q/.....Y
                *pdwDataLen: 0x10
               OUT:
                pbData:
            0000   33 43 54 55 65 66 66 D6 2D CE 66 CC EE 9E A9 AA    3CTUeff.-.f.....
                *pdwDataLen: 0x10
```

Des données inconnues `data1` sont déchiffrées (RC4) avec la clé obtenue précédemment. Appelons-le résultat `plain1`.

```
314A79B1    CryptCreateHash
               IN:
                hProv: hCrypt0
                AlgId: CALG_SHA1
                hKey: 0x0
                dwFlags: 0
               OUT:
                *phHash: hHash0
314A79C9    CryptHashData
               IN:
                hCryptHash: hHash0
                pbData:
            0000   33 43 54 55 65 66 66 D6 2D CE 66 CC EE 9E A9 AA    3CTUeff.-.f.....
                dwDataLen: 0x10
                dwFlags: 0
314A7A10    CryptGetHashParam
               IN:
                hHash: hHash0
                dwParam: HP_HASHVAL
                *pdwDataLen: 0x14
                dwFlags: 0x0
               OUT:
                pbData:
            0000   28 22 E3 F7 88 93 20 62 93 68 DA 85 B0 DB C3 A9    (".... b.h......
            0010   F4 BC 8D C4                                        ....
                *pdwDataLen: 0x14
```

`plain1` est haché. Le résultat est noté `h2`.

```
314A5580    CryptDecrypt
               IN:
                hKey: hKey0
                hHash: hHash0
                Final: FALSE
```

```
        dwFlags: 0
        pbData:
0000   4F 48 49 AA 72 A6 E3 91 FA C6 9D 33 14 CB F2 C0    OHI.r......3....
0010   0D 1D 38 BF                                        ..8.
        *pdwDataLen: 0x14
       OUT:
        pbData:
0000   28 22 E3 F7 88 93 20 62 93 68 DA 85 B0 DB C3 A9    ("....  b.h......
0010   F4 BC 8D C4                                        ....
        *pdwDataLen: 0x14
```

D'autres données inconnues `data2` sont déchiffrées. Nous remarquons que le buffer obtenu est identique à `h2`. En faisant le test avec plusieurs fichiers, nous nous apercevons que :
- les buffers sont identiques si nous entrons le vrai mot de passe ;
- ils sont différents dans le cas contraire.

Nous obtenons donc l'algorithme complet de déchiffrement sans avoir à analyser le code. Reste à déterminer d'où proviennent les valeurs `data1`, `data2`, `salt` et `salt2`. La recherche des paramètres dans le fichier chiffré répond à la question : ils sont tous contenus dans le fichier chiffré, ainsi que le nom du fournisseur cryptographique et les identifiants des algorithmes cryptographiques utilisés (cf. Fig. 3).

On en déduit l'algorithme de vérification utilisé :
ENTRÉE : le mot de passe *pass*, deux valeurs $data_1$ et $data_2$ fixées, deux valeurs $salt_1$ et $salt_2$ fixées.
SORTIE : *VRAI* si le mot de passe entré est correct, *FAUX* sinon.

1. Calculer $c = h(salt_1 + pass + salt_2)$, h étant la fonction de hachage SHA-1 ;
2. Calculer $k = KDF(c)$, en notant KDF la fonction de dérivation de clé ;
3. Calculer $m_1 = e_k(data_1)$ le chiffrement RC4, de clé k, de $data_1$;
4. Calculer $m_2 = h(m_1)$;
5. Calculer $m_3 = e_k(data_2)$;
6. Retourner *VRAI* si $m_2 = m_3$ et *FAUX* sinon.

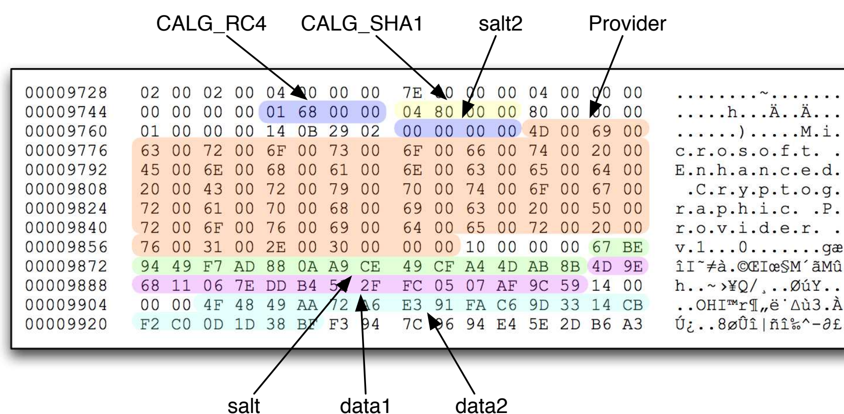


Fig. 3: Informations contenues dans un fichier Word chiffré

Plusieurs points restent à éclaircir : la provenance des valeurs `data1` et `data2` dans notre exemple, générées lors du chiffement d'un fichier, ainsi que la méthode de dérivation de clé utilisée. Néanmoins, le déroulement de l'algorithme apparaît directement sans étude intensive.

Cette méthodologie, associée à une étape de reconnaissance des algorithmes largement automatisable comme nous l'avons vu précédemment, permet de se concentrer sur les points critiques d'un logiciel (génération d'aléa et dérivation de clé) en faisant gagner du temps sur l'étude des mécanismes de chiffrement globaux utilisés.

3.3 Détection de flux chiffrés

La problématique est maintenant, étant donné un volume non négligeable de trafic, de déterminer les communications qui sont réellement chiffrées.

Le principe de base est dans un premier temps de disposer d'une ou plusieurs mesures permettant de discriminer les différents types de flux : chiffré, compressé, texte, image, son, texte, couple codage/langage ... L'approche consiste à définir des estimateurs caractéristiques pour chaque flux ou format possibles et ensuite à les utiliser pour l'identification de la nature du flux, une technique classique de tests statistiques [2].

Cependant, cette approche est longue – il faut appliquer autant de tests que de formats possibles – et les erreurs inévitables – défauts inhérents à l'approche statistique. Il est plus utile de faire un premier tri entre deux grandes classes de données : celles possédant peu ou pas de redondance (données chiffrées, données compressées) et celles possédant de la redondance (les autres données). Pour chaque classe ensuite isolée, un second traitement avec des outils statistiques plus fins est effectué. Dans ce qui suit, nous allons donc considérer comment isoler le premier groupe des données non redondantes, qui pour cet article sera restreint aux données chiffrées[13].

La technique de tri de flux se fonde sur la notion d'entropie définie par C. E. Shannon [23]. Considérons une source d'information (typiquement un flux de données) S constituée d'une succession de réalisation d'une variable aléatoire X, laquelle peut prendre un ensemble fini de valeurs $x_1, x_2, \ldots, x_n$ avec une probabilité respectivement de $p_1, p_2, \ldots, p_n$ (autrement dit $P[X = x_i] = p_i$). L'entropie de la source est définie par :

$$H(X) = -\sum_{i=1}^{n} p_i . \log_2(p_i).$$

Cette valeur est maximale lorsque toutes les valeurs de X sont équiprobables (et donc de probabilité égale à $\frac{1}{n}$), ce qui est le cas lorsque les données sont chiffrées (idéalement). Une manière équivalente consiste à considérer la notion de redondance, donnée par la formule :

$$R(X) = 1 - \frac{H(X)}{\log_2(|\Sigma|)},$$

où Σ est l'alphabet réalisé par la source X. L'intérêt de la redondance est, comme nous le verrons plus loin, de pouvoir en calculer un taux et donc permet de comparer différentes mesures d'entropie.

En pratique, l'entropie d'un flux est calculée à partir de la fréquence observée pour chaque caractère puis en appliquant la formule précédente.

Exemple 1 *Soient les deux chaînes de caractères suivantes :*[14]

$$C_1 = S\ S\ T\ I\ C\ 2\ 0\ 0\ 8 \quad \text{et}\ C_2 = S\ X\ T\ B\ C\ 2\ 1\ A\ 8.$$

[13] Le cas des données compressées ne sera pas abordé ici car il est assez complexe à traiter. Distinguer des données chiffrées de données compressées peut se révéler très difficile, pour ne pas dire impossible, selon les algorithmes de compression utilisés, du moins par une analyse brute des données. Le plus souvent, l'étude se limite, avec succès, à l'étude de l'entête du fichier.

[14] Tous les exemples donnés dans la suite sont volontairement courts par manque de place. Cependant, sans restriction conceptuelle, et même s'ils n'épuisent pas tout l'alphabet, ils illustrent suffisamment la

Le calcul de leur entropie donne :

$$H(C_1) = 2.72548 \text{ et } H(C_2) = 3.16992.$$

alors qu'en terme de redondance nous avons :

$$R(C_1) = 0.029164 \text{ et } H(C_2) = 0.$$

On voit que la chaîne C_1 est redondante (2.91% de redondance) alors que la chaîne C_2 ne l'est pas.

En pratique, pour détecter un flux chiffré, il suffit de rechercher ceux qui exhibent une entropie maximale (ou de manière équivalente, une redondance minimale).

L'intérêt d'utiliser l'entropie ou la redondance, tient au fait qu'elle résume utilement, en première approche, beaucoup de caractéristiques considérées généralement par d'autres tests (comme ceux recommandés par le NIST [4,5] : tests de fréquences, test d'autocorrélation, ...). Cependant, le défaut important de l'entropie est qu'elle est très délicate à interpréter en pratique et ne peut être considérée seule et seulement sous cette forme. Pour s'en convaincre, considérons ces quelques exemples :

– Considérons les chaînes

$$\text{S S T I C 2 0 0 8} \quad \text{et } C_2 = \text{8 T 0 S I 0 S 2 C}$$

Elles ont même entropie et même redondance. La chaîne C_2 ne sera pas détectée comme étant un flux chiffré alors qu'il s'agit de la version chiffrée par transposition de la chaîne C_1.

– L'entropie dépend de l'alphabet considéré. Regardons maintenant la chaîne suivante :

$$C = \text{S S T I C 2 0 0 8 A R E N N E S.}$$

Nous avons sur un alphabet à 256 caractères $H(C) = 3.327819$ et $R(C) = 0.168045117$. Considérons maintenant l'alphabet de 65 536 caractères formés de couples d'octets :

$$\Sigma = AA, AB, \ldots, ZZ.$$

L'entropie, sur ce nouvel alphabet, devient alors $H(C) = 8$ et $R(C) = 0$. La chaîne C est détectée, à tort, comme étant chiffrée.

– L'entropie dépend également fortement du codage. Si le codage ASCII à 8 bits est encore le plus utilisé, ce n'est pas le seul : de l'Unicode aux codes télex sur 5 bits (comme le CCITT-2) il en existe une grande variété selon le type de communication. Prenons la chaîne de caractères C = E V M K C R G T, chaîne apparemment aléatoire. Considérons l'alphabet quaternaire (ou alphabet de *nibbles* sur 4 bits) et deux codages différents de C.

