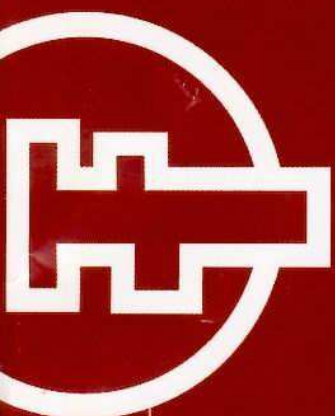


100% SÉCURITÉ INFORMATIQUE

France Metro : 7,45

Eur - 12,5 CHF - BEL, LUX, PORT.CONT : 8,5 Eur - CAN : 13 \$C - MAR : 75 DH

Mars - Avril 2004



misc

12

MULTI-SYSTEM & INTERNET SECURITY COOKBOOK

LA FAILLE VENAIT DU LOGICIEL !

Découvrez comment élaborer, développer et évaluer
votre sécurité logicielle

- WiFi, un an après : WPA, 802.11 i

- Cryptographie quantique :
chiffrer avec un photon

- La fin des "Buffer Overflow"
dans Windows (?)

- Introduction** —> Haute Disponibilité, concepts et principes
- Serveur** —> Utilisez Heartbeat pour assurer un service
 - > Sécurisez vos liens réseaux par le Channel Bonding
 - > Synchronisez simplement vos données avec Unison
- Stockage** —> Configuration du RAID sous Linux
 - > Basculez votre Debian sur du RAID logiciel
 - > Construisez un cluster HA avec Heartbeat, ENBD et le RAID Logiciel
 - > DRBD : le mirroring de périphériques block via le réseau
- Bases de données** —> Introduction à la réplication de bases de données
 - > La Haute Disponibilité sous PostgreSQL
 - > Réplication avec MySQL

Déjà chez votre marchand de journaux !

HORS SÉRIE N°18





Murailles en paille

Promis, cette fois je ne vous ennuierais pas avec mes neveux : ils vont bien et les parents deviennent de plus en plus inquiets à mesure que mes neveux gambadent de mieux en mieux. Quant à ma grand-mère, elle fête ses 82 ans et est toujours aussi rayonnante.

En revanche, je ne peux pas m'empêcher de vous rappeler - honteusement bien évidemment, devant une publicité à peine masquée - les dates du Symposium pour la Sécurité des Technologies de la Communication et de l'Information (SSTIC) : ce sera du 2 au 4 Juin 2004, à Rennes. Le programme devrait être disponible vers fin Mars et les inscriptions ouvertes dans la foulée.

Cela étant établi, passons aux choses sérieuses. Le thème du dossier de ce numéro : concevoir, réaliser et évaluer la sécurité de ses logiciels. Généralement, dès qu'il s'agit de mettre en place "de la sécurité", la tendance naturelle est de se focaliser sur le périmètre de ce qu'il faut protéger. Toutefois, historiquement, ce modèle a montré ses limites : les châteaux forts aux murailles épaisses et robustes ont peu à peu été abandonnés.

De plus, l'efficacité des murailles dégarnies bâties sur des sables mouvants est douteuse. En fait, la sécurité se construit à différents niveaux, les fondations, la muraille elle-même, et les petits soldats dessus.

D'un point de vue stratégique maintenant, demandons nous comment nous pouvons nous emparer de notre château fort muni d'épaisses et robustes murailles. Dans le cas d'un siège par exemple, nous attaquons les dépendances du château : nous abattons systématiquement les pigeons voyageurs, et polluons toutes les sources d'eau possibles. Il ne reste plus qu'à attendre. Par rapport à notre système, les dépendances sont : les routeurs, les DNS, le fournisseur d'accès...

Malheureusement, il est rarement possible de sécuriser soit même de toutes ces dépendances. C'est là que la confiance intervient, mais il faut conserver cela à l'esprit.

En mettant en place notre forteresse, nous avons posé une hypothèse très très forte : la solidité du ciment et des parpaings (<lolo>du tech Jean-Pierre </lolo> ;-). Si ce ciment est particulièrement friable, on risque d'entendre résonner encore longtemps les trompettes de Jericho. De même si les parpaings sont en craie.

Tout ça pour dire que si c'est nécessaire de fortifier son périmètre, encore faut-il que la muraille tienne le coup. Concevoir des gros systèmes afin de se prémunir contre toutes les attaques (ou presque) ne constitue pas uniquement un problème d'architecture. À un niveau plus microscopique, il faut encore que ce système soit lui-même bien sécurisé, ce qui passe entre autre par certaines contraintes. Son architecture est certes cruciale, mais la manière dont il sera programmé puis testé et évalué aura un impact sur sa sécurité. La tendance générale est de passer beaucoup de temps sur la conception, moins sur la programmation et presque pas sur l'évaluation (forcément, la dead line approche, les ressources sont épuisées, bref, il est trop tard).

Cette approche est différente et complémentaire de la notion de défense en profondeur (mise en place de multiples lignes de défenses, indépendantes les unes des autres), vision horizontale du problème. Là, il s'agit pour une ligne de défense, de bien s'assurer que la sécurité est présente à tous les niveaux : du système qui fournit la sécurité aux liens qu'il entretient avec d'autres systèmes, en passant par les briques élémentaires qui composent le système. Avec la défense en profondeur, on fait l'hypothèse raisonnable qu'une ligne de défense ne tiendra pas éternellement et qu'il faut donc les multiplier. Par rapport à notre château, cela revient à ajouter une herse, un pont-levis, des douves, etc., à notre muraille.

Je conclurai par un exemple réel. Assez périodiquement revient la rumeur d'un 0-day sur la pile IP des Windows, Linux et autres BSD. Une personne prétendant avoir cet exploit discute avec un de ses amis, et lui dit de blinder son firewall autant qu'il veut, il ne pourra rien y faire. L'autre, ne le croyant pas une seconde, bloque tout le trafic entrant, et se dit qu'il va lancer un sniffer pour récupérer l'exploit en question. Manque de chance, la faille était dans le sniffer, et il s'est fait rooter sa machine.

Bonne lecture,

Frédéric Raynal

MISC est un magazine consacré à la sécurité informatique sous tous ses aspects (comme le système, le réseau ou encore la programmation) et où les perspectives techniques et scientifiques occupent une place prépondérante. Toutefois, les questions connexes (modalités juridiques, menaces informationnelles) sont également considérées, ce qui fait de MISC une revue capable d'appréhender la complexité croissante des systèmes d'information, et les problèmes de sécurité qui l'accompagne.

MISC vise un large public de personnes souhaitant élargir ses connaissances en se tenant informées des dernières techniques et des outils utilisés afin de mettre en place une défense adéquate. MISC propose des articles complets et pédagogiques afin d'anticiper au mieux les risques liés au piratage et les solutions pour y remédier, présentant pour cela des techniques offensives autant que défensives, leurs avantages et leurs limites, des facettes indissociables pour considérer tous les enjeux de la sécurité informatique.



GUERRE DE L'INFO

DE LA GUERRE DE L'INFORMATION
À L'INFO-SÉCURITÉ ; p 6 à 9



SYSTÈME

LA FIN DES "BUFFER OVERFLOW"
DANS WINDOWS (?) ; p 54 à 59



RÉSEAU

LA SÉCURITÉ DES RÉSEAUX 802.11 :
QUOI DE NEUF DEPUIS UN AN ? ; p 60 à 66



FICHES TECHNIQUES

DU MIEL EN POT VIRTUEL AVEC VMWARE ;
p 67 à 71



SCIENCE

LA CRYPTOGRAPHIE QUANTIQUE :
À LA CONQUÊTE DES PHOTONS ; p 72 à 79

LA FAILLE
VENAIT DU LOG
Découvrez comment élaborer, développer
votre sécurité logique

Ont rédigé
ce numéro
et y ont collaboré :

Danielle Kaminsky

Philippe Biondi

Pascal Junod

Georges Bart

Nicolas Brulez

Victor Vuillard

Nicolas Ruff

Cédric Blancher

Olivier Gay

Grégoire Ribordy

Frédéric Raynal

Denis Bodor



misc

MULTI-SYSTEM & INTERNET SECURITY COOKBOOK

est édité par Diamond Editions
B.P. 121 - 67603 Sélestat Cedex
Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09
E-mail : lecteurs@miscmag.com
Abonnement : abo@miscmag.com
Site : www.miscmag.com

Directeur de publication : Arnaud Metzler

Rédacteur en chef : Frédéric Raynal

Rédacteur en chef adjoint : Denis Bodor

Conception graphique : Katia Paquet

Impression : LeykamDruck
Graz

Secrétaire de rédaction : Carole Durocher

Responsable publicité : Véronique Wilhelm
Tél. : 03 88 58 02 08

Distribution :

(uniquement pour les dépositaires de presse)

MLP Réassort :

Plate-forme de Saint-Barthélemy-d'Anjou.

Tél. : 02 41 27 53 12

Plate-forme de Saint-Quentin-Fallavier.

Tél. : 04 74 82 63 04

Service des ventes : Distri-médias :

Tél. : 05 61 72 76 24

Service abonnement :

Tél. : 03 88 58 02 08

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

Dépôt légal : 2^e Trimestre 2001

N° ISSN : 1631-9030

Commission Paritaire : 02 04 K 81 190

Périodicité : Bimestrielle

Prix de vente : 7,45 euros

Printed in Austria
Imprimé en Autriche

★ **Bulletin d'abonnement** ; p 81

★ **Commandez les anciens numéros de Misc et hors-série de Linux Magazine !** p 82

DOSSIER

LA FAILLE VENAIT DU LOGICIEL !

Découvrez comment élaborer, développer
et évaluer votre sécurité logicielle

■ **SÉCURITÉ DES LOGICIELS ;**
p 10 à 17

■ **DIX DANGERS QUI GUETTENT
LE PROGRAMMEUR
DE CRYPTOGRAPHIE ;**
p 18 à 24

■ **MÉTHODES D'ÉVALUATION
DE SÉCURITÉ : TCSEC, ITSEC ET CC ;**
p 25 à 33

■ **RECHERCHE DE VULNÉRABILITÉS
PAR DÉSASSEMBLAGE ;**
p 34 à 46

■ **SÉCURITÉ LOGICIELLE :
ÉTUDE DE CAS ;**
p 47 à 52



De la guerre de l'information à l'info-sécurité



Empruntées aux militaires, les techniques de guerre de l'information à la portée de tous brouillent les cartes.

L'information procure du sens : elle nous permet d'appréhender la réalité, et nous attendons d'elle un lien de connaissance et de confiance. Qu'on ne s'y trompe pas : l'information est à prendre dans tous les sens du terme : il s'agit aussi bien du contenu que des systèmes qui permettent de la capter et de la transmettre, parmi lesquels figurent les systèmes d'information [1].

L'expression "guerre de l'information" est tirée de l'art militaire. Elle a toujours été l'une des composantes importantes de la stratégie, utilisée dans un schéma global de ralliement, d'affaiblissement, de désorientation ou de destruction, en passant par tout un nuancier d'effets souhaités. L'information n'est pas dissociable de la guerre, elle fait partie depuis toujours de ses moyens d'action, et en conséquence de ses cibles chez l'adversaire. Dans tous les cas, la guerre de l'information se décompose en un contexte, un objectif, une stratégie, une ou des cible(s), des actions de renseignement sur la cible et le terrain (au sens large), des armes et des outils, des acteurs, un mode opératoire (technique et psychologique), un résultat ou effet : réussite ou échec [2].

TOUS LES DOMAINES SONT DÉPENDANTS DE L'INFORMATION, LE CHAMP DE BATAILLE EST PARTOUT

Renseignement, acquisition de l'information, moyens d'écoute et de vision, cartographie, moyens de communication et de transmission, moyens de calcul et d'analyse, moyens de coordination et de contrôle, représentation de la situation, sens de l'action, moral des troupes et des populations, les siennes, les adverses et les autres : l'information agit dans tous les champs du processus de guerre.

Mais hors du seul domaine militaire, l'information agit aussi dans tous les processus de cognition, de relations, d'affrontement, de conflit, de compétition dans la vie personnelle des individus, les rapports sociaux et l'activité économique des entreprises. Il n'est donc pas étonnant que l'expression "guerre de l'information" soit passée dans le langage courant pour désigner des actions qui ne relèvent pas du domaine militaire [3]. Qu'ils visent des états, des entreprises ou des individus, ou qu'ils soient commis par eux, les méthodes et outils de la guerre de l'information sont quasiment les mêmes. Les objectifs aussi : déstabiliser, désorienter, aveugler, affaiblir, détruire, influencer,

ou toute action pour agir sur la représentation de la situation, y compris pour motiver son propre camp.

Dans le monde d'aujourd'hui, toute organisation est devenue fortement dépendante de l'information qu'elle produit, acquiert, stocke, analyse, transmet. Il n'y a pas d'un côté les militaires et de l'autre la société : les intérêts civils et militaires convergent et se chevauchent. Le théâtre d'opération et l'espace des réseaux se superposent : le champ de bataille est partout. C'est pourquoi la guerre de l'information concerne aussi les professionnels de la sécurité informatique [4] [5].

Les aspects de guerre de l'information sont trop nombreux pour être tous détaillés dans un seul article. Nous avons sélectionné des cas concrets militaires et civils. Leur analyse transversale est susceptible de mettre en évidence des caractéristiques significatives pour les professionnels de la sécurité informatique qui ne sont pas militaires. La guerre de l'information agit sur les moyens de connaissance, de compréhension, d'analyse, de perception d'une situation ou d'une action, et donc sur le processus de décision et de contrôle. Pour celui qui la mène, le but est de favoriser ses propres actions.

RÉSEAUX : LA DISTANCE NE COMPTE PAS POUR ATTEINDRE SA CIBLE

Pendant la deuxième guerre du Golfe, les Américains ont recouru à des actions directes pour dissuader les leaders irakiens de continuer à soutenir le régime de Saddam Hussein [6]. Ils leur ont envoyé des messages SMS sur leurs téléphones portables et leurs adresses emails en leur faisant entrevoir les risques qu'ils couraient personnellement s'ils persistaient à appuyer leur chef.

Il s'agit ici d'action ciblée à distance, menée sans recourir à la moindre destruction physique ou logique. En agissant de cette manière sur le moral des proches du chef adverse, le but est de briser la cohésion de son commandement et d'isoler le chef. Cette méthode veut agir directement sur le processus de décision des supporters en injectant de la peur : si vous ne faites pas ça, vous allez au-devant de graves ennuis. Sans aller jusqu'à vouloir les rallier véritablement à sa propre cause, le commandement américain a visé ici à affaiblir la force adverse en paralysant l'action de soutien des leaders. Dans ce cas de figure : l'objectif peut être réalisé à distance, à court terme et par de petites équipes. L'émetteur des messages n'est pas masqué.

Pour agir, l'attaquant n'a pas de contrainte de distance pour atteindre sa cible [7]. Les hommes qui effectuent cette opération n'ont pas

besoin de se transporter physiquement sur le terrain, au plus près de leurs cibles. Or, toute l'histoire de la guerre comporte la contrainte de la distance. Les moyens - téléphone et emails - sont utilisables de loin par les opérateurs. Peut-être même en restant à leur base, ou dans leur pays, ou n'importe où ailleurs, du moment qu'il y a une liaison téléphonique et un accès à Internet. Pas de fusil ou de tank dans cette séquence d'action : le téléphone et le courrier électronique sont des outils de la vie de tous les jours.

Ce sont également les outils des pirates ou autres attaquants. Et ils leur procurent les mêmes avantages : pour eux aussi, l'attaque via les moyens électroniques et informatiques donne le moyen de s'affranchir du problème de la distance. Ils peuvent agir de loin tout en restant hors de portée. Lancer une offensive sans s'exposer physiquement.

RÉSEAUX : L'OBSTACLE DU TEMPS AISÉMENT FRANCHI

Il suffit d'utiliser les ordinateurs et le téléphone pour être instantanément connecté sur un monde sans limites, affranchi des contraintes physiques de distance et de temps. Constamment opérationnels 24 heures sur 24, les moyens de communication et les réseaux informatiques donnent le moyen d'agir sans contrainte de jour ou de nuit, et quelles que soient les conditions météo. Dans une opération de guerre de l'information via ou contre les systèmes d'information, peu importe qu'il y ait du brouillard, qu'il fasse chaud ou froid. L'action est possible 24 heures sur 24, et si l'attaquant le désire, en temps continu.

Les moyens informatiques sont également propices à des actions simultanées : attaques en déni de service. Ou activées en différé : effet retard inclus dans le code de programmes malicieux, activation de programmes déposés sur les machines pénétrées, communications avec des portes dérobées.

MÉTHODES REINES : TROMPERIE ET PRODUCTION D'ILLUSION

Les acteurs de la guerre de l'information ne se désignent pas forcément eux-mêmes. Ainsi, dans une opération psychologique (PSYOPS) dont l'objectif est pourtant similaire à celui décrit dans l'exemple précédent, c'est au contraire la mystification sur l'émetteur de l'information qui est utilisée.

En 2003, dans le but de retourner l'opinion des partisans de Saddam Hussein contre lui, les Américains ont mis en place des opérations de PSYOPS dites "Noires". L'une d'entre elles a consisté à monter une radio, Radio Tikrit [8]. L'opération se déroule en plusieurs étapes. Pour asseoir sa crédibilité auprès des auditeurs ciblés, on fait croire que la radio est dirigée par des loyalistes de Saddam Hussein. Dans un premier temps, la ligne éditoriale de la radio va dans leur sens et manifeste un support ardent envers le chef d'état irakien. Puis une fois que le public a mordu à l'hameçon, qu'il est capté et confiant, le ton change progressivement : il devient de plus en plus critique vis-à-vis du chef d'état irakien. Il s'agit d'amener petit à petit les auditeurs à changer d'avis. Evidemment, cette opération demande plus de temps. Elle repose sur le fait que les auditeurs ciblés doivent croire que les émissions proviennent de sources qui ne sont pas celles qu'ils pensent être. Et que la mystification ne soit pas découverte.

Les "acteurs" agissent donc de manière masquée. Dans le cas contraire, les choses peuvent virer à l'effet inverse. Cette opération utilise le *spoofing* pour agir dans le champ de la représentation de la situation par les cibles. En altérant et faussant de cette manière leur processus de décision, il s'agit de les induire au comportement souhaité : lâcher leur chef.

Autre utilisation du *spoofing*, ou usurpation de l'émetteur d'information ou de sa source : pendant la guerre de Kippour en 1973, les divisions égyptiennes s'engagent dans le désert du Sinaï. Dans l'étendue de sable et de dunes, elles reçoivent les instructions sur la direction de leur déplacement. Mais ce que les soldats ignorent, c'est que les instructions radio de leur commandement ont été confisquées en cours de route et remplacées par des instructions israéliennes. Les soldats égyptiens reçoivent alors dans leur propre langue des directives... qui les font tourner inlassablement en rond dans les sables, jusqu'à leur capture. Cette opération de désorientation des troupes adverses utilise aussi le *spoofing*, ainsi que la pénétration des communications originales des véritables commandants.

LA MÊME MÉTHODE DE SPOOFING À LA PORTÉE DES ACTEURS NON MILITAIRES

Le *spoofing* est une tactique dont les responsables sécurité informatique se méfient. Comme dans les cas de Radio Tikrit et des soldats désorientés dans le désert, le *spoofing* trompe sur la source véritable de l'information et son but est de conduire l'autre à faire ce qu'on attend de lui. Cette méthode est fréquemment utilisée par les auteurs de programmes malveillants pour endormir la méfiance des utilisateurs. Par exemple, en les trompant sur l'expéditeur véritable d'emails porteurs de vers ou de chevaux de Troie. Quel email vérolé ne se fait pas passer pour un courrier du support technique de Microsoft, une information d'une banque, un message de l'administrateur, ou d'un de ses amis ou correspondants habituels ? Un grand nombre de vers collectent les adresses emails dans les machines contaminées pour s'auto-envoyer en usurpant ces adresses expéditeurs. Bien que ce stratagème ne soit pas une nouveauté, les utilisateurs continuent à se faire piéger parce qu'ils croient que l'email provient d'une personne qu'ils connaissent. Et c'est exactement l'illusion voulue par le diffuseur de vers.

Le ver Swen usurpe une adresse email de Microsoft, des variantes de Mimail font croire à des emails en provenance de services bancaires. Plusieurs moyens sont conjugués pour renforcer la crédibilité en plus de la tromperie sur l'adresse d'expéditeur des emails, comme aller jusqu'à revêtir l'apparence de la société dont l'identité est usurpée. Swen cumule ainsi plusieurs signes trompeurs : non seulement il prétend provenir de Microsoft, mais en plus sa présentation visuelle renforce l'illusion : graphisme, couleurs, mise en page, logo sont empruntés au site Web de l'éditeur original, le nom de sa pièce jointe infectée se fait passer pour un *patch* correctif de sécurité. Si le faux patch est exécuté, l'illusion se prolonge puisque l'exécutable malveillant imite un vrai programme d'installation : il affiche des écrans d'installation, des messages d'erreur ressemblant à ceux d'un correctif. De nombreux emails frauduleux utilisés dans des opérations de "phishing" (pêche aux données personnelles) présentent une interface identique à celle de sociétés connues, banques ou un site d'enchères en ligne, ou faux liens vers des sites Web au look très crédible pour induire l'utilisateur à donner ses coordonnées personnelles et celles de sa carte bancaire [9].



GUERRE DE L'IN

GUERRE DE L'INFORMATION

A la longue, si elles se répètent trop souvent, de telles usurpations pourraient provoquer la méfiance à l'encontre de vraies sources d'information et troubler l'activité d'entreprises. Quand telle société enverra un vrai message, si son identité est trop fréquemment usurpée, les interlocuteurs pourraient finir par ne plus le considérer. Confusions en tous genres et confiance entamée. Pour l'instant, il est vrai, nous n'en sommes pas tout à fait encore là, mais...le risque n'est pas à écarter.

SPOOFING UTILISÉ DANS LES ATTAQUES INFORMATIONNELLES

La tromperie sur la qualité réelle de l'émetteur d'un message est également utilisée dans des opérations de déstabilisation d'une entreprise. Dans des attaques informationnelles, le *spoofing* sert à faire croire qu'on est quelqu'un d'autre pour mieux influencer l'auditoire cible. Faux internautes lambdas, en réalité agents de la concurrence, qui s'en prennent à des sociétés dans les forums de discussion et distillent des rumeurs. Faux consommateurs, faux clients qui sèment le doute sur la fiabilité ou la qualité de produits ou services de telle entreprise pour favoriser à la place ceux d'un concurrent. L'attaque informationnelle consiste à entamer la réputation de personnes clé ou de produits pour affaiblir, parfois même ruiner une entreprise. Nul besoin de troupes nombreuses pour ce type d'action : un seul individu suffit à miner la confiance. Une opération de déstabilisation peut même être lancée par une personne isolée pour des motifs strictement personnels tels que la rancœur ou la vengeance, tout en ayant le même impact que si elle avait été conduite par un concurrent organisé pour des raisons de profit. Le *spoofing* servira encore à induire en erreur sur l'origine de l'attaque, pour mieux crédibiliser l'action et tenter d'empêcher de remonter à l'auteur des faits.

C'est exactement ce qui est arrivé à un groupe pharmaceutique multinational : plus de 700 000 messages sont envoyés par email à des dirigeants et collaborateurs de l'entreprise, en France et dans plusieurs pays du monde. Il s'agit de messages dénigrants, qui affirment que ses produits sont défectueux ou mortels et ses dirigeants corrompus, que l'entreprise est mise en cause pour malfaçon de ses produits. Le personnel de l'entreprise n'est pas seul à être bombardé d'emails : des destinataires extérieurs au groupe, entreprises partenaires ou concurrentes, des analystes financiers, des journalistes en reçoivent aussi. L'adresse d'expéditeur est fausse, soit inventée, soit empruntée à des dirigeants réels du groupe ciblé. Une attaque informationnelle d'une telle envergure, d'une teneur si grave, en plein contexte de mouvements de fusion-acquisition dans le secteur pharmaceutique aurait pu être lancée par un concurrent. Et non. L'auteur de cette opération s'est révélé être un ex-salarié du groupe licencié trois ans avant. A lui seul, et grâce aux moyens informatiques, l'attaquant a un énorme potentiel d'impact [9].

MÉTHODE DE DECEPTION : TROMPER SUR L'EXISTENCE DE L'ACTION OU SUR SES PROPRES INTENTIONS

Dans les exemples suivants, ce n'est pas le *spoofing* qui est utilisé mais une autre méthode de "deception" : la tromperie sur l'existence ou les intentions réelles de son action. Il s'agit encore d'influencer l'adversaire sur sa représentation de la situation afin qu'il ne se doute

de rien et qu'il comprenne autre chose. La "deception" altère le processus d'analyse des données en créant du "bruit", en pourvoyant des données dans le but qu'elles soient faussement interprétées.

Dans le domaine militaire, l'une des opérations de "deception" les plus célèbres est l'opération Fortitude. Elle a été menée pendant la Seconde Guerre mondiale pour induire en erreur les Allemands sur la date et le lieu du débarquement des troupes alliées sur les côtes françaises. L'opération de guerre de l'information a consisté entre autres à servir de fausses informations à des agents, procurer un bon volume de messages radios codés destinés tout exprès aux interceptions allemandes, créer des mouvements réels de troupes vers le Nord. Bref, il s'agissait de renforcer la certitude des Allemands sur la nécessité de fortifier leurs défenses dans la zone de Calais, pour mieux lancer le débarquement ailleurs, sur les plages de Normandie [10].

Tromper sur ses intentions pour surprendre l'adversaire est une ruse de guerre vieille comme le monde qui passe par la guerre de l'information. Et c'est une ruse qui n'est pas prête d'être abandonnée par les stratèges [11]. C'est également une technique connue dans le domaine de la sécurité informatique pour noyer le poisson ou obtenir des informations [12].

En position défensive, mais aussi offensive, un système de déception peut consister à présenter un pot de miel (*honeypot*). C'est-à-dire attirer l'attaquant dans un système leurre où il va perdre son temps à s'aventurer, tout en fournissant des renseignements aux administrateurs du dispositif. De fausses données peuvent être placées en appât, tout en tenant à l'abri ailleurs les données sensibles que l'assaillant peut convoiter. Une gamme de systèmes de "deception" gradués peut même trier et observer les types d'attaquants et renseigner leurs profils et modes opératoires.

DECEPTION PAR IMPRÉGNATION

Dans une attaque surprise, le succès préalable d'une opération de guerre de l'information est déterminant. Cette guerre-là n'est pas déclarée. Elle doit envoyer des signes destinés à noyer le poisson, tout en masquant son existence. La technique utilisée pour fausser le processus d'analyse peut être celle de l'imprégnation : distiller des données de manière répétée et convergente, qui vont orienter imperceptiblement le schéma de représentation de la situation par l'adversaire ou la cible.

En 1973, pour préparer leur offensive surprise de Kippour, les Egyptiens et les Syriens ont mis en place un plan de tromperie ("deception") destiné à faire croire qu'ils n'avaient ni les capacités ni l'intention de s'engager dans une guerre contre Israël [13]. Pour cela, plus d'un an avant le déclenchement de l'attaque surprise, ils ont recouru non seulement à toute une série d'événements politiques et diplomatiques destinée à servir cette croyance, mais encore, ils ont organisé des manœuvres militaires répétées pour fausser les analyses des services de renseignements israéliens en les "endormant". L'opération se sert aussi de la presse : plusieurs mois avant l'assaut, les journaux publient des articles sur les difficultés dont souffrent les armées arabes. La caractéristique de ce type d'opération préalable à l'attaque surprise est l'imprégnation de l'adversaire par différents moyens concourant à produire ce même effet de représentation faussée de la situation. Ici, donner à déduire : "nous n'avons pas l'intention de vous attaquer et d'ailleurs nos armées sont en état de faiblesse".



Or, quelques temps plus tard, en octobre 1973, l'offensive avec avions, troupes et chars est lancée par surprise et devient brutalement "visible" et interprétable comme telle.

A la différence, la guerre de l'information peut rester "invisible", non détectée pendant un laps de temps plus ou moins long. Sa séquence de temps va au-delà de celui de la guerre "visible". De nombreuses opérations de déstabilisation d'entreprises ou de décrédibilisation de *personnes clés* utilisent la technique d'imprégnation. Des opérations via et contre les systèmes d'informations procèdent par étapes échelonnées dans le temps. Il est donc extrêmement important de savoir en déceler et décoder les signaux, aussi faibles soient-ils. Et le plus tôt possible [14, 16].

OUTILS ET MÉTHODES À LA PORTÉE DE TOUS : BROUILLAGE DE LA SITUATION

Il est évident que les systèmes informatiques et électroniques peuvent servir dans de telles opérations. La détection de l'attaque est un des points fondamentaux en matière de sécurité informatique. Le caractère diffus et parfois ignoré des attaques augmente les risques et rend très complexe la détermination de la situation. Lorsqu'ils sont détectés, les signaux savent-ils être corrélés ? Les méthodes et outils de guerre de l'information sont à la portée d'acteurs supplémentaires. Plus seulement les militaires, mais aussi des acteurs économiques, des groupes d'intérêts, des individus isolés. Les attaques sont devenues multidirectionnelles et dès lors, comment faire la distinction ?

Or, les moyens informatiques sont particulièrement propices à produire de l'illusion, aussi sont-ils efficaces pour conduire des opérations offensives de "déception". Une opération de "deception" assaillante peut consister à diffuser massivement des vers informatiques ou des virus, dont le nombre peut faire croire à une opération sans cible précise, alors qu'en réalité le nombre et le "bruit" masquent une opération réellement ciblée. Comment discerner qui l'a lancée ? Diffuseurs de virus isolés ou opération menée par un service d'Etat ?

Dans le domaine de la sécurité informatique, la guerre de l'information permet également de dissimuler une action offensive en simple panne ou incident : l'attaquant peut porter ses coups sans que ceux-ci soient même perçus comme tels. Ou identifiés comme provenant de lui. Ainsi, la cible ne saura pas nécessairement si elle est assiégée, qui l'assiège ni à l'encontre de qui riposter. Les techniques d'évasion dans les réseaux informatiques posent à ce propos un vrai problème.

Cerner immédiatement la véritable nature d'un incident peut s'avérer un véritable casse-tête.

Revenons sur le cas des pannes d'électricité qui ont touché les Etats-Unis et le Canada en août 2003. Une coupure de courant touche plusieurs états. A New York, Détroit, Toronto, Ottawa, des millions de personnes voient les feux de signalisation s'arrêter, les métros se figer, les ascenseurs se coincer, les télévisions et appareils stoppés, et la nuit noire s'abat sur des villes habituées à un déluge de lumières.

Virus ? Panne technique ? Exercice sur infrastructure sensible ? Sabotage ? Toutes ces questions se sont posées et chacune de ces hypothèses envisagées [15].

Un problème important se pose donc de plus en plus : une situation doit être analysée et son tableau clarifié, sinon le processus de décision est aveuglé. Or, la guerre de l'information peut être conduite par de tout petits effectifs, de manière diffuse. Les outils de guerre de l'information sont désormais à la portée de tous. La compréhension de la situation se complique, en raison de ce brouillage. Quand il s'agit d'actions conduites via ou contre les systèmes d'information si accessibles à une multitude d'acteurs et propices à l'illusion, comment déceler les prémisses d'une attaque, comment détecter les signaux et comment les corrélés ? Comment faire la distinction entre les actions causées par un individu qui agit pour perturber ou détruire sans objectif particulier autre que de se faire plaisir ou de se venger, et les actions menées par des professionnels pour des raisons économiques ou encore des professionnels servant des objectifs guerriers ?

Cela demande d'affiner les techniques d'intelligence, ce dont nous aurons l'occasion de reparler.

Danielle Kaminsky
dkam2@wanadoo.fr

RÉFÉRENCES

- [1] *Infosphère et intelligence stratégique, les nouveaux défis*, Economica et Institut des hautes études de défense nationale, Paris, 2002.
- [2] *La guerre du sens, Pourquoi et comment agir dans les champs psychologiques*, Loup Francart, éditions Economica, 2000.
- [3] *La guerre cognitive, à la recherche de la suprématie stratégique*, Christian Harbulot, Nicolas Moinet, Didier Lucas, communication, septembre 2002, http://www.strategic-road.com/intellig/infostrategie/pub/guerre_cognitive_superiorite_strategique_txt.htm
- [4] Dépendance accrue : https://www.clusif.asso.fr/fr/production/ouvrages/pdf/Presentation_DRIRE-200311.pdf
- [5] *L'infoguerre, Stratégies de contre-intelligence économique pour les entreprises*, Philippe Guichardaz, Pascal Lointier et Philippe Rosé, éditions Dunod, Paris, 1999.
- [6] [8] PSYOPS Guerre du Golfe : <http://www.iwar.org.uk/psyops/resources/iraq/mind-games.htm>
- [7] *Guerre cyber : pour en finir avec les préjugés*, Danielle Kaminsky, Pirates Mag N°6, 2000.
- [9] Virus, Phishing : <https://www.clusif.asso.fr/fr/production/ouvrages/pdf/PanoCrim2k3-fr.pdf>
- [10] Opération Fortitude : <http://www.militaryhistoryonline.com/wwii/dday/overlord.aspx>
- [11] Deception : <http://www.college.interarmees.defense.gouv.fr/03pub/tribune/articles29/desaintsalvy.pdf>
- [12] *The Art of Deception, controlling the human element of security*, Kevin D. Mitnick, éditions John Wiley & Sons, 2002
- [13] Guerre Kippour, analyse de Pierre Razoux : http://www.stratisc.org/strat/strat_073_dRazouxdoc.html
- [14] Attaques informationnelles : <http://www.infoguerre.com/article.php?sid=648>
- [15] Déni de service sur les infrastructures : www.hsc.fr/ressources/presentations/celar2003/deni-services-hsc-cyberdefense.pdf
- [16] *La guerre de l'information*, MISC 3 : <http://www.miscmag.com/articles/index.php3?page=404>



LA FAILLE VENAIT DU LOGICIEL !

4 Recherche de vulnérabilités par désassemblage

2 Dix dangers qui guettent le programmeur de cryptographie

3 Méthodes d'évaluation de sécurité : TCSEC, ITSEC et CC

5 Sécurité logicielle : étude de cas

1

Sécurité des logiciels



La sécurité des logiciels est un point central de la sécurité informatique. Toute opération informatique est effectuée par un logiciel, et toute infraction informatique est effectuée via un logiciel. Toute architecture de sécurité repose en grande partie sur la confiance en des logiciels. Pare-feu, routeurs, VLAN, serveurs Web, mail, tout cela n'est à la base qu'un logiciel. Nous partirons donc à la recherche de notre plus grand ennemi : la faille de sécurité. L'ayant jaugé, nous verrons comment tenter de le vaincre. Mais cela ne sera pas aussi facile, et nous verrons pourquoi nous, êtres humains, sommes à l'origine de ce mal. Résignés, nous finirons par étudier comment vivre avec.

INTRODUCTION

Cet article pourrait très bien être sorti du contexte de la sécurité informatique pour être replongé dans celui du génie logiciel. Cela est dû au fait que tout bogue est un trou de sécurité en puissance. Autant donc parler de tous les bogues plutôt que de se concentrer sur ceux connus pour leurs implications en sécurité.

L'utilitaire `whois` était autrefois victime d'un débordement de tampon concernant un de ses paramètres en ligne de commande. Cela était considéré comme un bogue bénin (appréciez l'oxymore). En effet, `whois` n'est qu'un bête utilitaire non SUID, sans privilège particulier. Autant se coder soi-même son programme vulnérable. Et puis un jour, l'on se rendit compte que toutes les interfaces Web qui avaient fleuri deci-delà en proposant d'aller questionner pour vous les bases Whois se reposaient sur ce bête utilitaire sans privilège particulier. Ainsi donc, ce "bogue bénin" avait ouvert une brèche béante pour prendre le contrôle de toutes ces machines.

Aussi partirons-nous en chasse à tous les bogues, et non uniquement à ceux qui ont une implication visible pour la sécurité.

CONNAÎTRE SON ENNEMI : QUELQUES GRANDES CLASSES DE TROUS DE SÉCURITÉ (ET PAS L'INVERSE)

Afin d'avoir une idée précise sur les maux qui guettent nos logiciels et mettent en péril la sécurité de nos systèmes d'information, nous présentons quelques classes de failles ainsi que des exemples réels.

Il y a bien entendu beaucoup d'autres classes, que nous n'avons pas la place de détailler dans ces pages, mais celles que nous allons voir suffiront à illustrer notre propos.

LES PROBLÈMES D'INTERFACE

Une interface est le lieu de rencontre de deux mondes. Lorsque ces deux mondes souhaitent communiquer, les informations transitent par cette interface. Afin qu'ils se comprennent, ils doivent se mettre d'accord sur le format des données échangées. Cet accord est appelé contrat d'interface. Les problèmes surgissent quand l'un des deux mondes ne respecte pas ce contrat ou ne vérifie pas qu'il a été respecté. Par exemple, lorsqu'un programme



reçoit des données de l'extérieur, il doit vérifier que le contrat est rempli. Mais le problème se pose également dans l'autre sens : lorsqu'un sous-programme en appelle un autre, et que le deuxième est conçu pour faire confiance au premier, le premier doit s'assurer qu'il respectera l'interface. Enfin, il n'est parfois pas possible de vérifier que le contrat d'interface est rempli, par exemple quand il s'agit d'interpréter une chaîne : si je vous dis "%s" pensez-vous que c'est une chaîne de format, ou est-ce simplement une chaîne de caractères ?

Voyons donc quelques exemples de failles de sécurité engendrées par un problème d'interface.

Débordement de tampon

Dans le cas du débordement de tampon, l'interface précise qu'un champ est de taille limitée. Cela se retrouve sous forme d'un *buffer* de cette même taille dans le programme appelé. Le problème est que ce dernier ne vérifie pas que l'appelant respecte le contrat d'interface, et si le champ fourni est trop long, l'appelé écrit un peu à côté du *buffer*, d'où le bogue. Une fois que l'on a le bogue, l'exploitation n'est plus qu'une formalité [aleph1].

Par exemple, le pare-feu personnel Kerio, dans les versions 2.1.4 et précédentes, permettait à l'administrateur de se connecter à distance sur le pare-feu. Il avait prévu un tampon de 64 octets pour recevoir la clef de l'administrateur et ainsi l'authentifier. Cependant, il ne vérifiait pas que la clef fournie était de la taille attendue. Il en résulta une faille qui permettait de prendre la main à distance sur les machines équipées de ce pare-feu, censé les protéger [kerio].

Débordement d'entiers et problèmes de signes

Nous avons ici un cas beaucoup plus vicieux. Avant de le décrire, il faut tout d'abord savoir que les nombres, en informatique, sont bornés, qu'on les représente sur 8, 16, 32 ou plus de bits. Les bornes varient également, selon qu'on décide que le nombre est signé ou non. Par exemple, sur 8 bits, un nombre est compris entre 0 et 255 s'il est non signé et entre -128 et 127 s'il l'est. Cependant, en binaire, il n'y a que des zéros et des uns, pas de signe plus ou moins. Aussi utilise-t-on un bit pour indiquer le signe (à peu de chose près). Ainsi 11111111 en binaire peut aussi bien signifier 255 si on considère qu'il est non signé, que -1 si on l'interprète comme un nombre signé.

Vous avez sans doute déjà compris le problème d'interface : dans un monde on considère qu'un nombre est signé, et pas dans l'autre (ou l'inverse, il n'y a pas qu'une façon de faire l'erreur). Par exemple, on peut vouloir vérifier que la taille annoncée d'un *buffer* est bien inférieure à 64, taille du *buffer* de destination. Or, on utilise une interprétation signée de cette taille. La taille annoncée est de -1. -1 est bien inférieur à 64, donc le test ne détecte pas de problème. On passe ensuite la main à une fonction de copie qui, elle, interprète le -1 en arithmétique non signée, c'est-à-dire qu'elle comprend 255. Pas la peine de faire un dessin.

Dans le cas du débordement d'entier, c'est le fait que le programmeur a oublié qu'il travaille en arithmétique modulo.

Autrement dit, $255+1=0$ (sur 8 bits). Si on prend deux *buffers* A et B de taille maximale 128 octets que l'on veut concaténer pour mettre dans un *buffer* de 128 octets, on peut être tenté de vérifier que $\text{longueur}(A)+\text{longueur}(B) \leq 128$. Ce calcul semble juste, mais si l'on utilise des nombres sur 8 bits, dans le cas où $\text{longueur}(A)=\text{longueur}(B)=128$, on se retrouve avec $128+128=0$ et $0 \leq 128$.

Par exemple, les versions 1.8, 1.9 et pré-2.0 de Snort étaient victimes d'un débordement d'entier dans le code de réassemblage des flux TCP [snort]. La condition $(\text{spd} \rightarrow \text{seq_num} + \text{spd} \rightarrow \text{payload_size}) \leq \text{s} \rightarrow \text{last_ack}$, censée détecter qu'un paquet dépasserait du *buffer*, est contournable quand le premier membre de la comparaison dépasse la capacité d'un nombre sur 32 bits, et redevient tout petit. C'est donc un équipement de sécurité central qui donne accès à une machine au sein du système d'information, machine ayant souvent accès au LAN d'administration ou de monitoring. À noter que ne pas mettre d'IP sur l'interface qui écoute le réseau n'empêchera pas un pirate de rester en communication avec son *shellcode*, pour peu qu'il ait prévu le coup. Il arrive de plus que, sur certains *switches*, les ports en mode *monitor* permettent également d'envoyer des paquets. Coupez plutôt le brin Tx de votre câble Ethernet, et on n'en parle plus.

Bogues de formats

Le bogue de format consiste à passer une chaîne quelconque à une fonction qui va l'interpréter comme une chaîne de format. Une chaîne de format est une chaîne de caractères qui donne le gabarit d'un affichage, et dont certaines parties vont être remplacées par des vraies valeurs passées en paramètres ; par exemple, `printf("age:%i, sexe:%s, ville:%s", age, sexe, ville)`. Si vous voulez afficher la chaîne quelconque "%s" en la passant directement à `printf()`, ça ne va pas marcher. En effet, la fonction va chercher à remplacer les commandes qu'elle trouve (%s, %i, etc.) par ce qui se trouve à l'endroit où auraient dû figurer les paramètres. C'est un bogue. Quand une personne extérieure peut fournir cette chaîne, c'est un trou de sécurité. Les commandes classiques permettent en général d'avoir dans la chaîne résultante n'importe quelle partie de la mémoire à partir d'une certaine adresse (la *stack frame* de la fonction). La commande %n permet d'écrire dans la mémoire.

Par exemple, les produits Checkpoint Firewall-1 ont été récemment touchés par plusieurs bogues de format dans leurs *proxies* HTTP [fw1], qui pouvaient amener à prendre directement le contrôle du pare-feu en super-utilisateur. C'est quand même dommage pour ce qui est, en général, la clef de voûte de la sécurité du système d'information.

Injectons

« - Il m'appelle souvent.
- Souvent ? Quel drôle de nom ! »

Même si la blague ne vous fait pas rire, elle illustre bien le problème de l'interprétation, et des différents niveaux sémantiques qui peuvent exister dans une expression. Nous sommes capables, grâce au contexte et à notre expérience, de faire la différence



avec « Il m'appelle "Jean-Kevin" ». Les machines n'ont pas accès à la compréhension sémantique, aussi faut-il leur préciser chaque changement sémantique. Vous noterez par exemple que j'ai mis des guillemets dans mon expression, pour accentuer le changement, même si nous n'en avons nullement besoin.

Dans le cas de l'injection, nous avons un premier niveau sémantique, par exemple `SELECT * FROM utilisateurs WHERE login='%s' and passwd='%s'`. Dans cette expression (qui est une chaîne de format), les *quotes* sont utilisées pour signifier le changement de niveau sémantique. Les *login* et mot de passe entrés par l'utilisateur seront donc placés entre les quotes à la place des %s et seront interprétés à un niveau sémantique inférieur. L'attaque consiste à essayer de placer dans la chaîne résultante les caractères spéciaux utilisés pour changer le niveau sémantique. Dans notre exemple, une quote dans le login permet de remonter au niveau SQL : `SELECT * FROM utilisateurs WHERE login='toto' OR 1=1# and passwd='aybattu'`. Ainsi, le `OR 1=1` est interprété comme du SQL et non comme la chaîne de caractères quelconque qu'elle était à l'origine.

Cette attaque n'est pas valable uniquement pour le SQL. On peut par exemple la retrouver en XML ou en HTML. En HTML, votre nom : %s
, si l'on n'y prête pas attention, va poser problème. Le changement de niveau sémantique se fait grâce aux caractères < et >. S'ils ne sont pas correctement filtrés, et si l'utilisateur peut choisir son nom, on se retrouve avec une faille de Cross Site Scripting (XSS). En XML, il existe une construction qui permet de ne pas avoir à trop utiliser de caractères d'échappement : <![CDATA[%s]]>. Tout ce qui est entre <![CDATA[et]]> sera ignoré par le parseur, peut-on lire. Et si on remplaçait %s par xx]]><MONTANT>30</MONTANT><![CDATA[xx ? Il semblerait que l'on vienne d'injecter un élément XML.

Par exemple, `ifsPROTECT`, un système "de protection de mots de passe et d'authentification haute performance", avait une interface d'administration en HTTP, et ne nettoyait pas certaines variables, ce qui permettait d'injecter du SQL. En particulier, il était possible d'appeler une procédure stockée pour exécuter des commandes directement sur la machine.

LES CONTOURNEMENTS

Typiquement, c'est le cas de la personne qui a pris toutes les précautions les plus paranoïaques concernant sa porte d'entrée et qui oublie la fenêtre. Cette image est bien sûr exagérée (quoiqu'il faille toujours se méfier des fenêtres) ; s'il est aisé, dans une maison, de ne pas oublier de point d'entrée, ce n'est plus aussi évident lorsqu'on s'attaque à des choses aussi complexes qu'un système d'exploitation avec ses applications.

Par exemple, sous Unix, connaissez-vous exactement le comportement du chargeur dynamique en temps normal (bibliothèques utilisées, variables d'environnement, etc.) ? Et lorsque le binaire exécuté est SUID ? Et lorsqu'il est "ptracé" ? Et si c'est un script ? Et sur l'Unix du voisin ?

Un exemple assez frappant est l'utilisation de l'option de montage `noexec`. L'idée est de monter toutes les partitions en lecture seule. Mais le système d'exploitation a besoin de laisser des zones

d'écriture temporaires, où tout le monde peut écrire. Donc le `/tmp` est monté en lecture/écriture. D'aucuns prétendent qu'il suffit alors de rajouter l'option `noexec`, qui, rappelons-le, va interdire l'exécution de binaires situés sur cette partition, pour se protéger de l'exécution de binaires étrangers. En effet, le seul endroit où on peut les écrire est `/tmp`, et c'est également un endroit d'où on ne peut les exécuter.

Nous sommes donc en sécurité. En êtes-vous sûr ? N'avez-vous donc point oublié une fenêtre ?

Savez-vous que le chargeur dynamique, présent sur le système sous forme de bibliothèque partagée (`/lib/ld-2.3.2.so` chez moi), est également un exécutable (comme tous les `.so`), et qu'il peut prendre un binaire ELF en argument, puis effectuer son boulot comme s'il avait été appelé par ce binaire pour l'exécuter.

```
$ cp /usr/bin/id /tmp
$ /tmp/id
uid=1000(pbi) gid=1000(pbi) groups=1000(pbi)
$ sudo mount /tmp -o remount,noexec
$ /tmp/id bash:
/tmp/id: Permission denied
$ /lib/ld-2.3.2.so /tmp/id
uid=1000(pbi) gid=1000(pbi) groups=1000(pbi)
```

Et pour ceux qui auraient cru trouver la parade en mettant hors d'état de nuire le chargeur dynamique, qu'ils jettent un œil au Userland `execve [gruegq]`.

Un autre cas réel est celui beaucoup plus répandu de l'utilisation directe d'une ressource sans passer par la phase d'authentification. Avec TrendMicro InterScan Viruswall, il était par exemple possible d'accéder à la console d'administration sans passer par la phase d'authentification en utilisant directement l'URL des CGI, par exemple `http://victime.com/intercan/cgi-bin/smtpscan.dll`. On pouvait ainsi, par exemple, désactiver l'antivirus pour pouvoir semer la pagaille.

LES CONDITIONS DE CONCURRENCE (RACE CONDITIONS)

Les conditions de concurrence apparaissent dès lors que plusieurs actions qui travaillent sur la même ressource puissent se dérouler en même temps (même si l'on n'y a qu'un processeur ; ce qui compte c'est que le début de l'une soit entre le début et la fin de l'autre).

Si tout bon programmeur sait maintenant faire attention, lorsqu'il développe des tâches qui vont entrer en concurrence, à bien gérer les ressources partagées (faire attention ne suffit pas, mais c'est un bon début), il en est autrement quand il faut également prévoir la concurrence avec des tâches extérieures qui pourraient être développées par quelque pirate.

Par exemple, dans le pare-feu IPFilter, la règle qui permettait de bloquer une connexion TCP en envoyant un paquet RST souffrait d'une condition de concurrence `[ipfilter]`. En effet, afin d'émettre le paquet RST, un état était rajouté à la table d'états. Un attaquant pouvait utiliser cet état avant qu'il n'expire pour faire passer des paquets SYN.



COMBATTRE SON ENNEMI

Maintenant que nous avons pu jauger notre ennemi, nous pouvons le combattre. Nous connaissons les pièges à éviter. Nous pouvons le traquer dans notre code source.

GÉNIE LOGICIEL

La discipline du génie logiciel s'attache à trouver des méthodes de développement qui permettent de spécifier et de découper de manière exhaustive et complète un projet logiciel. Le but est de ne rien oublier en route, et de mettre en place une meilleure communication entre les différentes personnes du projet en spécifiant les communications. Les méthodes d'écriture de spécifications transforment en quelque sorte l'homme en machine en lui faisant exécuter une suite d'instructions simples et en lui faisant oublier par exemple ses capacités à imaginer des raccourcis, et éviter ainsi les écueils que nous avons vus plus haut.

Parmi les éléments les plus remarquables apportés par le génie logiciel, on notera les *design patterns*. Ce sont en quelque sorte des briques algorithmiques de base dont le but est de répondre à des problèmes récurrents en programmation. Ainsi, un programmeur moins expérimenté, lorsqu'il sera confronté à un problème que des milliers de programmeurs ont déjà résolu avant lui, pourra réutiliser cette brique certifiée comme la meilleure solution au problème par des développeurs plus expérimentés.

Mais ces méthodes ne sont pas infaillibles. L'élément humain est toujours présent. On pourra voir par exemple que le *double check locking pattern* était en fait victime d'une condition de concurrence sur certaines JVM [dclp].

AUDITS

Étant entendu que, malgré les conditions de développement les plus drastiques, des erreurs peuvent quand même passer, on peut essayer d'auditer le code pour trouver les failles restantes. Bien évidemment, on ne peut jamais être sûr de n'avoir rien oublié, d'autant que vu l'énormité de la tâche, on part plutôt à la recherche d'erreurs classiques ou de failles connues. Un jour, quelqu'un découvrit le potentiel malfaisant du débordement de tampon. Le bogue étant devenu une classe de failles de sécurité, la chasse aux mauvaises pratiques de codage et autres `strcpy()` fut lancée. Les programmes audités sortaient donc (à peu près) exempts de cette classe de faille. Grâce à leur estampille "auditée", ils étaient donc réputés plus sûrs que d'autres. Mais on découvrit plus tard une autre classe, les bogues de format. En fait, l'estampille aurait dû être "à peu près garanti sans débordements de tampon".

Ce qui va de pair avec les audits qu'on paye cher, ce sont les certifications. Il faut garder à l'esprit qu'un audit ne peut garantir l'absence de bogues, et qu'une certification est délivrée après un audit, et pour des conditions d'utilisation particulières. Par exemple, il ne faut pas dire Windows NT est certifié C2, mais Windows NT 3.51sp3 sans réseau ni modem est certifié C2. Un peu comme si votre médecin vous annonçait votre immunité contre les accidents de voiture, à condition que vous gardiez le lit.

AUDITS AUTOMATIQUES

Il existe pas mal de petits programmes pour repérer les points sensibles, l'utilisation de fonctions risquées ou encore des comportements dangereux dans du code source. Ils sont souvent bien modestes mais, s'ils ne sont pas une garantie de code sans faille, ils permettent souvent d'éviter les failles bien connues. Parmi les plus connus, on pourra citer `its4`, `lclint`, `pychecker`, `flawfinder`, `rats`, `cqual`, `pscan`. Le mieux est que vous consultiez ces méta-références : [meta1] et [meta2].

LA SÉCURITÉ EST DANS LA SIMPLICITÉ

La meilleure façon de ne pas faire d'erreur est de faire des choses simples (en anglais, KISS : *Keep It Simple, Stupid*). Il vaut mieux résoudre 50 petits problèmes qu'un seul gros problème. Essayez de calculer 21×13 directement. Maintenant, faites 2×13 , puis 1×13 puis $260 + 13$.

Prenons par exemple le problème de réaliser un serveur de nom. Ce dernier doit remplir plusieurs tâches, comme servir les zones pour lesquelles il est autoritaire, permettre les transferts de zones pour ses esclaves, télécharger celles pour lesquelles il est lui-même esclave, traiter les requêtes récursives des clients... bref, un gros problème. Une solution compliquée est BIND. D'un autre côté, `Djbdns` décompose le problème en plusieurs petits et les résout séparément. Par exemple, il y a un programme pour chaque fonctionnalité. Vous trouverez plus de détails dans l'article suivant de ce dossier [etudecas].

Un autre exemple est la mise au point des règles d'un pare-feu à plusieurs pattes. Il est possible, avec `Netfilter`, de créer des chaînes de règles en plus de celles préexistantes et d'aiguiller leur parcours en les agençant à la façon d'un arbre. Ainsi, on peut par exemple décomposer le problème de filtrer le trafic entre 3 pattes A, B et C en 6 problèmes plus petits : filtrer le trafic de A vers B, de B vers A, de A vers C, de C vers A, de B vers C et de C vers B.

MÉTHODES FORMELLES, LANGAGES PROUVABLES

Lorsque des vies humaines ou des dollars sont en jeu, l'erreur est interdite. Pour cela, on peut utiliser des méthodes extrêmement contraignantes et des langages spécialisés comme le B ou une sous-partie de l'ADA qui sont dits prouvables. On ne peut pas tout faire avec ces langages, mais on peut mathématiquement prouver qu'ils font exactement ce qu'on souhaite qu'ils fassent... enfin, s'il n'y a pas d'erreur dans le compilateur et si on ne se trompe pas dans la démonstration.

OPEN SOURCE OU CLOSED SOURCE

Si vous aviez une voiture, seriez-vous plus tranquille de la laisser garée si elle ne ferme pas à clef mais, ayant les vitres teintées, personne ne puisse voir qu'il n'y a pas de loquets, ou si elle a les vitres transparentes, mais on puisse réellement fermer les portes à clef ?



Tout dispositif de sécurité qui repose sur le secret de sa composition (clef sous le paillason, URL cachée, IP secrète, etc.) est voué à l'échec. Cela s'illustre particulièrement bien en cryptographie : lorsque la connaissance de l'algorithme le fragilise, c'est qu'il était lui-même sa propre clef. Lorsqu'une clef est corrompue, on la change. Donc, soit on passe son temps à changer d'algorithme, soit on utilise des algorithmes dont la connaissance n'apporte rien. Et l'histoire [hdes] montre que les cryptographes ont raison et que le commercial de la boîte d'en face qui vend des produits révolutionnaires à base de cryptographie maison a tort.

Certaines personnes sont capables de lire un binaire comme nous lisons son code source. Le *closed source* ne freine pas énormément les personnes qui recherchent une faille car elles n'en ont besoin que d'une. Les personnes qui tentent de sécuriser une application, en revanche, doivent en trouver le maximum. Donc, concrètement, le *closed source* ne ralentit que ceux qui recherchent la sécurité.

Un exemple est le débordement de buffer dans les serveurs TFTP des IOS 11.x. Cisco a longuement soutenu que le fait que personne n'avait les sources d'IOS, que tout pouvait "crasher" au moindre bit déplacé, etc. rendait toute exploitation impossible. Un peu de *reverse engineering*, un peu de malice, et hop, un exploit [ciscoheap].

Un autre exemple est la découverte d'une porte dérobée oubliée dans Interbase [interbase]. Celle-ci a survécu sans être remarquée pendant 6 ans. Puis Borland a ouvert les sources de son produit. Il n'a pas fallu plus de 6 mois pour que celle-ci soit mise à jour.

EXTREME PROGRAMMING

L'*extreme programming* [xtreme] est une autre façon de mener un projet de développement logiciel. Nous allons voir les quelques points les plus marquants de cette discipline, mais il y a d'autres règles. Les développeurs programment par binômes, ainsi le code est en permanence relu par un autre programmeur, et de façon plus approfondie que lors d'un audit. Le code résultant est donc de bien meilleure facture. De plus, comme les binômes tournent, les connaissances de chacun peuvent se partager et le niveau de l'équipe augmente énormément.

Les développements sont guidés par des tests automatisés. Les premières lignes de code écrites concernent les premiers tests que doit passer le programme. Au fur et à mesure de l'avancement du projet, de nouveaux tests sont codés et ajoutés. Ainsi, les développements du logiciel peuvent être testés très souvent, et il n'y a pas de régression possible.

QUELQUES CONSEILS AVANT DE COMMENCER

Utiliser un architecte

Tout gros projet a besoin d'un architecte, dont le rôle est uniquement de garantir la cohérence globale du projet. Il n'est pas censé développer quoi que ce soit mais orienter les

développeurs, choisir les solutions, spécifier les interfaces et s'assurer de la bonne communication entre les différents développeurs et de la bonne compréhension des spécifications.

Pour du développement sécurisé, cet architecte **doit** avoir de bonnes connaissances en sécurité. Sinon, ce serait un peu comme faire dessiner les plans techniques d'une tour de 50 étages par un étudiant des beaux-arts.

Le choix des développeurs

Lorsqu'on cherche à développer une application où la sécurité est critique, il faut des développeurs expérimentés dans les technologies utilisées. Ne croyez pas les commerciaux, n'hésitez pas à faire passer des tests techniques aux "experts ès (*mettez votre technologie ici*)".

Il est rare de trouver des développeurs ayant de bonnes compétences en sécurité. Veillez cependant à en avoir au moins un dans votre équipe de développement, car la sécurité, c'est un état d'esprit. De plus, cela vous coûtera moins cher de faire tout de suite un programme sécurisé que de devoir le corriger plus tard.

Le choix du langage

Étant donné un problème, certains langages se prêtent mieux que d'autres pour le résoudre. Il est souvent possible de dévisser avec une pince, mais le tournevis reste toujours la meilleure façon de faire. De plus, certains langages demandent une charge cognitive plus importante pour faire la même chose. Comparez l'ouverture d'une *socket* en Python (1 ligne) et en C (10 lignes).

Les langages orientés objet sont un plus pour les projets même de taille moyenne. Le penser objet reste possible avec des langages non orientés objet, mais le coût cognitif n'est plus le même.

D'autres points de comparaison sont possibles. Les langages où il faut déclarer les variables aux débuts de bloc imposent un coût cognitif supplémentaire. Lorsqu'on a besoin d'une variable, on doit penser à la déclarer.

Certains langages sont plus risqués que d'autres. Le C, par exemple, en contrepartie de sa puissance, permet tout et n'importe quoi. Au programmeur donc de garder en tête les limites à ne pas franchir et validations à effectuer. Le PHP également, qui est paradoxalement trop simple par rapport à sa complexité. Il est trop facile de faire un site qui marche sans rien y comprendre que nombre de failles bêtes sont présentes. Les *scripts shells* sont également très risqués, en raison de la complexité de l'évaluation de chaque ligne (substitutions diverses, nombreux caractères spéciaux) et également de l'évolution de ceux-ci (rajout de fonctionnalités).

Coding Style

Un bon style de code est important pour l'homogénéité du code créé en équipe, et pour sa relecture. Il faut donc en avoir un, le faire lire à tous les développeurs et veiller à ce qu'il soit respecté.

Attention cependant à ne pas pondre des contraintes aberrantes qui auront l'effet contraire à celui recherché.



J'ai par exemple vu un vrai *coding style*, vraiment appliqué, où l'on imposait que chaque ligne de code soit précédée d'un commentaire qui paraphrasait en français ce qu'elle faisait. On se retrouvait ainsi avec le programme en double : un coup normal et un coup traduit en français, dans les commentaires. Ces commentaires n'apportaient absolument rien et diminuaient la lisibilité du code, tout en diluant les commentaires utiles. Inspirez-vous plutôt de celui du noyau Linux [**codingstyle**] : direct, pragmatique, efficace.

LA SOURCE DE TOUS LES MAUX

Malgré tout cet attirail, toutes ces études, le mal persiste, l'ennemi semble invincible. Pourquoi n'arrivons-nous pas à nous en débarrasser définitivement ?

Derrière chaque bogue, il y a un développeur, un homme qui s'est trompé (bon, OK, parfois, ils s'y mettent à plusieurs). L'être humain a des limites et le mieux qu'il puisse faire c'est de les connaître afin de ne pas trop se surestimer. La variante de l'être humain que l'on nomme développeur, car sa fonction est de créer le code de logiciels, hérite de toutes les faiblesses de l'être humain. Mais certaines qui n'étaient pas gênantes pour un être humain de base vont le devenir pour un développeur, en particulier celle de ne pas être doué pour la programmation. De plus, un autre type d'être humain entre dans l'équation. Il s'agit de l'utilisateur.

L'ÊTRE HUMAIN DE BASE

Nous passons rapidement sur la grande variabilité de ses performances (digestion, mauvaise humeur, manque de motivation, petite déprime, soucis personnels, lendemain de fête, etc.) car d'autres limites guettent.

Les limites cognitives

La règle des 7 ± 2 stipule que tout être humain est capable de manipuler entre 5 et 9 objets (*chunks*) dans sa mémoire de travail. On peut par exemple lire un numéro de téléphone de 5 nombres et le composer tout de suite après, mais nous n'arriverons pas à faire la même chose avec le même numéro, s'il est présenté sous forme de 10 chiffres (étonnant, non ?). Il en va de même avec les variables d'un programme, ou des éléments d'un algorithme à agencer. On comprend mieux pourquoi on doit se contenter de faire des choses simples si on ne veut pas se tromper.

De plus, le contenu de cette mémoire est très volatile. La moindre interruption (petite sœur qui a des problèmes avec son Yop, collègue qui vous presse pour aller nourrir son estomac...) peut être fatale à son contenu. Par ailleurs, sa durée de stockage sans répétition (boucle vocale) est de l'ordre de 20 secondes.

Les problèmes de communication

Souvent, derrière un logiciel, il y a une équipe de développeurs, chapeauté par un architecte. Le travail de l'architecte consiste à avoir une vue globale du projet et à le découper de manière

cohérente en petits bouts indépendants qui pourront être donnés aux différents développeurs. Il devra maintenir la cohérence globale du projet en spécifiant les différentes interfaces entre les petits bouts et en orientant les solutions d'implémentation possibles de chaque petit bout.

Tout cela nécessite une bonne communication. Traduire une image, une idée, une représentation en mots et schémas lors de réunions ou d'écritures de spécifications, ce n'est pas si évident. Et relire une spécification afin de reproduire l'idée de l'architecte dans sa propre tête, c'est encore plus dur.

En guise d'exemple, souvenez-vous de la mission Mars Climate Orbiter, où la sonde s'est lamentablement crashée sur Mars parce qu'une équipe de développement utilisait dans leurs programmes le système métrique international et l'autre le système métrique anglo-saxon. Le problème de communication entre les deux équipes s'est directement traduit par le problème de communication entre leurs deux programmes.

L'ÊTRE HUMAIN DIT UTILISATEUR

L'utilisateur n'a *a priori* rien à voir avec la sécurité des logiciels qu'il utilise. Lorsqu'il se trompe en les configurant, ne respecte pas les *best practices*, et qu'il ouvre ainsi des trous béants, il le fait tout seul comme un grand.

Pour autant, la façon dont le logiciel a été conçu peut parfois provoquer ces erreurs.

Si la logique de configuration ou l'interface sont à l'opposée du modèle mental de l'utilisateur, l'outil aura beau être sécurisé, il ne sera jamais bien utilisé. Un logiciel sécurisé est donc également un logiciel facile à appréhender.

L'ÊTRE HUMAIN DIT DÉVELOPPEUR

Entre le recyclage d'ingénieurs de tout poil en développeurs, le *turnover* de certaines équipes et les apprentissages sur le tas, la qualité du code n'est pas toujours à la hauteur.

On ne peut pas transformer n'importe qui en développeur. Même s'il suffit parfois d'un peu de logique, de quelques connaissances d'anglais et d'une maîtrise de copier/coller pour arriver à des résultats corrects en apparence, on se retrouve toujours avec du code mal pensé, marchant dans 90% des cas mais ayant des comportements catastrophiques sur les cas particuliers.

Une équipe de développement doit pouvoir compter en son sein des compétences en sécurité mais également en ergonomie. Ces deux qualités ne s'improvisent pas.

L'ÊTRE HUMAIN DIT DÉCIDEUR

Un mot tout de même sur les décideurs qui se laissent facilement bernier par le discours du commercial, et qui optent parfois pour des produits bien marketés plutôt que bien faits [**mmotu**]. Cela ralentit fortement le travail de la sélection naturelle des meilleurs éléments.



VIVRE AVEC SON ENNEMI

Maintenant que la faillibilité de l'être humain est mise à jour, il faut se résigner à côtoyer l'ennemi. Si nous ne pouvons pas le vaincre, il faut l'enfermer. Il faut le réduire à l'impuissance. Lorsque la sécurité est présente dès le début du projet, il est facile de compartimenter, de placer des garde-fous, de prévoir une architecture qui saura contenir une intrusion.

LA SÉPARATION DE PRIVILÈGES

La séparation de privilèges consiste à découper l'application en plusieurs morceaux, à correctement les isoler, à les faire communiquer à travers des interfaces très bien définies, et à faire tourner les parties les plus exposées avec le moins de privilèges possibles. Concrètement, cela veut dire identifier les différentes fonctions du programme et évaluer les privilèges dont chacune a besoin ainsi que son exposition. Si une partie nécessite beaucoup de privilèges tout en étant très exposée, il faudra essayer de la découper encore un peu. Il est évident que si cette architecture a été pensée dès le début, la tâche est grandement simplifiée. Vous pouvez par exemple vous référer à l'article suivant de ce dossier, qui explore l'anatomie de Postfix [etudecas].

LES SUPERVISEURS

Les superviseurs sont des programmes dont l'unique but est de vérifier en permanence le bon fonctionnement d'une tâche particulière. Ils sont, bien entendu, extérieurs à cette tâche. Leur usage est assez courant dans le domaine de la sûreté de fonctionnement. Leur but est simple : si la tâche qu'ils supervisent semble ne plus fonctionner correctement, ils la relancent ou la court-circuitent. C'est tout bête. C'est d'ailleurs l'intérêt du superviseur : étant extrêmement simple, on peut arriver à ne pas faire (trop) d'erreurs le programmant.