  – La représentation binaire de C codée en ASCII est donnée par (représentée ici en notation hexadécimale par souci de compacité) :

$$C_{\text{ASCII}} = 0x45564D4B43524754.$$

  Son entropie et sa redondance, relativement à un alphabet quaternaire, sont données par :

$$H(C_{\text{ASCII}}) = 2.5306390 \quad \text{et } R(C_{\text{ASCII}}) = 0.367340.$$

  Selon ce test, cette chaîne possède une forte redondance et sera rejetée comme non chiffrée.

problématique de la détection de flux chiffrés et en particulier les techniques de simulabilité qui peuvent être appliquées pour contourner cette détection, même sur de très longues chaînes (voir à ce propos les deux fichiers de 64 Mo proposés en challenge dans [2].

– La représentation binaire de C codée en CCITT-2 est donnée par (représentée ici en notation hexadécimale par souci de compacité) :

$$C_{\text{CCITT-2}} = 0x83CFE72961.$$

Son entropie et sa redondance, relativement à un alphabet quaternaire, sont donnés par :

$$H(C_{\text{CCITT-2}}) = 3.3219281 \quad \text{et } R(C_{\text{CCITT-2}}) = 0$$

Selon ce test, cette chaîne ne possède aucune redondance et sera considérée comme chiffrée.

Tout ceci montre que l'utilisation de l'entropie doit être multiple et considérer plusieurs codages et alphabets possibles, ce qui calculatoirement devient vite complexe. Nous nous limitons donc à des profils entropiques plutôt qu'à une mesure unique. L'expérience montre que sous l'hypothèse où aucune technique de simulabilité n'a été mise en œuvre, ces profils sont très efficaces.

3.4 Identification du chiffrement

Une fois un flux supposé chiffré repéré, avant de pouvoir l'attaquer, il peut s'avérer utile de déterminer le chiffrement employé. On suppose donc ici que l'attaquant n'a pas la connaissance du protocole sous-jacent. Dans le cas général, le problème est particulièrement ardu. Trois cas de figures se présentent en pratique :

- Nous connaissons par avance le type de chiffrement utilisé. Du renseignement humain, de l'analyse de trafic et/ou de l'exploitation des données techniques ouvertes (un trafic IP chiffré, de nos jours à de fortes chances de relever de l'IPSec, une liaison Wi-Fi de RC4 ou de l'AES, une liaison Bluetooth de l'algorithme E0 . . .). En pratique, cela concerne près de 90% des cas. Mais, il est nécessaire que dans un domaine comme la sécurité, la norme puisse varier. Ainsi, le chiffrement TrueCrypt autorise un large choix d'algorithmes, tout comme le logiciel GPG, ce qui amène à considérer les deux autres possibilités.
- Nous disposons de ce que nous dénommons un «*distingueur*». Il s'agit d'un procédé plus ou moins complexe permettant d'identifier et de caractériser de manière univoque un système de chiffrement donné. En théorie, il existe toujours au moins un tel distingueur, déterministe, sauf dans le cas des systèmes à secret parfait (type Vernam). Le problème est que ce distingueur possède en général une taille exponentielle et est donc inutilisable en pratique. Il est alors possible de considérer des distingueurs probabilistes, mais les quelques proposés dans la littérature ouverte sont également d'une complexité trop importante.

  Des résultats récemment obtenus grâce à des techniques d'apprentissage couplées à des considérations combinatoires [24] semblant mener à de nouveaux distingueurs efficaces. Certes, la taille de ces outils est encore énorme mais des techniques combinatoires devraient permettre d'optimiser les formes algébriques obtenues. D'autre part, les puissances de calcul actuelles permettent de traiter – même *inline* – des distingueurs de plusieurs méga-octets. À titre d'exemple, sur le chiffrement E0 de la norme Bluetooth, l'analyse combinatoire des formes normales algébriques caractérisant chaque bit de sortie [25] couplée à des techniques d'apprentissage de formules booléennes, il a été possible d'extraire une famille de distingueurs impliquant les bits de sortie de la suite chiffrante dont les indices sont[15] :
  - bits chiffrants $b_0, b_1, \ldots, b_{127}, b_{128}$,
  - bits chiffrants $b_{129}, \ldots, b_{141}, b_{145}, \ldots, b_{169}, b_{171}, b_{184}, \ldots$, au total environ 130 bits compris entre b_{129} et b_{300}.

  La taille de chaque distingueur – une fonction booléenne sous forme conjonctive normale – se situe autour de 10 Mo et 100 Mo. Le taux de succès est encore réduit (0.55) mais les recherches en cours

[15] Sans perte de portée opérationnelle, nous supposons connus les quatre bits de mémoire. Cependant, il est possible d'établir et de stocker les familles de distingueurs probabilistes pour les 16 valeurs possibles de ces 4 bits.

permettent d'espérer de faire mieux notamment grâce à des optimisations combinatoires. Les techniques de cryptanalyse à chiffré seul peuvent être utilisées (théorème des probabilités totales) pour utiliser ces distingueurs sur les flux chiffrés réels. Le taux de détection est bien sûr moindre (effet du clair agissant comme bruit).

Il est intéressant de préciser que les aspects combinatoires utilisés pour établir cette famille de distingueurs sont essentiellement les mêmes que ceux utilisés dans la cryptanalyse de type zéro-knowledge [25,26]. Rappelons que ce type de cryptanalyse permet de prouver que nous disposons d'une attaque meilleure que celles connues mais sans en révéler la teneur. Il suffit d'exhiber une suite de sorties particulières ou présentant des propriétés particulières telles qu'elles ne puissent être obtenues par hasard, ni même par une approche exhaustive. La preuve consiste alors à produire la clé permettant de produire cette suite. À titre d'exemple, la clé de 128 bits `K[0] = 0x104766230DF89169 K[1] = 0xC95B9D50C7DF0C57` (voir [25] pour les notations) : produit une suite chiffrante de poids 29 et commençant par 69 zéros. La meilleure attaque connue (sur une suite de sortie aussi courte, à savoir 128 bits, il s'agit de la recherche exhaustive) a une complexité en $\mathcal{O}(2^{72.28})$.

– La troisième solution consiste à faire un nombre minimal de suppositions (par exemple, le système est un système par flot) et à reconstruire le système par des méthodes mathématiques, à partir des seuls flux chiffrés [27]. Cette approche est complexe mais à l'avantage de fonctionner sur des flux chiffrés avec des clés différentes. En outre, le travail n'est à faire qu'une seule fois. Il suffit que le temps de reconstruction n'excède pas la durée de vie du système, ce qui est souvent le cas.

4 Attaquer le canal de communication

Dans cette partie, nous nous mettons dans la position d'un attaquant qui n'a pas accès au contenu des échanges, comme dans le cas où la cryptographie employée est trop forte pour être cassée. Nous verrons que la simple existence de communications, même inintelligibles, constitue déjà une fuite d'information qui peut s'avérer intéressante pour l'attaquant. L'objectif n'est donc pas d'accéder au contenu de l'échange, mais de montrer comment la mise en relation suffit parfois.

Nous supposons que nous disposons de traces de communications. Par exemple, il pourrait s'agir des logs d'un fournisseur d'accès ou d'un opérateur téléphonique, d'interceptions, ou de n'importe quelle entité disposant d'un accès à une infrastructure de communication. Nous considérons que nous disposons de peu d'informations :

– qui discute avec qui ;
– la durée ou le volume.

Ces éléments s'avèrent suffisants, sous réserve qu'ils soient en volume raisonnable, pour en tirer de l'information. Nous nous retrouvons confrontés à deux cas de figure :

1. l'opération s'apparente plus à de la supervision ou à de l'analyse post-mortem afin de déterminer les anomalies. Nous retrouvons là les outils habituels en sécurité informatique, comme l'analyse de flux réseau et le problème de leur représentation. De plus, le data mining peut se révéler également intéressant afin de faire apparaître des structures ou des comportements donnés. Cette méthode est utilisée par exemple par les opérateurs téléphoniques pour détecter des fraudes, ou par des services de police pour reconstruire des cellules terroristes ou mafieuses à partir de leurs communications.
2. l'opération vise un individu (ou un groupe) précis, auquel cas nous favoriserons les algorithmes spécifiques afin de déterminer les groupes et relations liés à la cible.

Avant de réaliser ces analyses se pose la question de l'accès aux éléments à analyser. La première question à se poser est « qui peut mettre en place une telle analyse ? » Réponse évidente : toute personne ayant accès au support de communication ! Pour simplifier, nous nous limiterons au cas du réseau classique[16]. Dans cet environnement, n'importe quel administrateur est à même de récupérer ces traces et de les analyser. Pour un attaquant, cela nécessite une exposition supérieure :

– parce qu'il bénéficie d'une complicité interne lui donnant accès à ces informations ;
– parce qu'il a compromis une (ou plusieurs) machine centrale (routeurs, serveurs, . . .) ;
– parce qu'il dispose d'un moyen de re-router le trafic pour y accéder.

Nous ne spéculerons pas sur les multiples possibilités offertes à un attaquant décidé, hors de propos, et nous concentrons dans la suite sur l'analyse elle-même, supposant ainsi que l'analyste, qu'il soit mal intentionné ou non, a accès aux informations.

Petite mise en garde avant de commencer : dans cette section, nous ne parlons pas de cryptographie, alors que c'est le thème de l'article. L'objectif est de montrer comment l'étude du canal de communication apporte aussi des éléments d'informations, ceux-ci pouvant être suffisants en eux-mêmes, ou servir à casser/affaiblir la cryptographie mise en place. Pourquoi attaquer une porte blindée lorsque nous pouvons passer par la fenêtre restée ouverte . . .

4.1 Supervision et forensics : faire parler les traces

Dans ce cas, nous cherchons à regarder comment les communications évoluent, et non les anomalies comme en détection d'intrusion (bien qu'elles puissent également apporter des éléments). À titre d'exemple, nous prendrons le cas d'une analyse post-mortem réalisée à partir de traces réseau.

Il est possible de faire exactement la même chose en analysant un système, par exemple un ordinateur portable volé [28,29] mais nous ne détaillerons pas cela dans ces pages.

[16] Nous considérons indistinctement la couche IP ou les couches supérieures, applicatives en particulier.

Par définition, l'analyse post-mortem cherche à récupérer des informations sur un système compromis. Il n'est toutefois pas nécessaire que le système soit compromis pour appliquer ces méthodes à des fins de collecte d'information.

Dans le cadre du réseau, ces analyses ne permettent que de trouver les liens entre des machines. Cette information est utile en soi par exemple dans le cas où un attaquant souhaite mener une attaque ciblée : il a besoin de cartographier au mieux sa cible.