Un bon exemple de leur utilisation est ce qui a été fait par l'équipe de l'INRIA pour l'ICFP'99 [icfp99]. Le but était de transformer une description de personnage en une description équivalente mais plus petite. Une telle optimisation est complexe et une erreur est vite arrivée. En revanche, il est relativement aisé de détecter que deux descriptions ne correspondent pas, en les évaluant pour un grand nombre de valeurs et en comparant les résultats. Ainsi, le superviseur laissait l'optimiseur tourner et vérifiait le résultat. En cas d'erreur, il rendait la description originale. Ainsi le résultat était médiocre, mais juste. Stackguard, Stackshield, Propolice et Libsafe, même s'ils font partie de la tâche qu'ils surveillent, peuvent être vus comme des superviseurs. Les 3 premiers sont des extensions du compilateur qui rajoutent le code nécessaire pour vérifier que l'adresse de retour n'a pas été écrasée. Le quatrième surcharge certaines fonctions dangereuses, les exécute, et vérifie qu'aucun débordement n'a eu lieu.

PAX

Il y a deux manières d'éteindre un feu : soit on coupe le carburant, soit on le prive du comburant. Plutôt que s'attaquer à la faille, on peut protéger ce dont toute exploitation de faille a besoin : le

détournement du flot d'exécution. Toutes les techniques d'exploitation ont besoin d'écrire l'adresse du code qu'elles veulent exécuter à un endroit où il sera interprété comme tel (adresse de retour, *global offset table*, section *dtors...*). PaX rend aléatoire l'adresse des différentes sections. Si le pirate n'est plus en mesure de calculer l'adresse où se trouve le code visé ou l'adresse du pointeur qu'il doit écraser, il ne pourra rien exploiter.

LA GESTION DES ERREURS

Une habitude à prendre dans les langages qui n'ont pas d'exceptions, est la gestion **systématique** des codes de retour des fonctions, y compris et surtout dans les prototypes. En effet, c'est lors de la conception du prototype que le plus d'erreurs vont se produire (normal, ce n'est qu'un prototype). Si vous ne le faites pas, vous perdrez énormément de temps à développer ce prototype car vous ne pourrez pas identifier facilement quelles sont les erreurs qui se produisent et où elles se produisent. Vous perdrez de plus une bonne opportunité d'analyser des cas extrêmes auxquels vous n'aviez pas pensé.

LES LOGS DE DÉBOGAGE

La partie de code consacrée au débogage doit faire partie intégrante du processus de développement. En effet, si un bogue apparaît, modifier le code pour le trouver peut modifier ses conditions d'apparition, voire le masquer totalement. De plus, du code de débogage ajouté et enlevé un peu au hasard, ça laisse toujours des traces, ça fait perdre du temps, et ça n'est jamais idéal. Dans le cas de langages avec préprocesseur, ce code de débogage peut être totalement retiré pour la compilation, ce qui ne pénalise plus les performances du programme. Dans les autres cas, le gain en performances ne justifie pas, en général, l'ablation du code de débogage. À noter qu'on peut décider de rajouter une étape de pré-processeur à n'importe quel langage. Avoir le code de débogage présent dans le programme en production, mais désactivé par une option est également un plus pour trouver rapidement les problèmes et les résoudre au mieux. En conclusion, il faut prévoir d'intégrer les procédures de débogage dans le processus de développement.

CONCLUSION

La sécurité des logiciels n'est pas pour demain, y compris pour les logiciels sur lesquels repose la sécurité du système d'information. La première partie de cet article est largement illustrée par des exemples de logiciels programmés par des éditeurs de produits de sécurité. L'incompétence est souvent la première cause de bogues en général, et de failles de sécurité en particulier. Mais c'est de moins en moins vrai et c'est la complexité des programmes face aux limites cognitives de l'être humain qui commence à se ressentir. Je conclurai donc par ce conseil : ne faites qu'une chose, et faites-la bien !

KISS,

Philippe Biondi

<phil@secdev.org> <philippe.biondi@arche.fr>



RÉFÉRENCES

[mco] NASA Advisors Explain Mars Mission Failures to a Concerned Congress, <http://www.spaceref.com/news/viewnews.html?id=115>

[grugq] The Grugq Userland Execve post, <http://seclists.org/lists/bogueraq/2004/Jan/0002.html>

[dclp] David Bacon et al. The "Double-Checked Locking is Broken" Declaration, <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

[xtreme] What is extreme programming, <http://www.xprogramming.com/xpmag/whatisxp.htm>

[aleph1] Smashing The Stack For Fun And Profit, <http://www.phrack.org/show.php?p=49&a=14>

[hdes] Simon Singh. *Histoire des codes secrets*, JC Lattès, 1999.

[codingstyle] Linux kernel coding style, linux/Documentation/codingstyle ou <http://lwn.net/2001/1115/a/CodingStyle.php3>

[mmotu] Harry Martin. *Marketing mysteries of the Universe*, <http://www.vcnet.com/~harry/OS2020.JPG>

[icfp99] Camls 'R Us team Composite, <http://www.ai.mit.edu/extra/icfp-contest/>

[etudecas] Victor Vuillard. *Sécurité logicielle : étude de cas*, MISC 12, Sécurité des logiciels.

[faq] Thamer Al-Herbish. *Secure UNIX Programming FAQ*, <http://www.whitefang.com/sup/>

[tpics] Jerome H. Saltzer et Michael D. Schroeder. *The Protection of Information in Computer Systems*, <http://web.mit.edu/Saltzer/www/publications/protection/>

[wheeler] David Wheeler. *Secure Programming for Linux and Unix HOWTO*, <http://www.dwheeler.com/secure-programs/>

[peonguide] Michael Bacarella. *The Peon's Guide To Secure System Development*, <http://michael.bacarella.com/papers/secsoft/html/>

[meta1] Sardonix Security Portal Auditing Resources, https://sardonix.org/Auditing_Resources.htm

[meta2] Tools & Tips for auditing code, <http://www.vanheusden.com/Linux/audit.html>

[kerio] Vulnerabilities in Kerio Personal Firewall, <http://www.securiteam.com/windowsntfocus/5VP10009PQ.html>

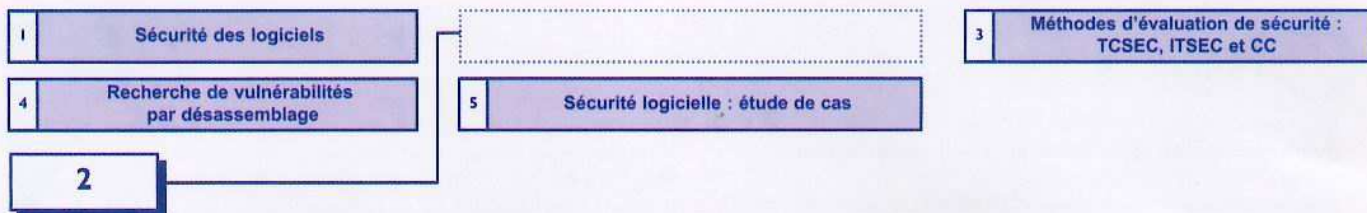
[snort] Snort TCP Stream Reassembly Integer Overflow Vulnerability, <http://www.securiteam.com/securitynews/5HP0C2A9QC.html>

[fw1] Checkpoint Firewall-1 HTTP Parsing Format String Vulnerabilities, <http://www.securiteam.com/securitynews/5CP071FC0W.html>

[viruswall] InterScan VirusWall Remote Configuration Vulnerability, <http://www.securiteam.com/windowsntfocus/5MP051F4KK.html>

[ipfilter] IPFilter race condition enables partial firewall penetration, <http://www.securiteam.com/unixfocus/5QP080A1UY.html>

[ciscoheap] FX Cisco IOS Heap Exploit Proof of Concept, <http://www.securiteam.com/exploits/5KP000A81K.html>



Dix dangers qui guettent le programmeur de cryptographie



Vous êtes en train d'écrire une application qui traite des données sensibles et bien évidemment, vous faites appel à de la cryptographie. Mais, en fait, avez-vous bien pensé à tout ? Le but de cet article est de soulever quelques problèmes auxquels un programmeur développant une application qui utilise de la cryptographie se verra confronté.

RÉFLÉCHIR À LA SÉCURITÉ TROP TARD

Lorsqu'un architecte dessine les plans d'une banque, il ne va pas dessiner le bâtiment dans un premier temps, puis se demander comment le protéger dans un second temps. Dès la première minute, il réalise l'inventaire des besoins en sécurité du bâtiment, puis il dessine les plans en ayant cet inventaire en tête.

De façon analogue, il est parfaitement illusoire de vouloir intégrer de la sécurité et de la cryptographie à une application une fois qu'elle existe déjà en version alpha, son architecture devant être pensée *dès le début* en tenant compte des contraintes de sécurité. De plus, il est impossible de définir des contraintes de sécurité précises sans connaître le mieux possible les risques inhérents à l'application. Cet inventaire des risques permet également de fixer des priorités en matière d'investissement en temps et en argent : pourquoi protéger la confidentialité des disques sur lesquels est stockée une base de données alors qu'on peut la consulter à distance et que les données transitent en clair sur Internet ?

RÉINVENTER LA ROUE... CARRÉE

Une qualité nécessaire pour un programmeur de cryptographie est d'être conservateur. Il est toujours tentant (et flatteur pour l'ego) de se lancer dans la conception d'un nouvel algorithme, ou d'un nouveau protocole cryptographique, comme il est tentant de vouloir réécrire complètement une bibliothèque de fonctions cryptographiques. Ce n'est malheureusement pas une bonne idée, *a priori*. Concevoir un nouvel algorithme ou protocole cryptographique sûr est une tâche *terriblement* difficile. Les

meilleurs chercheurs inventent et développent continuellement de nouvelles solutions... qui ne sont pas sûres et qui seront cassées à court ou moyen terme. Il est donc *indispensable* de s'appuyer sur des algorithmes et protocoles existant depuis un certain nombre d'années et qui ont ainsi passé l'épreuve du temps en termes de sécurité.

Toutefois, même en choisissant des algorithmes réputés, il faut encore être certain de les utiliser correctement. Par exemple, l'algorithme RSA, tel qu'exposé dans l'extrême majorité des livres traitant de cryptographie (c'est-à-dire sans formatage des entrées), n'est pas sûr ! Si le principe est connu, il est des précautions indispensables à prendre dans le choix des nombres qui serviront à générer les clefs (voir par exemple [10]).

Une solution incontournable à ce problème consiste à s'appuyer sur des standards reconnus, tel que la série des standards PKCS, IEEE et autres ISO, non sans s'être renseigné préalablement sur leur sécurité auprès de personnes compétentes (certaines versions de standards ne pouvant raisonnablement pas être considérées comme sûres). Le même principe vaut pour les bibliothèques cryptographiques. Pourquoi vouloir tout recommencer à zéro, et prendre le risque de faire des erreurs, alors que des bibliothèques (OpenSSL [1], Microsoft CryptoAPI [2],...) toutes faites, écrites par des spécialistes, sont disponibles.

CHOISIR UNE SOURCE D'ALÉA AU HASARD

La plupart des algorithmes et protocoles cryptographiques doivent pouvoir disposer d'une source d'aléa cryptographiquement sûre, c'est-à-dire d'une source de nombres pour laquelle il est virtuellement impossible de deviner à l'avance les valeurs qu'elle



fournit. Cette source sera typiquement utilisée pour générer des clefs, des nonces ainsi que d'autres secrets. Le problème est que les ordinateurs sont des machines déterministes, et qu'il leur est difficile de produire des valeurs vraiment aléatoires, donc imprédictibles.

On peut grossièrement classer les générateurs d'aléa en trois catégories : les générateurs non-cryptographiques, les générateurs cryptographiques, et les générateurs fondés sur un matériel dédié. La première catégorie englobe la quasi-totalité des générateurs pseudo-aléatoires dont dispose tout programmeur, comme `rand()`, `random()`, `drand48()` et consœurs offertes par les bibliothèques de programmation standard. On peut partir du principe qu'un adversaire sera toujours capable de prédire leurs sorties ; leur utilisation est donc prohibée dans un contexte cryptographique. La deuxième catégorie englobe des algorithmes cryptographiques capables, une fois initialisés par une valeur sûre, de produire une suite de valeurs imprédictibles aussi longue que nécessaire, pour autant que certaines conditions soient réunies (notamment au niveau de leur initialisation). Enfin, la troisième catégorie est constituée de solutions matérielles qui collectent des valeurs aléatoires issues de phénomènes physiques et les mette à disposition du programmeur. Un de leurs défauts majeurs est leur faible débit ; en revanche, elles offrent une sécurité souvent très élevée. Les valeurs générées par ces dernières seront typiquement utilisées pour initialiser des générateurs de la seconde catégorie.

Les systèmes d'exploitation modernes fournissent souvent un accès à un générateur pseudo-aléatoire de qualité cryptographique. Dans le monde Microsoft, la CryptoAPI offre un générateur via la fonction `CryptGenRandom()` qui utilise un algorithme fondé sur SHA-1 et RC4. La documentation indique que la CryptoAPI stocke une valeur d'initialisation aléatoire pour chaque utilisateur, sans toutefois préciser comment cette valeur est générée. Il est donc *fortement conseillé* de fournir à `CryptGenRandom()` une valeur initiale supplémentaire dont on est sûr qu'elle possède assez d'entropie (d'aléa). L'utilisation de cette fonction est assez simple, voir l'exemple en C du **Tableau 1**.

Dans le monde UNIX, la plupart des OS (au moins Linux, les BSD, Mac OS X, AIX, HP-UX, IRIX et Solaris) fournissent deux *devices* (`/dev/random` et `/dev/urandom`) qui génèrent un flux de valeurs aléatoires de qualité cryptographique. Le premier nommé est bloquant par défaut, c'est-à-dire que si l'OS estime qu'il n'y a pas assez d'entropie à livrer, une opération `read()` sera suspendue jusqu'à ce qu'il y ait à nouveau assez d'entropie ; le second (sauf circonstances extraordinaires) fournira toujours le nombre d'octets aléatoires demandé.

On notera qu'il est possible de rendre un appel à `/dev/random` non-bloquant (c'est-à-dire qu'une erreur sera retournée si il n'y a pas assez d'entropie) en mettant le bit `O_NONBLOCK` à 1 sur les drapeaux associés au descripteur de fichier, ceci à l'aide de l'appel système `fcntl()`. Il faut juste savoir que les OS ne garantissent pas, en général, que `/dev/urandom` soit correctement initialisé. Soit on peut sauver une valeur d'initialisation sûre avant un boot (et ne pas se la faire voler), soit on peut générer artificiellement de l'entropie à l'aide d'une commande `cat /dev/hda > /dev/null &` puisque dans la plupart des implémentations, le temps entre les interruptions est pris en compte comme entropie.

Il existe une solution fonctionnant uniquement en *user-land* : l'*Entropy Gathering Daemon* [3], qui est un substitut de `/dev/random`. Enfin, la bibliothèque OpenSSL dispose également d'un bon générateur pseudo-aléatoire.

Tableau 1

```
* handle sur un Cryptographic Service Provider */
/* acquis à l'aide de CryptAcquireContext() */
HCRYPTPROV hCryptProv;

/* tampon qui recevra les données aléatoires */
/* son contenu initial servira de "seed" */
BYTE pbBuffer[16];

if (CryptGenRandom(hCryptProv, 16, pbBuffer)) {
    printf ("Random sequence generated. \n");
}
else {
    printf ("Error during CryptGenRandom.\n");
    exit (-1);
}
```

Exemple d'utilisation de `CryptGenRandom()`

VALIDER LES ENTRÉES À LA LÉGÈRE

Une source classique de problèmes apparaît dès que les entrées d'un programme ne sont pas contrôlées correctement. La règle est simple : tout ce qui rentre dans un programme doit être validé, et rejeté par défaut.

La question à se poser ici est donc mais d'où proviennent les entrées de mon programme ? La réponse est simple : partout ! Certes, les arguments passés en ligne de commande sont une possibilité, mais une parmi ... tant d'autres ! Pensez aussi aux variables d'environnement, au(x) fichier(s) de configuration, aux résultats retournés par les DNS, aux paquets capturés sur les réseaux (pour les sniffers). Bref, tout ce qui ne vient pas directement du programme doit être considéré comme suspect.

La solution à cela n'est, théoriquement, pas très compliquée : toutes les entrées doivent être contrôlées et la politique par défaut doit être de tout rejeter.

Quelles sont les exemples de telles failles ? Les classiques *buffer overflows* ou autres *format bugs*, mais aussi des attaques aussi simples que le *Cross Site Scripting* qui consiste à faire exécuter du code injecté dans une page web par le navigateur (client) du site en question ou une bête manipulation de cookies ou URL (voir **Tableau 2**).

À titre d'illustration, nous prendrons un cas particulier de *format bug*. Ce programme propose de se connecter à un serveur ultra-sensible. Pour éviter les tentatives de brute-force des comptes/mots de passe le nombre de tentatives est limité à 3. Toutefois, cette mesure ne sert à rien puisque le programme contient un *format bug* évident.



Tableau 2

```

/* auth.c */
#define MAX_TRIES 3

/* Remplace le \n final par un \0 */
void chomp(char *str);

/* Fonction vérifiant les autorisations */
int do_auth(char *user, char *pass);

/* Fonction exécutée lorsque l'accès est autorisé */
void run_secret_command(void);

int
main(int argc, char **argv)
{
    int try = 0;
    char user[64], pass[64], msg[64];

    do {

        printf("user=");
        fgets(user, sizeof(user)-1, stdin);
        user[sizeof(user)-1] = 0;
        chomp(user);

        printf("pass=");
        fgets(pass, sizeof(pass)-1, stdin);
        pass[sizeof(pass)-1] = 0;
        chomp(pass);

        if (do_auth(user, pass))
            run_secret_command();
        else
            snprintf(msg, sizeof(msg), "Access refused for %s (%d/%d)\n",
                     user, try+1, MAX_TRIES);
            printf(msg);
    } while (++try < MAX_TRIES);

    printf("Too many tries ... sleep now!\n");
    exit (-1);
}

```

Exploitation inhabituelle d'un bug de format

En revanche, il n'est pas exploitable au sens habituel (sur x86) puisque même si on va modifier l'adresse de retour du `main()`; on ne l'utilisera jamais... puisqu'il n'y a aucun `return`. Cependant, avec un peu d'astuce, d'espièglerie, ce n'est pas que la vie de Candy. On va exploiter le bug de format pour ré-initialiser le compteur de tentative à 0, ce qui nous permettra de "bruteforcer" tranquillement :)

Le principe de l'attaque est simple : à l'aide du *format bug*, on contrôle la valeur de la variable `try`. On pourrait réinitialiser cette variable à 0 après deux tentatives infructueuses, puis rejouer.

Cependant, outre le *format bug*, profitons également de la négligence du programmeur : la variable `try` est de type `int` et le test qui est fait consiste à vérifier que cet entier est bien inférieur à `MAX_TRIES` (3). Or, un nombre négatif est toujours inférieur à 3. Il nous suffit donc d'initialiser `try` à un nombre négatif plutôt qu'à 0 pour disposer de beaucoup d'essais.

Pour l'exploitation, rien de neuf :

- 1 on recherche l'adresse du début du buffer `user` à grands coups de `AAAA%[1:]$x` :

```

$ ./auth
user=AAAA%1$x
pass=plop
Access refused for AAAA3 (2/3)
user=AAAA%35$x
pass=plop
Access refused for AAAA13e78 (1/3)
user=AAAA%36$x
pass=plop
Access refused for AAAA1414141 (3/3)
too many tries ... sleep now!

```

Il faut certes faire plusieurs tentatives, mais on trouve finalement l'offset voulu.

- 2 On recherche l'adresse de la variable `try` dans la pile. Pour cela, on pourrait utiliser l'information précédente pour transformer le `printf()` en débogueur afin d'explorer la pile. Toutefois, `gdb` reste un meilleur débogueur que `printf()` donc ne nous privons pas. On obtient alors que l'adresse de `try` est `0xbffff878`

- 3 Exploitions nos informations :

```

$ (perl -e 'print "\x7f\xfb\xff\xbfq%.104x%36$\n\nAAAA\n"; cat dico.txt') | ./auth
user=pass=Access refused for 0ÿ;q00...001e (1/3)
user=pass=Access refused for toto (-2147483646/3)
user=pass=Access refused for aaaa (-2147483645/3)
user=pass=Access refused for cccc (-2147483644/3)
user=pass=
*** Welcome master ***

```

Le premier appel à `perl` injecte la chaîne de format dans la variable `user` pour exploiter le *format bug*. Il est suivi d'une première chaîne `AAAA` qui fait office de mot de passe. Là, comme on s'y attend, la tentative est un échec, le premier sur trois (1/3). Toutefois, l'affichage de la chaîne que nous avons injectée provoque le remplacement de la valeur de la variable `try`, comme on le remarque sur la ligne pour l'utilisateur `toto` : nous venons d'échouer à la tentative -2147483646 sur 3. Finalement, à la quatrième tentative (-2147483644/3), le bon couple est trouvé, et le programme nous donne l'accès.

Tout ça pour dire que *tout doit être validé*. Dans le cas de caractères, il faut vérifier qu'ils sont explicitement autorisés ; dans le cas de nombres, il faut contrôler les valeurs minimales et maximales de l'ensemble de définition, etc.

NE PAS DIVULGUER
DES INFORMATIONS INUTILES

Ce qui pourrait paraître comme étant une lapalissade informatique revêt en fait une importance singulière en sécurité. On le sait, l'information est capitale dans ce domaine. Par conséquent, il est capital d'évaluer quelle information donner, et à qui.



Dix dangers qui guettent le programmeur de cryptographie

Prenons deux exemples de serveurs particulièrement bavards :

- ① Les serveurs telnet qui, par chance, se font de plus en plus rares :

```
raynal@alfred raynal>> telnet batman
Trying 192.168.0.66...
Connected to batman.gotham-city.org.
Escape character is '^J'.
Red Hat Linux release 7.3 (Valhalla)
Kernel 2.4.24 on an i686
login:
```

Alors que je demande simplement à me connecter à batman, j'ai droit à la distribution, l'architecture et la version du noyau ... alors que l'option -h aurait évité tout ce verbiage.

- ② Les serveurs web qui racontent leur vie en annonçant leur version et plus si affinités :

```
$ telnet www.microsoft.com 80
Trying 207.46.134.221...
Connected to www.microsoft.com.
Escape character is '^J'.
HEAD / HTTP/1.0
```

```
HTTP/1.0 200 OK
Date: Sun, 25 Jan 2004 17:21:29 GMT
Content-Length: 37700
Content-Type: text/html
Expires: Sun, 25 Jan 2004 17:21:29 GMT
Cache-Control: private
Server: Microsoft-IIS/6.0
P3P: CP="ALL IND DSP COR ADM CONO CUR CUSO IVAO IVDO PSA PSD TAI
TELO OUR SAMO CNT COM INT NAV ONL PHY PRE PUR UNI"
X-Powered-By: ASP.NET
Connection closed by foreign host.
```

```
$ telnet www.mandrakesoft.com 80
Trying 212.43.244.27...
Connected to www.mandrakesoft.com.
Escape character is '^J'.
HEAD / HTTP/1.0

HTTP/1.1 302 Found
Date: Sun, 25 Jan 2004 18:06:03 GMT
Server: Apache-AdvancedExtranetServer/1.3.27 (Mandrake Linux/Bmdk)
mod_ssl/2.8.12 OpenSSL/0.9.7 PHP/4.3.1
Location: http://www.mandrakesoft.com/
Connection: close
Content-Type: text/html; charset=iso-8859-1
Connection closed by foreign host.
```

Sans compter que cela est souvent pire quand vous demandez une page qui n'existe pas (pour mémoire, avec Apache, il suffit de mettre ServerToken Prod pour déjà empêcher l'annonce de la bannière).

Ce genre de problèmes apparaît également en cryptographie. Récemment, des démonstrations d'attaques par effet de bord ont été proposées contre des implémentations logicielles trop bavardes, alors qu'elles étaient plutôt l'apanage du monde matériel

(comme les cartes à puce). Ces attaques utilisent de l'information issue de fichiers de logs, ou de l'information temporelle, par exemple, et l'exploitent pour casser la cryptographie sous-jacente.

Deux exemples classiques sont l'attaque de Bleichenbacher contre la version 1.5 du standard PKCS#1, qui définit comment bien utiliser l'algorithme RSA et l'attaque de Vaudenay contre la méthode de bourrage CBC-PAD, qui est utilisée dans nombre de standards cryptographiques (pour plus de détails, voir [4]). Alors qu'il est concevable qu'un programme utilisant de la cryptographie soit bavard lors de la phase de déverminage, il est préférable de le faire taire le plus possible lorsqu'il entre en phase de production !

SE FAIRE TRAHIR PAR LA GESTION DE LA MÉMOIRE

Une application utilisant de la cryptographie est forcément amenée à traiter des données secrètes (clefs, vecteurs d'initialisation, numéros de sessions,...) qui ne doivent tomber sous aucun prétexte dans les mains d'un adversaire. Ces données doivent donc être traitées avec la plus grande prudence.

Premièrement, il est absolument impensable de *hard-coder* une valeur secrète dans un programme: l'exemple **Tableau 3** parle de lui-même.

Tableau 3

```
pjunod@machine:~/temp/mem> cat > motdepasse.c << EOF
#include <string.h>

const char *motdepasse = "dhjskjhd";

int main(int argc, char **argv)
{
    if (argc == 2)
        return (strncmp (motdepasse, argv[1], strlen (motdepasse)) ? 0 : 1);

    return -1;
}
EOF
pjunod@machine:~/temp/mem> gcc -o motdepasse motdepasse.c
pjunod@machine:~/temp/mem> strings motdepasse
/lib/ld-linux.so.2
libc.so.6
strncmp
_IO_stdin_used
__libc_start_main
strlen
__gmon_start__
GLIBC_2.0
PTRh
QVht
dhjskjhd
pjunod@machine:~/temp/mem>
```

● Mot de passe hard-codé



L'exemple reste valable si le secret est une valeur générée aléatoirement: rien ne ressemble plus à une valeur aléatoire qu'une valeur aléatoire lorsqu'on examine le fichier exécutable à l'aide d'un éditeur hexadécimal. De plus, il est relativement facile d'automatiser la recherche de clefs cryptographiques en utilisant quelques notions de statistiques et de théorie des nombres s'il s'agit de clefs asymétriques.

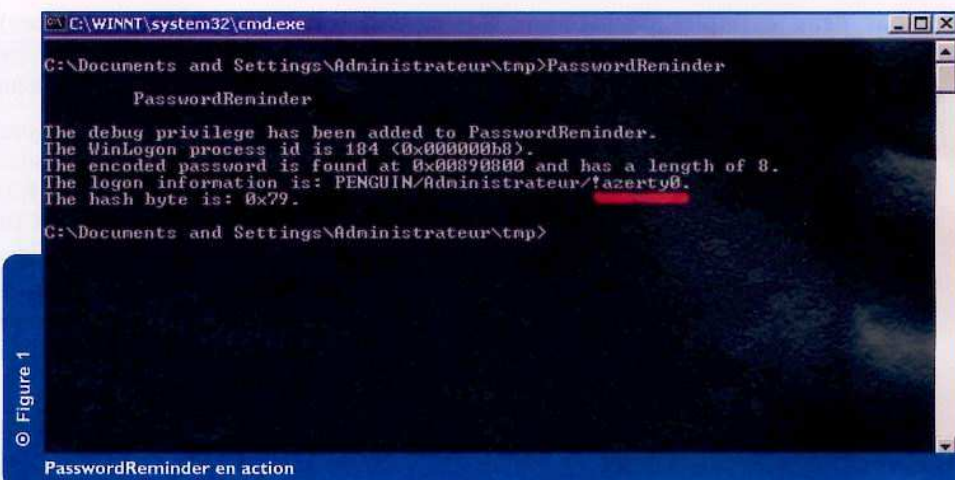
Un principe de base stipule que les secrets ne devraient être visibles que le temps de les utiliser. Cela implique plusieurs choses. Premièrement, les secrets qui nécessitent d'être stockés quelque part doivent l'être de façon sûre, ce qui signifie qu'ils doivent être stockés soit dans la mémoire humaine, soit sur un support qui ne peut tomber entre les mains de l'adversaire (par exemple, dans un CD-ROM mis dans un coffre-fort ...) ou qui peut tomber dans les mains de l'adversaire sans dommage (dans une carte à puce ou un module *tamper-proof*).

Deuxièmement, les secrets, une fois devenus inutiles, doivent impérativement être détruits pour éviter qu'ils ne tombent en des mains inamicales: les tampons contenant des secrets doivent être effacés (à l'aide d'un `memset (buffer, 0x00, sizeof(buffer))` par exemple) avant d'être libérés. Ceci implique deux choses: l'application doit savoir quand le secret n'est plus utilisé, ce qui n'est pas toujours le cas, et elle doit gérer les tampons sensibles de manière prudente, pour éviter d'en oublier. C'est pourquoi les bibliothèques cryptographiques bien écrites possèdent leur propre système de gestion de la mémoire.

Il faut également se méfier des compilateurs trop entreprenants qui, lors de la phase d'optimisation, éliminent les `memset()`, ainsi que des arrêts brutaux de l'application (dus à un bug, à une exception, à un signal, etc.) et s'assurer dans la mesure du possible que les données sensibles soient effacées. Certains langages de haut-niveau (comme Java, qui utilise un *garbage collector*) peuvent rendre ce processus difficile à gérer.

MS Windows 2000 est un excellent exemple de ce qu'il ne faut pas faire puisque le mot de passe est présent en mémoire *en clair* tant que la session de l'utilisateur dure (d'où l'importance des écrans de veille), comme le montre la **figure 1**.

Enfin, un mécanisme propre aux systèmes d'exploitation modernes rend les choses encore plus difficiles: la mémoire virtuelle. Les pages de mémoire allouées par un programme peuvent très bien se retrouver sur le disque dur dans le fichier de *swap* sans avertissement, et ceci pour de multiples raisons. Une fois de retour en RAM, une page de mémoire aura laissé une trace sur le disque dur, ce qui pose problème si cette page contenait des données sensibles. La plupart des OS possèdent un appel système qui désigne une page de mémoire comme étant inéligible pour être swappée (`mlock()` et ses variantes dans le monde UNIX). Malheureusement, certains OS (Linux est un cas typique) exigent de disposer de droit *root* pour en bénéficier, ce qui va à l'encontre



© Figure 1

PasswordReminder en action

du principe de moindre privilège. Les variantes de BSD (dont Mac OS X), par contre, permettent à n'importe quel utilisateur de verrouiller des pages de mémoire, mais en nombre limité.

Toutefois, en combinant ce problème de gestion de la mémoire, et celui vu précédemment de gestion des erreurs, un attaquant arrive à des situations intéressantes, où le développeur s'arrache les cheveux de la tête. Par exemple, si l'attaquant parvient à faire planter un programme au moment où celui-ci contient des informations sensibles, celles-ci risquent alors de se retrouver dans les pages mémoires fautives.

Sous Windows, pour peu que Visual Studio soit installé, il a même la gentillesse de vous proposer de debugger l'application. Et si vous avez effectivement les droits de debug, vous pouvez aller lire tout ce que vous voulez dans la mémoire du processus qui vient de planter.

Sous Unix, un plantage génère éventuellement un fichier *coredump* contenant également la mémoire du processus fautif et les droits de ce fichier sont assez restrictifs (*rw* pour le propriétaire, rien pour les autres).

Si vous voulez vous amuser sur un problème de ce genre, allez voir le niveau 10 du challenge *ngsec2* [11]. Si vous ne voulez pas lire la solution, passez à la partie suivante ;-). Vous êtes sûrs ? Bon, alors, allons-y. Le niveau 10 commence par un script (<http://quiz.ngsec.biz:8080/game2/level10/FHER.php>), dont une partie du code source est disponible (http://quiz.ngsec.biz:8080/game2/level10/validate_FHER.txt).

En examinant le code, on se rend compte d'un problème de validation d'entrées :

```
/* Get Username from Query String */
1: memset(user, 0, sizeof(user));
2: ch_ptr_begin = (char *)strstr(argv[1], "login=");
3: if (ch_ptr_begin == NULL)
4:     show_error();
5: ch_ptr_begin+=6;
6: ch_ptr_end = (char *)strstr(ch_ptr_begin, "&");
7: *(ch_ptr_end++)='\0';
8: strncpy(user, ch_ptr_begin, sizeof(user)-1);
```




Cet extrait du programme se charge de décomposer la requête pour en extraire le login. Une requête normale serait de la forme `login=johndoe&password=secret`. Or, ici, si des mesures sont prises pour contrôler la taille des buffers, il n'en est pas de même sur la validité syntaxique de la requête.

Que se passe-t-il si elle ne contient pas le caractère `&` ? En ligne 6, la fonction `strstr()` renvoie `NULL`, et par conséquent, la ligne 7 devient (symboliquement) `NULL='\0'`, ce qui provoque une *segmentation fault* due à une tentative d'écriture invalide.

La première étape de l'exploitation est donc de provoquer cette *segmentation fault*, qui génère un core dump sur le serveur, et ensuite de récupérer ce fichier (<http://quiz.ngsec.biz:8080/game2/level10/core>).

En examinant ce fichier (`objdump -D`, section `load8`), on trouve les fonctions qui composent le programme. En recherchant les `push %ebp`, on retrouve l'adresse du `main()` (`0x8049528`).