Un ouvrage de référence en la matière est [30]. Un bon aperçu est disponible dans [31]. En paraphrasant les références précédentes, la méthodologie pour analyser des traces réseau se décompose en 4 étapes :

1. analyse statistique : il s'agit de regarder la capture à partir de ses statistiques (nombre de paquets, durée des sessions, volume, etc.) ;
2. analyse des sessions : il s'agit de retracer les liens entre les machines, ce qui permet entre autres de déterminer leur rôle respectifs (serveurs de fichiers, d'impression ou d'authentification, poste de travail, et ainsi de suite) ;
3. analyse des anomalies : à partir de signatures, de règles, nous déterminons les événements étranges ;
4. analyse du contenu : nous descendons au niveau du paquet et nous examinons les données.

Dans notre contexte, les deux derniers points ne sont pas nécessaires. Nous supposons que nous sommes dans une situation normale, et nous ne cherchons qu'à comprendre comment le réseau vit.

En guise d'illustration, nous nous appuyons sur la compromission d'une machine[17] [32,33]. À la base de l'analyse, 19Mo de traces au format pcap. En examinant les caractéristiques du trafic, nous ne relevons pas de protocoles inhabituels : 143 paquets ICMP, 41 UDP et 192700 TCP.

C'est connu, Internet est une jungle sauvage où de méchants pirates et des bots passent leur temps à faire des scans de ports : il nous faut donc faire du ménage dans les paquets TCP. Si nous regardons les tentatives de connexion (paquets `SYN`), nous récupérons trop de faux positifs liés aux scans. Nous nous concentrons alors sur les paquets `SYN|ACK` pour déterminer les flux. La figure 4 page suivante montre le graphe complet des connexions TCP et des ports concernés. Si cet exemple est assez simple, les choses sont souvent plus touffues. Il convient dès lors mieux de séparer les connexions entrantes et sortantes.

La figure 5(a) montre les connexions entrantes. Un rapide examen révèle :
- que le port 139 a intéressé beaucoup de monde ;
- des connexions vers les ports 31337, 50 et 45295 alors que ceux-ci n'étaient pas ouverts à l'installation de la machine ;
- que seules 2 machines, `200.30.109.90` et `61.62.84.30`, se sont connectées à la fois aux ports 139 et 45295.

La figure 5(b) montre les connexions sortantes. Un rapide examen révèle :
- qu'il y a eu de nombreuses connexions vers `81.196.20.133` et `217.215.111.13`, mais à chaque fois le port 21 est présent : il s'agit de transferts FTP ;
- que 2 serveurs de messagerie (port 25), 4 serveurs IRC (port 6667) et un serveur web (port 80) ont été contactés ;
- que des connexions sont initialisées vers le port 139 d'autres machines.

Ces hypothèses permettent déjà d'avoir une idée de ce qui se passe. Bien sûr, en situation d'analyse post-mortem, il faudra analyser, confirmer ces hypothèses en regardant le contenu des paquets, les fichiers corrompus, etc. Donc, il s'agit vraisemblablement d'une attaque sur le port 139 qui ouvre un shell sur le port 45295. Étant donné que rien n'a été installé par l'administrateur sur le port 50, et comme il faut des privilèges root pour cela, nous supposons que le pirate a installé une backdoor sur ce port. Pourquoi cette attribution des rôles des ports 50 et 45295, alors que nous pourrions *a priori* intervertir le rôle de ces ports (*i.e.* le port 50 pour le shell et le 45295 pour la backdoor) ? Parce qu'un shellcode est rarement mis en écoute sur un port inférieur à 1024, ce qui demanderait des privilèges root . . .et il n'est pas du tout certain que le démon exploité donne ces droits, donc les shells sont installés sur des ports hauts.

[17] Il s'agissait en réalité d'un pot à miel, mais cela n'a pas d'impact sur la suite.

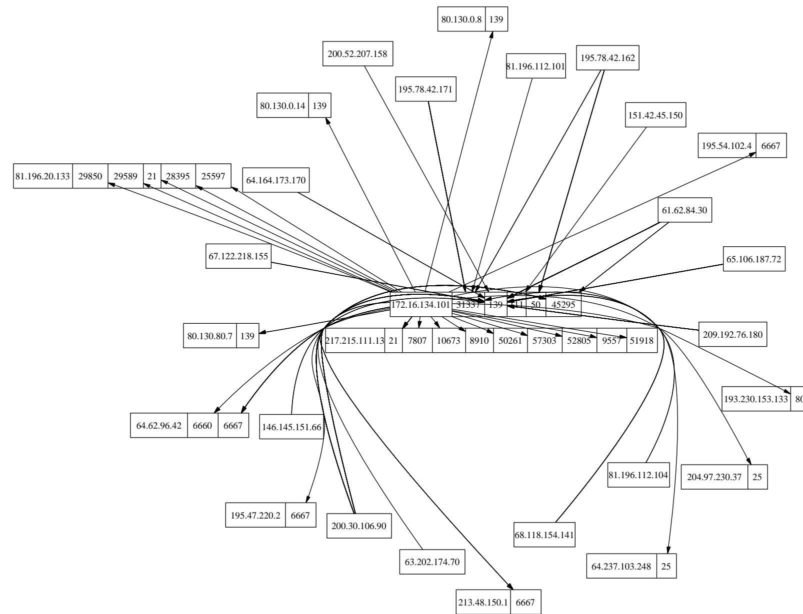


Fig. 4: Graphe des connexions TCP

Les machines avec le port 21 ouvert sont sans doute des bases du pirate sur lesquelles il récupère des outils. Habituellement, ils font aussi cela sur des serveurs web. Encore une question d'habitude, mais bien souvent, les scripts de compromission automatique font une photo du système (mémoire, règles de filtrage, utilisateurs, uptime, . . .) qu'ils envoient à leur maître : les connexions vers les ports 25 y sont probablement liées.

Enfin, le port 31337 sert traditionnellement à un relais IRC (psyBNC), ce qui semble correct étant données les connexions sortantes vers de tels serveurs.

Cet exemple (certes simple) illustre comment une bonne connaissance de certaines pratiques et du fonctionnement d'un réseau permet rapidement de tirer de l'information sur les communications entre plusieurs machines, tout cela sans même tenir compte du contenu des échanges, comme si ces échanges étaient inaccessibles, correctement protégés par de la cryptographie.

4.2 Réseaux sociaux : faire parler les relations

Dans la section précédente, nous traitions du réseau. Dans celle-ci, nous abordons ses utilisateurs. Bien souvent les communications chiffrées ne sont pas anonymisées (contrairement à ce qui se fait avec Tor par exemple). Il est dès lors très facile pour quelqu'un ayant un accès au support de communication de déterminer les relations d'une personne (ou un groupe) avec d'autres. On parle alors de *réseau social* en tant que mise en relation d'entités sociales reliées par des liens crées lors d'interactions sociales (une communication chiffrée ici).

Cette notion de réseau social est introduite dans [34], puis développée par d'autres sociologues. Elle s'appuie essentiellement sur les graphes, objets bien connus en informatique. Entre autres, les réseaux sociaux permettent de déterminer le capital social d'un individu, en particulier grâce à la théorie des liens

faibles [35] et des trous structuraux [36]. Les notions importantes sont la diversité des liens, le temps, la complexité du graphe,

Les auteurs de cet article n'étant pas sociologues, nous nous intéresserons pour le moment à la construction d'un tel réseau. À titre illustratif, nous nous appuyons sur la messagerie (le mail). Il serait cependant intéressant de renforcer l'analyse en ajoutant d'autres sources :

- la messagerie instantanée, en examinant les contacts d'un individu et le volume, la fréquence des échanges ;
- ses communications téléphoniques ;
- sa participation à des sites comme facebook ou linkedin ;
- et ainsi de suite.

Nous avons développé un outil qui permet, à partir de l'analyse de boîtes de messagerie de construire des graphes de relations entre les personnes. Le réseau social est construit à partir des éléments suivants :

- une arête signifie que 2 personnes ont été parties prenantes dans un mail ;
- un nœud est une personne mentionnée dans un des champs `From`, `To` ou `Cc` d'un mail.

Les arêtes sont pondérées par le nombre de relations entre ses extrémités, soit le nombre de fois où ces entités sont apparues dans un échange. La figure 6 page 37 présente le graphe d'une boîte mail sur une durée de presque deux ans.

Il est alors possible d'extraire le sous-graphe composé des nœuds ayant un degré élevé (10 ou plus dans la figure 7 page 38)

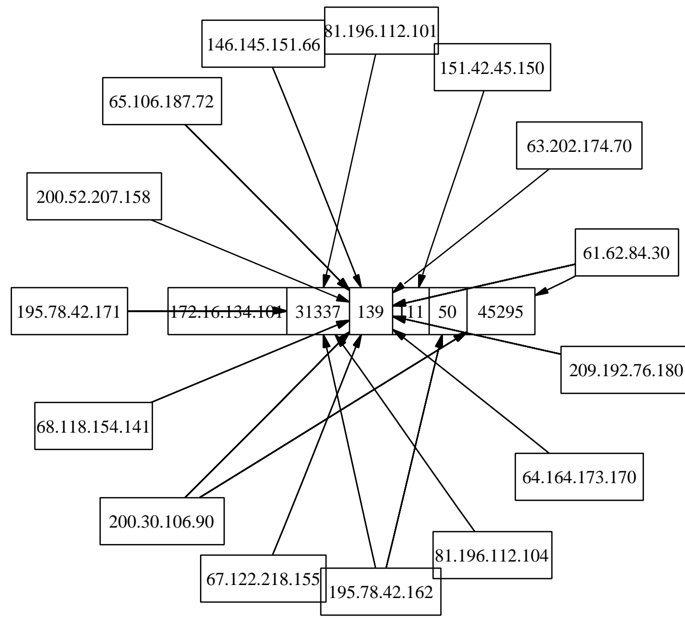
Il est aussi possible de calculer l'intersection entre des graphes (cf.fig 8(a)), et la mesure de centralité[18] (cf.fig 8(b)).

La génération de tels graphes n'est pas une difficulté majeure. Alors, pourquoi se préoccuper de ce type d'analyse ? Reprenons le célèbre exemple du BlackBerry de la société RIM en passant outre sur toutes les spéculations possibles, par exemple quant au fait que les messages soient obligés de transiter par un serveur aux États-Unis ou en Angleterre. Faisons l'hypothèse que la cryptographie est parfaite, que RIM n'a introduit aucune porte dérobée, aucune trappe à la demande d'on ne sait quel gouvernement. Que reste-t-il ? Un outil sûr, certes.