Ensuite, en comparant les instructions assembleur au code C récupéré précédemment, on se rend compte d'une totale similitude, jusqu'à tomber sur une suite d'instructions du type `movb valeur, %ebp - offset` allant de `0x80495fa` à `0x8049704`. Le but du jeu est alors de réordonner les valeurs en fonction des offsets, grâce au script `biondi.py` :

```
>>> import sys
>>> l=sys.stdin.readlines()
00495fa: c6 85 b8 fa ff ff 59 movb $0x59,0xfffffab8(%ebp)
...
0049704: c6 85 fa fe ff ff 00 movb $0x0,0xfffffefa(%ebp)
>>> l=map(lambda x: (x.split()[3],chr(int(x.split()[7],16))),l)
>>> l.sort()
>>> reduce(lambda x,y:x+y[1] , x, '')
'S331oU6ne81\X00Yihd45ue7d99i11\X00coredump\X00'
```

OUBLIER D'ATTAQUER SA PROPRE APPLICATION

Une fois écrite, déverminée et testée, une étape *incontournable*, à notre sens, dans le développement de toute application prétendant à un minimum de sécurité consiste à se mettre dans la peau d'un pirate et à tenter *réellement* de l'attaquer. Force est de constater que cette étape est souvent négligée pour des raisons de temps, d'argent ou de manque de compétences.

L'idéal serait de confier cette étape à une personne indépendante du développement de l'application, possédant quelques petits talents en rapport avec la tâche, de lui exposer le cahier des charges en matière de sécurité, et de la laisser s'amuser en paix quelques temps.

Le résultat d'une telle étape est souvent édifiant : une approche différente, l'absence de préjugé (lorsqu'on est le développeur, on est souvent convaincu de la sécurité de son produit) permettent de mettre en évidence des problèmes inattendus ou des scénarios d'attaque originaux.

FAIRE REPOSER (TOUTE) LA SÉCURITÉ SUR LES UTILISATEURS

Inévitablement, une application sécurisée ne peut fonctionner sans un minimum d'interaction avec ses utilisateurs, qui doivent, par définition, être considérés comme des ignares en sécurité. Quelle(s) responsabilité(s) leur laisser ? Le problème est épineux... Combien de personnes parmi nous vérifient réellement l'empreinte de la clef RSA lorsqu'elles sont en face de l'annonce suivante ?

```
pjunod@machine:~> ssh -l admin autremachine
The authenticity of host 'autremachine (192.168.12.13)' can't be
established.
RSA key fingerprint is 92:df:eb:22:aa:37:09:41:50:a1:67:da:df:1a:00:4c.
Are you sure you want to continue connecting (yes/no)?
```

Si *autremachine* est le serveur d'impression d'une usine de recyclage de verre ou la base de données des comptes d'une banque privée suisse, les problèmes potentiels issus d'une négligence de la part de l'utilisateur ne seront pas identiques. Cet exemple (certes bidon) illustre le fait qu'il est *crucial* de maîtriser parfaitement *toutes* les implications des actions susceptibles d'être prises par les utilisateurs en matière de sécurité, et le cas échéant, d'en tirer certaines conséquences. On pourrait imaginer dans le cas présent que le client `ssh` de l'administrateur système de la banque suisse refuse de poursuivre le processus de connexion au cas où il ne pourrait contrôler l'authenticité de la clef RSA de son interlocuteur.

NÉGLIGER DE SUIVRE L'ACTUALITÉ

Inévitablement, le monde de la sécurité informatique, et donc de la cryptographie, qu'elle soit théorique ou appliquée, sont des domaines qui évoluent terriblement rapidement : de nouveaux algorithmes apparaissent, d'autres disparaissent, les chercheurs trouvent continuellement de nouvelles attaques, de nouvelles techniques de piratage de logiciel s'imposent (les techniques d'exploitation de *buffer overflow*, bien qu'utilisées par Morris pour écrire son célèbre ver en 1988, ne se sont répandues que depuis la seconde moitié des années 90), de nombreuses personnes découvrent des problèmes de sécurité dans les bibliothèques cryptographiques les plus diverses, les standards évoluent, bref, ce qui est vrai aujourd'hui ne sera plus vrai demain avec une grande probabilité.

Une fois écrite, une application utilisant de la cryptographie se doit (plus que tout autre type d'application) d'être confrontée d'une manière continue à ces évolutions, et le cas échéant, d'être adaptée ou corrigée lorsque les événements l'exigent. Dans le cas contraire, la valeur des services offerts par la cryptographie décroît extrêmement vite. Concrètement, un simple suivi des listes de diffusion dédiées à la sécurité (il en existe des centaines), ainsi que de celles des bibliothèques et des standards utilisés



permet déjà de se maintenir à un niveau d'actualité intéressant, sachant qu'il est difficile (et coûteux en temps), voire utopique de vouloir suivre l'évolution complète au jour le jour du monde de la sécurité informatique.

SOUS-ESTIMER LA DIFFICULTÉ DE LA TÂCHE

Bruce Schneier, l'expert le plus médiatique du monde de la cryptographie, relève ce fait étrange dans un de ces derniers ouvrages : en cryptographie, tout le monde pense en savoir assez pour être capable de concevoir et de construire son propre système, alors qu'on ne laisserait pas la responsabilité de concevoir un réacteur nucléaire à un étudiant en physique, ou une infirmière opérer un patient parce qu'elle prétend savoir comment le guérir. Si l'on enlève le côté potentiellement intéressé de cette remarque, force est de constater qu'elle n'est pas absurde...

En réalité, concevoir et écrire une application sécurisée, ou qui utilise de la cryptographie est, il faudrait constamment le garder à l'esprit, une tâche *très* difficile à mener à bien pour de multiples raisons, certaines ayant été effleurées dans cet article. Des applications conçues et écrites par des spécialistes de la question souffrent régulièrement de problèmes graves. L'exemple récent du problème dans l'implémentation des signatures ElGamal de GnuPG [5] (plus d'un millier de paires de clés ont dû être révoquées dans l'urgence parce qu'il était possible de récupérer la clé secrète à partir de la clé publique correspondante en quelques minutes de calcul sur un simple ordinateur personnel) en est une cruelle illustration.

POUR EN SAVOIR PLUS

Évidemment, le but de cet article n'était pas de viser l'exhaustivité, mais de donner l'envie d'en savoir plus. Pour une première approche, le *Secure-Coding Howto* permet de bien débroussailler le sujet. Une référence dans le monde de Microsoft dans le domaine de la programmation sécurisée (donc pas spécifiquement orientée sur la cryptographie) est le livre [7] de Howard et LeBlanc disponible depuis peu en français. Cet ouvrage donne quelques bonnes ficelles, sans toutefois aller vraiment dans les détails.

Les deux autres ouvrages que nous recommandons ne sont malheureusement pas (encore) traduits en français : *Practical Cryptography* [8], de Niels Ferguson et Bruce Schneier, expose en détails tous les points sensibles que nous avons abordés (et bien d'autres), sous le point de vue (plutôt théorique) du cryptographe.

Enfin, *Secure Programming Cookbook for C and C++* [9] de Viega et Messier va aussi profondément dans les détails et est orienté pratique (en exposant des solutions toutes faites), que ce soit dans les mondes UNIX ou Microsoft.

Ces trois livres sont incontournables pour qui se soucie vraiment d'implémenter de la cryptographie dans les règles de l'art. Enfin, rien ne vaut l'expérience acquise en étudiant le code source de GnuPG, OpenSSL et autres, en écrivant du code soi-même, en essayant de le casser, bref, en faisant des erreurs et en les réparant !

RÉFÉRENCES

- [1] The Open Source toolkit for SSL/TLS, <http://www.openssl.org>
- [2] Microsoft CryptoAPI, http://msdn.microsoft.com/library/en-us/security/security/cryptography_portal.asp
- [3] EGD: The Entropy Gathering Daemon, <http://egd.sourceforge.net/>
- [4] Pascal Junod, "Problèmes d'implémentation de la cryptographie: les attaques par effet de bord", MISC 4, novembre-décembre 2002.
- [5] <http://lists.gnupg.org/pipermail/gnupg-announce/2003q4/000276.html>
- [6] David Wheeler, "Secure Programming for Linux and Unix HOWTO", <http://www.dwheeler.com/secure-programs/>
- [7] Michael Howard, David LeBlanc, "Ecrire du code sécurisé", seconde édition, Microsoft Press, 2003.
- [8] Niels Ferguson, Bruce Schneier, "Practical Cryptography", Wiley, 2003.
- [9] John Viega, Matt Messier, "Secure Programming Cookbook for C and C++", O'Reilly, 2003.
- [10] Robert Erra, "Attaques sur le protocole RSA", MISC 10.
- [11] <http://quiz.ngsec.biz:8080/>

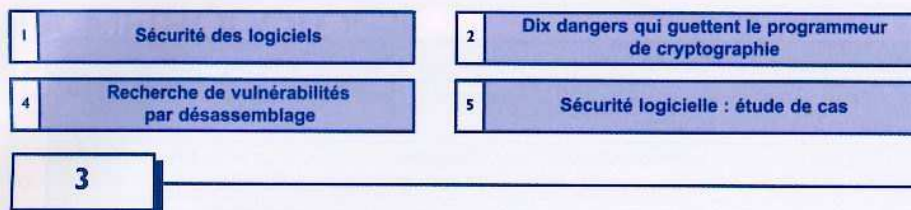
Pascal Junod

<pascal.junod@epfl.ch>

Laboratoire de Sécurité et de Cryptographie (LASEC), École Polytechnique Fédérale de Lausanne

Frédéric Raynal

<pappy@miscmag.com>



Méthodes d'évaluation de sécurité : TCSEC, ITSEC et CC



Après des exemples de techniques de réalisation de logiciels plus sûrs, le dossier sur la sécurité des logiciels continue avec l'étude d'une méthode d'évaluation et de sa genèse.

HISTORIQUE

L'évaluation de logiciels n'est pas une activité très nouvelle. Examinons ensemble l'évolution des critères d'évaluation depuis les vingt dernières années.

TCSEC

Les États-Unis d'Amérique ont ouvert le bal avec les *Trusted Computing System Evaluation Criteria* (TCSEC) dont la première édition a été publiée en 1983 et l'édition finale en 1985 [1] par le *National Computer Security Center* (NCSC). C'est une initiative essentiellement militaire permettant d'évaluer le niveau de confiance dans un système d'exploitation.

Le document TCSEC fait partie d'une série de documents et guides couvrant différents domaines de sécurité informatique. Chaque volume de la série avait une couverture de couleur différente. La série de volume est connue sous le nom de série arc-en-ciel (*rainbow serie*) [2]. Le document TCSEC avait une couverture orange et il est donc aussi très connu sous le nom de livre orange ou *orange book*.

TCSEC définit une liste ordonnée de niveaux d'évaluation et de mécanismes de sécurité fournis. Le tableau suivant donne un très rapide aperçu des différents niveaux (**Tableau 1**). Par exemple, Windows 98 serait de niveau D. Un système Unix de base est de niveau C1 au minimum.

Les TCSEC ont été définis pour l'évaluation de systèmes d'exploitation. Ils ne sont pas adaptés pour l'évaluation d'une application comme une base de données par exemple. De plus, à l'époque, Internet et les réseaux n'étaient pas très développés et TCSEC ne considère qu'une machine seule et en particulier ne considère que son système d'exploitation.

D	Aucune sécurité (échec lors de l'évaluation)
C1	Protection discrétionnaire
C2	C1 avec audit
B1	Protection obligatoire (<i>mandatory</i>)
B2	Protection structurée par un modèle de sécurité
B3	Domaines de sécurité
A1	Conception vérifiée (prouvée)

Tableau 1

ITSEC

De leur côté, plusieurs pays européens (Allemagne, France, Pays Bas et Royaume-Uni) travaillent ensemble à l'écriture de nouveaux critères d'évaluation permettant d'éviter les limitations des TCSEC. Ces critères sont appelés *Information Technology Security Evaluation Criteria* (ITSEC) et sont publiés en 1991 [3].

La grande nouveauté par rapport aux critères TCSEC est que les mécanismes de sécurité à évaluer ne sont plus définis par la méthode d'évaluation mais par celui qui demande l'évaluation en rédigeant une cible d'évaluation ou *Target of Evaluation* (TOE). Cela ajoute beaucoup de souplesse aux critères d'évaluation qui peuvent maintenant être utilisés pour évaluer un système d'exploitation, une application de base de données, un pare-feu, etc. Tous ces produits sont très différents les uns des autres et ont donc une cible d'évaluation très différente.

Comme pour les TCSEC, plusieurs niveaux d'évaluation sont définis (**Tableau 2**). Le niveau d'évaluation correspond au niveau d'assurance que les mécanismes de sécurité décrits dans la cible d'évaluation (TOE) correspondent à ce qui est implanté dans le produit. Comme on peut le voir dans le **Tableau 2**, il s'agit



E0	Ce niveau représente une assurance insuffisante (échec de l'évaluation).
E1	À ce niveau, il doit exister une cible de sécurité et une description informelle de la conception générale de la TOE. Les tests fonctionnels doivent indiquer que la TOE satisfait à sa cible de sécurité.
E2	Outre les exigences du niveau E1, il doit exister une description informelle de la conception détaillée. Les éléments de preuve des tests fonctionnels doivent être évalués. Il doit exister un système de gestion de configuration et un processus approuvé de diffusion.
E3	En plus des exigences du niveau E2, le code source et/ou les schémas descriptifs des matériels correspondant aux mécanismes de sécurité doivent être évalués. Les éléments de preuve des tests de ces mécanismes doivent être évalués.
E4	En plus des exigences du niveau E3, il doit exister un modèle formel sous-jacent de politique de sécurité supportant la cible d'évaluation. Les fonctions dédiées à la sécurité, la conception générale et la conception détaillée doivent être spécifiées en style semi-formel.
E5	En plus des exigences du niveau E4, il doit exister une correspondance étroite entre la conception détaillée et le code source et/ou les schémas descriptifs des matériels.
E6	En plus des exigences du niveau E5, les fonctions dédiées à la sécurité ainsi que la conception générale doivent être spécifiées en style formel de manière cohérente avec le modèle formel sous-jacent de politique de sécurité.

Tableau 2

principalement (mais pas uniquement) de produire de la documentation. Le niveau de détail de la documentation augmente avec le niveau d'évaluation.

CRITÈRES COMMUNS

Pendant que les Européens réalisaient les ITSEC, les États-Unis d'Amérique mettent en œuvre, en 1993, de nouveaux critères : FC (*Federal Criteria for IT security*) et le Canada développe les CTCPEC (*Canadian Trusted Computer Product Evaluation Criteria*). Toutefois, bien que ces critères soient largement utilisés par leurs pays respectifs, ils ne bénéficient d'aucune reconnaissance mutuelle. Un produit évalué selon les critères FC "étatsuniens" doit à nouveau être évalué en Europe selon les critères ITSEC. C'est une grande perte de temps et d'argent.

Face à cette constatation, les différents gouvernements initient, en 1996, le projet Critères Communs, projet visant à fournir un ensemble harmonisé de critères d'évaluation de la sécurité des systèmes d'information. Ayant évolué durant plus de 4 années, et bénéficiant de l'influence de l'ISO (*International Standard Organization*), les Critères Communs Version 2.1 sont reconnus comme norme ISO 15408 depuis juin 1999.



ÉQUIVALENCES ENTRE LES TROIS MÉTHODES

Il n'y a pas d'équivalences entre les trois méthodes TCSEC, ITSEC et CC. Ces trois méthodes sont assez différentes. Il est cependant possible d'établir un tableau grossier de correspondances entre les différents niveaux des trois méthodes (cf **Tableau 3**).

ACCORDS DE RECONNAISSANCE MUTUELLE

L'accord de reconnaissance mutuelle SOG-IS signé en mars 1998 couvre les certificats ITSEC et Critères Communs quels que soient les niveaux d'assurance atteints. Les organismes de certification qualifiés à émettre des certificats, dans le cadre de leurs schémas respectifs, au titre de cet accord sont : le BSI (Allemagne), le CESG (Royaume Uni), la DCSSI (France).

Les pays suivants, signataires de l'accord, reconnaissent les certificats émis par ces organismes qualifiés : Espagne, Finlande, Grèce, Italie, Norvège, Pays-Bas, Portugal, Suède et Suisse.



Les certificats émis dans le cadre de cet accord sont reconnaissables au logo de la figure ci-dessus.

L'accord de reconnaissance mutuelle MRA signé en mai 2000 couvre les certificats exigeant les composants d'assurance compris entre les niveaux EAL1 et EAL4. Les organismes de certification qualifiés pour émettre des certificats, dans le cadre de leurs schémas respectifs, au titre de cet accord sont : le BSI (Allemagne), le CESG (Royaume Uni), le CSE (Canada), la DCSSI (France), le DSD et le GCSB (Australie et Nouvelle-Zélande), le NIST et la NSA (États-Unis).

Les pays suivants, signataires de l'accord, reconnaissent les certificats émis par ces organismes certifiés : Autriche, Espagne, Finlande, Grèce, Israël, Italie, Norvège, Pays-Bas, Suède.



Les certificats émis dans le cadre de cet accord sont reconnaissables au logo ci-contre.

TCSEC	D	-	C1	C2	B1	B2	B3	A1
ITSEC	E0	-	E1	E2	E3	E4	E5	E6
Common Criteria	-	EAL 1	EAL 2	EAL 3	EAL 4	EAL 5	EAL 6	EAL 7

Tableau 3



LA MÉTHODOLOGIE CRITÈRES COMMUNS

La méthodologie Critères Communs est décrite dans plusieurs documents.

- ⊙ Trois documents descriptifs des critères [4] :
 - Motivations et concepts de base de Critères Communs ;
 - Liste des exigences fonctionnelles de sécurité ;
 - Liste des exigences d'assurance de sécurité.
- ⊙ Deux documents descriptifs de la méthode d'évaluation [5] :
 - Introduction et modèle général ;
 - Méthode d'évaluation.
- ⊙ Un document d'aide à l'interprétation :
 - CCIMB
(*Common Criteria Interpretation Management Board*).

EXIGENCES FONCTIONNELLES ET D'ASSURANCE

Les Critères Communs définissent deux types d'exigences :

- Exigences fonctionnelles : ce que fait le produit (**Tableau 4**).
- L'implantation de ces exigences donne les fonctions de sécurité. C'est une liste des spécifications sécuritaires.

→ Exigences d'assurance : garantie que le but est atteint (**Tableau 5**). Assure la correction de l'implantation. C'est une liste de contraintes qui permet d'évaluer le niveau de sécurité que l'on veut atteindre avec l'implantation fournie.

NIVEAUX D'ÉVALUATION : EAL

Comme pour les autres critères d'évaluation, les CC (Critères Communs ou *Common Criteria*) établissent plusieurs niveaux (**Tableau 6**).

EAL 1	Produit testé fonctionnellement
EAL 2	Produit testé structurellement
EAL 3	Produit testé et vérifié méthodiquement
EAL 4	Produit conçu, testé et vérifié méthodiquement
EAL 5	Produit conçu et testé de façon semi-formelle
EAL 6	Produit conçu, testé et vérifié de façon semi-formelle
EAL 7	Produit conçu, testé et vérifié de façon formelle

Tableau 6

L'EAL (*Evaluation Assurance Level*) est le niveau d'assurance de l'évaluation. Comme pour les ITSEC, ce qui est mesuré n'est pas le niveau de sécurité, mais le niveau d'assurance que le produit

CLASSES	OBJECTIFS
FAU Audit de la sécurité	Reconnaissance, enregistrement, stockage et analyse d'informations relatives à la sécurité.
FCO Communication	Garanties de l'identité d'une partie participant à un échange de données, de la non-répudiation d'émission et de la non-répudiation de réception de données.
FCS Support cryptographique	Fonctionnalités cryptographiques à des fins de communication, d'identification, d'authentification et de séparation de données.
FDP Protection des données de l'utilisateur	Protection des données de l'utilisateur durant leur importation, leur exportation et leur stockage au sein du produit.
FIA Identification et authentification	Garantie de l'identification non ambiguë des utilisateurs autorisés par vérification d'identité, détermination des droits d'interaction avec le produit et définition des attributs de sécurité.
FMT Gestion de la sécurité	Gestion des attributs de sécurité, des données et des fonctions de la TSF (<i>TOE Security Functionalities</i> , ensemble des fonctions de sécurité de la TOE).
FPR Protection de la vie privée	Protection de l'utilisateur contre la connaissance ou l'utilisation induite de son identité, couverture de l'anonymat, de la possibilité d'agir sous un pseudonyme, de l'impossibilité d'établir un lien entre l'utilisateur et les opérations effectuées et de la non-observabilité de l'utilisation des ressources.
FPT Protection des fonctions de sécurité de la TOE	Protection des données de la TSF et couverture de l'intégrité et de la gestion des mécanismes et des données de la TSF.
FRU Utilisation des ressources	Couverture de la disponibilité des ressources, des capacités de stockage, des tolérances aux pannes, de la priorité des services et de l'allocation des ressources.
FTA Accès à la TOE	Contrôle de l'établissement d'une session utilisateur.
FTP Chemin et canaux de confiance	Couverture des chemins de communication de confiance entre l'utilisateur et la TSF, et entre plusieurs TSF.

Tableau 4



CLASSES	OBJECTIFS
APE Évaluation d'un profil de protection	Démontrer qu'un Profil de Protection (PP ou <i>Protection Profil</i>) est complet, cohérent et techniquement correct.
ASE Évaluation d'une cible de sécurité	Démontrer qu'une cible de sécurité (ST ou <i>Security Target</i>) est complète, cohérente et techniquement correcte.
ACM Gestion de configuration	Préservation de l'intégrité de la TOE et de la documentation associée et garantie que la TOE évaluée est identique à la TOE distribuée.
ADO Livraison et exploitation	Couverture des mesures, procédures et normes pour une livraison, une installation et une utilisation sûre de la TOE.
ADV Développement	Couverture du raffinement de la TSF à partir des spécifications définies dans la ST jusqu'à la phase d'implémentation et de la traçabilité des exigences pour tous les niveaux de représentation de la conception.
AGD Guides	Couverture des guides nécessaires à une utilisation opérationnelle sûre de la TOE par les utilisateurs et les administrateurs.
ALC Support au cycle de vie	Couverture du cycle de vie du produit, des outils et techniques de développement, de la sécurité de développement et de la correction des erreurs découvertes par les utilisateurs.
ATE Tests	Démonstration que la TOE satisfait à ses exigences fonctionnelles.
AVA Estimation des vulnérabilités	Couverture de la recherche et de l'analyse de vulnérabilités exploitables qui pourraient être introduites lors du développement, de l'exploitation ou à cause d'une utilisation impropre ou d'une configuration incorrecte de la TOE.
AMA Maintenance de l'assurance	Garantie qu'une TOE certifiée continuera à satisfaire à sa cible de sécurité après que des changements aient été effectués sur la TOE ou sur son environnement.

Tableau 5

respecte ses spécifications. C'est une nuance subtile qui fait que le niveau de sécurité n'est pas uniquement lié au niveau d'EAL mais aussi, et surtout, au choix de la cible d'évaluation.

Le niveau d'évaluation atteint par le produit peut aussi se trouver entre deux niveaux. Par exemple, il peut être EAL 4+ ou EAL 4 augmenté. Cela signifie que le niveau est EAL 4 et qu'au moins une (et en général une seule) des exigences du niveau supérieur est satisfaite.

CIBLE D'ÉVALUATION

La cible d'évaluation (*Target Of Evaluation* ou TOE) est l'un des points les plus importants dans une évaluation suivant les Critères Communs. C'est la TOE qui spécifie ce qui est évalué dans le produit.

La TOE est l'objet de l'évaluation. Elle est composée d'un produit ou d'un système IT (*Information Technology*) ainsi que de la documentation associée pour l'administrateur et pour l'utilisateur.

CIBLE DE SÉCURITÉ

La cible de sécurité (*Security Target* ou ST) contient un ensemble d'exigences de sécurité qui peut être constitué par référence à un PP (profil de protection), ou directement par référence à des composants fonctionnels ou d'assurance des CC (critères communs), ou encore formulés explicitement. Une ST permet l'expression d'exigences de sécurité pour une TOE spécifique

dont l'évaluation montre qu'elles sont utiles et efficaces pour satisfaire aux objectifs identifiés.

Une ST contient les spécifications globales de la TOE ainsi que les exigences et les objectifs de sécurité et leurs argumentaires respectifs. Une ST constitue la base de l'accord entre toutes les parties sur la sécurité offerte par la TOE.

PROFIL DE PROTECTION

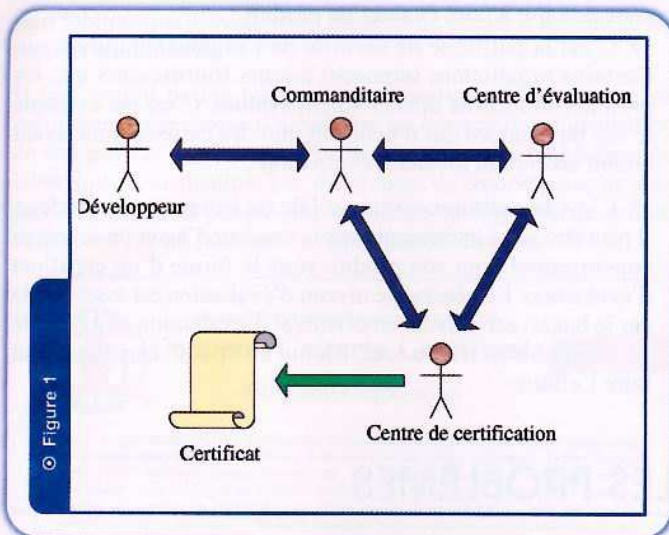
Le profil de protection (*Protection Profile* ou PP) contient un ensemble d'exigences de sécurité qui sont soit issues des CC, soit formulées explicitement. Il devrait inclure un EAL (éventuellement augmenté par des composants d'assurance supplémentaires). Le PP permet, de façon indépendante de l'implémentation, l'expression d'exigences de sécurité pour un ensemble de TOE qui se conformeront complètement à un ensemble d'objectifs de sécurité. Un PP est prévu pour être réutilisable et pour définir des exigences pour la TOE qui sont connues pour être utiles et efficaces pour répondre aux objectifs identifiés, à la fois pour les fonctions et pour l'assurance. Un PP contient également l'argumentaire pour les objectifs de sécurité et les exigences de sécurité.

Un PP pourrait être développé par des communautés d'utilisateurs, des développeurs de produits IT, ou d'autres parties intéressées par la définition d'un tel ensemble commun d'exigences. Un PP donne aux utilisateurs un moyen de se référer à un ensemble spécifique de besoins de sécurité et facilite une évaluation future par rapport à ces besoins.



LES ACTEURS

Les différents acteurs intervenant lors d'une évaluation Critères Communs sont décrits sur la **figure 1** ci-dessous.



Développeur du produit

Le développeur est celui qui développe et réalise le produit. Le développeur rédige les différents documents nécessaires à l'évaluation Critères Communs.

Commanditaire

Le commanditaire est celui qui demande et paye l'évaluation. Il arrive souvent que le commanditaire et le développeur soient la même entité.

Organisme de certification

L'organisme de certification est l'organisme légalement autorisé à délivrer le certificat. Il y a un organisme de certification par pays. En France, il s'agit de la Direction Centrale de la Sécurité des Systèmes d'Information (DCSSI).

L'organisme de certification est aussi en charge de contrôler et d'habiller les centres d'évaluation.

Centre d'évaluation

Les Centres d'Évaluation de la Sécurité des Technologies de l'Information (CESTI) réalisent les évaluations : ils agissent en tant que tierce partie indépendante des développeurs de produits et des commanditaires. Ils doivent être agréés par l'organisme de certification.

Il existe plusieurs CESTI en France. La liste des CESTI agréés est disponible sur le site de la DCSSI [6]. Tous les CESTI ne sont pas équivalents. Ils ont des domaines de prédilection : "informatique et réseaux" ou "composants électroniques et logiciels embarqués".

La DCSSI et le Centre d'Électronique de l'Armement (CELAR) disposent chacun d'un centre d'évaluation agréé d'office. Le centre d'évaluation de la DCSSI peut procéder à l'évaluation des produits spécialement développés ou modifiés par les ministères autres que celui de la Défense en vue de répondre à leurs besoins propres.

Le centre d'évaluation du CELAR effectue l'évaluation des produits et systèmes développés ou utilisés par le ministère de la Défense.

ANALYSE DES VULNÉRABILITÉS

Parmi toutes les classes d'assurance, une classe me semble intéressante : la classe d'exigence d'assurance AVA (Estimation des vulnérabilités).

Elle se divise en quatre familles :

- AVA_CCA : analyse des canaux cachés ;
- AVA_MSU : utilisation impropre ;
- AVA_SOF : résistance des fonctions de sécurité de la TOE ;
- AVA_VLA : analyse des vulnérabilités.

Chaque famille se redécompose ensuite en composants. Par exemple AVA_VLA se décompose en 4 composants : AVA_VLA.1 à AVA_VLA.4. Le niveau d'évaluation atteint (EAL1 à EAL7) est fonction de la rigueur de l'analyse de vulnérabilité faite par le développeur et l'évaluateur (décrite dans les composants AVA_VLA.1 à AVA_VLA.4).

Pour chaque niveau de composants (.1 à .4), les Critères Communs définissent :

- une liste d'actions pour le développeur (notées AVA_VLA.x.yD) ;
- une liste de documents à fournir (notés AVA_VLA.x.yC) ;
- et une liste d'actions pour l'évaluateur (notées AVA_VLA.x.yE).

Une analyse de vulnérabilités est effectuée par le développeur afin de vérifier la présence de vulnérabilités de sécurité et devrait au moins prendre en compte le contenu de toutes les fournitures de la TOE, y compris la ST, pour le niveau d'assurance de l'évaluation visé. Le développeur doit documenter les caractéristiques des vulnérabilités identifiées afin de permettre à l'évaluateur d'utiliser cette information, si elle se révèle utile, pour appuyer son analyse indépendante de vulnérabilités.

Le but de l'analyse du développeur est de confirmer qu'aucune vulnérabilité de sécurité identifiée ne peut être exploitée dans l'environnement prévu pour la TOE et que la TOE résiste aux attaques de pénétration évidentes.

L'analyse de vulnérabilités indépendantes va au-delà des vulnérabilités identifiées par le développeur. Le principal but de l'analyse de l'évaluateur est de déterminer si la TOE est résistante aux attaques de pénétration effectuées par un attaquant ayant un potentiel d'attaque élémentaire (pour AVA_VLA.2), moyen (pour AVA_VLA.3) ou élevé (pour AVA_VLA.4).

Pour atteindre ce but, l'évaluateur fait d'abord une estimation du caractère exploitable de toutes les vulnérabilités identifiées. Cela



est accompli en effectuant des tests de pénétration. L'évaluateur devrait assumer le rôle d'un attaquant doté d'un potentiel d'attaque élémentaire (pour AVA_VLA.2), moyen (pour AVA_VLA.3) ou élevé (pour AVA_VLA.4) quand il essaye de pénétrer la TOE. Toute exploitation des vulnérabilités par un tel attaquant devrait être considérée par l'évaluateur comme des "attaques de pénétration évidentes" (par rapport aux éléments de AVA_VLA.*.2C) dans le contexte des composants AVA_VLA.2 à AVA_VLA.4.

Quelques exemples

⊙ L'AVA_VLA.2.5E est la 5^{ème} action de l'évaluateur pour le niveau 2 du composant AVA_VLA. Cette action est :

"L'évaluateur doit déterminer si la TOE est résistante aux attaques de pénétration effectuées par un attaquant ayant un potentiel d'attaque élémentaire."

⊙ L'AVA_VLA.1.1D, qui est la première action de développeur pour le niveau 1 du composant AVA_VLA, dit :

"Le développeur doit effectuer et documenter une analyse des fournitures de la TOE pour rechercher les moyens évidents par lesquels un utilisateur peut violer la TSP (*TOE Security Policy* ou Politique de sécurité de la TOE)."

⊙ L'AVA_VLA.1.2D est :

"Le développeur doit documenter les caractéristiques des vulnérabilités évidentes."

⊙ L'AVA_VLA.1.1C (action de documentation) est :

"La documentation doit montrer, pour toutes les vulnérabilités identifiées, que la vulnérabilité ne peut pas être exploitée dans l'environnement prévu pour la TOE."

Vous pouvez trouver la liste des actions pour toutes les classes d'exigence d'assurance dans [7] (version française dans un PDF ou version en ligne mais en anglais).

Intérêt

Cette classe d'exigence d'assurance est importante parce qu'elle oblige le développeur à réfléchir aux menaces qui pèsent sur son produit, et pour chacune des menaces, à justifier pourquoi la menace n'est pas réalisable en fonction des procédés techniques, de gestion de la TOE, ou organisationnels.

COÛTS D'UNE ÉVALUATION

Une évaluation Critères Communs est relativement chère. Le coût externe (facturation par le laboratoire d'évaluation) va de 30 K€ pour une évaluation EAL 1 à 400 K€ pour une évaluation EAL 7. Il faut aussi ajouter le coût du temps passé par les développeurs à rédiger les divers documents et le temps passé à suivre l'évaluation en relation avec le laboratoire d'évaluation.

Ces coûts sont donc loin d'être négligeables et ont un impact certain sur le coût final du produit. Les organismes de certification sont des organismes d'État et délivrent les certificats gratuitement (ce sont nos impôts qui payent donc).

POURQUOI

FAIRE UNE ÉVALUATION DE SÉCURITÉ ?

Il existe principalement deux raisons qui poussent un commanditaire à faire évaluer un produit :

→ C'est la politique de sécurité de l'organisme utilisateur. Certains organismes imposent à leurs fournisseurs que les produits de sécurité utilisés soient évalués. C'est par exemple le cas de banques qui n'achètent que des cartes à puce ayant atteint un certain niveau d'évaluation ;

→ C'est la politique commerciale de l'organisme vendeur. Il peut être aussi intéressant pour le vendeur d'avoir un avantage concurrentiel pour son produit sous la forme d'un certificat d'évaluation. En général, le niveau d'évaluation est assez faible car le but ici est d'avoir un certificat d'évaluation alors que le concurrent n'a rien du tout. Même un "petit" certificat peut faire l'affaire.

LES PROBLÈMES

Après cette présentation de la méthodologie d'évaluation Critères Communs et de son historique, voyons les problèmes et limites de cette méthode.

ATTENTION À LA CIBLE D'ÉVALUATION

La cible d'évaluation est définie par le client (le commanditaire). C'est lui qui définit le périmètre de ce qui sera évalué ainsi que le niveau d'évaluation qu'il souhaite obtenir.

Le problème est que la cible peut être n'importe quoi, y compris une toute petite partie seulement du produit. Il est alors relativement facile et peu coûteux d'évaluer une toute petite partie à un niveau élevé. Lorsqu'on vous dit que le produit A est évalué, regardez bien quelle est la cible de sécurité utilisée pour l'évaluation.

La cible d'évaluation étant un document à part entière, il n'est pas inclus dans le certificat final délivré par l'autorité de certification. Le certificat ne contient que le nom commercial du produit (ou le nom que le commanditaire veut indiquer sur le certificat) et le niveau de l'évaluation.

ATTENTION AU CHOIX

DU LABORATOIRE D'ÉVALUATION

Tous les laboratoires d'évaluation n'ont pas les mêmes compétences. Par exemple, si le produit du commanditaire est un peu léger sur certains aspects de sécurité, il est intéressant, pour le commanditaire, de faire évaluer ce produit par un laboratoire moins expert sur les points faibles du produit. Le commanditaire espère alors que le laboratoire ne trouvera pas les faiblesses du produit. Grâce aux accords de reconnaissances mutuelles, un certificat issu de l'évaluation par un laboratoire un peu faible a la même valeur que le même certificat issu de



l'évaluation par un laboratoire expert dans le produit évalué et dans les techniques d'attaques sur ce type de produits. On trouve ainsi des produits d'un fabricant français évalués par un laboratoire allemand ou l'inverse. Bien sûr, le choix du laboratoire d'évaluation sera officiellement justifié par le fait que c'était le seul laboratoire d'évaluation disponible ou qu'il a le meilleur rapport qualité/prix.

Il faut quand même nuancer ma remarque. L'organisme de certification a la charge d'accréditer les laboratoires d'évaluation de son pays et donc de vérifier les compétences techniques du laboratoire d'évaluation. Les différences de compétences ne sont pas flagrantes et jouent sur quelques techniques de pointe maîtrisées par un laboratoire et pas par un autre.

Bien sûr, Windows 2000 n'est pas le seul produit évalué qui a encore des trous de sécurité. Il faut juste ne pas croire qu'un produit évalué est plus sûr qu'un produit non évalué. Il faut aussi noter qu'une évaluation Critères Communs ne prend pas en compte la vitesse de réaction du développeur pour corriger un problème de sécurité. Ce critère est pourtant important quand il faut choisir une solution informatique.

RENSEIGNEMENT ET INTELLIGENCE

Contrôler les évaluations et les laboratoires d'évaluation est un bon moyen pour les organismes de certification, et indirectement pour les "services secrets" du pays, d'apprendre beaucoup de détails très intéressants sur les produits soumis à la certification.

La liste des vulnérabilités étant fournie par le développeur lui-même, il est plus facile de les exploiter en sachant où elles se trouvent et comment faire pour réussir. Les niveaux élevés d'évaluation (à partir d'EAL4) impliquent aussi de donner le code source au laboratoire d'évaluation. La vulnérabilité est jugée non exploitable par le laboratoire d'évaluation car les moyens dont disposent l'attaquant (moyens techniques et connaissances) sont jugés insuffisants. Mais les "services secrets" ont des moyens relativement plus importants que l'attaquant dans son garage.

Il est d'ailleurs intéressant de noter que l'accord MRA de reconnaissance mutuelle entre les pays européens et les pays nord-américains (USA et Canada) se limite aux niveaux EAL1 à EAL4. Un commanditaire désirant évaluer son produit à un niveau supérieur devra faire une évaluation en Europe et une autre en Amérique du Nord. Peut-être est-ce pour que les "services secrets" étatsuniens puissent avoir toutes les informations sur un système européen avant qu'il ne soit vendu comme système critique aux services gouvernementaux.

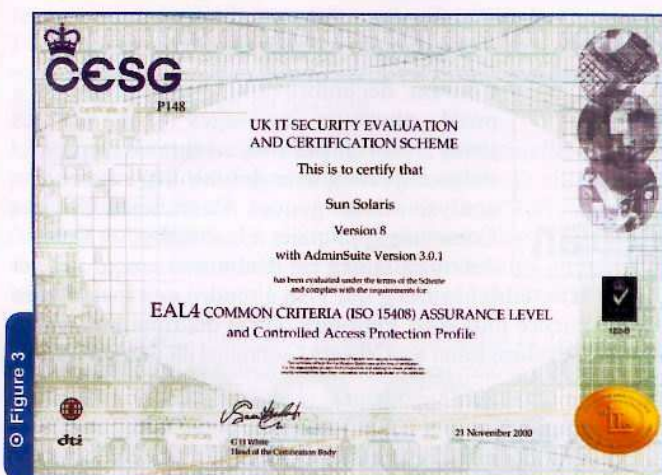
La même remarque s'applique à un système étatsunien qui serait vendu en Europe à un client exigeant un niveau de certification élevé.

DURÉE DE VIE D'UN CERTIFICAT

La durée de vie d'un certificat est très courte. Le certificat est en effet périmé le jour de sa publication. Cela est dû au fait que de nouvelles attaques et de nouvelles failles sont découvertes chaque jour. Donc, un produit évalué un jour J peut ne plus passer la même évaluation avec le même niveau le jour J+1. Il est donc nécessaire de repasser régulièrement l'évaluation. Plutôt que de recommencer toute la procédure à chaque réévaluation, il existe une procédure simplifiée dite de maintenance.

LES SOLUTIONS

Pour pouvoir comparer les évaluations de deux produits concurrents, le plus simple est que ces deux produits utilisent le même profil de protection. Il existe par exemple un profil de protection "Firewall à exigences élevées" [10] dont le commanditaire est la Délégation Générale pour l'Armement



ARGUMENT MARKETING

Un certificat Critères Communs est souvent un argument marketing. Par exemple, Windows 2000 a été évalué au niveau EAL4 augmenté [8] (certificat en figure 2). Cela n'a pas empêché les failles sérieuses dans le service RPC, qui permettent d'exécuter du code malveillant avec les privilèges système ainsi que d'autres problèmes [9].



(DGA). Dans ce cas, il est clair que le client, la DGA, impose que les fournisseurs potentiels de firewall fassent évaluer leurs produits selon ce profil de protection. Ça permet d'éviter que chaque fournisseur utilise une cible d'évaluation différente, et donc que les résultats des évaluations soient incomparables entre eux.

Cependant, certains clients n'utilisent pas les évaluations Critères Communs, même si les produits sont évalués par rapport à un profil de protection. Ces clients sont en particulier VISA et MasterCard, qui ont leurs propres critères, leurs propres méthodes d'évaluation et des laboratoires d'évaluation accrédités par eux.

LES ÉVALUATIONS DE GNU/LINUX ET AUTRES

Quasiment tous les systèmes d'exploitation propriétaires actuels ont un certificat d'évaluation Critères Communs. Une évaluation étant relativement chère, l'évaluation d'un projet collaboratif comme GNU/Linux est difficilement imaginable. Cependant, certains acteurs de GNU/Linux sont très gros et jugent stratégique l'obtention d'un certificat Critères Communs.

SUSE



C'est ce qu'ont fait SuSE et IBM en évaluant "SUSE LINUX Enterprise Server 8" sur une machine "IBM eServer xSeries" [11] en obtenant un certificat EAL2+ sur une cible de sécurité spécifique (et pas sur un profil de protection). La cible de sécurité ainsi que le rapport d'évaluation sont disponibles [11] sur le site du BSI : l'organisme de certification allemand.

L'histoire de l'évaluation, les justifications et les problèmes ont été présentés lors de la quatrième conférence ICCC (*International Common Criteria Conference*). Les transparents sont disponibles [12] et instructifs sur l'état de Linux dans une évaluation. Les principaux problèmes de Linux et des logiciels libres en général sont le manque de documentation de conception, de documentation utilisateur ou que la documentation n'est plus à jour et que les processus d'écriture et de gestion du code source et des documentations ne sont, en général, pas formalisés.

Certaines très bonnes documentations existent mais ne sont pas toujours orientées sur la sécurité et/ou ne sont pas spécifiques à une distribution particulière (SuSE dans notre exemple). Les documentations manquantes pour l'évaluation ont été rédigées et ont donc profité à la communauté.

Dernière minute : IBM et SuSE Linux (maintenant propriété de Novell) ont obtenu un certificat EAL3+ sur le profil de protection CAPP (*Controlled Access Protection Profile*) pour SUSE LINUX Enterprise Server 8 avec Service Pack 3 sur IBM eServers. Il s'agit du même profil de protection que celui utilisé par Microsoft pour l'évaluation dont je parlais un peu plus haut. L'annonce [17] est très claire sur le fait que l'obtention du certificat permet à Novell/SuSE et IBM de vendre leurs produits à l'armée étatsunienne :

"Certification under Common Criteria is a requirement for security related products in our environment," said William Wolf, U.S. Navy, Space & Naval Warfare Systems Center, San Diego. "We are encouraged by EAL 3 certification for Linux, as new doors will open to build flexible, cost effective solutions for our end users."

Qui peut se traduire par :

"La certification par les Critères Communs est une exigence pour les produits liés à la sécurité dans notre environnement" a dit William Wolf, U.S. Navy, Space & Naval Warfare Systems Center, San Diego. "Nous sommes encouragés par la certification EAL 3 pour Linux car de nouvelles portes vont s'ouvrir pour construire des solutions flexibles et peu chères pour nos utilisateurs."

RED HAT



De son côté, Oracle reçoit des demandes de certains de ses gros clients (comme le département de la défense étatsunien) de pouvoir utiliser le logiciel Oracle sur le système GNU/Linux. Mais pour que le DoD puisse utiliser Oracle sur GNU/Linux, il faut que le produit soit évalué selon les Critères Communs. Oracle a déjà fait évaluer son produit sur Windows et Solaris, mais les clients veulent du GNU/Linux. Oracle est donc le commanditaire d'une évaluation de Red Hat Enterprise Linux 3. Ensuite, Oracle pourra faire évaluer son produit Oracle 9i Release 2 sur le Red Hat précédemment évalué [13]. Le budget de cette évaluation de Red Hat est de 1 million de dollars pour une évaluation sur 10 mois.

DEBIAN



Courant décembre 2003, une personne a posté plusieurs messages [14] sur les listes developers-l@adamantix.org et debian-devel@lists.debian.org avec une analyse des exigences d'assurance Critères Communs appliquées à la distribution Debian. Adamantix.org est le nouveau nom du site et groupe trusteddebian.org qui vise à rendre une installation Debian encore plus sûre. Peut-être que la distribution Debian sera aussi évaluée selon les Critères Communs un jour prochain ?

TRUSTEDBSD



Un effort commun financé par la DARPA (*Defence Advanced Research Projects Agency*), la NSA (*National Security Agency*) et quelques autres entreprises, appelé TrustedBSD [15], a pour but d'ajouter des extensions de sécurité à FreeBSD pour passer une évaluation Critères Communs. La majorité des développements effectués dans le cadre de TrustedBSD sont destinés à être intégrés dans FreeBSD.



CONCLUSION

Une évaluation selon les Critères Communs coûte (très) chère et n'est pas forcément (très) utile. Elle est indispensable si un client impose que le produit soit évalué avec un niveau suffisant pour un profil de protection donné. Cela peut être le cas de certains grands comptes ou clients particulièrement exigeants et qui ont des moyens financiers (comme les armées par exemple).

Il faut garder à l'esprit qu'une évaluation Critères Communs n'évalue pas la sécurité du produit, mais évalue l'assurance que le produit réalise correctement les fonctions de sécurité décrites dans la documentation. L'utilisation de profils de protection permet d'imposer à différents fournisseurs un certain niveau de services de sécurité, et donc d'éviter qu'un produit obtienne un niveau d'évaluation élevé mais limité à une ou deux fonctions de sécurité.

Il faut aussi savoir que le niveau atteint n'est pas le maximum que le produit peut atteindre mais le niveau que le commanditaire a voulu atteindre en fonction de ses besoins et de ses moyens financiers. Un produit évalué au niveau EAL2 n'échouera pas forcément à une évaluation EAL3 par exemple. C'est juste que le commanditaire n'a pas voulu atteindre le niveau EAL3 parce que ça impliquerait plus de documentations et de travail pour le développeur.

La méthodologie Critères Communs n'est pas inintéressante. Même si vous n'avez pas besoin d'obtenir un certificat, vous pouvez essayer de l'appliquer à votre produit et vous servir de la méthodologie comme d'une *check list* des points à vérifier et des documentations à écrire. Cela ne peut que profiter aux clients/utilisateurs.

Je ne suis pas expert en évaluation selon les Critères Communs. J'ai aussi omis certains points et fait quelques raccourcis pour ne pas occuper tout un numéro entier de MISC. J'espère que les experts ne m'en voudront pas trop.

L'interprétation des "problèmes" m'est personnelle. Les avis peuvent être différents pour d'autres personnes. D'ailleurs, rien ne vous empêche de donner votre avis en m'écrivant ou en écrivant au rédacteur en chef. Ce sera l'occasion d'ouvrir une section "courrier des lecteurs" dans les prochains numéros de MISC.

Si le sujet vous intéresse, je vous conseille la lecture des documents [5, 6] et autres documents décrivant la méthode. Ces documents de spécification de la méthode Critères Communs sur le "serveur thématique de la sécurité des systèmes d'information" (serveur Web de la DCSSI) sont en français et donc beaucoup plus agréables à lire que leur version anglophone. Il existe même une conférence internationale sur les Critères Communs [16]. C'est lors de cette conférence qu'ont été présentés les résultats sur l'évaluation de SuSE Linux par exemple.

Georges Bart

georges.bart@free.fr

Références

- [1] *Trusted Computing Systems Evaluation Criteria*, National Computer Security Center, DOD 5200.28-STD, décembre 1985, <http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html>
- [2] *Rainbow Series Library*, National Computer Security Center, <http://www.radium.ncsc.mil/tpep/library/rainbow/>
- [3] *Critères d'évaluation de la sécurité des systèmes informatiques (ITSEC)*, version 1.2, juin 1991, <http://www.ssi.gouv.fr/fr/confiance/documents/itsec.pdf>
- [4] *Evaluation - Certification, Critères Communs version 2.1*, août 1999, <http://www.ssi.gouv.fr/fr/confiance/cc21f.html>
- [5] *Evaluation - Certification, Critères Communs version 2.1*, août 1999, <http://www.ssi.gouv.fr/fr/confiance/cem.html>
- [6] *Evaluation - Certification, Les Centres d'Evaluation de la Sécurité des Technologies de l'Information*, <http://www.ssi.gouv.fr/fr/confiance/cesti.html>
- [7] *Critères Communs pour l'évaluation de la sécurité des technologies de l'information, Partie 3 : Exigences d'assurance de sécurité*, Août 1999, Version 2.1, <http://www.ssi.gouv.fr/fr/confiance/documents/p3.zip>, <http://csrc.nist.gov/cc/Documents/CC%20v2.1%20-%20HTML/PART3/Part314-4.htm>
- [8] *Microsoft Windows 2000 Awarded Common Criteria Certification*, "Achieves Highest Level of Security Evaluation for the Broadest Set of Real-World Scenarios", 29 octobre 2002, <http://www.microsoft.com/presspass/press/2002/Oct02/10-29CommonCriteriaPR.asp>
- [9] *CERT Summary CS-2003-04*, 24 novembre 2003, <http://www.cert.org/summaries/CS-2003-04.html>
- [10] *Firewall à exigences élevées v2.2*, PP/9905, septembre 1998, <http://www.ssi.gouv.fr/fr/confiance/documents/PP9905.pdf>
- [11] *IBM and SUSE LINUX Earn First Security Certification of Linux*, 5 août 2003, http://www.suse.com/us/company/press/press_releases/archive03/security_certification.html, <http://www.bsi.bund.de/zertifiz/zert/reporte.htm>
- [12] *Evaluating and Certifying Open Source - The Linux Experience*, Kittur Shankar (IBM, USA), Helmut Kurth (Atsec, Germany), septembre 2003, http://www.atsec.com/downloads/pdf/Shankar_Kurth_Stockholm.pdf
- [13] *Red Hat, Oracle to certify Linux for gov't*, 12 février 2003, <http://news.com.com/2100-1001-984383.html>
- [14] *Assurance measures: ADO*, <http://lists.debian.org/debian-devel/2003/debian-devel-200312/msg00158.html>, *Assurance measures: ADV*, <http://lists.debian.org/debian-devel/2003/debian-devel-200312/msg00185.html>, etc.
- [15] *TrustedBSD*, <http://www.trustedbsd.org/>
- [16] *The 4th International Common Criteria Conference*, Septembre 7-9, 2003, Folkets Hus, Stockholm, SWEDEN <http://www.icconference.com/>
- [17] *IBM and SUSE LINUX achieve a higher level of Linux security certification across all IBM eServer systems*, 21 janvier 2004, http://www.suse.com/us/company/press/press_releases/archive04/eal3.html



1 Sécurité des logiciels

2 Dix dangers qui guettent le programmeur de cryptographie

3 Méthodes d'évaluation de sécurité : TCSEC, ITSEC et CC

5 Sécurité logicielle : étude de cas

4

Recherche de vulnérabilités par désassemblage



Dans les articles précédents, je présentais les techniques de Reverse Engineering comme un moyen de lutter contre les risques d'infections virales. Cet article se focalise lui sur la recherche de vulnérabilités par désassemblage et présente les techniques d'analyse de code, permettant la découverte de vulnérabilités dans des applications dont le code source n'est pas disponible. Pour les problèmes classiques, nous verrons qu'il n'est pas vraiment plus difficile de détecter ses problèmes dans un désassemblage que dans un code source.

INTRODUCTION

Dans cet article, je commencerai tout d'abord par présenter les méthodes d'analyse de binaire en *closed source*. Je montrerai les avantages et défauts de chaque méthode avant d'illustrer une des méthodes retenues, le désassemblage.

Pour une meilleure approche, je rappellerai les bases en assembleur, les conventions d'appels classiques et leur reconnaissance dans un exécutable désassemblé, la reconnaissance d'arguments et variables locales, tout ce qui permettra ensuite d'étudier des exemples simples, illustrant les méthodologies présentées dans l'article.

Quelques applications réelles ayant des vulnérabilités dues à des erreurs de programmation illustreront ensuite les explications précédentes.

Enfin, la présentation d'un script permettant de trouver dans un désassemblage des vulnérabilités de type débordement de *buffer*.

LES DIFFÉRENTES MÉTHODES D'ANALYSE EN "CLOSED SOURCE"

LE STRESS TESTING / FUZZLING

Le "Fuzzling", utilisé lors de la recherche de vulnérabilités sur des applications en Closed Source, consiste à générer de façon automatique, des chaînes de caractères de tailles importantes et croissantes. Ces chaînes sont ensuite utilisées pour tester différents protocoles (FTP, HTTP, SMTP, etc.), et pour permettre de découvrir rapidement si les applications testées contiennent des débordements de *buffer*.

L'avantage de cette méthode est d'avoir un aperçu rapide sur la fiabilité d'une "application serveur" et cela, de façon automatisée sans avoir à recourir à un personnel hautement qualifié.

Toutefois, cette méthode possède quelques inconvénients majeurs. Tout d'abord, le protocole à tester doit être connu. S'il s'agit d'un protocole propriétaire ou exotique, il est alors impossible d'effectuer le Stress Testing sans analyse plus poussée. Par ailleurs, un problème plus important persiste. Cette méthode ne permet pas de trouver les problèmes complexes (ex. : bogues liés à la manipulation de pointeurs), et il est donc possible que des failles ne soient pas mises en évidence par ces tests. Pour pallier ces inconvénients, il faut alors avoir recours au Reverse Engineering.



REVERSE ENGINEERING : AUDIT MANUEL

A la manière d'un audit de code classique, l'auditeur examine le listing assembleur du programme, à la recherche d'appels de fonctions réputées pour leurs problèmes de sécurité. Il orientera ses recherches vers les parties de code traitant les données dynamiques, pouvant être entrées par un utilisateur. Pour un serveur FTP par exemple, il examinera le code traitant des commandes du protocole (USER, PASS, etc.) à la recherche de code pouvant mener à un débordement de buffer. Bien que cette méthode demande excessivement de temps et un auditeur qualifié, sans parler de la difficulté à analyser une application de taille importante, le point positif de cette méthode est que les problèmes les plus complexes peuvent être trouvés.

REVERSE ENGINEERING : AUDIT SEMI-AUTOMATISÉ - RECHERCHE DE CODE SUSPECT

Une autre méthode consiste à repérer dans le listing assembleur, des constructions suspectes, comme des appels mal formés ou des fonctions dites non sûres, etc., pouvant entraîner des problèmes de sécurité. L'avantage, par rapport à la seconde méthode (cf paragraphe précédent), c'est la possibilité d'automatiser l'investigation, ce qui implique un gain de temps non négligeable. Une fois l'analyse terminée, l'auditeur étudiera les éventuels problèmes manuellement.

Cette technique permet de découvrir des problèmes non manifestes lors d'un Stress Testing. Cette méthode ne peut remplacer un audit manuel complet, car toutes les anomalies ne seront pas découvertes, mais elle permet d'effectuer un audit relativement sérieux d'un programme dont le code source n'est pas disponible.

LES OUTILS DE L'AUDITEUR

DÉSASSEMBLEUR : IDA PRO - DATA RESCUE

IDA [1] est un désassembleur multiplateforme interactif réalisé par Ilfak Guilvanov. Il offre de nombreuses caractéristiques intéressantes, qui en font probablement 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.

• Ses principaux points forts de cet outil sont :

- support d'une impressionnante série de processeurs (Intel, AMD64, Motorola, Hitachi, GameBoy, HP, MIPS, SPARC, Alpha, Playstation, Siemens, Toshiba, NEC, Java VM...);
- possibilité de récupérer un code source en assembleur compilable, à partir de l'exécutable binaire;
- FLIRT, "Fast Library Identification and Recognition Technology" - Reconnaissance des appels systèmes et des appels de bibliothèques, ainsi que l'identification des paramètres de fonctions avec l'ajout des commentaires appropriés;

- manipulation de structures complexes;
- identification automatique des variables locales des procédures, des arguments, etc.;
- langage de script propre, similaire au langage C, utilisé pour automatiser des tâches, comme nous allons le voir plus tard pour, par exemple, rechercher des problèmes de sécurité dans un binaire;
- architecture permettant l'ajout de plugins, et une "SDK" complète est fournie;
- 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;
- support de la plupart des formats exécutables et binaires (PE, ELF...);
- interface graphique (GUI) simple et complète à la fois;
- possibilités étendues de navigation dans le code: définition des variables, labels et structures, ajout de commentaires automatiques;
- 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.

DÉBOGUEUR : SOFTICE - NUMEGA/COMPUWARE

Soft Ice [2] est un débogueur kernel très puissant. Les professionnels de la sécurité informatique s'en servent pour étudier des applications susceptibles de contenir des failles de sécurité (*buffer overflow*) ou même pour étudier des protections, algorithmes propriétaires, etc.

Il est bien sûr utilisé par les développeurs pour les aider à trouver les bogues dans leurs applications, ou plus généralement, dans leurs drivers. Soft Ice est un outil performant et complexe qui nécessite une bonne maîtrise de l'assembleur, ainsi que du système d'exploitation pour être utilisé efficacement.

• Ses principaux atouts sont :

- kernel débogueur qui permet de déboguer des drivers ou tout code fonctionnant en *ring 0* (ex. : vulnérabilité dans le driver de Norton Anti Virus);
- localisation dynamique des zones à étudier ensuite sous IDA, très utile pour les applications de taille importante;
- utilisation du désassemblage commenté d'IDA sous Soft Ice (Plugin IDA).

NOTIONS ÉLÉMENTAIRES

LES REGISTRES

Les registres généraux servent à contenir ou manipuler des données, ainsi qu'à passer des paramètres aux fonctions.



REGISTRES DE TRAVAIL

EAX (accumulateur)	Ce registre reçoit la valeur de retour des API Windows.
EBX	Base, utilisé pour l'adressage indirect.
ECX	Compteur utilisé en particulier dans les boucles.
EDX	Données et opérations mathématiques.

REGISTRES D'OFFSET

ESI (Index de Source) et EDI (Index de Destination)	Ils sont utilisés dans la manipulation de données, notamment dans le traitement de chaînes de caractères.
ESP (Stack Pointer - pointeur de pile)	Il pointe vers la dernière donnée empilée.
EBP (Base Pointer - pointeur de base)	Adresse base pour les <i>frames</i> des fonctions.
EIP (Instruction Pointer - pointeur d'instruction)	Adresse de la prochaine instruction à exécuter.

ASSEMBLEUR

Pour un guide plus complet, vous pouvez vous reporter à [3].

Push

L'instruction push permet de placer une donnée sur la pile (empilement). Celle-ci provient d'un registre, d'un emplacement mémoire ou d'une valeur passée au push directement, et sa taille est de deux octets (mot) ou de quatre octets (double mot). Techniquement, Le registre (E)SP (Stack Pointer) est décrémenté de deux ou quatre selon la taille de la donnée avant que celle-ci ne soit copiée à l'adresse pointée par le registre (E)SP.

Exemple

PUSH EAX

Avant l'instruction PUSH. EAX = 00000015h ESP = 0006FFC4
Dump de la pile: E7 87 E9 77 00 00 00 00 18 DA 41 07 00 F0 FD 7F
Après l'instruction PUSH. EAX = 00000015h ESP = 0006FFC0
Dump de la pile: 15 00 00 00 E7 87 E9 77 00 00 00 00 18 DA 41 07

Pop

L'instruction pop permet de récupérer une donnée placée sur la pile (désempilement). Celle-ci peut être placée dans un registre ou dans un emplacement mémoire et sa taille est de deux octets (mot) ou de quatre octets (double mot).

Techniquement, la donnée est d'abord récupérée à l'adresse pointée par le registre (E)SP. Celui-ci est ensuite incrémenté de deux ou quatre selon la taille de la donnée.

Exemple

POP EAX.

Avant l'instruction POP. Le registre EAX = 00004552 et ESP = 0006FFC0
Dump de la pile: 15 00 00 00 E7 87 E9 77 00 00 00 00 18 DA 41 07

Après l'instruction POP. Le registre EAX = 00000015 ESP = 0006FFC4
Dump de la pile: E7 87 E9 77 00 00 00 00 18 DA 41 07 00 F0 FD 7F

Call

L'instruction call permet d'appeler une sous-routine. Techniquement, l'adresse de l'instruction qui succède au call est placée sur la pile. Le registre EIP (Instruction Pointer) prend ensuite la valeur de l'adresse de la sous-routine.

Ret

L'instruction Ret permet de sortir d'une sous-routine quand celle-ci est terminée. Techniquement, l'adresse de retour est récupérée sur la pile (pointée par ESP) et l'adresse de la pile est incrémentée de quatre, s'il s'agit d'un programme 32 bit.

Exemple

```
401000 call sous_routine
401005 instructions suivantes
..
..
```

A l'emplacement de l'instruction Call, le registre EIP = 00401000 et ESP = 0006FFC4

Dump de la pile: E7 87 E9 77 00 00 00 00 18 DA 41 07 00 F0 FD 7F

401020 sous_routine:

```
..
..
..
```

Une fois dans la sous-routine, le registre EIP = 401020 et le registre ESP = 0006FFC0

Dump de la pile : 05 10 40 00 E7 87 E9 77 00 00 00 00 18 DA 41 07

```
..
..
```

Juste avant d'exécuter le ret, EIP = 401034 et la pile n'a pas changé.

401034 ret

L'exécution du RET récupère l'adresse de retour sur la pile 00401005 et la place dans le registre EIP. La pile est incrémentée de quatre.

EIP: 401005 et ESP = 0006FFC4.

Dump de la pile: E7 87 E9 77 00 00 00 00 18 DA 41 07 00 F0 FD 7F

Nous sommes sortis de la sous-routine et le registre EIP pointe vers la prochaine instruction à exécuter.



CALLING CONVENTIONS

Pour comprendre le code désassemblé, il est utile de connaître les différents types de conventions d'appels. Cela permet de saisir comment les paramètres sont passés aux fonctions, et comment la pile et les valeurs de retour sont gérées.

Spécification Générale de Microsoft

Les paramètres passés à une fonction sont "élargis" à 32 bits ainsi que la valeur de retour qui elle est placée dans le registre EAX. Pour les structures de huit octets, la combinaison des registres EDX:EAX est utilisée. Si la structure s'avère plus grande, le registre EAX contient un pointeur vers cette structure. Le passage des paramètres se fait de la droite vers la gauche, c'est-à-dire du dernier au premier.

Convention `_cdecl`

Cette convention est utilisée par défaut par le C/C++ :

- les paramètres sont passés de droite à gauche ;
- la gestion de la pile se fait après l'appel de la fonction.

Voici une fonction qui utilise la convention `_cdecl` :

```
int _cdecl somme(int x, int y, int z)
{
    int somme;
    somme=x+y+z;
    return somme;
}
```

Elle est appelée via `somme(10, 20, 30);`, ce qui donne en assembleur :

```
push 30 ; param 3
push 20 ; param 2
push 10 ; param 1
call somme ; fonction somme
add esp, 0Ch ; 0Ch = 12. Trois Paramètres.
```

Convention `_stdcall`

Cette convention est utilisée par les API Windows. C'est une convention hybride entre la Convention C et la convention Pascal :

- les paramètres sont passés sur la pile de la droite vers la gauche ;
- la gestion de la pile se fait par la fonction elle-même.

Voici maintenant la même fonction, mais cette fois avec pour convention `_stdcall` :

```
int _stdcall somme(int x, int y, int z)
{
    int somme;
    somme=x+y+z;
    return somme;
}
```

Son appel, `somme(10, 20, 30);`, devient en assembleur :

```
push 30 ; param 3
push 20 ; param 2
push 10 ; param 1
call somme ; fonction somme
```

Comme nous pouvons le voir, il n'y a plus de code pour rectifier la pile juste après le `call` vers la fonction. En revanche, dans la fonction elle-même, nous retrouvons la régularisation de la pile suivante :

```
ret 0Ch ; 0Ch = 12. Trois Paramètres.
```

L'instruction `ret` ici, afin d'utiliser la bonne valeur de retour, contient la taille des paramètres passés sur la pile depuis que l'adresse de retour a été sauvegardée sur la pile. Cela permet d'ajuster la pile et d'utiliser la vraie adresse de retour.

Trois paramètres qui sont des doubles mots (4 octets), on obtient donc $3 * 4 = 12$ octets. (0Ch octets).

Convention `_fastcall`

Selon les éditeurs, cette convention comporte quelques différences.

● Spécificités Microsoft :

- les paramètres sont passés de droite à gauche sur la pile exceptés les deux premiers ;
- ils utilisent les registres ECX et EDX ;
- la gestion de la pile se fait par la fonction elle-même comme pour `stdcall`.

```
int _fastcall somme(int x, int y, int z, int a, int b, int c, int d, int e)
{
    int somme;
    somme=x+y+z+a+b+c+d+e;
    return somme;
}
```

```
void main(void)
{
    somme2(10, 20, 30, 40, 50, 60, 70, 80);
}
```

Le code assembleur produit par le compilateur Microsoft Visual C++ est le suivant :

```
push 80 ; int
push 70 ; int
push 60 ; int
push 50 ; int
push 40 ; int
push 30 ; int
mov edx, 20 ; int
mov ecx, 10 ; int
call somme
```

● Spécificités Borland :

- les paramètres sont passés de gauche à droite à partir du quatrième paramètre ;
- les trois premiers utilisent les registres EAX, EDX et ECX.

Le compilateur Borland produit le code assembleur suivant :

```
push 40 ; int
push 50 ; int
push 60 ; int
push 70 ; int
push 80 ; int
mov ecx, 30 ; int
```




```

mov     edx, 20      ; int
mov     eax, 10      ; int
call    somme2

```

Convention thiscall

Il s'agit d'une convention spécifique aux fonctions membres des classes C++ qui n'utilisent pas un nombre variable de paramètres. Il n'est pas possible de spécifier cette convention dans un programme.

- les paramètres sont passés de la droite vers la gauche et placés sur la pile ;
- le pointeur `this` est passé dans le registre ECX ;
- si le nombre de paramètres est variable, `__cdecl` est utilisé et le pointeur `this` est passé sur la pile en dernier ;
- la gestion de la pile se fait par la fonction elle-même. (comme `stdcall`).

PARAMÈTRES ET VARIABLES LOCALES

Nous venons de voir comment les paramètres étaient passés aux fonctions. Il est aussi utile de savoir comment différencier les paramètres des variables locales, dans un désassemblage.