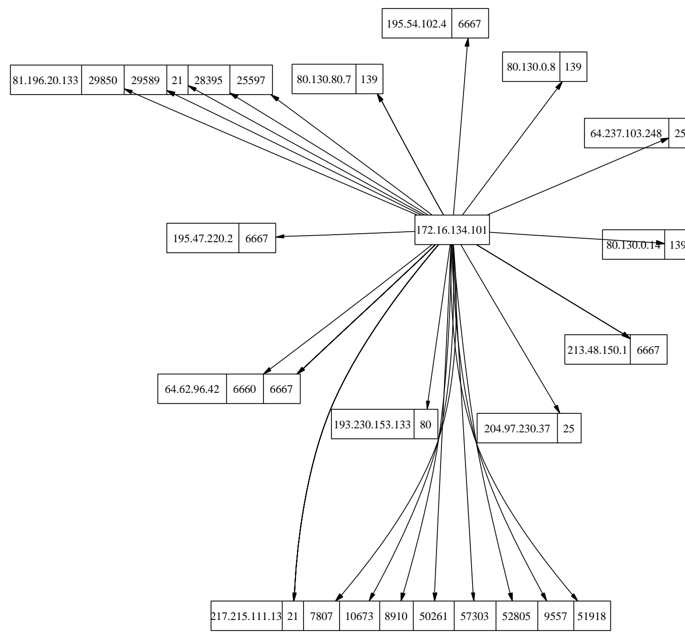
En se plaçant dans la position de RIM, que peuvent-ils faire ? Juste surveiller qui communique avec qui. Et à quelle fréquence. Et à quel volume. Etc. Bref, tous les éléments nécessaires à une très fine analyse de réseau social telle que nous l'avons rapidement présentée. Alors que les autres risques sont spéculatifs, celui-ci ne fait aucun doute et ne demande aucune intervention pour exister (à la différence d'une trappe qui demande une modification logicielle) puisque la simple observation du système donne ces informations.

Deux questions viennent alors immédiatement à l'esprit. Premièrement, quel avantage cela apporterait à RIM, surtout en regard du risque encouru si cela venait à être découvert ? Deuxièmement, pourquoi focaliser ce type de risque sur RIM, alors que tous les opérateurs téléphoniques et autres fournisseurs d'accès Internet peuvent faire exactement la même chose ? D'ailleurs, n'analysent-ils pas déjà nos échanges à des fins de profilage marketing et ciblage publicitaire ?

[18] En réalité, il existe plusieurs (pour le graphe, on a considéré le poids des arêtes).



(a) (a) Connexions entrantes



(b) (b) Connexions sortantes

Fig. 5: Connexions TCP

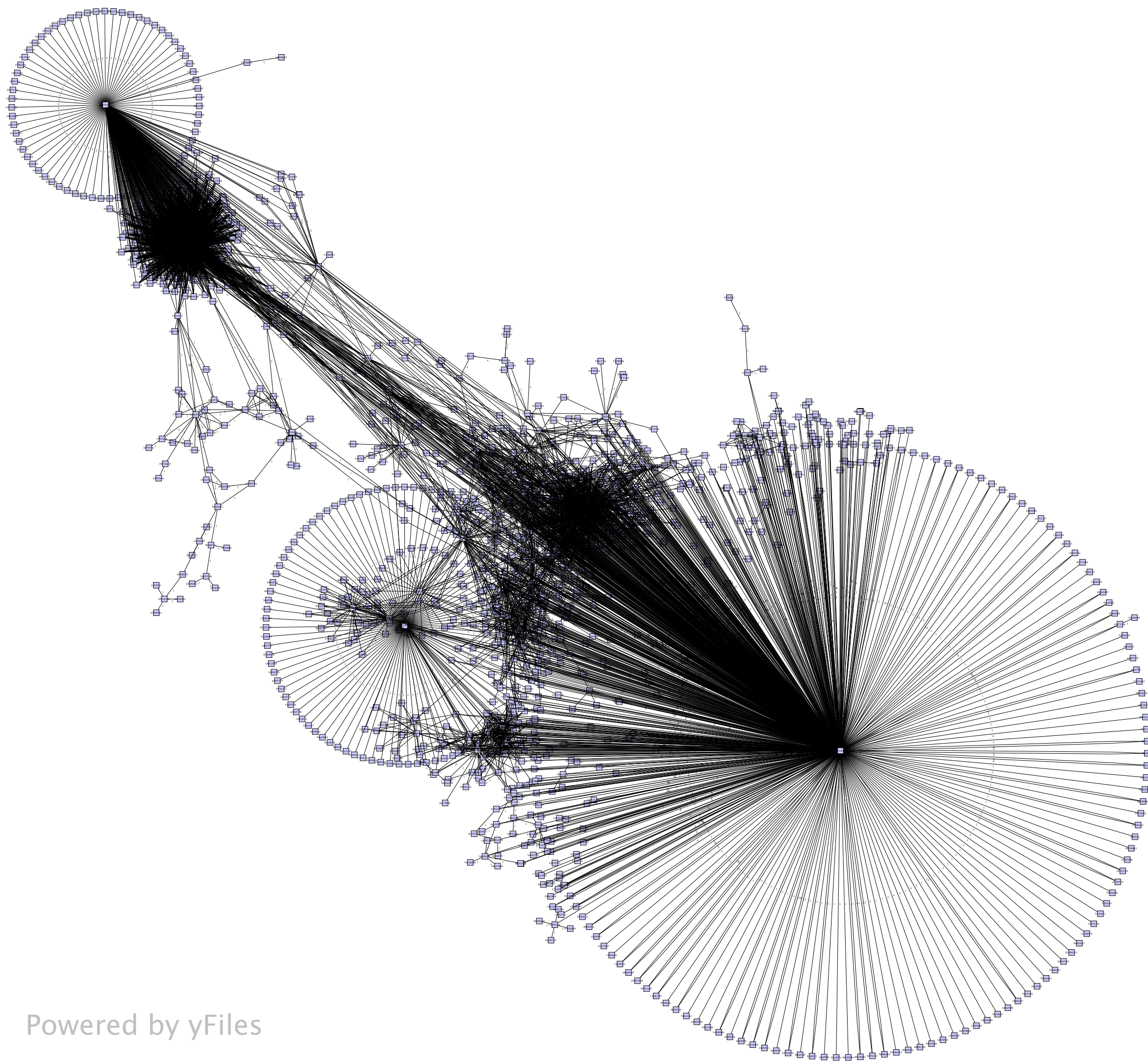


Fig. 6: Graphe d'une boîte de messagerie

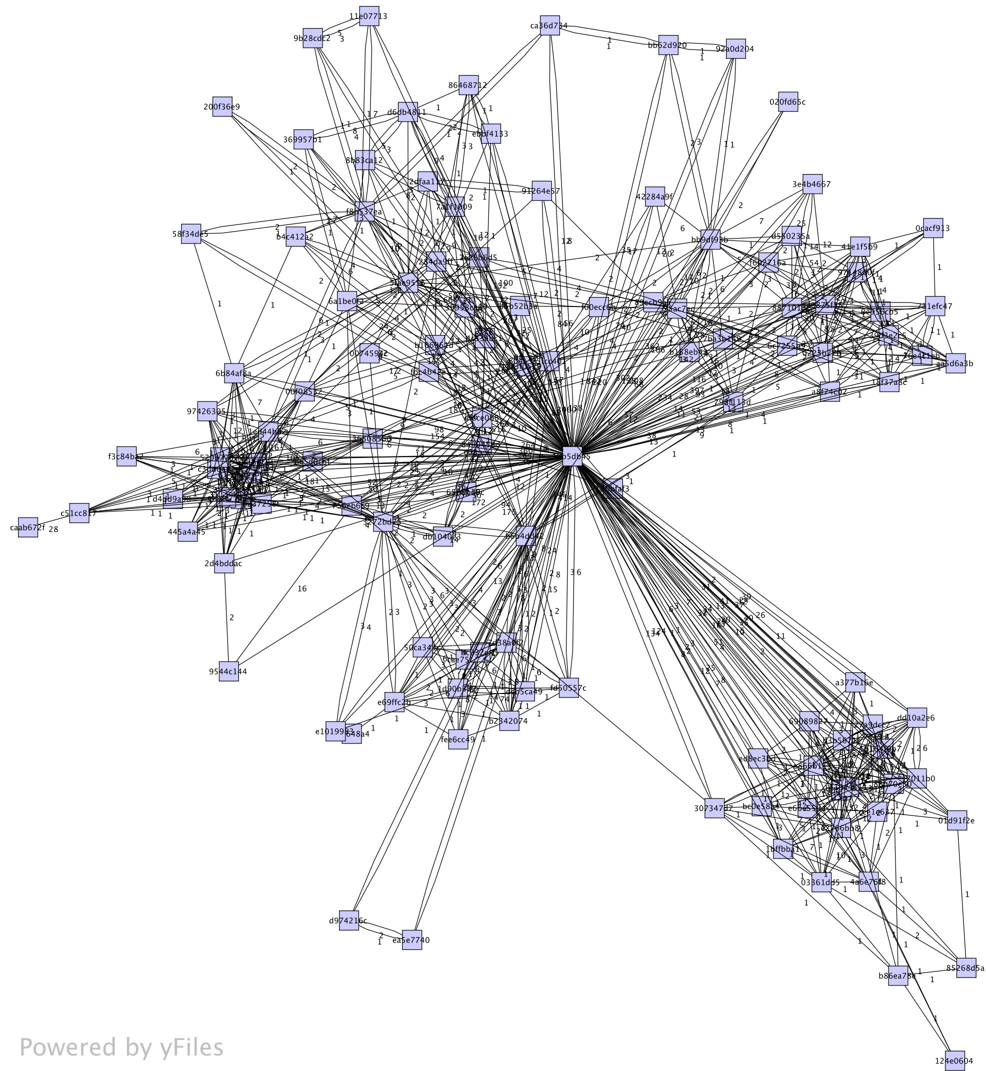
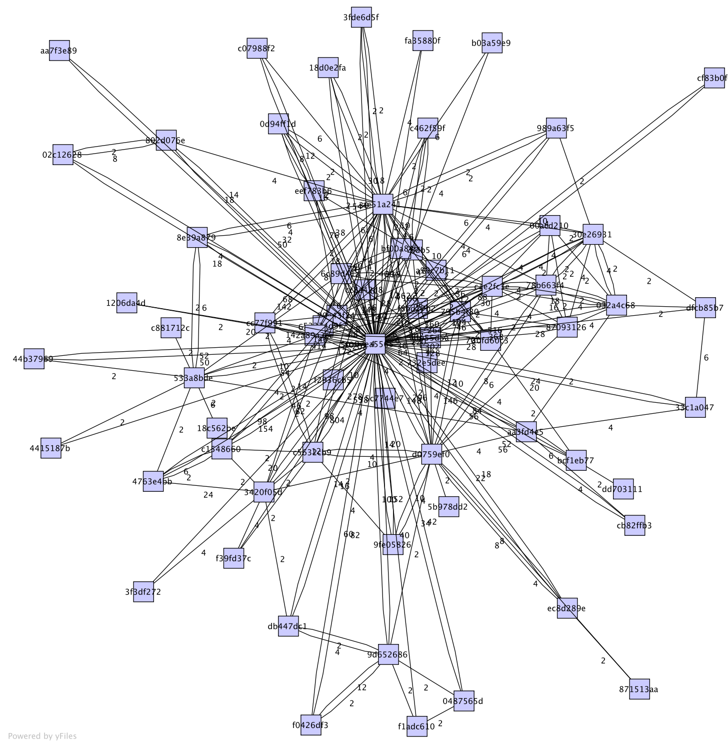
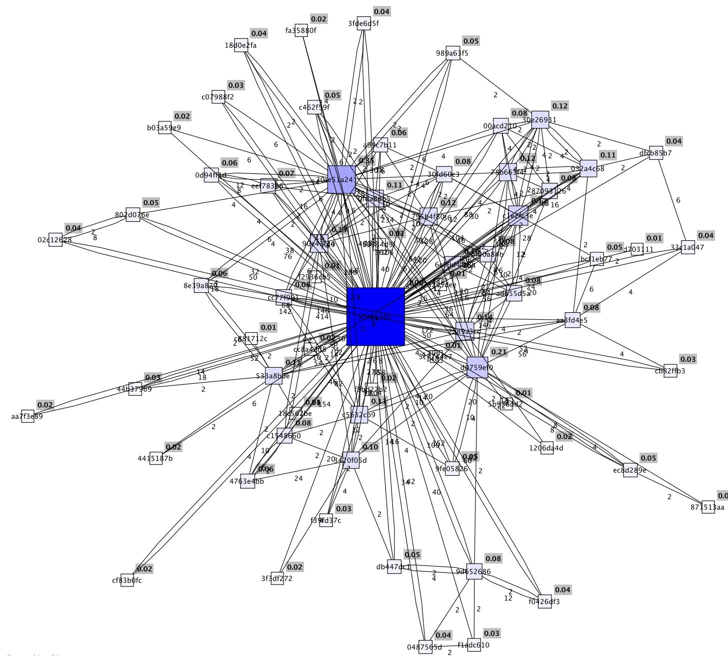


Fig. 7: Extraction d'un sous-graphe fortement connexe



(a) (a) intersection de 2 réseaux



(b) (b) centralité

FIG. 8: Représentation de réseau social

5 Analyse d'un protocole fermé

Au travers d'un exemple, nous montrons comment il est possible de reconstruire un protocole non documenté et chiffré.

OneBridge est un logiciel de synchronisation développé par Sybase. Il permet aux utilisateurs nomades d'accéder aux fonctionnalités de leur Groupware, telles que les e-mails, la liste des contacts, les mémos, en servant de passerelle entre ces serveurs et les terminaux mobiles. De plus en plus d'entreprises se tournent vers cette solution, au détriment de celle qui apparaît comme son concurrent, le BlackBerry. Afin de l'évaluer, nous avons dû préalablement *reverser* son protocole de communication afin de développer notre propre client. L'étude a été faite sur la version 5.5.

L'analyse de la cryptographie se déroule selon deux axes. D'une part, elle est employée dans les échanges entre les clients et le serveur : il s'agit donc de comprendre le protocole pour voir l'utilisation qui est faite de la cryptographie[19]. D'autre part, comme tout logiciel s'appuyant sur de la cryptographie, il y a une gestion des clés, des utilisateurs, etc., qu'il convient également d'examiner. Nous suivons donc cette double approche dans cette section.

5.1 Comprendre le protocole

Premières trames, premières intuitions On capture une session entre un client OneBridge et le serveur. La connexion se fait en TCP. Les paquets envoyés ont un en-tête `HTTP 1.1`. Ensuite, les données envoyées sont dans un format propriétaire qu'il faudra analyser.

Voici deux exemples de paquets afin d'illustrer les intuitions initiales. Le premier paquet est envoyé par le client pour établir la connexion :

```
0000  50 4F 53 54 20 2F 74 6D 3F 63 69 64 3D 30 26 76  POST /tm?cid=0&v
0010  65 72 73 69 6F 6E 3D 35 2E 35 2E 32 30 30 36 2E  ersion=5.5.2006.
0020  31 31 31 32 20 48 54 54 50 2F 31 2E 31 0D 0A 68  1112 HTTP/1.1..h
0030  6F 73 74 3A 31 39 32 2E 31 36 38 2E 31 30 30 2E  ost:192.168.100.
0040  32 31 30 0D 0A 54 72 61 6E 73 66 65 72 2D 45 6E  210..Transfer-En
0050  63 6F 64 69 6E 67 3A 63 68 75 6E 6B 65 64 0D 0A  coding:chunked..
0060  0D 0A 31 63 0D 0A 03 1B 21 00 00 15 00 00 00 01  ..1c....!.......
0070  00 00 00 06 74 6D 3A 2F 2F 00 06 74 6D 3A 2F 2F  ....tm://..tm://
0080  00 00 0D 0A 30 0D 0A 0D 0A                       ....0....
```

Le deuxième paquet est la réponse du serveur :

```
0000  48 54 54 50 2F 31 2E 31 20 32 30 30 20 53 75 63  HTTP/1.1 200 Suc
0010  63 65 73 73 0D 0A 53 65 72 76 65 72 3A 20 4F 6E  cess..Server: On
0020  65 42 72 69 64 67 65 0D 0A 73 6F 75 72 63 65 75  eBridge..sourceu
0030  72 69 3A 20 74 6D 3A 2F 2F 0D 0A 74 72 61 6E 73  ri: tm://..trans
0040  66 65 72 2D 65 6E 63 6F 64 69 6E 67 3A 20 63 68  fer-encoding: ch
0050  75 6E 6B 65 64 0D 0A 0D 0A 31 33 61 0D 0A 03 82  unked....13a....
0060  39 02 00 81 0D 82 30 81 89 02 81 81 00 9F 98 D0  9.....0.........
0070  33 E8 90 14 D5 D4 AC 3F 80 84 77 DA 96 0B 52 A5  3......?..w...R.
0080  1F AC 08 CD BC 3C B5 CF E0 82 8B 66 19 3F 71 F0  .....<.....f.?q.
0090  AC 0B 05 0E F1 13 E8 88 72 B5 09 82 E0 FA 88 68  ........r......h
00a0  F3 4F 86 1B C0 51 91 D3 FB 8B CC D9 B7 39 9F 21  .O...Q.......9.!
00b0  49 C7 E3 65 63 82 F6 13 74 01 05 BB C0 CD 35 69  I..ec...t.....5i
00c0  B4 95 9C 84 26 BE 0C 32 E2 C6 7F 64 15 C7 EB B6  ....&..2...d....
00d0  35 2E 78 21 C9 5E 96 50 54 85 B1 F0 6B 6C 32 C7  5.x!.^.PT...kl2.
00e0  C7 87 30 C0 F0 5E 9D C6 DF 79 F2 B8 21 02 03 01  ..0..^...y..!...
00f0  00 01 81 23 81 0D 82 30 81 89 02 81 81 00 C4 D1  ...#...0........
0100  81 F3 32 10 B7 E6 15 E3 AE F7 84 A2 B1 73 1D 2B  ..2..........s.+
0110  9E 32 CF 57 A1 AB 7F FC 73 F7 7B CA A5 D0 8C 41  .2.W....s.{....A
0120  AF DA 24 A0 28 74 61 CA 8D 66 3E 8B A0 7C A1 8B  ..$.(ta..f>..|..
```

[19] Cela est encore plus important quand on cherche à évaluer la sécurité du logiciel afin de pouvoir échanger avec lui.

```
0130  21 4E 8B 11 DE BA 46 81 49 71 CE 25 B4 82 D1 E6  !N....F.Iq.%....
0140  41 C1 87 68 F7 B2 C9 5A 05 7D E7 C1 22 0B 80 1C  A..h...Z.}.."...
0150  31 84 F5 12 C7 44 46 3F DA 87 C3 7F 7F D6 68 27  1....DF?......h'
0160  6E BD F6 DD 62 11 64 4A 40 5F CA E4 26 AC E3 C3  n...b.dJ@_..&...
0170  FA D4 68 A3 DE 80 EB 11 86 F6 EF 1B 88 3F 02 03  ..h..........?..
0180  01 00 01 00 00 01 00 00 00 06 74 6D 3A 2F 2F 00  .........tm://.
0190  06 74 6D 3A 2F 2F 00 00 0D 0A 30 0D 0A 0D 0A     .tm://....0....
```

En analysant l'ensemble des trames, on peut dégager plusieurs observations :
- l'en-tête HTTP est toujours le même pour le client et pour le serveur, pour plusieurs échanges ;
- `cid` est le *Company ID*, fourni dans les paramètres de connexion du client, et `version` la version du client ;
- l'en-tête est, dans les deux cas, suivi d'une longueur l en hexadécimal et d'un retour à la ligne. l octets après ce retour à la ligne, on trouve un marqueur de fin `\r\n0\r\n`.
- plusieurs chaînes semblent précédées de leur longueur, par exemple 06 suivi de `tm://\0` ;
- plusieurs blocs apparaissent, chacun précédé de sa longueur. Par exemple, on découpe le premier paquet de la manière suivante :
  `(0x15, 00 00 00 (01, 00) 00 00 (06, «tm://\0»), (06, «tm://\0»))`
  Ce même découpage fonctionne également pour le deuxième paquet.

L'expérience montre que la plupart des protocoles propriétaires sont construits sur des modèles assez semblables. On retrouve très souvent le triplet TLV (Type, Length, Value). Par-dessus cet encodage, on a bien souvent de la compression et/ou du chiffrement, ce qui semble fortement probable ici puisque les messages ne sont pas lisibles. Il s'agit donc maintenant de trouver les bits des en-têtes des paquets, bits qui décriront la construction du paquet, puis les données transportées.

Rendre le protocole compréhensible L'analyse du protocole repose en grande partie sur l'analyse des trames entre un client et le serveur. D'après la documentation, les communications sont chiffrées et compressées.

OneBridge utilise `zlib`, une bibliothèque de compression : nous créons une DLL pour espionner `zlib`. Les sources de `zlib` sont disponibles et il est aisé de les modifier pour que toutes les données compressées et décompressées par la bibliothèque soient enregistrées dans un fichier. Il suffit ensuite de remplacer la DLL initiale par notre DLL modifiée afin d'identifier toutes les données qui transitent sous forme compressée entre le client et le serveur. Les fonctions à *hooker* sont `deflate` pour la compression de données, et `inflate` pour la décompression. On écrit toutes les données qui transitent par ces deux fonctions dans un fichier de log sous forme de dumps hexadécimaux.

Les seuls flux traités par `zlib` sont les zones de données compressées dans les échanges entre le client et le serveur. En plaçant notre bibliothèque sur le serveur, on saura que :
- les données décompressées sont des données envoyées par le client ;
- les données à compresser sont des données à envoyer au client.