IDA

```

.text:0040103A mov [ebp+var_8], edx ; variable locale
.text:0040103D mov [ebp+var_4], ecx ; variable locale
.text:00401040 mov eax, [ebp+var_4] ; variable locale
.text:00401043 add eax, [ebp+var_8] ; variable locale
.text:00401046 add eax, [ebp+arg_4] ; Paramètre
.text:00401049 add eax, [ebp+arg_8] ; Paramètre
.text:0040104C add eax, [ebp+arg_C] ; Paramètre
.text:0040104F add eax, [ebp+arg_10] ; Paramètre
.text:00401052 add eax, [ebp+arg_14] ; Paramètre
.text:00401055 add eax, [ebp+arg_18] ; Paramètre
.text:00401058 mov [ebp+var_C], eax ; variable locale

```

IDA reconnaît et renomme automatiquement les variables locales et les arguments.

Soft ICE

```

001B:0040103A 8955F8 MOV [EBP-08],EDX ; variable locale
001B:0040103D 894DFC MOV [EBP-04],ECX ; variable locale
001B:00401040 8B45FC MOV EAX,[EBP-04] ; variable locale
001B:00401043 0345F8 ADD EAX,[EBP-08] ; variable locale
001B:00401046 034508 ADD EAX,[EBP+08] ; Paramètre
001B:00401049 03450C ADD EAX,[EBP+0C] ; Paramètre
001B:0040104C 034510 ADD EAX,[EBP+10] ; Paramètre
001B:0040104F 034514 ADD EAX,[EBP+14] ; Paramètre
001B:00401052 034518 ADD EAX,[EBP+18] ; Paramètre
001B:00401055 03451C ADD EAX,[EBP+1C] ; Paramètre
001B:00401058 8945F4 MOV [EBP-0C],EAX ; variable locale

```

Pour différencier une variable locale d'un paramètre, il suffit de regarder comment se fait l'indexage. S'il s'agit d'un indexage négatif (EBP-) alors c'est une variable locale, sinon en cas d'indexage positif (EBP+), c'est un paramètre.

Nous allons maintenant voir quelques problèmes de sécurité, et surtout comment les détecter dans un binaire désassemblé. Je présenterai la méthodologie à appliquer et divers exemples de binaires désassemblés contenant une faille de sécurité, pour plusieurs types de problèmes : débordement de buffer (*stack overflow*), format *string*, etc.

MÉTHODOLOGIE : VULNÉRABILITÉS DE TYPE DÉBOREMENTS DE BUFFER

Les questions à se poser pour identifier un problème par débordement de buffer.

- Pour les fonctions du type `strcpy`, `strcat`, `scanf`, `gets`, etc. :
 - présence d'un buffer de taille fixe comme destination ?
 - présence de données dynamiques (source) pouvant être placées dans ce buffer ?
- Pour les fonctions de type `sprintf`, `wsprintf`, `fprintf`, etc. :
 - la somme de tous les arguments formatés et du format est-elle supérieure à la taille du buffer de destination ?
- Pour les fonctions `strn*`, `snprintf`, ... fonctions qui permettent de contrôler la taille des buffers :
 - la taille spécifiée correspond-elle bien à celle de la destination ?
 - la taille spécifiée est-elle statique ou calculée dynamiquement, et dans ce cas, est-il possible d'en influencer la valeur ?

CAS DÉTAILLÉ : STRCPY

Voici un programme contenant une faille de sécurité :

```

void hole(char *str)
{
    char buf[30];
    strcpy(buf, str);
    printf("param is: %s\n", buf);
}

int main(int argc, char **argv)
{
    hole(argv[1]);
    return 0;
}

```

Nous allons maintenant rechercher la faille à partir d'un binaire compilé avec IDA. Après avoir désassemblé le fichier, nous allons appliquer la méthodologie vue précédemment.

① TROUVER LA PRÉSENCE D'UNE FONCTION NON SÛRE :

Pour cela, il suffit d'utiliser la fenêtre d'affichage des fonctions d'IDA : **Shift + F3**. Un double clic nous amène au code de cette fonction. Il suffit de regarder les références à cette fonction pour trouver le code qui l'appelle :



Recherche de vulnérabilités par désassemblage

```
CODE:00401304 ; char *__cdecl strcpy(char *dest,const char *src)
CODE:00401304 _strcpy      proc near      ; CODE XREF: sub_401108+Dp
CODE:00401304
CODE:00401304 dest          = dword ptr  8
CODE:00401304 src            = dword ptr  0Ch
CODE:00401304
```

On apprend ici que la fonction `strcpy()` utilise la convention `__cdecl`, et deux paramètres : `dest` et `src`. Les paramètres seront donc passés sur la pile de droite à gauche, et la gestion de la pile se fera après l'appel de la fonction.

Nous apprenons aussi qu'il n'y a qu'un seul appel à cette fonction (nombre de XREF).

Un double clic sur ce XREF nous permet de nous rendre à l'appel de cette fonction.

```
; int __cdecl sub_401108(char *src)  <== Un seul Argument.
sub_401108      proc near          ; CODE XREF: _main+9p

dest          = byte ptr -20h    <== Valeur Négative: Variable Locale.
src           = dword ptr  8     <== Valeur Positive: Argument.

push  ebp
mov   ebp, esp
add   esp, 0FFFFFFF0h
push  [ebp+src]      ; src
lea   eax, [ebp+dest]
push  eax            ; dest
call  _strcpy
add   esp, 8
lea   edx, [ebp+dest]
push  edx
push  offset aParamIsS ; format
call  _printf
add   esp, 8
mov   esp, ebp
pop   ebp
retn

sub_401108      endp
```

Cette fonction contient, comme souligné dans le désassemblage ci-dessus, un paramètre `src` et une variable locale `dest`.

2 LA PRÉSENCE D'UN BUFFER DE TAILLE FIXE COMME DESTINATION :

```
push  [ebp+src]      ; src
lea   eax, [ebp+dest]
push  eax            ; dest
call  _strcpy
```

Pour obtenir des informations sur la taille du buffer `dest`, il suffit de presser **Ctrl + K** pour ouvrir la fenêtre qui contient les informations sur la pile de la fonction étudiée.

A partir de là, il est possible d'obtenir la taille de notre buffer en se positionnant sur notre variable et en pressant la touche `***`. Cette touche permet de créer un tableau sous IDA. Le nombre d'éléments est trouvé automatiquement par le désassembleur.

La taille de notre buffer est de 32 octets, comme nous pouvons le voir sur l'image ci-dessus, ce qui correspond à l'instruction `char buf[30]` dans le code source. La différence de taille provient du compilateur car celui-ci a aligné la taille du buffer pour qu'elle soit un multiple de 4 ($30 \bmod 4 = 2$, mais la valeur venant juste après satisfaisant cette règle est 32, $32 \bmod 4 = 0$). Nous avons donc un buffer de 32 octets. Passons à la dernière étape.

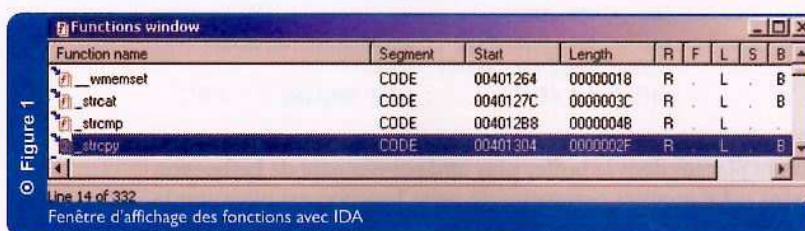


Figure 1

Fenêtre d'affichage des fonctions avec IDA

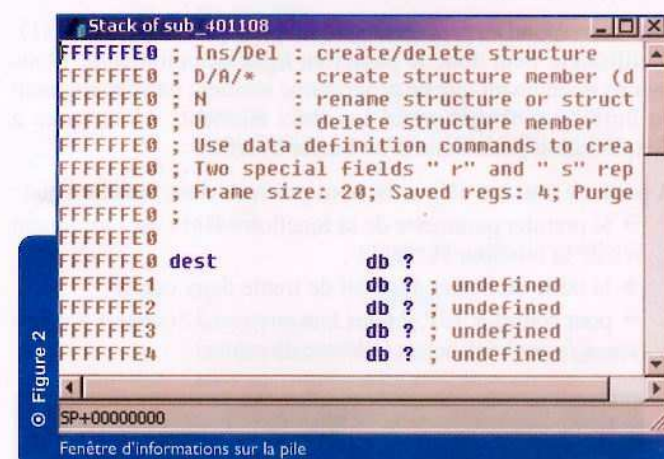


Figure 2

Fenêtre d'informations sur la pile

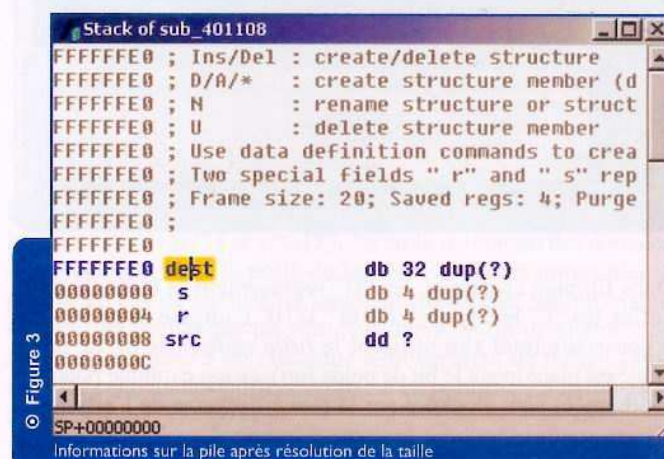


Figure 3

Informations sur la pile après résolution de la taille

3 RECHERCHE DE DONNÉES DYNAMIQUES (SOURCE) POUVANT ÊTRE PLACÉES DANS CE BUFFER :

Nous savons que le paramètre `src` est passé à la fonction. Nous allons maintenant voir si la valeur de ce paramètre est définie par un utilisateur, et donc si elle est dynamique. Nous retournons au programme appelant la fonction en double cliquant sur le XREF comme nous l'avons précédemment fait.



```
; int __cdecl main(int argc, const char **argv, const char *envp)
_main proc near ; DATA XREF: DATA:0040B044o

argc = dword ptr 8
argv = dword ptr 0Ch
envp = dword ptr 10h

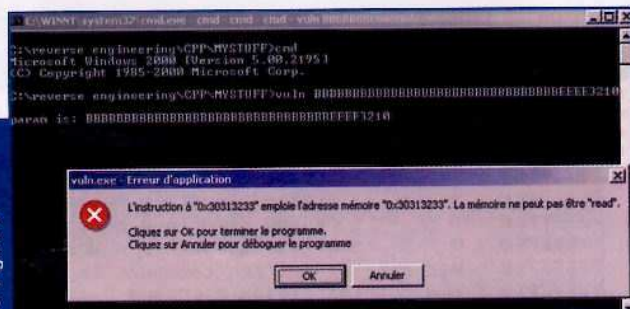
push ebp
mov ebp, esp
mov eax, [ebp+argv] ; 1er paramètre de main()
push dword ptr [eax+4]; src
call sub_401108 ; notre fonction
```

D'après le code ci-dessus, nous apprenons que les données qui seront placées dans le buffer plus tard proviennent de la fonction `main()`. Le premier paramètre de cette fonction est placé sur la pile ici par les instructions `mov eax, [ebp+argv]` `push dword ptr [eax+4]` ci-dessus.

`argv` correspond au premier argument passé à la fonction `main()`. L'utilisateur peut donc le passer en ligne de commande. Nous savons maintenant que ce programme contient un débordement de buffer car il comprend les trois éléments nécessaires à l'exploitation potentielle d'un buffer overflow.

A partir de tous ces éléments, nous pouvons donc conclure que :

- le premier paramètre de la fonction `main()` est l'argument `src` de la fonction `strcpy()` ;
- la taille du buffer `dest` est de trente deux octets ;
- pour contrôler EIP, il nous faut envoyer 32 octets + 4 octets (*stack frame*) + 4 octets (adresse de retour).



Débordement de buffer

Dans l'image ci-dessus, les "B" représentent la taille de notre buffer, les "E" le registre EBP et "3210" l'adresse de retour. Les processeurs Intel x86 utilisent le *little endian* (le bit de poids faible est placé avant le bit de poids fort), ce qui explique pourquoi les données sont stockées sur la pile à l'inverse de l'affichage.

Le programme a effectué une erreur en essayant d'exécuter l'adresse `0x30313233` (0123), ce qui veut dire que nous contrôlons bien l'adresse de retour, il est possible d'exploiter la faille.

SCANF()

```
; int __cdecl main(int argc, const char **argv, const char *envp)
_main proc near ; DATA XREF: DATA:0040C044o

var_54 = dword ptr -54h
```

```
var_4 = dword ptr -4
argc = dword ptr 8
argv = dword ptr 0Ch
envp = dword ptr 10h
```

```
push ebp
mov ebp, esp
add esp, 0FFFFFFACh
push ebx
mov ebx, offset a5 ; "%s"
lea eax, [ebx+3] ; "Entrez votre nom: "
```

```
push eax
push ebx ; format
call _printf
add esp, 8
lea edx, [ebp+var_54] ; var_54 db 80 dup(?)
push edx
lea ecx, [ebx+16h] ; "%s"
push ecx ; format
call _scanf
```

```
add esp, 8
lea eax, [ebx+1Ch] ; "Entrez votre age: "
push eax
lea edx, [ebx+19h] ; "%s"
push edx ; format
call _printf
```

```
add esp, 8
lea ecx, [ebp+var_4]
push ecx
lea eax, [ebx+2Fh] ; "%d"
push eax ; format
call _scanf
```

```
add esp, 8
push [ebp+var_4]
lea edx, [ebp+var_54]
push edx
lea ecx, [ebx+32h] ; "Bonjour. %s, Vous avez %d ans.\n"
push ecx ; format
call _printf
```

```
add esp, 0Ch
call _getch
xor eax, eax
pop ebx
mov esp, ebp
pop ebp
retn
endp
```

_main

En regardant le listing assembleur, on retrouve une fonction non sûre `scanf()`, et une variable locale `var_54` d'une taille fixe de 80 caractères (on utilise la "stack window" pour déterminer la longueur du buffer). Pour déterminer le nombre de caractère à entrer pour écraser l'EIP, il suffit de regarder l'état de la pile dans cette fonction :

Pour obtenir les informations sur la pile, un double clic sur une variable locale suffit (ou une pression des touches Ctrl+K).

```
FFFFFFAC var_54 db 80 dup(?) ; buffer de taille fixe
FFFFFFFC var_4 dd ? ; entier utilisé dans le
second scanf
00000000 s db 4 dup(?) ; Registre sauvegardé (EBP)
00000004 r db 4 dup(?) ; Adresse de retour
```




Pour écraser l'EIP, il faut fournir 80 octets **:(Buffer) + 4 octets (Entier) + 4 octets (Stackframe) + 4 octets (adresse de retour)** soit **BBBBBBBBBBBB...BBBBBBBBBBBBBBBBEEEESSSSRRRR**.

Le code C suivant a été utilisé pour l'exemple :

```
int main ()
{
    char str [80];
    int age;

    printf ("%s","Entrez votre nom: ");
    scanf ("%s",str);
    printf ("%s","Entrez votre age: ");
    scanf ("%d",&age);
    printf ("Bonjour. %s , Vous avez %d ans.\n",str,age);
    getch();
    return 0;
}
```

GETS()

```
; int __cdecl main(int argc,const char **argv,const char *envp)
_main      proc near      ; DATA XREF: DATA:0040C044o
```

```
s          = byte ptr -3Ch
argc       = dword ptr  8
argv       = dword ptr  0Ch
envp       = dword ptr  10h

        push    ebp
        mov     ebp, esp
        add     esp, 0FFFFFFC4h
        lea     eax, [ebp+s] ; stack variable : s db 60 dup(?)
        push    eax          ; s
        call    _gets
```

La combinaison d'un buffer de taille fixe et d'une fonction n'effectuant aucun contrôle sur la taille des données entrées engendrent un problème de sécurité.

- le buffer `s` est de taille fixe : 60 octets ;
- la fonction `gets()` ne vérifie en aucun cas la taille des données entrées ;

Pour contrôler EIP, il suffit d'envoyer 60 octets (buffer) + 4 octets (stack frame) + 4 octets (adresse de retour). Le code utilisé dans l'exemple ci-dessus est le suivant :

```
int main ()
{
    char name[60];
    gets(name);
    getch();
    return 0;
}
```

FONCTION (PRESQUE) SÛRE : STRNCPY()

Il existe des fonctions présentées comme plus sûres, permettant d'éviter les débordements de buffer. Malheureusement, elles sont souvent mal utilisées et des problèmes de sécurité subsistent, comme l'illustre l'exemple suivant :

```
void func(char *str) {
    char buffer[12];
    strcpy(buffer, str);
    printf("%s",buffer);
}
```

```
main(int argc, char *argv[])
{
    char arg2[24];
    char arg1[8];
```

```
    if (argc > 2) {
```

```
        strncpy(arg1, argv[1], sizeof(arg1);
        strncpy(arg2, argv[2], sizeof(arg2);
        func(arg1);
    }
}
```

```
mov     ebx, [ebp+argv]
cmp     [ebp+argc], 2 ; 2 parametres?
jle     short non_on_sort
push    8 ; sizeof(dest);
push    dword ptr [ebx+4] ; argv[1] : (+4 = premier arg)
lea     eax, [ebp+dest] ; dest db 8 dup(?) = char dest[8];
push    eax ; dest
call    _strncpy
add     esp, 0Ch ;
; strncpy(dest, argv[1], sizeof(dest));
;
push    24 ; sizeof(dest2);
push    dword ptr [ebx+8] ; argv[2] : (+8 = second arg)
lea     edx, [ebp+dest2] ; dest2 db 24 dup(?) = char dest2[24];
push    edx ; dest2
call    _strncpy
add     esp, 0Ch ;
; strncpy(dest2,argv[2], sizeof(dest2));
;
lea     ecx, [ebp+src] ; argv[1]
push    ecx ; src
call    func_strcpy
pop     ecx
```

```
non_on_sort: ; CODE XREF: _main+Ej
xor     eax, eax
```

Les deux `strncpy()` utilisent respectivement les données passées en paramètre `argv[1]` et `argv[2]`. La taille maximum des données à copier est égale à la taille du buffer. Ce qui est un problème puisqu'en cas de chaînes de longueur maximale, les buffers ne se termineront pas par un zéro (caractère de fin de chaîne), ce qui peut conduire à un classique débordement de buffer. Pour illustrer cela, un des buffers est passé en paramètre à la fonction nommée `func_strcpy()` :

```
push    [ebp+src] ; src = buffer de 8 octets
lea     eax, [ebp+dest] ; dest db 12 dup(?)
push    eax ; dest
call    _strcpy
```

Nous avons vu précédemment les risques de la fonction `strcpy()`. Cependant, ici, cette fonction ne fait que copier un buffer de 8 octets dans un buffer de 12 octets, ce qui ne pose aucun problème. Or, le principe de la fonction `strcpy()` est de copier les données du buffer `src` vers le buffer `dest` tant qu'un caractère nul n'est pas rencontré.



Revenons maintenant à la mauvaise utilisation des fonctions `strcpy()`. Si une personne mal intentionnée emploie une chaîne de caractères de 8 octets, le buffer ne se terminera donc pas par un zéro. En passant une seconde chaîne de caractères, il est probable qu'il n'y ait aucune séparation entre les deux buffers. Par conséquent, la chaîne `src`, qui était initialement longue de 8 octets, sera mesurée avec une taille supérieure maintenant.

Lors de la copie de cette chaîne par la fonction `strcpy()`, l'octet nul sera rencontré bien plus tard, impliquant un buffer d'une taille bien supérieure au buffer de destination, ce qui provoque un débordement de buffer.

Voici une autre erreur souvent rencontrée :

```
push 7 ; sizeof(dest)-1;
push dword ptr [ebx+4] ; src = argv[1] : (+4 = premier arg)
lea eax, [ebp+dest] ; dest db 8 dup(?) = char dest[8];
push eax ; dest
call _strcpy
add esp, 0Ch
```

`strcpy(dest,argv[1],sizeof(dest)-1);`

```
push 23 ; sizeof(dest2)-1;
push dword ptr [ebx+8] ; src = argv[2] : (+8 = second arg)
lea edx, [ebp+dest2] ; dest2 db 24 dup(?) = char dest2[24];
push edx ; dest2
call _strcpy
add esp, 0Ch
```

`strcpy(dest2,argv[2],sizeof(dest2)-1);`

Cette fois-ci, la valeur maximum passée dans le buffer est la taille du buffer-1 pour éviter tout problème de débordement. Cependant, nous ne connaissons pas la valeur du dernier octet du buffer. Celui-ci peut très bien être différent de zéro, comme dans le premier exemple et engendrer un débordement de buffer. La pile contient souvent ce que l'on appelle du *garbage* et il n'est pas improbable d'avoir un octet différent de zéro.

Voici maintenant une implémentation sécurisée de `strcpy()` :

```
push 7 ; sizeof(dest)-1;
push dword ptr [ebx+4] ; src = argv[1] : (+4 = premier arg)
lea eax, [ebp+dest] ; dest db 8 dup(?) = char dest[8];
push eax ; dest
call _strcpy
add esp, 0Ch
mov [ebp+var_1], 0 ; arg1[sizeof(arg1)-1]='\0';
```

`strcpy(arg1, argv[1], sizeof(arg1)-1);`
`arg1[sizeof(arg1)-1]='\0';`

```
push 23 ; sizeof(dest2)-1;
push dword ptr [ebx+8] ; src = argv[2] : (+8 = second arg)
lea edx, [ebp+dest2] ; dest2 db 24 dup(?) = char dest2[24];
push edx ; dest2
call _strcpy
add esp, 0Ch
mov [ebp+var_9], 0 ; arg2[sizeof(arg2)-1]='\0';
```

`strcpy(arg2, argv[2], sizeof(arg2)-1);`
`arg2[sizeof(arg2)-1]='\0';`

Cette fois, la taille des données copiées est égale à la taille du buffer-1 et le dernier caractère du buffer est initialisé à zéro. Dans ces conditions, l'appel à la fonction `func_strcpy()` se fait sans problème ; il n'y a plus aucun risque de débordement car le buffer `src` fera bien 8 octets, et sera donc inférieur aux 12 octets du buffer de destination.

OFF BY ONE : PROBLÈME D'INDEXAGE

Voici un exemple d'erreur de *parsing* :

```
jmp short start_parsing
; -----
parsing:
mov cl, [eax] ; CODE XREF: sub_401100+2Fj
mov [ebp+edx+var_2B], cl ; cl = caractère pointé par EAX.
inc eax ; on sauvegarde le caractère
inc edx ; on change de caractère
; on change de position dans le buffer

start_parsing: ; CODE XREF: sub_401100+Bj
xor ecx, ecx ; ECX = 0
mov cl, [eax] ; cl = caractère pointé par EAX.
cmp ecx, VK_SPACE ; s'agit-il d'un espace?
jz short bad_char_stop_parsing ; oui, on sort
xor ecx, ecx ; ECX = 0
mov cl, [eax] ; cl = caractère pointé par EAX
cmp ecx, VK_TAB ; s'agit-il d'une tabulation?
jz short bad_char_stop_parsing ; oui, on sort
cmp byte ptr [eax], 0 ; s'agit-il d'un caractère nul?
jz short bad_char_stop_parsing ; oui, on sort
cmp edx, 40 ; index de boucle = 40
jl short parsing ; boucle tant que l'index est < 40

bad_char_stop_parsing: ; CODE XREF: sub_401100+1Cj
; sub_401100+25j ...
mov [ebp+edx+var_2B], 0 ; Le caractère en [index] est mis à zéro.
```

Le bout de code suivant recherche dans la chaîne entrée la présence d'un espace, d'une tabulation, ou d'un caractère nul. Tant qu'aucun de ces caractères n'est trouvé, chaque caractère validé est placé dans un buffer. Nous sommes en présence d'une boucle de 40 itérations, et d'un buffer de 40 octets.

En revanche, en sortie de boucle, un caractère nul est placé en position `buffer[index]`. Lorsque l'index est de 40, le programme écrit un `\0` en `buffer[40]`, mais nous sommes ici hors des limites du buffer, donc le programme place un `\0` sur des données existantes. Dans cet exemple, le registre `EBP`, qui est sauvegardé lors du stack frame, est modifié d'un octet sur la pile.

Voici le code utilisé pour illustrer l'exemple :

```
void do_nothing(char *cp) {
    char caCmd[40];
    char *cpArgs;
    int i;

    /* parse input */
    for (i = 0; *cp != ' ' &&
```




```
*cp!=='\t' &&
*cp!=0 &&
i<40; )
caCmd[i++] = *cp++;
caCmd[i] = NULL;
cpArgs = cp;
}
```

Pour éviter ce problème, il aurait fallu utiliser :

```
/* parse input */
for (i = 0; *cp!=' ' &&
*cp!=='\t' &&
*cp!=0 &&
i < sizeof(caCmd)-1;)
```

SPRINTF()

```
push    0Ah
push    400h                ; 1024 octets (0x400) peuvent
                             être entré au clavier.
lea     eax, [ebp+var_400]   ; buffer de 1024 (0x400) Octets
push    eax
push    offset dword_411098
call    @istream@getline$qpcc ; Entrée clavier.
add     esp, 10h
lea     edx, [ebp+var_400]
push    edx
push    offset aS            ; Format: %s
lea     ecx, [ebp+buffer]
push    ecx                  ; buffer db 700 dup(?)
call    _sprintf
add     esp, 0Ch
```

La fonction `printf()` convertit une ou plusieurs variable(s) suivant les formats précisés et place les résultats dans son premier argument. Il s'agit ici de `buffer` et celui-ci fait 700 octets. (`buffer db 700 dup(?)`). Le format utilisé est de type chaîne de caractères `%s`. La variable qui est convertie est initialisée par une entrée clavier. Les données sont donc dynamiques et sont placées dans un buffer de 1024 octets. La fonction `getline()` limite la taille des données à 1024 octets pour éviter un débordement.

Cependant, la fonction `printf()` convertit ces données dans un buffer plus petit. Une personne mal intentionnée entrant plus de 700 octets peut donc engendrer un débordement de buffer et contrôler le registre EIP.

```
int main(int argc, char **argv)
{
    char nom[1024];
    char str[700];
    cin.getline(nom, 1024);
    printf(str, "%s", nom);
    printf("%s\n", str);
}
```

MÉTHODOLOGIE : VULNÉRABILITÉS DE TYPE FORMAT STRING.

La méthode pour détecter les problèmes de format string par désassemblage est très simple. En effet, pour détecter ces erreurs de programmation, il suffit de se poser deux questions très simples :

→ Tout d'abord, le nombre de paramètres est-il suffisant ?

Lorsqu'un programmeur oublie ou néglige la format string, le nombre de paramètres passés aux fonctions est inférieur au nombre attendu. Les programmes en C utilisent la convention `_cdecl` par défaut, ce qui nous permet de voir la correction de la pile juste après l'appel : `add esp, xx`. A partir de cette correction, il est très simple de déterminer si un paramètre est manquant.

→ Le format string est-il dynamique ?

Pour que l'erreur soit potentiellement exploitable, il faut que le format string soit dynamique. Si l'utilisateur peut contrôler celui-ci, il est alors possible d'exécuter du code arbitraire.

Voici quelques exemples vulnérables ou non:

ERREUR AVEC SPRINTF()

```
lea     edx, [ebp+format]
push    edx                  ; format dynamique
lea     ecx, [ebp+buffer]
push    ecx                  ; buffer
call    _sprintf
add     esp, 08              ; Nombre d'arguments insuffisants
```

`sprintf()` accepte trois paramètres au minimum, ce qui implique une correction de la pile de 12 octets. (3*4). Après le call `_sprintf` nous devrions donc avoir un `add esp, 0Ch`, synonyme d'une utilisation correcte de la fonction. Un paramètre s'avère donc manquant étant donné la correction de seulement 8 octets (`add esp, 8`).

Le seul paramètre pouvant manquer à l'appel étant la format string, il nous faut maintenant déterminer si elle peut être modifiée dynamiquement. La présence d'une variable locale nous indique qu'il est sûrement possible d'entrer nous-même la chaîne de format.

Code utilisé : `printf(buffer, str);`

SPRINTF() AVEC FORMAT NON DYNAMIQUE

```
lea     ecx, [ebp+var_200]
push    ecx
lea     eax, [ebp+var_100]
push    eax
push    offset aLesParamsSontS ; "les params sont: %s et %s"
lea     edx, [ebp+buffer]
push    edx                  ; buffer
call    _sprintf
add     esp, 10h             ; 16 octets
```

Dans l'exemple suivant, nous avons deux arguments de format de type chaîne (`%s`). En temps normal, la fonction `printf()` attend au minimum trois paramètres, mais ici nous avons une format string en plus, les arguments sur la pile doivent être de 16 (10h) octets afin d'assurer un formatage correct.

Le nettoyage de la pile traite bien 16 octets, cet appel à `printf()` n'est donc pas vulnérable.



PRINTF() VULNÉRABLE

```

push [ebp+format] ; format dynamique!
call _printf
pop ecx           ; Ajustement de pile:
                  1 seul paramètre.

```

Ici, le programmeur n'a pas utilisé correctement la fonction printf(), il est donc ainsi possible de passer la chaîne de format manuellement, ce qui implique un problème de sécurité.

D'ailleurs, l'ajustement de pile nécessaire ne suit pas l'appel de printf, il manque bien un paramètre.

Code de l'exemple :

```

void hole(char *str)
{
    printf(str);
}

```

PRINTF() CORRECTEMENT UTILISÉ

```

push [ebp+arg_0]
push offset aParamIsS ; "param is: %s\n"
call _printf
add esp, 8             ; Ajustement pour deux
                       paramêtres : 8 octets

```

La format string est ici statique et la pile est correctement ajustée. Deux arguments sont attendus, impliquant un ajustement de huit octets.

```

void hole(char *str)
{
    printf("%s",str);
}

```

LES API WINDOWS

SONT AUSSI VULNÉRABLES : WSPRINTFA

```

int wprintf(
    LPTSTR lpOut,      // pointer to buffer for output
    LPCSTR lpFmt,       // pointer to format-control string
    ...                // optional arguments
);

```

L'API wprintf() fonctionne exactement comme la fonction sprintf().

wsprintfA() vulnérable

```

push [ebp+arg_4] ; Format dynamique!
push [ebp+arg_0]
call wsprintfA

```

Dans l'exemple ci-dessus, la format string n'est pas statique, et seulement deux buffers sont passés en paramètres. La format string est un argument, il est donc probablement possible de la définir nous-mêmes.

wsprintfA() correctement employé

```

push [ebp+arg_4]
push offset aPrsnomS ; "prénom: %s ; format Satique!"
push [ebp+arg_0]
call wsprintfA

```

CAS RÉELS
D'APPLICATIONS VULNÉRABLES

WINDOWS: FTP.EXE

Comme premier exemple, j'ai commencé par désassembler FTP.exe à la recherche d'erreurs de programmation, et j'en ai trouvée plusieurs. Voici un exemple :

```

push dword ptr [edi+4]
lea eax, [ebp-200h]
push eax
call ds:_mbscopy

```

La fonction _mbscopy() est similaire à la fonction strcpy() dans son fonctionnement. Cette fonction copie des données sans aucun contrôle. Ce bout de code est présent dans la routine qui gère l'envoi d'une commande raw, telle que SITE par exemple. Le nom de cette commande est littéral. Un argument est passé en paramètre à la fonction, et celui-ci est copié dans un buffer de taille fixe. L'argument est dynamique et il s'agit de notre commande. L'adresse de retour est présente un peu plus loin sur la pile : Stack[000003D8]:0006F774 retaddr dd 10019FDh

Celle-ci se fait écraser lorsque la commande envoyée est trop longue, il est alors possible de contrôler EIP et donc d'exécuter du code arbitraire. Même s'il s'agit d'une vulnérabilité locale et sans grand intérêt pour un attaquant, cet exemple montre bien qu'on peut trouver des erreurs de programmation, pas toujours standard aux exemples, sans avoir le code source de l'application.

L'exploitation est par exemple envisageable au travers d'un script malicieux (ftp -s:ftp.txt), préalablement téléchargé par un autre biais. En conclusion, FTP.exe contient de nombreuses erreurs de programmation, il s'avère être un très bon exemple pour se faire la main.

WAR FTP : CÉLÈBRE

POUR SES DÉBORDEMENTS DE BUFFER

Voici maintenant une erreur de programmation dans un serveur FTP. Cette erreur se trouve dans l'écriture des logs du serveur. Le serveur logue tous les événements, telle qu'une connexion au serveur FTP. Lors de la création du log, le programmeur passe directement un argument dynamique à un sprintf() de taille fixe :

```

mov eax, [esp+210h+arg_0]
lea ecx, [esp+210h+var_200] ; var_200 db 512 dup(?)
push eax
push offset aS_3 ; "%s\r\n"
push ecx
call ds:sprintf

```




Le buffer a une taille fixe de 512 octets, ce qui permet un débordement si une commande trop longue est envoyée. Cette vulnérabilité est exploitable sans compte, il suffit d'envoyer un `USER buffer_de_grande_taille` pour déclencher le débordement, et de contrôler EIP. A partir de là, il est possible d'exécuter du code arbitraire.

SERV-U 4

Alors que j'écrivais cet article, une nouvelle vulnérabilité pour Serv-U [5] a été publiée. Je me suis empressé de télécharger la version vulnérable pour présenter la vulnérabilité. Une fois de plus, il s'agit d'une erreur de programmation classique. Comme pour WarFTP, il s'agit d'un `sprintf()` prenant pour source un argument de taille variable, et pour destination, un buffer de taille fixe. Dans notre exemple, il s'agit d'un buffer de 256 octets. La commande incriminée est `chmod()`.