Enfin, en créant des profils de connexion personnalisés via l'utilitaire d'administration, il est possible de ne pas chiffrer les communications entre le client et le serveur. Les sessions seront capturées en clair, leur analyse en sera bien plus aisée.

Identifier les bibliothèques tierces Le programme utilise *RSA BSAFE Crypto-C* pour le chiffrement, et des outils *DataViz* pour la conversion de fichiers (lors d'une synchronisation de fichiers, certains documents sont convertis dans un format compatible avec les terminaux mobiles avant leur envoi). L'analyse des en-têtes des bibliothèques, ou au moins des recherches sur leurs fonctionnalités et leurs possibilités, renseigne sur le fonctionnement global du programme. Ces bibliothèques n'ont pas besoin d'être analysées, on peut les considérer comme des boîtes noires utilisées par le programme. Seules leur utilité et leur rôle importent.

Il faut également penser à regarder les noms des fonctions exportées par toutes les bibliothèques : leur nom est souvent explicite et suffit pour comprendre leur utilité. Il n'y a pas besoin de les analyser ensuite si leur rôle est déjà compris.

BSAFE est linké statiquement. Pour reconnaître ces fonctions nous avons créé des signatures pour IDA, à l'aide de FLAIR (fourni avec IDA).

Disséquer les paquets La compréhension du protocole fait appel à différentes techniques :
– le *debuggage* pour suivre le comportement de l'application une fois un paquet reçu ou pour son émission, ce qui permet d'identifier les fonctions clés ;
– le *reverse engineering* pour disséquer les fonctions clés ;
– l'expérimentation et l'intuition pour deviner le rôle de certains bits ou octets dans les paquets.

Le code à analyser étant relativement important et les trames compressées et chiffrées, l'utilisation d'un debugger est inévitable afin d'identifier les zones à analyser. Une analyse plus poussée pourra ensuite être faite sur des fonctions précises à l'aide d'un désassembleur.

Que regarder ? Trouver à quel endroit sont disséqués les paquets serait une bonne chose, afin d'analyser le parser et de dégager les différentes zones de chaque paquet. Les transmissions se font via TCP. On peut contrôler tout ce qui est envoyé ou reçu en posant des points d'arrêt sur `send` et `recv`. Les fonctions de dissection ne devraient pas être très éloignées.

Une fois ces fonctions bien identifiées, une grosse partie de *reverse engineering* commence. Il s'agit de comprendre comment est traité chaque paquet. Le langage utilisé étant de haut niveau dans ce cas-là (C++ avec de nombreux constructeurs, destructeurs et fonctions virtuelles), une analyse ligne par ligne serait trop fastidieuse.

Une représentation des fonctions par graphes est assez intéressante : en effet, cela permet de voir les chemins exécutés selon les valeurs rencontrées dans les paquets. Et ainsi, de forcer une fonction à rentrer dans un chemin qu'elle n'emprunte jamais pour comprendre son utilité, lors d'une session de debuggage.

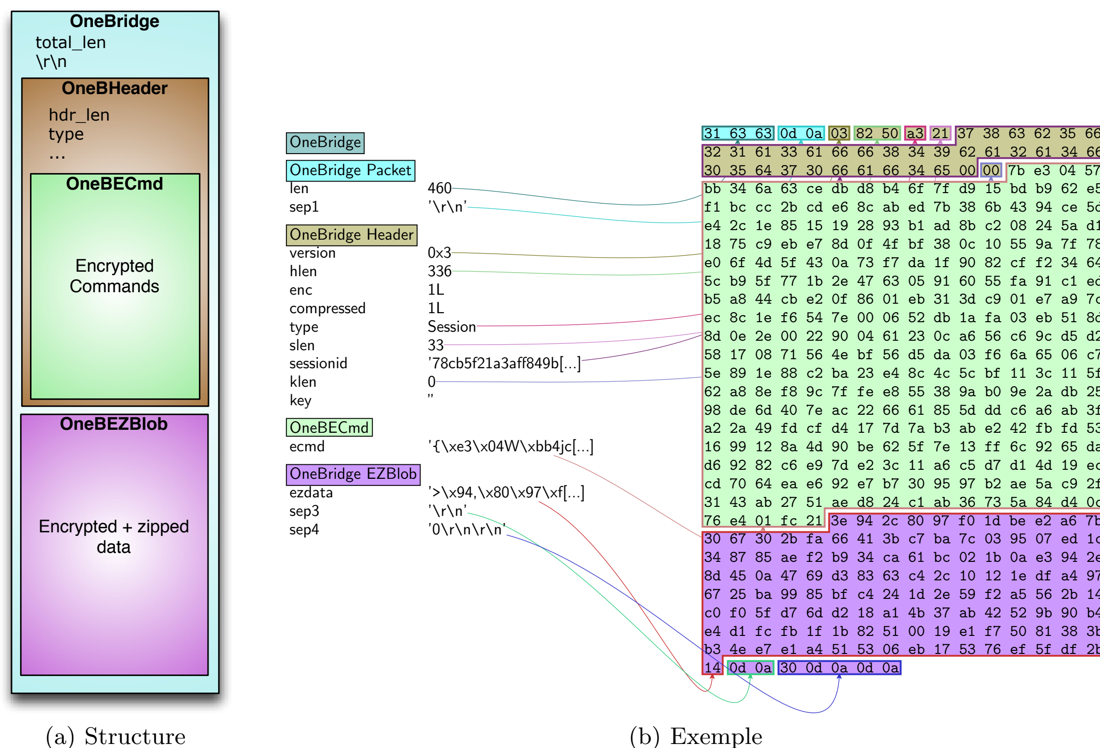
Comprendre la dissection des paquets est important, mais l'autre partie l'est tout autant : l'étape de construction des paquets à partir des données contenues par le processus. Le point d'arrêt sur `send` permet de se rapprocher des fonctions de construction. En « remontant » dans le code et en analysant les références à la procédure qui envoie les paquets, on retrouve la méthode qui les construit.

Format des paquets Lors de l'analyse, nous avons écrit un client OneBridge en Python, à l'aide de scapy. Le client a été écrit au fur et à mesure que le format a été compris. Il est un outil de test, mais permet également de finaliser la compréhension du protocole, en envoyant des paquets spécifiques et en analysant les réponses obtenues.

Les figures 9 et 10 détaillent les structures d'un paquet OneBridge. Un tel paquet est essentiellement divisé en deux parties : un en-tête suivi optionnellement des données.

La figure 9 page suivante illustre un paquet lorsqu'il est chiffré. La seule partie en clair est l'en-tête, qui contient le type du paquet : il détermine la nature des octets placés dans la suite de l'en-tête et dans la partie de données. Comme on pouvait s'y attendre, certains champs de l'en-tête déterminent la suite :
– deux bits `enc` et `compressed` signalent respectivement le chiffrement et la compression des données ;
– un champ `type` précise le type de paquet (cf. tab. 1 page 45) ;

Le récepteur du paquet déchiffre, si nécessaire, le champ `OneBECmd` (*Encrypted Command*), qui se subdivise en 3 :
– un premier bloc, `OneBKey`, contient des informations sur les clés de chiffrement, l'utilisateur et la session ;
– deux blocs indiquent ensuite respectivement les modules source et destination pour le paquet (en simplifié, comment manipuler les données). Par exemple, une commande venant d'un terminal mobile sera codée dans une chaîne commençant par `tm://`.



(a) Structure (b) Exemple

Fig. 9: Paquet chiffré

Si besoin, des données sont également présentes. C'est par exemple le cas lors d'un transfert de fichier. Ces données sont compressées et chiffrées par défaut (avec la même clé que l'en-tête). Elles sont découpées en bloc d'au plus 127 octets, et ajoutées au format wbxml[20].

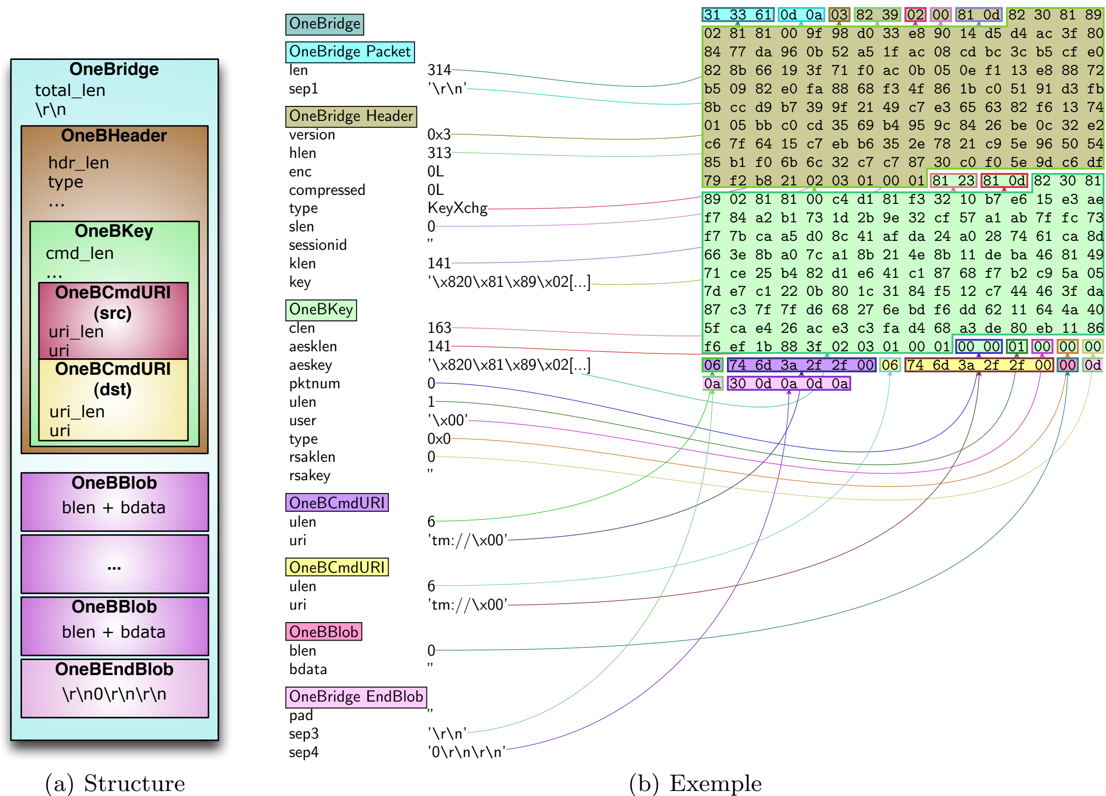
5.2 Gestion de la cryptographie

Le chiffrement des données s'effectue par *BSAFE*, une des bibliothèques de chiffrement les plus utilisées au monde. Il est illusoire de trouver une faille rapidement au niveau des algorithmes. Il est en revanche nécessaire de vérifier leur implémentation. C'est de là que viennent en général la plupart des faiblesses.

Échange de clés Le client stocke les clés publiques du serveur dans la base de registres. S'il ne possède pas les clés, il peut les demander au serveur en spécifiant `PKReq` comme type de paquet.

Le serveur lui renvoie alors deux clés RSA : la première, `RSAHostKey`, est placée dans le champ `key` de l'en-tête de la réponse (`OneBHdr`), et sera utilisée par le client pour chiffrer sa première clé symétrique. La seconde, `RSACredKey`, est placée dans le champ `aeskey` de `OneBKey`. Elle sera utilisée par le client pour

[20] Le WAP Binary XML (WBXML) est une représentation binaire du XML, développé par l'Open Mobile Alliance, afin de transmettre du XML sous forme compacte.



(a) Structure (b) Exemple

Fig. 10: Paquet déchiffré (sans donnée dans `OneBBlob`)

chiffrer son mot de passe. Le type de paquet envoyé par le serveur est `KeyXChg`. Ces clés sont émises au format ASN1. Les chiffrements à clé publique sont tous RSA-OAEP, le condensat est un SHA-1.

Le client s'authentifie avec une combinaison login / mot de passe. Le mot de passe est chiffré avec la clé `RSACredKey`. Dans le cas d'une connexion chiffrée (par défaut), la première graine utilisée pour le chiffrement symétrique (voir section 5.2 est chiffrée avec `RSAHostKey`.

Si l'authentification est valide, le serveur renvoie un paquet de type `Session`, ainsi qu'un identifiant de session dans le champ `sessionid` de l'en-tête, chaîne hexadécimale de 32 octets. Dans le cas contraire l'identifiant de session est une chaîne vide et une valeur d'erreur est retournée dans `type`.

Comme il n'y a aucun aléa envoyé du serveur au client, on peut donc rejouer une authentification capturée. En outre, notre étude a montré qu'un client peut imposer ses exigences de sécurité, y compris le chiffrement. Il semblerait donc possible de :

– capturer puis rejouer une authentification ;
– en extraire le `sessionid` qui passe dans l'en-tête en clair du paquet ;
– continuer la transaction avec le `sessionid`, mais en spécifiant d'abandonner le chiffrement (champ `enc` de `OneBHdr`.

Nous n'avons pas encore testé cette attaque, car elle nous semble peu réaliste opérationnellement.

Type	Nature	Description
1	PKReq	Demande des clés publiques
2	KeyXchg	Demande d'un échange de clés
3	Session	Données post-authentification
5	PwdErr	Erreur d'authentification
6	SidErr	Identifiant de session incorrect
7	PktErr	Erreur dans les échanges
8	ReqEnd	Demande de fin de session
9	AckEnd	Acquittement de fin de session

Tab. 1: Types de paquet déjà identifiés

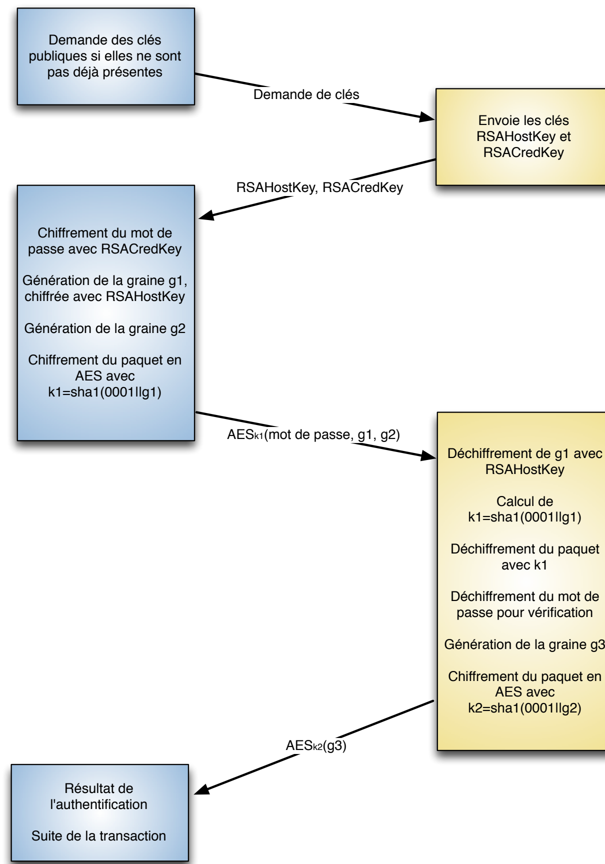


Fig. 11: Échange de clés

Chiffrement des paquets Comme vu précédemment, les données peuvent être chiffrées. Le chiffrement est actif si le flag `enc` de l'en-tête `OneBHdr` est mis à 1. Les clients et le serveur chiffrent les données en fonction du précédent paquet qu'ils ont reçu : ils gardent l'état du paquet et chiffrent le paquet suivant en conséquence.

Seuls `OneBKey` et les blobs de données sont chiffrés. L'en-tête est toujours en clair. Ces données sont chiffrées avec AES en mode CFB. La clé de chiffrement est dérivée de la graine se trouvant dans le paquet reçu. Ainsi, chaque clé n'est utilisée qu'une fois.

La graine est dérivée ainsi : $k_n = SHA-1(prefix.g_n)$, en notant ¿ g_n la n$^{\text{ième}}$ graine et k_n la k$^{\text{ième}}$ clé. $prefix$ est une chaîne constante de 4 octets. Les 128 premiers bits du résultat sont utilisés.

Stockage des clés Un des gros problèmes concernant le chiffrement des données est le stockage des clés de chiffrement. Les communications sont chiffrées, mais il faut bien enregistrer les clés quelque part, et de là peuvent découler bien des problèmes.

Sur le serveur

Le serveur doit stocker les clés publiques et privées utilisées pour chiffrer le mot de passe et la première clé AES envoyée par le client. Il doit aussi garder en mémoire la dernière clé de chiffrement symétrique envoyée par chaque client pour déchiffrer le message, ainsi que la graine envoyée par le client pour qu'il puisse chiffrer sa réponse.

Les clés RSA sont stockées dans la base de registres (`HKLM\SOFTWARE\Extended Systems\OneBridge Sync Server`). Cette clé contient deux autres clés, `tmaininfo` (notée `RsaHostKey`) et `tidentinfo` (notée `RsaCredKey`). La première contient les clés nécessaires au chiffrement de la clé AES envoyée par le client , et `tidentinfo` celle utilisée pour le chiffrement du mot de passe de l'utilisateur . En plus de ces clés, deux autres clés sont utilisées pour vérifier l'authentification du serveur par le client (signatures ECDSA).

Chacune de ces clés contient 4 valeurs de type `REG_BINARY` :
- `rp`, la clé publique RSA du serveur, en clair ;
- `rs`, la clé privée du serveur, chiffrée avec AES mais la clé de chiffrement est une clé fixe, présente dans le programme ;
- `p`, la clé publique ECDSA d'authentification. La courbe et le générateur G étant dans le programme, seul le point $K = kG$ est stocké ;
- `s`, la clé privée k d'authentification, chiffrée avec RC4, dont la clé de chiffrement est également fixe (en fait la même clé que pour l'AES).

Ces deux clés de registre sont accessibles uniquement avec un compte administrateur.

La clé fixe utilisée par l'AES et le RC4 a été retrouvée à partir du programme *Key Manager* permettant l'export des clés de chiffrement, notamment pour faire communiquer les serveurs OneBridge entre eux si on a installé un proxy. Sa valeur est `57 43 BC EE D0 32 31 28 FB CB D4 32 AC 8F 01 86 4F A1 55 73`. Lors d'un export, les clés privées sont tout d'abord déchiffrées (avec la clé fixe), puis chiffrées à partir d'un mot de passe spécifié par l'utilisateur afin qu'elles n'apparaissent pas en clair une fois exportées. Lors d'un import, elles sont déchiffrées avec le mot de passe de l'utilisateur puis rechiffrées avec la clé fixe.

Sur le client

Le client peut, si l'administrateur n'a pas désactivé cette possibilité, enregistrer son mot de passe. Il est stocké dans la base de registres. Le mot de passe n'est jamais en clair : il est chiffré avec RSA-OAEP, avec la clé `RSACredKey`. Le registre contient donc le mot de passe tel qu'il doit être envoyé au serveur. Ainsi, on ne peut pas récupérer le mot de passe de l'utilisateur via la base de registres, mais on peut se connecter avec ses identifiants. Il conviendrait donc de désactiver ce stockage.

Base de données du serveur Toutes les informations nécessaires au serveur lors des connexions sont enregistrées dans une base de données dans un format propriétaire (Advantage Database). En particulier, on trouve des informations relatives aux utilisateurs connectés dans la table `TMS`. Les fichiers de cette table se trouvent dans le dossier `Data` : `TMS.ADT`, `TMS.ADI` et `TMS.ADM`. Elle contient les logins des utilisateurs, leur identifiant de session, les clés de chiffrement des paquets, le condensat de leur mot de passe, etc.

Un outil a été codé pour lister le contenu de toutes les tables du serveur. Voici ce que donne la table `TMS` après que l'utilisateur `fred` se soit connecté :

```
uid            fred
gid            Starter Group
devid          b5b3152c687142d69f3f5eb475e58025
devtype        windows
seqnum         4
sessionid      8e24aba27f277045b7c1b6695ff6f9bb
starttime      39287.62521991
reqtime        39287.62521991
resptime       0.00000000
endtime        0.00000000
action
actionmsgcount 0
nextpacketkey
[0000]   0C 14 00 00 00 A8 33 25 9F 1A DB C6 CB 6E E9 D7     ......3%.....n..
[0010]   38 A5 01 47 CB 95 96 C0 10 00 00 00 00 00 00 00    8..G............
[0020]   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
...
prevpacketkey
[0000]   0C 14 00 00 00 10 A8 A1 3C D8 97 FF 40 6D 1A 04     ........<...@m..
[0010]   E8 15 00 E0 30 72 C0 41 99 00 00 00 00 00 00 00     ....0r.A........
[0020]   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
...
authresult
...
creddigest     AAE865D3638CDFFA67378F5B30F5380B
connectstart   20070724T130019Z
```

Les champs `prevpacketkey` et `nextpacketkey` ne contiennent pas véritablement la clé de chiffrement, mais la graine qui a servi à la générer. Ils sont composés d'un octet correspondant certainement à l'algorithme utilisé, de la taille de la graine sur 4 octets, et de la valeur de la graine.

`creddigest` est un condensat SHA-1 du login et du mot de passe concaténés et convertis en Unicode. Il peut être bruteforcé pour obtenir les *credentials* de l'utilisateur sur un serveur compromis.