Voici le désassemblage de la partie vulnérable :

```
lea    eax, [esi+arg_0480]
push   eax
call   sub_414350
add    esp, 0Ch
push   eax                ; format
lea    edx, [ebp+buffer]
push   edx                ; buffer db 256 dup(?)
call   _sprintf
add    esp, 0Ch
```

En envoyant une commande mal formée, il est possible d'exécuter du code arbitraire : `SITE chmod 777 [Buffer>256]`

SEMI-AUTOMATISATION DES RECHERCHES

Voici ici un exemple de script très simple pour rechercher des problèmes de débordements de buffer. Ce script recherche les `strcpy()` d'une application qui ont pour paramètre un argument pour la source (potentiellement de taille variable), et un buffer de taille fixe pour la destination (et donc potentiellement vulnérable). Afin de limiter la taille du script, celui-ci n'est pas très intelligent, mais permettra aux personnes intéressées de se lancer dans l'écriture de scripts plus performants. Il existe un projet Open Source du nom de Bugscam [4], qui comporte une multitude de scripts d'audits plus puissants et plus performants. Je vous invite à les télécharger et les étudier.

```
//-----strcpy.idc-----
// Script IDC Audit strcpy par Nicolas Brulez.
#include <idc.idc>
```

```
static GetArg(lpCall,n)
{
    auto TempReg;

    while(n>0)
    {
        lpCall=RfirstB(lpCall);
```

```
        if(GetMnem(lpCall) == "push")
            n=n-1;
    }
    if(GetOpType(lpCall,0)==1)
    {
        TempReg=GetOpnd(lpCall,0);
        lpCall=RfirstB(lpCall);
        while(GetOpnd(lpCall,0)!=TempReg)
            lpCall=RfirstB(lpCall);
        return(GetOpnd(lpCall,1));
    }
    else
        return(GetOpnd(lpCall,0));
}

static AuditStrcpy(lpCall)
{
    auto buffSource, buffTarget, addr, offset, offset2;
    buffTarget = GetArg(lpCall,1);
    // Récupère le premier paramètre: dst
    buffSource = GetArg(lpCall,2);
    // Récupère le second paramètre: src

    buffTarget = substr(buffTarget, 5, strlen(buffTarget));
    // retire le "ebp+" ou le "esp+"
    buffTarget = substr(buffTarget, 0, strlen(buffTarget)-1);
    // retire le "]" restant.

    buffSource = substr(buffSource, 5, strlen(buffSource));
    // retire le "ebp+" ou le "esp+"
    buffSource = substr(buffSource, 0, strlen(buffSource)-1);
    // retire le "]" restant.

    offset = GetMemberOffset(GetFrame(lpCall), buffTarget);
    offset2 = GetMemberOffset(GetFrame(lpCall), buffSource);

    if (offset < GetFrameLvarSize(lpCall))
    {
        // La destination est une variable locale, donc de taille fixe.

        if (offset2 > GetFrameLvarSize(lpCall))
            Message("%lx -> Possibilité de Buffer Overflow\n", lpCall);

        // La source est un Argument. Il peut y avoir un probleme!
    }
}

static main()
{
    auto FuncAddr, xref;                // déclaration des variables

    FuncAddr = AskAddr(-1, "Enter Address:");
    // Demande l'adresse du strcpy.
    Message("Adresse de strcpy: %lx\n\n", FuncAddr);
    // Affiche l'adresse de la fonction strcpy
    xref = RfirstB(FuncAddr);
    // Prends la première référence à strcpy
    Message("Xref: %lx\n", xref);
    // Affiche son adresse.
```




```

while(xref != -1)
// Tant qu'il y a des références on boucle
{
    if(GetMnem(xref)=="call")
        AuditStrcpy(xref);
// Si l'instruction qui référence strcpy est un call
// On audite l'adresse.
    xref = RnextB(FuncAddr, xref);
// sinon on récupère la prochaine référence.
    Message("xref = %lx\n", xref);
// Sinon on affiche l'adresse qui référence le strcpy
}

xref=DfirstB(FuncAddr);
while(xref!=-1)
{
    if(GetMnem(xref)=="call")
        AuditStrcpy(xref);
    xref = DnextB(FuncAddr, xref);
}
Message("Audit des Strcpy Terminé!\n"); // Message de fin.
}

```

Voici les résultats du script :

Compiling file 'C:\reverse engineering\IDA\IDC\strcpy.idc'...
 Executing function 'main'...
 Adresse de strcpy: 401304

Xref: 401115
 401115 -> Possibilité de Buffer Overflow
 Audit des Strcpy Terminé!

Il nous indique la présence possible d'un débordement de buffer à l'adresse 401115. Ce script a été testé sur une application utilisant strcpy() une centaine de fois. Celui-ci m'a indiqué que dix des utilisations étaient probablement exploitables. C'est un gain de temps considérable, mais il ne faut pas oublier que certains problèmes ne sont pas toujours mis en évidence.

CONCLUSION

Dans cet article, j'espère avoir une fois de plus prouvé l'utilité du Reverse Engineering en démontrant qu'il ne s'agissait pas seulement d'une technique utilisée par les "crackers de code", mais qu'elle pouvait aussi être employée par les professionnels de la sécurité pour trouver des vulnérabilités dans les produits commerciaux dont le source n'est pas disponible. Je n'ai ici bien sûr présenté que les cas les plus simples et je vous invite à lire les conférences d'Halvar Flake [6], auteur de Bugscam, qui traitent le problème plus en profondeur.

Nicolas Brulez - 0x90@rstack.org

Chief of Security - The Armadillo Software Protection System
<http://www.siliconrealms.com/armadillo.htm>

Remerciements à Nabakelti et à Eric Landuyt pour leur aide.

Références

- [1] IDA (c) Datarescue :
<http://www.datarescue.com/ibase/>
- [2] Soft ICE (c) Compuware :
<http://www.compuware.com/products/driverstudio/softice/>
- [3] Art of Assembly :
http://webster.cs.ucr.edu/Page_asm/ArtofAssembly/ch06/CH06-1.html
- [4] Bugscam par Halvar Flake :
sourceforge.net/projects/bugscam/
- [5] Serv-U Remote Buffer Overflow :
<http://www.securityfocus.com/bid/9483>
- [6] Halvar Flake :
 - > <http://www.blackhat.com/presentations/bh-asia-01/halvar.ppt>
 - > <http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-halvar.pdf>
 - > <http://www.blackhat.com/presentations/win-usa-03/bh-win-03-halvarflake.pdf>
 - > <http://www.blackhat.com/presentations/bh-federal-03/bh-fed-03-halvar.pdf>

www.miscmag.com

Le site 100% sécurité informatique !

MISC

www.mis
 Multi-System & I

Le site web de M.I.S.C. le m

Accueil
 Sommaires
 Articles
 Commande
 Rendez-vous
 Contacts

Accueil

Bienvenu sur le site de MISC, le magazine qui

- les articles de MISC 0, le hors série 8
- des articles de nos anciens numéros.

L'objectif est de concentrer ici toutes les inform

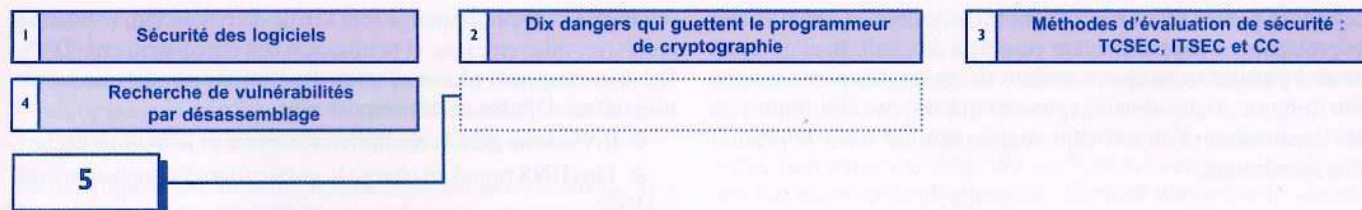
Si votre buraliste n'a pas le dernier MISC, dem

SYMPOSIUM

SSTIC

2-4 juin 2004 à Rennes

SUR LA SÉCURITÉ DES TECHNOLOGIES DE L'INFORMATION ET DES COMMUNICATIONS



Sécurité logicielle : étude de cas

INTRODUCTION

Donner des exemples de programmes sécurisés n'est pas une tâche aisée. Contrairement à bien des domaines, la sécurité en génie logiciel ne suit pas de réglementation particulière : les constructeurs automobiles ou les architectes du bâtiment doivent eux suivre des règles connues et imposées (par exemple sur les ceintures de sécurité ou les airbags, sur les normes anti-sismiques). Le thème qui nous intéresse s'appuie beaucoup plus sur le bon sens, l'expérience et des recommandations ou critères de certification de différents organismes.

Les cas que nous traiterons par la suite ne sont pas des programmes complètement sûrs. Ce ne sont pas non plus de programmes qui ne connaîtront aucune vulnérabilité dans le futur ou qui n'en ont connu aucune dans le passé. Toutefois, ils montrent clairement que la sécurité est un paramètre qui a été pris en compte à un ou plusieurs niveau(x) du cycle de développement du logiciel et ils sont souvent considérés comme "suffisamment sûr pour ce qu'on souhaite en faire". Les différents exemples pourront inspirer le lecteur développeur et lui donner des pistes sur les bons usages à adopter. Ils pourront aussi donner à l'utilisateur ou au décideur des critères de sécurité à prendre en compte. En effet, avoir une idée de la sécurité d'un programme est important : les personnes qui achètent ou utilisent des logiciels vulnérables de manière récurrente [MURPHY] sont autant coupables du manque de sécurité qui règne sur Internet que les éditeurs.

ARCHITECTURE ET IMPLÉMENTATION

Bien que le cycle de développement d'un logiciel soit bien plus complexe, nous allons nous focaliser sur les deux points les plus significatifs que sont l'architecture et l'implémentation. Le lecteur souhaitant avoir une vue plus complète de ces aspects pourra

également consulter le très bon livre *Secure Coding* de Mark G. Graff et Kenneth R. van Wyk [OREILLY].

L'architecture et le design d'un logiciel correspondent à l'étude qui a lieu en amont et qui vise à choisir d'une part l'organisation générale du programme, les structures de données utilisées et les principaux algorithmes... D'autre part, il est nécessaire de qualifier les risques liés à l'utilisation du logiciel et de définir le niveau de sécurité souhaité dont découleront les mesures prises lors de l'implémentation. Il est vital lors de cette phase d'avoir un œil très critique et de garder à l'esprit que l'environnement peut être hostile, que les protocoles peuvent ne pas être suivis à la lettre, que l'utilisateur final ne sera pas forcément sans mauvaise intention, etc.

Toutefois, le temps investi en mise en place de la sécurité sera du temps qui aurait pu servir au développement de nouvelles fonctionnalités.

Le but premier d'un logiciel étant rarement d'être sécurisé, des compromis à ce sujet doivent souvent être faits au profit du fonctionnel. Il faut tout de même noter que si la sécurité est intégrée dès le design du logiciel, le coût de développement est plus faible que de l'incorporer après la détection d'un problème. Modifier l'architecture du logiciel pour supprimer un problème logique ou structurel est bien moins trivial que de corriger une faute d'implémentation.

Les failles dues à une erreur d'implémentation peuvent être multiples. La cause peut en être une maîtrise imparfaite du langage de programmation par le développeur, une inattention, l'inconscience d'un problème ou d'un type de problème... Sans être exhaustif, nous pouvons citer en guise d'exemples :

- les *buffer overflows* ;
- l'injection de commandes dans une saisie de l'utilisateur ;
- la mauvaise utilisation des mécanismes de gestion des privilèges ;
- le stockage de mots de passe non chiffrés ;
- la trop grande confiance lors de la récupération de variables d'environnement ou lors de l'invocation de commandes shells ;
- l'utilisation intempestive de répertoires de stockage accessibles à tous en lecture/écriture.



Le lecteur pourra se référer à l'article de ce dossier de MISC sur les problèmes d'implémentation pour plus de détail. Bien qu'ayant attiré à l'aspect technique, certaines de ces erreurs peuvent aussi être dues au circuit de management qui impose des durées de développement et de test trop courtes afin de sortir le produit plus rapidement.

DJBDNS

Djbdns [**DJBDNS**], le serveur DNS du très controversé D.J. Bernstein [**DJB**], est une bonne illustration de l'adage *small is beautiful*. Il vient en remplacement de Bind [**BIND**], le serveur DNS fourni par l'ISC (*Internet Software Consortium*). Alors que Djbdns compte moins de 16 000 lignes de code source, la dernière version 9.2.3 de Bind en totalise plus de 300 000. La proportion des vulnérabilités est équivalente puisque Djbdns n'en a connu aucune alors que Bind en enregistre une multitude.

L'architecture complexe et difficile à maintenir de Bind est souvent montrée du doigt et joue très certainement un rôle dans les différentes failles de sécurité découvertes dans celui-ci. Comme le montre l'étude *Bound by Tradition* de Mike Schiffman [**BOUND**], cela n'empêche pas les utilisateurs de choisir, pour les trois quarts d'entre eux, Bind comme serveur DNS !

L'œil très critique de D.J. Bernstein a mis en cause la sécurité de bien des logiciels et a souvent attiré vers lui la rancœur de certains de leurs programmeurs. D.J. Bernstein conserve toutefois le même comportement lorsqu'il s'agit de ses propres réalisations : il a montré à de nombreuses occasions son souci de passer en revue tout ce qui pourrait ne pas aller, quels pourraient être les problèmes. Dans le cas de Djbdns, cet état d'esprit est flagrant et amplifié par les faiblesses du protocole lui-même [**MISC-DNS**]. Il a donc tenté de minimiser l'effet de plusieurs de ces problèmes liés au protocole [**DJB-NOTES**].

Il est possible de prendre en exemple le *cache poisoning* ou encore la menace due aux chaînes de confiance entre serveurs - il cite le cas du DNS de **w3.org** contrôlé par le serveur **w3csun1.cis.rl.ac.uk**, où **ac.uk** l'est par le serveur DNS **ns.eu.net**, le domaine **eu.net** étant sous la coupe de **beer.pilsnet.sunet.se...**

Un point important est donc que D.J. Bernstein :

- se pose les bonnes questions au moment du design du logiciel ;
- remet les choses en question ;
- reste proactif en proposant une solution adaptée à chaque problème lorsque cela est possible ;
- avertit l'utilisateur du logiciel des failles potentielles de sécurité dues au protocole et n'ayant pu être résolues ou minimisées.

Dans le cas du *cache poisoning* et contrairement à Bind, DNSCache ne gardera en mémoire que les réponses provenant d'un des domaines du niveau supérieur. Par exemple, pour **xxx.titi.net**, il ne gardera que les réponses venant de **titi.net** ou **net** mais en aucun cas de **toto.com** qui peuvent être plus facilement faussées.

Alors que **named** remplit à la fois le rôle de résolution, résolution récursive, mise en cache et publication des enregistrements DNS, Djbdns comprend plusieurs sous-programmes ayant chacun un rôle défini. Djbdns se décompose ainsi :

- **DNSCache** gère la résolution récursive et la mise en cache ;
- **TinyDNS** prend en charge la publication d'enregistrements DNS ;
- **Axfrdns** est responsable du transfert de zones (via TCP) ;
- **Axfr-get** est le penchant client de Axfrdns ;
- **DNSwall** filtre et répond aux requêtes inverses de DNS.

Que ce soit DNSCache, TinyDNS ou DNSwall, chacun est exécuté dans un environnement *chrooté* et comme processus utilisateur non-root. DNSCache possède une liste d'accès en dehors de laquelle elle ne répond pas aux requêtes DNS. Il utilise également un nouveau numéro de port UDP aléatoire pour éviter à un attaquant de prédire et de forger une réponse erronée (sauf s'il peut écouter le trafic réseau bien sûr). Quant à TinyDNS et walldns, ils ne supportent pas la récursion. Dans un souci de simplicité, D.J. Bernstein a refusé d'inclure certaines fonctionnalités dans Djbdns, comme par exemple DNSSEC [**DJB-DNSSEC**]. Dans le même souci d'efficacité, de clarté et de simplicité, les fichiers de configuration ont un format bien plus simple, ce qui d'une part réduit le risque d'erreur au niveau de l'administrateur système et d'autre part permet d'avoir du code source plus court et plus lisible pour ce qui concerne l'interprétation de ces fichiers. La configuration est divisée en de nombreux fichiers dans une hiérarchie claire. Pour la réplication, D.J. Bernstein a préféré laisser les administrateurs utiliser des outils éprouvés tels que SSH et Rsync plutôt que d'inclure ce genre de fonctionnalités dans Djbdns. Enfin, d'un point de vue général, les algorithmes utilisés montrent un bon compromis entre efficacité et simplicité : la solution retenue est rarement idéale d'un point de vue performance, mais est souvent claire et auditable tout en étant *une des plus performantes*.

Cette minutie dans le design et l'architecture ainsi que les nombreuses précautions prises par rapport à la sécurité placent donc Djbdns dans la catégorie des logiciels sûrs. D.J. Bernstein le clame bien haut et propose une récompense de 500\$ à quiconque trouvera une faille dans Djbdns !

POSTFIX

Postfix [**POSTFIX**] est sans doute un exemple des plus parlant lorsque l'on évoque la sécurité logicielle. Postfix est un MTA (*Mail Transfer Agent*) développé par Wietse Venema [**WIETSE**] et visant à remplacer Sendmail qui a à son passif une sérieuse liste de vulnérabilités.

• Les points les plus importants sont les suivants :

- Dès l'origine du projet, la sécurité a tenu un rôle important. En effet, un des objectifs est de proposer une alternative sûre au vulnérable Sendmail. L'architecture et le design de Postfix en ont donc tenu compte [**POSTSEC**] ;
- La simplicité est de mise : Postfix est découpé en plusieurs sous-programmes ayant chacun une fonction. De plus, chacun



de ces programmes est confiné et possède les privilèges les plus bas possible ;

→ Une défense en profondeur est présente : les contrôles sont multiples et la confiance en l'environnement externe ou en la communication entre les sous-programmes est très limitée.

La décomposition en sous-programmes est décrite par la "Big Picture" de Postfix [BIGPIC], et suit le même principe que dans le cas précédant de Djbdns.

Voici les différentes parties :

→ Un démon `master` est le seul à avoir des privilèges importants et contrôle ou appelle la majorité des autres programmes. Il est capable de stopper un démon au cas où certaines erreurs surviendraient et il surveille certaines limites imposées dans `master.cf`.

→ Lorsque Postfix reçoit un mail, que ce soit en provenance du système local (**Sendmail** transmet le message à **Postdrop**), venant d'Internet (démon `smtpd`), généré en interne pour retourner un mail de notification (`bounce`) ou encore redirigé (`local`), un démon central `cleanup` a à sa charge de vérifier le contenu du message, des types MIME, de l'enveloppe du message, la taille de certains champs, le format des adresses et éventuellement de réécrire ou ajouter certains champs.

→ Lors de la délivrance d'un mail, le démon `qmgr` (*queue manager*) gère les files de messages *incoming* et *deferred*. Une troisième file active contenant les messages prêts à être délivrés sert de tampon entre les deux files précédentes et `qmgr` s'assure que seul un petit nombre de messages s'y trouve, pour éviter une consommation trop importante de la mémoire.

Cette très brève description du fonctionnement de Postfix souligne bien que chaque démon possède une tâche limitée, que les vérifications (autant sur les mails transmis que de l'utilisation des ressources système) prend une place importante. La limitation du nombre d'actions de chaque démon induit que le risque d'erreur est plus faible.

En supplément du concept *small is beautiful*, Wietse Venema a souhaité réduire la portée d'action de chacun des démons précités : la plupart des démons sont chrootés, aucun mécanisme de `setuid/setgid` n'est utilisé et `master` contrôle les sous-programmes.

Non content de vérifier les différentes parties des mails en transit, les démons de Postfix ne se font pas confiance entre eux et encore moins en l'environnement : les messages IPC sont filtrés (taille des messages, format du contenu, certains *flags*, etc.) et des vérifications sont appliquées aux files de messages. Cette démarche est certainement plus efficace que de rajouter `smrsh` (*Sendmail Restricted Shell*) [SMRSH] à Sendmail en guise de palliatif, surtout si nous prenons en compte la vulnérabilité que celui-ci a connu [SMRSH-VULN].

Le principe de défense en profondeur, souvent mis en avant en sécurité des réseaux mais moins souvent en sécurité logicielle est donc appliqué dans le cas de Postfix puisqu'un attaquant devra exploiter plusieurs programmes (procédant chacun à un certain nombre de vérifications et ayant tous des droits limités) entre le démon à l'écoute du réseau et les files de messages sur le système lui-même.

Enfin, pour conclure ce paragraphe sur Postfix, notons que l'implémentation n'est pas en reste puisque la mémoire des chaînes de caractères et des buffers est allouée dynamiquement, les lignes des mails sont tronquées si la taille est trop importante, tout comme le sont les messages d'erreur ou de notification avant d'être renvoyés à `syslog`. Wietse Venema avoue tout de même que son code contient approximativement une erreur touchant à la sécurité toutes les 1000 lignes de code, ce qui porte à une trentaine le nombre de failles éventuelles dans Postfix.

SÉPARATION DE PRIVILÈGES

La séparation de privilèges part du principe que chaque programme étant potentiellement vulnérable, il demeure important que celui-ci soit exécuté avec les privilèges les plus faibles possibles. Comme un programme peut être divisé en deux ou plusieurs ensembles de fonctionnalités ayant des niveaux nécessaires de privilèges différents, l'objectif de la séparation de privilèges est de diviser ces ensembles pour que chacun dispose du niveau de privilège le plus bas possible. Souvent, la partie du programme à séparer est celle qui traite des données potentiellement dangereuses : données transmises par le réseau, saisies de l'utilisateur, etc.

Au niveau implémentation, il s'agit en général de réaliser un `fork()` et de forcer un des processus à changer d'identité et éventuellement à renoncer à certains de ses droits (utilisation de `setuid()`, `setgid()`, `setreuid()`, `setregid()`), à s'enfermer dans un environnement chrooté (utilisation de `chroot()`), etc. Si différents processus doivent ensuite communiquer et partager des informations, ils peuvent évidemment le faire en communiquant par IPC, tubes, *socket* ou tout autre moyen similaire.

Bien que n'étant pas le précurseur dans ce domaine, le projet OpenBSD a été l'un des plus actifs au niveau de la séparation de privilèges dans différents démons : OpenSSH, `syslogd`, `bgpd`, X Window system, etc.

Pour reprendre l'exemple qui a le plus fait parler de lui, celui de OpenSSH [OPENSSH] :

→ le processus de base, privilégié, crée un nouveau processus par `fork()` ;

→ ce nouveau processus cède ses privilèges et s'enferme dans une `chroot()` ;

→ le processus non privilégié, qui effectue la plupart des opérations (partie cryptographie, compression) délègue au processus privilégié les quelques opérations (principalement l'authentification) qu'il ne peut pas réaliser lui-même. Une communication par tubes est utilisée pour cela.

Dans le cas de `syslogd`, un processus est créé. Celui-ci a pour rôle d'être à l'écoute des requêtes de log. Dès sa création, il prend les droits d'un utilisateur normal `_syslogd` et s'enferme dans une `chroot` dans `/var/empty/`. Le processus parent, privilégié, est ainsi le seul à pouvoir accéder et écrire dans les fichiers de logs.

Le projet Openwall nous donne un autre bon exemple de séparation de privilège : celui du démon POP3 `popa3d` [POPA3D].



Pour pouvoir écouter sur le port réseau 110, il est lancé avec des droits super-utilisateur. Dès son lancement, il réalise un `fork()`. Le premier processus fils qui en résulte récupère les données d'authentification (utilisateur, mot de passe) et les transmet au processus père privilégié. Celui-ci crée un deuxième fils après les avoir récupérées. Ce dernier a à sa charge de vérifier l'authentification et en retourne le résultat au père.

Le processus père abandonne alors ses privilèges pour ceux de l'utilisateur nouvellement authentifié. Tous les traitements sont ensuite réalisés sous cette nouvelle identité.

Privman [**PRIVMAN**] est une librairie permettant de simplifier la tâche difficile qui est de séparer manuellement les fonctionnalités d'un programme (déjà existant) ayant besoin de différents niveaux de privilèges. Privman permet de diviser le programme en deux parties : l'une réalise les opérations privilégiées, l'autre les traitements du programme initial. Une *police* permet de spécifier les actions privilégiées que la partie non privilégiée est autorisée à réaliser. Ensuite, dans le code de la partie non privilégiée, il faudra remplacer certaines fonctions par leur équivalent Privman.

Par exemple, il sera nécessaire de remplacer `fopen` par `priv_fopen` pour ouvrir un fichier :

```
pidfd = open(pidfile, O_RDWR | O_CREAT, 0644);
```

[...]

```
if ((aclfile = fopen(aclpath, "r")) == NULL) {
    syslog(LOG_ERR, "cannot open access file %s: %s", aclpath, strerror(errno));
    return (0);
}
```

[...]

```
if (bind(s, (struct sockaddr *) &data_source, sizeof(data_source)) >= 0)
    break;
```

seront remplacés par :

```
pidfd = priv_open(pidfile, O_RDWR | O_CREAT, 0644);
```

[...]

```
if ((aclfile = priv_fopen(aclpath, "r")) == NULL) {
    syslog(LOG_ERR, "cannot open access file %s: %s", aclpath, strerror(errno));
    return (0);
}
```

[...]

```
if (priv_bind(s, (struct sockaddr *) &data_source, sizeof(data_source)) >= 0)
    break;
```

En soi, Privman a été annoncé comme révolutionnaire mais l'approche est la même que Systrace [**SYSTRACE**] dans le sens où il repose sur une politique d'actions autorisées. Systrace a toutefois l'avantage de ne pas nécessiter la modification du code source du programme.

L'avantage de Privman par rapport à Systrace se placerait au niveau d'une moindre dégradation des performances.

La sécurité étant un tout, il est utile de garder à l'esprit que le fait de minimiser les privilèges d'une entité ne réduit pas obligatoirement les risques sur l'ensemble d'un programme : si la communication et les échanges de messages entre les différentes entités ne suivent pas les règles élémentaires de sécurité, le risque peut devenir plus élevé. Pour conclure ce paragraphe sur la séparation de privilèges, il est important de noter que réfléchir à une bonne architecture en amont, pendant la phase de développement correspondant au design et à la création de l'architecture, est plus économe que procéder *a posteriori* à une séparation de privilèges.

PROGRAMMES SUID

Se débarrasser des programmes **S[UG]ID** permet aussi de limiter les privilèges de différents programmes sensibles. Toujours sous la coupe du projet Openwall, tcb [**TCB**], un module PAM, permet d'abandonner le bit SUID root du programme `passwd`. Dans le cas traditionnel de `passwd`, des droits de super-utilisateur sont nécessaires pour lire ou écrire dans `/etc/shadow`. tcb supprime cette contrainte en modifiant la structure de `/etc/shadow` : au lieu d'avoir un seul fichier commun à tous les utilisateurs, chacun dispose d'un fichier unique dont il est le propriétaire (et non accessible aux autres utilisateurs). Par défaut, le fichier en question est placé dans `/etc/tcb/utilisateur/shadow` et ne contient qu'une ligne au même format que celle qui aurait concerné ledit utilisateur dans `/etc/shadow`. L'utilisation est transparente pour l'utilisateur.

Toutefois, il est parfois plus simple de conserver ce bit SUID. C'est l'option qui a été choisie pour le programme `ping` de OpenBSD. `ping` a besoin de privilèges élevés pour accéder aux sockets ICMP. Pour limiter les risques, il est alors possible de ne conserver les droits root que le temps de créer le descripteur de fichier pour une socket de type RAW et de les abandonner pour le reste de l'exécution.

```
int
main(int argc, char *argv[])
{
    /* déclarations de variables */
    ...

    /* création de la socket RAW */
    if ((s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0)
        err(1, "socket");

    /* révocation des privilèges */
    seteuid(getuid());
    setuid(getuid());

    /* suite du programme */
    ...
}
```

Les quelques lignes de code ci-dessus montrent que seule une instruction (l'appel à `socket()`) est exécutée avec des privilèges root. Ces privilèges sont ensuite définitivement et irrémédiablement abandonnés. Dans ce cas précis, l'existence d'un bit SUID n'apporte pas de danger.



BCRYPT

Alors que la puissance des ordinateurs, et donc la facilité de casser un mot de passe chiffré par attaque *brute force* croît sans cesse, la méthode classique de chiffrement des mots de passe sous Unix (`crypt()`) est restée inchangée depuis 10 ans. Niels Provos a fait évoluer les choses avec `bcrypt` [BCRYPT], en proposant une amélioration de l'ancienne méthode. Alors que `crypt` utilise DES et limite la taille de mots de passe à 8 caractères, `bcrypt` utilise Eksblowfish. Alors que la *graine* (*salt*) de `crypt()` est de 12 bits, celui de `bcrypt` est de 128. Cela réduit la possibilité d'attaques par dictionnaires.

Cet exemple qui est maintenant la solution par défaut dans OpenBSD et qui a été reprise par le projet OpenWall [OPENWALL] illustre le fait que la cryptographie a également un rôle à jouer dans la sécurité logicielle et que le choix d'un algorithme de chiffrement ou de hachage adéquat, la gestion et le stockage de clé privée, la conservation de la *passphrase* d'une clé privée a parfois un rôle aussi important que le choix d'une bonne architecture.

Comme précisé dans l'article de ce dossier "dix dangers qui guettent le programmeur de cryptographie", il ne suffit pas de faire les bons choix, encore faut-il une implémentation réfléchie.

L'IMPORTANCE DE L'IMPLÉMENTATION

Il est difficile de parler d'implémentation modèle en ce qui concerne la sécurité logicielle. Les problèmes potentiels varient pour la plupart suivant le langage choisi. Le plus grand atout du programmeur à ce niveau est très certainement d'avoir conscience des différentes attaques existantes et de suivre les bonnes pratiques et le bon sens en général.

La pro-activité du projet OpenBSD mérite d'être citée puisqu'elle entre dans cette logique :

- les différents membres du projet sont sensibles aux failles potentielles ;
- avant même qu'une faille soit découverte, les fonctions considérées comme dangereuses sont remplacées autant que possible par un équivalent moins risqué ;
- le code source écrit par une personne est systématiquement passé en revue par un ou plusieurs autres programmeurs.

La sécurité est donc prise en compte dans les choix d'implémentation. Par exemple :

- la gestion des chaînes de caractères avec `strncpy` et `strlcat` qui remplacent avantageusement `strcpy`, `strcat` et dans une moindre mesure `strncpy` et `strncat` [OBSD-STR] ;
- l'introduction de mécanismes de prévention d'exploitation (W^X, ProPolice) [PROPOLICE] ;
- la séparation de privilèges dans de nombreux démons, comme vu précédemment ;
- l'utilisation de la cryptographie.

L'exemple des fonctions `strl*` est très parlant : lors d'un audit portant sur l'utilisation des fonctions `str*`, plusieurs membres du projet OpenBSD se sont rendu compte que dans de nombreux cas, l'utilisation censée être sûre des fonctions `strn*` (au lieu de `str*`) n'était pas efficace. Le principal problème avancé à propos de `strncpy()` est que celui-ci termine correctement une chaîne de caractères par un caractère nul, mais ne le fait pas dans le cas où la chaîne de caractères à copier est plus grande que le nombre de caractères donné en paramètre.

D'autre part, il paraissait peu clair aux programmeurs que la taille donnée en paramètre à la fonction `strncat` (qui elle garantit la terminaison de la chaîne par un caractère nul) ne comprenne pas le dernier caractère marquant la fin de la chaîne. Les fonctions `strl*` proposent donc d'harmoniser les comportements pour éviter les fautes d'implémentation : la terminaison par un caractère nul est garantie et la taille donnée en argument à chacune des fonctions est effectivement la taille de la chaîne finale.

Voici un exemple de manipulation de la variable d'environnement `HOME` permettant de comparer l'utilisation de chacune des fonctions.

Tout d'abord avec `strn*` :

```
strncpy(path, homedir,
sizeof(path) - 1);
path[sizeof(path) - 1] = '\0';
strncat(path, "/",
sizeof(path) - strlen(path) - 1);
strncat(path, ".foorc",
sizeof(path) - strlen(path) - 1);
len = strlen(path);
```

Ensuite avec l'équivalent `strl*` :

```
strlcpy(path, homedir, sizeof(path));
strlcat(path, "/", sizeof(path));
strlcat(path, ".foorc", sizeof(path));
len = strlen(path);
```

Pour terminer cette partie, il est important de rappeler que l'audit et la validation du code permet de repérer et supprimer bon nombre de fautes liées à l'implémentation.

Pour conserver l'exemple du projet OpenBSD, notons que cette activité y prend une place importante :

- Le processus d'audit est en place depuis les premiers mois du projet ;
- Plusieurs programmeurs sont dédiés à cette tâche uniquement ;
- Les audits de sécurité ne sont pas axés uniquement sur les problèmes de sécurité mais aussi sur les simples bogues. Si à un temps *t* un problème est considéré uniquement comme un bogue, il pourrait s'avérer que dans le futur, il devienne une faille de sécurité (découverte de nouveaux vecteurs d'attaques, risques mal jugés) ;
- Une même partie du code peut être auditée plusieurs fois (en particulier en cas de nouveaux types d'attaques) et par des personnes ayant des champs de compétences différents ;
- Concernant la validation, le code passe automatiquement sous les yeux de Theo de Raadt.