Il n'y a pas de trou majeur dans l'implémentation des algorithmes de chiffrement dans le programme. Les permissions sur les clés secrètes sont correctement gérées, et les seules données récupérables sur les connexions ne sont que des condensats, les mots de passe des utilisateurs n'étant jamais stockés en clair.

6 Conclusion

Le plan de cet article représente la méthodologie d'un attaquant face à un logiciel de cryptographie. Il commence par rechercher les erreurs bêtes (et néanmoins courantes) comme la présence de secrets là où ils ne devraient pas se trouver, que ce soit en clair ou sous une forme quelconque : codés en dur dans le fichier, dans la mémoire, ou dans les fichiers système (swap et hibernation).

Si cela échoue, l'analyse doit se poursuivre en identifiant les primitives cryptographiques dans les binaires, ou en reconnaissant les flux chiffrés, ainsi que le chiffrement employé, dans des communications. Nous avons présenté comment, dans ces deux cas, il était envisageable d'aider voire d'automatiser cette reconnaissance.

Quand l'attaquant ne peut vaincre la cryptographie, il lui reste bien souvent la possibilité d'examiner le canal de communication. Bien souvent, cela s'avère déjà riche en informations : qui discute avec qui ? Avec quelle intensité ? Etc. Que ce soit en informatique avec la supervision ou l'analyse post-mortem, ou en sociologie avec les réseaux sociaux, dans les deux cas, on dispose de méthodes et d'outils pour tracer des relations entre les entités (respectivement des machines et des personnes).

Enfin, un exemple vient illustrer une partie de ce qui a été préalablement présenté. Il s'agit de réaliser un client pour un protocole, non documenté et chiffré par défaut. Cela passe autant par l'analyse du client officiel que par celle des trames capturées sur le réseau. Cette opération est souvent nécessaire dans le cadre d'audit de sécurité pour des applications.

Il est intéressant de constater qu'on peut lire cet article autrement ... À la démarche de l'attaquant, on peut substituer celle du fonctionnement normal d'un système d'information. On relève alors les multiples points où la cryptographie est susceptible de servir. On constate alors que nous avons également couvert toutes ces possibilités, comme illustré sur la figure 12. Ceci n'est guère surprenant dans la mesure où un attaquant va également chercher à agir en ne s'interdisant aucune action, voire en les combinant. Et encore, nous nous sommes restreints aux attaques sur la cryptographie ou le canal, pas sur l'environnement : l'utilisation de logiciels espions, comme les *keyloggers*, s'avère également redoutable pour venir à bout des codes les plus robustes.

On l'aura compris : oui, la cryptographie augmente la sécurité ... mais à condition qu'elle soit bien déployée et d'en comprendre aussi les limites. Ce n'est certainement pas un gage ultime de sécurité comme « on » tente souvent de nous en convaincre.

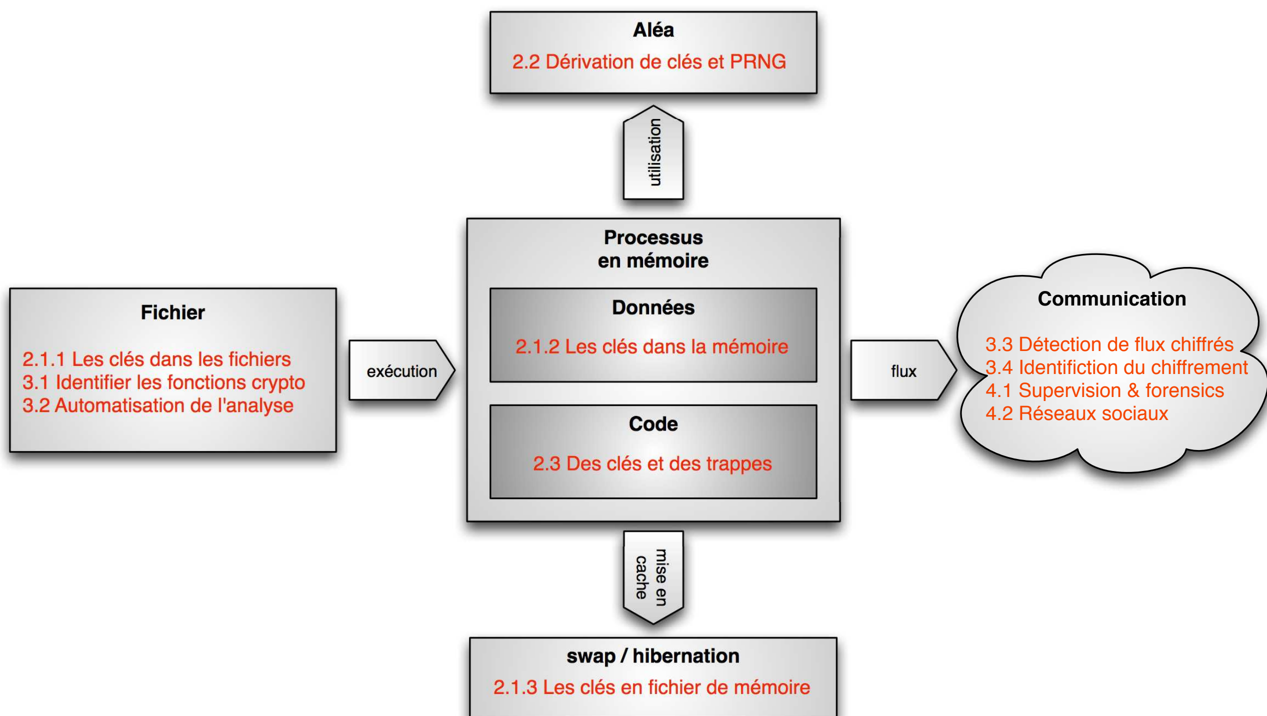


Fig. 12: Organisation du document en fonction de l'endroit où est utilisée la cryptographie

Références

1. Shannon, C.E. : Communication theory of secrecy systems. Bell Syst. Tech. Journal **28** (1949) 656–715
2. Filiol, É. : La simulabilité des tests statistiques. MISC magazine **22** (2005)
3. Filiol, É. : Techniques virales avancées. Collection IRIS. Springer Verlag France (2007)
4. of Standards, N.I., (NIST), T. : A statistical test suite for random and pseudorandom number generators for cryptographic applications (2001) `http://csrc.nist.gov/publications/nistpubs/800-22/sp-800-22-051501.pdf`.
5. of Standards, N.I., (NIST), T. : Recommendation for random number generation using deterministic random bit generators (March 2007) `http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf`.
6. Shamir, A., van Someren, N. : Playing "hide and seek" with stored keys. Lecture Notes in Computer Science **1648** (1999) 118–124
7. Carrera, E. : Scanning data for entropy anomalies (May 2007) `http://blog.dkbza.org/2007/05/scanning-data-for-entropy-anomalies.html`.
8. Carrera, E. : Scanning data for entropy anomalies ii (July 2007) `http://blog.dkbza.org/2007/05/scanning-data-for-entropy-anomalies-ii.html`.
9. Bordes, A. : Secrets d'authentification windows. In : Proc. Symposium sur la Sécurité des Technologies de l'Information et de la Communication (SSTIC). (June 2007) `http://actes.sstic.org/SSTIC07/Authentification_Windows/`.
10. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., P., W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W. : Lest we remember : Cold boot attacks on encryption keys. Technical report, Princeton University (2008) `http://citp.princeton.edu/memory/`.
11. Provos, N. : Encrypting virtual memory. Technical report, University of Michigan (2000) `http://www.openbsd.org/papers/swapencrypt.ps`.
12. Ruff, N., Suiche, M. : Enter sandman (why you should never go to sleep) (2007) `http://sandman.msuiche.net/`.
13. Johnston, M. : Mac OS X stores login/keychain/filevault passwords on disk (June 2004) `http://seclists.org/bugtraq/2004/Jun/0417.html`.
14. Appelbaum, J. : Loginwindow.app and Mac OS X (February 2008) `http://seclists.org/bugtraq/2008/Feb/0442.html`.
15. Liston, T., Davidoff, S. : Cold memory forensics workshop. In : CanSecWest. (2008)
16. Aumaitre, D. : Voyage au coeur de la mémoire. In : Proc. Symposium sur la Sécurité des Technologies de l'Information et de la Communication (SSTIC). (June 2008)
17. Dorrendorf, L., Gutterman, Z., Pinkas, B. : Cryptanalysis of the random number generator of the windows operating system. Cryptology ePrint Archive, Report 2007/419 (2007)
18. Kortchinsky, K. : Cryptographie et reverse-engineering en environnement win32. In : Actes de la conférence SSTIC 2004. (2004) 129–144 `http://www.sstic.org`.
19. Chow, S., Eisen, P., Johnson, H., van Oorschot, P. : White-box cryptography and an aes implementation (2002)
20. Jibz, Qwerton, snaker, xineohP : Peid `http://www.peid.info`.
21. Guilfanov, I. : Findcrypt (January 2007) `http://www.hexblog.com/2006/01/findcrypt.html`.
22. Immunity, I. : Immunity debugger `http://www.immunitysec.com/products-immdbg.shtml`.
23. Shannon, C.E. : A mathematical theory of communication. Bell Syst. Tech. Journal **27** (1948) 379 – 423 et 623 – 656

24. Filiol, É. : A family of probabilistic distinguishers for e0 - à paraître (2008)
25. Filiol, É. : Modèles booléens en cryptologie et en virologie. PhD thesis, Thèse d'habilitation - Université de Rennes (2007)
26. Filiol, É. : Preuve de type zero knowledge de la cryptanalyse du chiffrement bluetooth. MISC magazine **26** (2006)
27. Filiol, É. : Techniques de reconstruction en théorie des codes et en cryptographie. PhD thesis, École Polytechnique (2001)
28. Pilon, A. : Sécurité des secrets du poste nomade. MISC magazine **Hors série 1** (2007)
29. D. Aumaitre, J.B. Bedrune, B.C. : Quelles traces se dissimulent malgré vous sur votre ordinateur ? (February 2008) `http://esec.fr.sogeti.com/FR/documents/seminaire/forensics.pdf`.
30. Bejtlich, R. : The Tao of Network Security Monitoring : Beyond Intrusion Detection. Addison Wesley Professional (2004)
31. Arcas, G. : Network forensics : cherchez les traces sur le réseau. MISC magazine **35** (2008)
32. Raynal, F., Berthier, Y., Biondi, P., Kaminsky, D. : Honeypot forensics : Honeypot forensics : analyzing the network. IEEE Security & Privacy Journal (June 2004)
33. Raynal, F., Berthier, Y., Biondi, P., Kaminsky, D. : Honeypot forensics : analyzing system and files. IEEE Security & Privacy Journal (August 2004)
34. Barnes, J. : Class and committees in a norwegian island parish. Human Relations **7** (1954) 29–58
35. Granovette, M. : The strength of weak ties. American Journal of Sociology **78** (1973) 1360–1380
36. Burt, R. : Structural holes : the social structural of competition. Harvard University Press (1992)