D'AUTRES ASPECTS IMPORTANTS

Plusieurs points méritent d'être mentionnés pour terminer notre étude de cas :

→ L'organisation de la qualité, la revue de code et les tests doivent impérativement tenir une place importante dans le cycle de développement logiciel. Cette étape est souvent oubliée, volontairement (moins intéressant pour les développeurs, retarde la sortie des produits...) ou non. Dans le cas de logiciels ayant un rôle critique dans le système d'information, des audits de sécurité sont souvent indispensables.

→ La documentation utilisateur prend un rôle non négligeable : à quoi bon réaliser un logiciel ayant un haut niveau de sécurité si les utilisateurs le configurent de manière dangereuse ou s'ils n'ont pas conscience des mécanismes de sécurité qu'ils peuvent actionner ?

→ La documentation du développement est tout aussi importante puisqu'elle permet aux différents programmeurs qui collaborent de conserver une certaine cohérence. Les deux principaux cas que nous avons vus ont, en plus de code très aisément lisible, une documentation de l'architecture du logiciel ; le code source de Postfix est aussi abondamment et très clairement commenté.

→ Les mécanismes de sécurité ne doivent pas rebuter l'utilisateur : si ceux-ci sont trop complexes ou s'ils impactent de manière trop négative la productivité, ils seront tôt ou tard désactivés...

→ Les logs, voire même des alertes de sécurité, sont importants car ils permettent d'avertir en cas de problème ou abus quelconque. Prévoir dans le développement du logiciel des mécanismes d'alertes simples aide à la sécurité globale du système d'information.

→ Une procédure de *patch* de sécurité (pouvant être automatisée suivant les cas) ou une procédure de remise en fonctionnement (comme c'est possible avec les *daemontools* dans le cas de Djbdns) peuvent être ajoutées.

CONCLUSION

Les différents exemples exposés ici montrent que la sécurité est un élément à prendre en compte tout au long du développement logiciel. Même si les phases en amont d'architecture et de design permettent de pallier un grand nombre d'erreurs couramment rencontrées, l'attention portée à la bonne implémentation et la phase de qualité ne doivent pas être laissées de côté. Le nombre d'exemples que nous aurions pu prendre est conséquent mais j'espère que ceux-ci donneront certaines pistes à nos lecteurs programmeurs pour qu'ils créent à l'avenir des logiciels étudiés et incorporant dès le départ un modèle de sécurité valable. Le niveau de sécurité global d'Internet en dépendra probablement !

Victor Vuillard
victor.vuillard@utbm.fr

Références

- [MURPHY] : <ftp://ftp.porcupine.org/pub/security/murphy.txt.gz>
Murphy's law and computer security, Wietse Venema.
- [OREILLY] : <http://www.securecoding.org/>
Secure Coding - Designing and Implementing Secure Applications, Mark G. Graff and Kenneth R. van Wyk, Ed. O'Reilly.
- [POSTFIX] : <http://www.postfix.org/>
Site officiel de Postfix.
- [WIETSE] : <http://www.porcupine.org/wietse/>
Wietse Zweitze Venema's home page
- [POSTSEC] : <http://www.postfix.org/security.html>
Postfix Overview - Security
- [BIGPIC] : <http://www.postfix.org/big-picture.html>
Postfix - the Big Picture
- [SMRSH] : <http://www.sendmail.org/faq/section2.html#2.13>
Sendmail Restricted Shell
- [SMRSH-VULN] : <http://www.sendmail.org/smrsh.adv.txt>
Sendmail smrsh bypass vulnerabilities
- [DJB] : <http://cr.yp.to/djbdns.html>
Djbdns
- [DJB] : <http://cr.yp.to/djb.html>
D.J. Bernstein home page
- [BIND] : <http://www.isc.org/products/BIND/>
I.S.C. Bind
- [BOUND] : <http://www.packetfactory.net/papers/DNS-posture/DNS-posture-1.0.pdf>
Bound by Tradition, Mike Schiffman.
- [MISC-DNS] : <http://www.misemag.com/sommaire.php>
Exploitation malicieuse du protocole DNS, MISC n°4.
- [DJB-NOTES] : <http://cr.yp.to/djbdns/notes.html>
Notes on the Domain Name System
- [DJB-DNSSEC] : <http://cr.yp.to/djbdns/forgery.html>
DNSSEC and DNS forgery
- [OPENSSSH] : <http://www.openssh.com/>
OpenSSH
- [POPA3D] : <http://www.openwall.com/popa3d/DESIGN.shtml>
The design of popa3d
- [TCB] : <http://www.openwall.com/tcb/>
tcb - the alternative to shadow
- [PRIVMAN] : <http://opensource.nailabs.com/privman/>
Privman - A library to make privilege Separation easy
- [SYSTRACE] : <http://www.citi.umich.edu/u/provos/systrace/>
Systrace - Interactive Policy Generation for System Calls
- [OBSID-STR] : <http://www.courtesan.com/todd/papers/strlcpy.html>
strlcpy and strlcat - consistent, safe, string copy and concatenation.
- [PROPOLICE] : <http://www.trl.ibm.com/projects/security/ssp/>
GCC extension for protecting applications from stack-smashing attacks
- [BCRYPT] : http://www.usenix.org/events/usenix99/provos/provos_html/index.html
A Future-Adaptable Password Scheme, Niels Provos and David Mazières.
- [OPENWALL] : <http://www.openwall.com/crypt/>
Modern password hashing for your software and your servers



La fin des "Buffer Overflow" dans Windows (?)

INTRODUCTION

Suite aux failles RPC découvertes l'été dernier, de nombreux codes d'exploitation ont été publiés sur Internet, mais curieusement la totalité d'entre eux avaient pour cibles Windows 2000 et Windows XP - Microsoft indiquait pourtant dans son avis de sécurité que toutes les versions de Windows étaient impactées. Quelles sont les différences entre Windows 2000/XP et Windows 2003 qui ont rendu l'écriture d'un *shellcode* si compliquée ? Nous allons voir que ces différences sont nombreuses, fondamentales, et pourtant peu documentées.

UNE ANALYSE DE LA FAILLE RPC

Mon objectif ici n'est pas de reprendre la théorie de la compilation et de l'exploitation de failles dans les programmes en C : pour les lecteurs qui n'y sont pas familiarisés, voir les anciens articles de MISC [3] ou l'abondante littérature disponible sur Internet. Prenons un *shellcode* pour la faille RPC, parfaitement fonctionnel sur un système Windows 2000 ou XP, et analysons ses effets sur un Windows 2003 : il apparaît assez rapidement que l'adresse de retour placée sur la pile ne se trouve pas au bon endroit, car la pile contient 4 octets supplémentaires. Quel peut bien être le rôle de ces 4 octets, qui semblent parfaitement aléatoires ?

LES PROTECTIONS INDUITES PAR LE COMPILATEUR

PROTECTION DE LA PILE

Présentation du cookie

Nous sommes en présence d'un "canary" (ou "cookie" en bon français), le nouveau mécanisme de protection de pile proposé par Visual Studio.NET. Ce mécanisme n'est cependant pas totalement nouveau : il a déjà été implémenté sous Unix dans le produit StackGuard. Son principe est d'insérer une valeur "vérifiable" entre les données utilisateur et les données compilateur placées en pile, selon le schéma suivant :

Visual Studio 6

Sommet de la pile
... reste de la pile ...
Paramètres de la fonction appelée
Adresse de retour (EIP)
Pointeur de cadre (EPB)
Gestionnaire d'exceptions (SEH) - si utilisé
Variables locales
Registres sauvegardés
Espace libre
...

Visual Studio.NET

Sommet de la pile
... reste de la pile ...
Paramètres de la fonction appelée
Adresse de retour (EIP)
Pointeur de cadre (EPB)
Cookie
Gestionnaire d'exceptions (SEH) - si utilisé
Variables locales
Registres sauvegardés
Espace libre
...

Le "cookie" est un nombre aléatoire généré à chaque chargement de l'image d'exécution. Il est stocké dans la section de données (".DATA") et reste fixe pendant toute la durée d'exécution. D'après l'excellent article de David Litchfield, "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server" [2], l'algorithme de génération du cookie s'écrit en C de la manière suivante :

```
int _security_init_cookie() {
```

```
    FILETIME ft;  
    unsigned int Cookie=0;  
    unsigned int tmp=0;  
    unsigned int *ptr=0;  
    LARGE_INTEGER perfcount;
```

```
    GetSystemTimeAsFileTime(&ft);  
    Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;  
    Cookie = Cookie ^ GetCurrentProcessId();  
    Cookie = Cookie ^ GetCurrentThreadId();  
    Cookie = Cookie ^ GetTickCount();  
    QueryPerformanceCounter(&perfcount);  
    ptr = (unsigned int*)&perfcount;  
    tmp = *(ptr+1) ^ *ptr;  
    Cookie = Cookie ^ tmp;  
    printf("Cookie: %8X\n", Cookie);  
    return 0;
```

Cette idée d'introduire une protection de la pile en temps réel n'est pas nouvelle, même dans les environnements Windows, mais les mécanismes utilisés diffèrent selon les versions du compilateur :

- Option /GZ ("enable runtime checks") dans Visual Studio 6;
- Options /RTCs ("stack checks");



- ♦ /RTCu ("find uninitialized variables");
- ♦ /RTCc ("catches conversions that truncate information") dans Visual Studio.NET et 2003.

Malheureusement ces protections en temps réel ne sont pas recommandées par Microsoft en mode "release", car elles sont consommatrices de mémoire, de CPU, et modifient de manière sensible le code généré ainsi que les optimisations possibles.

A contrario l'option /GS a été conçue pour minimiser l'impact sur les applications. Les développeurs de IIS 6.0 annoncent d'après leurs benchmarks que la performance ne chute que de 2% (information à prendre avec des pincettes).

Effets du /GS (Visual Studio 6)

Considérons la fonction suivante :

```
void fonction(void) {
    char string[10];
    [...]
}
```

Le code généré par Visual Studio 6 avec et sans l'option /GS est le suivant :

Sans /GS	Avec /GS
push ebp	push ebp
mov ebp, esp	mov ebp, esp
sub esp, 50h	sub esp, 50h
push ebx	push ebx
push esi	push esi
push edi	push edi
	lea edi, [ebp+var_50]
	mov ecx, 14h
	mov eax, 0CCCCCCCch
	repe stosd
[...]	[...]
pop edi	pop edi
pop esi	pop esi
pop ebx	pop ebx
	add esp, 50h
	cmp ebp, esp
	call __chkesp
	mov esp, ebp
mov esp, ebp	pop ebp
pop ebp	popn
retn	

Comme on le voit dans cet exemple, on est loin du code "académique" présenté traditionnellement dans les tutoriaux sur le "buffer overflow". Les protections ajoutées par le compilateur sont les suivantes :

- ♦ Avec ou sans /GS, allocation dans la pile d'un espace sensiblement plus large que celui requis par les variables locales - toutes les variables sont complétées (padding) avec 4 octets au minimum (ceci n'apparaît pas clairement dans l'exemple ci-dessus qui ne compte qu'une seule variable locale).
- ♦ Protection contre les attaques en "off by one".

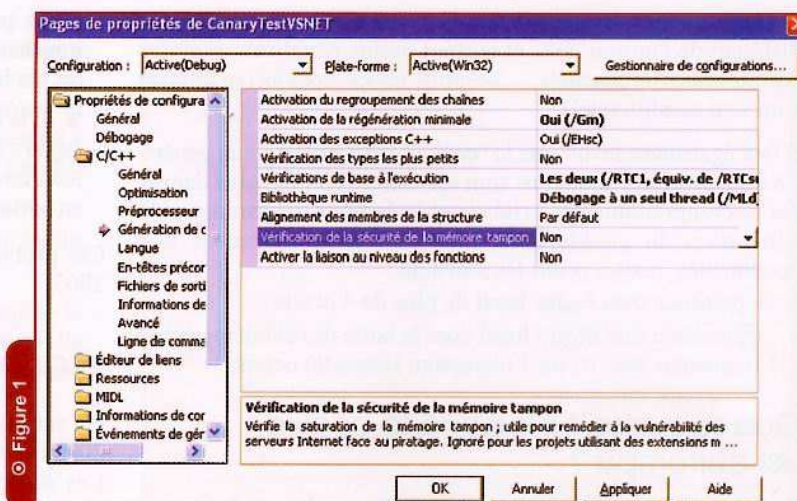


Figure 1

- ♦ Initialisation de cet espace avec l'octet 0xCC.
- ♦ 0xCC représente en assembleur l'instruction de débogage INT 3, qui a pour effet d'interrompre immédiatement l'exécution du programme si ce code vient à être exécuté hors débogage.
- ♦ Vérification en sortie de fonction que la taille de la pile n'a pas changé.
- ♦ Permet de détecter les fonctions ne nettoyant pas correctement la pile, par exemple à cause d'une convention d'appel différente entre l'appelant et l'appelé.

Effet du /GS (Visual Studio.NET)

Reprenons l'exemple précédent avec Visual Studio.NET. L'option /GS se trouve dans les propriétés du projet, rubrique C/C++, génération de code et se nomme en bon français "vérifier la sécurité de la mémoire tampon" (notez au passage la description de l'option) (voir figure 1).

Les résultats sont les suivants :

Sans /GS, sans /RTCsuc	Avec /GS, sans /RTCsuc
push ebp	push ebp
mov ebp, esp	mov ebp, esp
sub esp, 50h	sub esp, 54h
	mov eax, __security_cookie
	xor eax, [ebp+4]
	mov [ebp+var_4], eax
push ebx	push ebx
push esi	push esi
push edi	push edi
[...]	[...]
	mov ecx, [ebp+var_4]
	xor ecx, [ebp+4]
	call j__security_check_cookie
pop edi	pop edi
pop esi	pop esi
pop ebx	pop ebx
mov esp, ebp	mov esp, ebp
pop ebp	pop ebp
retn	retn



On constate que le mécanisme de protection est sensiblement différent de l'option /GZ, et surtout moins pénalisant pour les performances (la fonction `__security_check_cookie()` se limitant à un saut conditionnel).

Il faut également savoir que le "cookie" n'est pas toujours généré. En effet certaines situations sont identifiées comme sans danger par le compilateur. Les critères utilisés pour déterminer si la génération du cookie doit intervenir sont largement non documentés, mais a priori les suivants :

- présence d'un *buffer* local de plus de 4 octets ;
- présence d'un *buffer* local dont la taille des éléments est de 1 (caractère ASCII) ou 2 (caractère Unicode) octets.

Que se passe-t-il lorsque le cookie est corrompu ?

Tout d'abord, si le programme intercepte les exceptions de sécurité avec son propre gestionnaire, alors celui-ci est appelé. Microsoft ne recommande pas d'installer son propre gestionnaire d'exception de sécurité, car si celui-ci présente lui-même une faille, un attaquant peut tenter de l'exploiter comme on le verra par la suite.

Ensuite la fonction standard `UnhandledExceptionFilter()` est appelée - cette fonction affiche entre autres la possibilité de remonter la trace de l'erreur à Microsoft. Enfin un message d'erreur est affiché à l'utilisateur et le process se termine (**figure 2**).



Quels sont les problèmes posés par le cookie ?

Tout d'abord il existe de nombreux arguments philosophiques contre ce type de protection : les développeurs sont moins sensibilisés au développement sécurisé, les attaques en déni de service deviennent plus faciles, la taille du code généré augmente, la performance diminue, etc. Je ne m'étendrai pas plus sur ce terrain très subjectif. On se souvient également que la société Cigital avait été prompte à crier au loup sur la fiabilité de ce mécanisme. Ceci n'est pas un scoop, puisqu'on connaissait déjà des attaques sur StackGuard [6], mais leur vive réaction a probablement été motivée par une éviction de l'appel d'offre concernant l'audit de code des produits Microsoft.

D'un point de vue strictement technique, il existe effectivement deux problèmes identifiés dans Visual Studio.NET :

- ◆ Le cookie est initialisé au sein de la fonction `_CRT_INIT()`. Les applications ne faisant pas appel à cette fonction ne sont donc pas protégées. Ce cas peut se produire par exemple dans le cas d'une DLL possédant un point d'entrée non standard (autre que `DLLMain()`). Dans ce cas le compilateur ne génère

pas le prologue d'initialisation standard. Le cookie peut être initialisé "manuellement" via un appel explicite à `_CRT_INIT()` ou via la fonction `__security_init_cookie()` ;

- ◆ Si la fonction appelant l'initialisation du cookie possède des *buffers* locaux, alors le mécanisme de sécurité se déclenche de manière intempestive puisque la valeur du cookie en entrée et en sortie de l'appelant diffère.

Ces problèmes semblent avoir été corrigés dans Visual Studio 2003.

RÉORDONNANCEMENT DES VARIABLES

Un autre changement introduit dans le compilateur Visual Studio 2003 est le réordonnement automatique des variables locales. Les *buffers* (variables locales de type tableau définies par l'utilisateur) sont placés avant les variables locales de taille fixe (ex. entiers). Ainsi il devient plus difficile d'exploiter un débordement de *buffer* par rebond, c'est-à-dire en écrasant une variable locale judicieusement choisie (telle qu'un pointeur sur une fonction) pour tenter de faire exécuter un *shellcode* avant le retour de la fonction exploitée.

L'efficacité de cette méthode de protection dépend de l'algorithme de réordonnement, mais ne peut atteindre la perfection : par exemple le compilateur ne peut pas déterminer à coup sûr les variables exploitables dans une structure définie par l'utilisateur.

Visual Studio.NET

Sommet de la pile
... reste de la pile ...
Paramètres de la fonction appelée
Adresse de retour (EIP)
Pointeur de cadre (EPB)
Cookie
Gestionnaire d'exceptions (SEH) - si utilisé
Variables locales
Registres sauvegardés
Espace libre
...

Visual Studio 2003

Sommet de la pile
... reste de la pile ...
Paramètres de la fonction appelée
Adresse de retour (EIP)
Pointeur de cadre (EPB)
Cookie
Gestionnaire d'exceptions (SEH) - si utilisé
Buffers locaux
Variables locales
Registres sauvegardés
Espace libre
...

LES PROTECTIONS INDUITES PAR WINDOWS

STRUCTURED EXCEPTION HANDLING (SEH)

Les exceptions sont des événements imprévus (ex. division par zéro) qui se produisent au cours de l'exécution d'une thread. Toutes les conditions imprévues ne sont pas forcément fatales pour la thread qui peut être à même, elle ou son parent, de les traiter.



Lorsqu'une fonction veut intercepter des exceptions à l'aide de sa propre routine, elle installe alors un "gestionnaire d'exceptions utilisateur". L'adresse de l'ancien gestionnaire est sauvegardée sur la pile à des fins de chaînage. Le nouveau gestionnaire indique au système les événements qu'il prend à sa charge, et ceux qui doivent être remontés à son parent.

Au final si aucun gestionnaire d'exceptions utilisateur ne traite l'exception, Windows applique l'action par défaut (qui inclut en général un appel au débogueur ou à Dr Watson).

Bien entendu la présence en pile d'une adresse à laquelle le flot d'exécution peut être transféré a donné lieu à une nouvelle catégorie d'attaque, dont le plus illustre représentant est sans doute Code Red.

Pour pallier à ce problème, Windows XP introduit plusieurs nouveautés :

- ◆ Le format PE contient une entrée "Exception Directory", renseignée lorsque les options de compilation adéquates sont utilisées. Si cette entrée est présente, seuls les gestionnaires d'exceptions utilisateurs "enregistrés" peuvent être appelés. Le système refuse de transférer le contrôle à un gestionnaire inconnu, sauf si celui-ci se trouve hors de l'espace d'adressage du module courant (ex. gestionnaire système) (voir **Tableau 1** ci-dessous) ;

- ◆ Quelque part aux tréfonds de NTDLL.DLL (je laisse le soin aux lecteurs curieux de trouver où), les registres EBX, ESI et EDI sont sauvegardés, puis les registres EAX, EBX, ESI et EDI sont effacés avant l'appel du gestionnaire d'exception. Ceci permet de bloquer une partie des attaques en rebond ;

Dans le cas de Code Red, le gestionnaire d'exception malveillant pointait sur une instruction CALL EBX (sise dans MSVCRT.DLL), exploitant le fait que le registre EBX contenait à cet instant un pointeur vers la pile (et donc le shellcode).

- ◆ Le système refuse de transférer le contrôle à un gestionnaire d'exception situé dans la pile. Ce dernier point peut être mis en évidence à l'aide du code présenté ci-contre.

Ce code provoque une exception bien connue de type "Access Violation", mais un gestionnaire d'exceptions utilisateurs est installé. Sous Windows 2000, l'exception est interceptée et le programme termine normalement en affichant "Last ebx = ...". Sous Windows XP, le processus est tué car le gestionnaire installé sur la pile ne peut pas être appelé ;

```
#include <stdio.h>

void main() {
    unsigned int cint = 0;

    char unsigned bytes[] =
        "\xEB\x1B" // jmp gethandler
//start:
        "\x33\x0B" // xor ebx,ebx
        "\x33\xC9" // xor ecx,ecx
        "\x64\xFF\x31" // push dword ptr fs:[ecx]
        "\x64\x89\x21" // mov dword ptr fs:[ecx],esp
//search:
        "\x90" // nop
        "\x83\x3B\x00" // cmp dword ptr [ebx],0
        // ACCESS VIOLATION
//SEH cleanup
        "\x33\xC9" // xor ecx,ecx
        "\x8B\x04\x24" // mov eax,dword ptr [esp]
        "\x64\x89\x01" // mov dword ptr fs:[ecx],eax
        "\x83\xC4\x08" // add esp,8
        "\xEB\x1B" // jmp printit
//gethandler:
        "\xE8\xE0\xFF\xFF\xFF" // call start
//handler:
        "\x55" // push ebp
        "\x8B\xEC" // mov ebp,esp
        "\x53" // push ebx
        "\x8B\x5D\x10" // mov ebx,dword ptr [ebp+10h]
        "\x81\xC3\xA4\x00\x00\x00" // add ebx,0A4h
        "\xFF\x03" // inc dword ptr [ebx]
        "\x33\xC0" // xor eax,eax
        "\x5B" // pop ebx
        "\x8B\xE5" // mov esp,ebp
        "\x5D" // pop ebp
        "\xC3" // ret
//printit: (clean exit?)
        "\xFF\xD2" // jump edx

    __asm {
        lea eax,bytes
        lea edx,printit
        jmp eax
    }
    printit:
        mov [cint],ebx
    }

    printf("Last ebx = %x\n",cint);
}
```

Export	Certificates	Global Pointer	ImportAddress Table
Import	Base Relocation	Thread Storage	Delay Import
Resource	Debug	Load Configuration	COM Descriptor
Exception	Architecture	Bound Import	Reserved

Tableau 1 : Entrées du format PE, telles que reportées par la commande DUMPBIN de Visual Studio 2003



◆ Indépendamment de Windows, et sans rentrer dans les détails, Visual Studio 2003 intègre également des contrôles de cohérence à l'exécution sur la table des gestionnaires d'exceptions utilisés en C++ par la construction "try / catch / finally". Cette table est par exemple dans une zone mémoire en lecture seule.

QUELLES OPTIONS DE COMPILEMENT POUR WINDOWS ?

Comment détecter le compilateur utilisé pour générer Windows ? Les numéros de version interne des compilateurs sont :

- ◆ Visual Studio 6 : 6.0
- ◆ Visual Studio.NET : 7.0
- ◆ Visual Studio 2003 : 7.1

Le format PE stocke de nombreuses informations dans l'entête du fichier exécutable, ce qui permet de retrouver rapidement la version de l'éditeur de liens utilisé, par exemple avec la commande "DUMPBIN /HEADERS". Si on se base sur cette information dans le fichier RPCSS.DLL on trouve :

- ◆ Windows NT4 SP1 : 3.10
- ◆ Windows 2000 SP4 à jour des derniers correctifs : 5.12
- ◆ Windows XP SP1 à jour des derniers correctifs : 7.00
- ◆ Windows 2003 : 7.10

Windows XP a donc été compilé avec Visual Studio.NET, et Windows 2003 avec Visual Studio 2003. Le reste est plus difficile à interpréter : des outils internes à Microsoft ont probablement été utilisés. On notera que dans tous les cas l'entrée "Exception Table" du format PE est vide, donc peu utilisée par Microsoft en pratique.

Malheureusement il n'existe pas à ma connaissance de méthode pour retrouver les options de compilation utilisées, mais à l'analyse du code assembleur on peut supposer que seul Windows 2003 a été compilé avec le flag /GS. Microsoft indique également que Visual Studio 2003 et le ".NET Framework" ont eux-mêmes été compilés avec l'option /GS.

UNE PROTECTION PARFAITE ?

LES ATTAQUES PUBLIÉES PAR MICROSOFT

Malgré les contrôles et les protections intégrées à Visual Studio 2003 et au noyau Windows XP / 2003, des attaques restent possibles. On peut citer les attaques suivantes, reconnues par Microsoft [1], mais que je n'ai personnellement pas testées (je laisse ce soin aux lecteurs curieux) :

◆ L'attaque sur les tables de fonctions virtuelles ("V-Tables"). Grossièrement, le destructeur d'un objet C++ est un pointeur de fonction enregistré dans une table stockée en pile. Si un attaquant réussit à remplacer ce pointeur, il peut faire appeler la fonction de son choix au moment de la destruction de l'objet, donc potentiellement avant le contrôle du cookie.

Ce problème est corrigé dans Visual Studio 2003 car la V-Table se trouve derrière le cookie.

```
class Vulnerable {
public:
    int value;

    Vulnerable() { value = 0; }
    virtual ~Vulnerable() { value = -1; }
};

void vulnerable(char * pStr) {
    Vulnerable * vuln = new Vulnerable();
    char buf[20];

    strcpy(buf, pStr);
    delete vuln;
}
```

Programme vulnérable à l'attaque "V-Table Hijacking"

◆ Le remplacement de pointeurs ("pointer subterfuge").

Ce scénario est exploitable lorsqu'une fonction admet comme variables locales un pointeur sur une fonction et un buffer vulnérable. Le débordement de buffer permet d'écraser le pointeur de fonction. Il ne reste plus qu'à espérer que la fonction "pirate" soit appelée après le remplissage du buffer et avant le contrôle du cookie.

Ce problème est corrigé dans Visual Studio 2003 par le réordonnement des variables, de plus les pointeurs de fonctions sont placés derrière le cookie.

```
void vulnerable(char *bBuff, in cbBuff) {
    char bName[128];
    void (*func)() = MyFunction;

    memcpy(bName, bBuff, cbBuff);
    (func)();
}
```

Programme vulnérable à l'attaque "pointer subterfuge"

◆ Les attaques en deux étapes ("Two Stages Attacks").

La valeur d'un pointeur est altérée, et les données malveillantes copiées par le programme vers ce pointeur provoquent une défaillance ultérieure du programme.

Ce problème n'est pas corrigé à l'heure actuelle car très difficile à détecter.

```
void vulnerable6(char * pStr) {
    char buf[_MAX_PATH];
    int * pNum;

    strcpy(buf, pStr);
    sscanf(buf, "%d", pNum);
}
```

Programme vulnérable à une attaque en deux étapes



◆ Les attaques sur le tas (heap).

Par exemple un bogue exploitable de type "double free()" a été trouvé il y a quelques temps dans la librairie de compression ZLIB, ce qui prouve qu'il ne s'agit pas d'un cas d'école !

Il n'existe pas actuellement dans Windows ou dans Visual Studio de mécanisme de protection contre ce type d'attaque.

LES ATTAQUES PUBLIÉES PAR DAVID LITCHFIELD

David Litchfield [2] propose plusieurs méthodes innovantes mais assez similaires pour outrepasser les protections de la pile, et ce via le gestionnaire d'exceptions :

- ◆ La première consiste à provoquer une défaillance du programme et à exploiter une faille potentielle dans un gestionnaire existant.

→ Par exemple, le gestionnaire se trouvant dans NTDLL.DLL à l'adresse 0x77F45A34 semble être exploitable puisqu'il effectue un CALL EAX où EAX est calculé en fonctions de valeurs prises en pile.

- ◆ La deuxième consiste à faire pointer le gestionnaire d'exceptions sur une séquence de code dans le heap permettant un rebond en pile.

→ Par exemple, une séquence exploitable (à savoir CALL DWORD PTR [EBP+0x30]) a été trouvée dans le fichier UNICODE.NLS.

- ◆ La troisième consiste à faire pointer le gestionnaire d'exceptions directement sur du code injecté dans le heap. Cette méthode est beaucoup moins fiable car les adresses allouées dans le heap sont en général non prédictibles.

Litchfield propose également d'autres méthodes plus tordues mais fonctionnelles :

- ◆ Lorsque la fonction exploitée effectue une copie de pointeurs avant de retourner, il est possible d'écrire à des adresses arbitraires en mémoire avant que le cookie ne soit vérifié. Au moins trois méthodes d'exploitation de cette technique sont documentées :
 - ◆ Le cookie est une variable ordinaire, dont la valeur peut être remplacée par une valeur fixée par l'attaquant ;
 - ◆ S'il est possible d'écrire dans la section ".DATA" de KERNEL32.DLL à l'aide de cette technique, alors il est possible de modifier le chemin de chargement de FAULTREP.DLL, utilisée par le gestionnaire de sécurité par défaut, et d'injecter une DLL malveillante (attaque locale) ;
 - ◆ Une autre méthode consiste à écrire dans la section ".DATA" de NTDLL.DLL à un emplacement judicieux afin que la fonction LdrUnloadDll() exécute le code de l'attaquant ;
 - ◆ Si un gestionnaire d'exception de sécurité est installé, celui-ci est appelé lorsque le contrôle du cookie échoue - ce gestionnaire peut être lui même exploité. Cette attaque a été identifiée par Microsoft qui recommande clairement de ne pas tenter d'écrire soi-même son gestionnaire de sécurité.

CONCLUSION

Bien que les nouvelles protections intégrées discrètement par Microsoft dans Windows et dans Visual Studio rendent impossible l'exploitation "académique" des buffer overflow - provoquant des dénis de service -, ces protections restent largement contournables et ne dispensent pas d'une pratique de programmation sécurisée.

Le principal problème des systèmes basés sur les processeurs Intel x86 reste l'impossibilité de rendre des pages mémoire "non exécutables", contrairement à de nombreux autres processeurs, y compris l'AMD-64 et l'Itanium.

Après avoir intégré StackGuard dans ses produits, la prochaine étape pour Microsoft sera de porter PaX sous Windows. Le SP2 de Windows XP introduit le support du flag NX disponible sur AMD 32/64 et Itanium, ce qui est une avancée dans ce sens.

Nicolas RUFF / EdelWeb

Consultant expert en sécurité Windows

nicolas.ruff@edelweb.fr

Références

- [1] Présentation Microsoft sur les protections dans Visual Studio 2003

<http://std.dkuug.dk/JTC1/SC22/WG21/docs/papers/2003/n1462.pdf>

- [2] Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server

<http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf>

- [3] Articles MISC sur les "buffer overflows"

- ◆ MISC 1, 2, 3 : "protection contre l'exploitation des débordements de buffer"

<http://www.miscmag.com/articles/index.php3?page=212>

- ◆ MISC 8 : "shellcode sous Windows"

- [4] StackGuard

<http://www.immunix.org/stackguard.html>

- [5] Unmanaged C++ (MSDN)

http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vetchCompilerSecurityChecksInDepth.asp

- [6] Bypassing StackGuard and StackShield, Phrack n°56

<http://www.phrack.org/show.php?p=56&a=5>



La sécurité des réseaux 802.11 : quoi de neuf depuis un an ?



Il y a tout juste un an, MISC vous offrait un dossier [1] spécialement consacré à la sécurité des réseaux sans fil dont la conclusion se voulait plutôt pessimiste. Depuis, la Wi-Fi Alliance [2] s'est mise au travail pour proposer une solution, le WPA, en attendant la standardisation officielle de la norme 802.11i.

RAPPEL : LES PROBLÈMES DU WEP

Le WEP, pour *Wired Equivalent Privacy*, est le protocole qui devait assurer la sécurité des réseaux 802.11. Comme l'a montré un article du dossier [3], ce protocole souffrait de failles majeures :

- un mécanisme à clé partagée unique pour l'ensemble des participants et invariable dans le temps ;
- un mécanisme d'authentification unilatéral faible ;
- un mécanisme de chiffrement inadapté au contexte ;
- un mécanisme de contrôle d'intégrité faible.

RC4, UN CHOIX MALHEUREUX

L'algorithme de chiffrement utilisé par WEP est le système de chiffrement de flux RC4 [4]. Un tel système est en fait un générateur de séquence pseudo-aléatoire. Initialisé par une valeur (i.e. la clé), RC4 nous donne une séquence continue qui sera utilisée pour réaliser un XOR avec le flux de données à chiffrer. Les propriétés de cette séquence éliminent les problèmes de réutilisation des clés inhérents au XOR. Or, dans un contexte de chiffrement de trames, nous ne sommes pas en présence d'un flux, mais en présence de blocs de données (les trames) chiffrés indépendamment les uns des autres. Dès lors, la réinitialisation du générateur RC4 pour chaque trame induit une réutilisation de certaines séquences.

L'AUTHENTIFICATION

Le mécanisme d'authentification de WEP se fait sur la base d'un challenge aléatoire de 128 octets que l'on doit chiffrer avec la clé partagée pour s'authentifier. Ce mécanisme pose deux problèmes. D'abord, il ne permet pas à la station d'identifier le point d'accès dont un attaquant peut alors usurper l'identité pour mettre en place une redirection de type Man In the Middle de niveau 1. Ensuite, le challenge étant transmis en clair par le point

d'accès, un intrus peut, par déni de service (désassociation des clients valides), forcer des authentifications pour déduire les 128 premiers octets d'une séquence RC4 par attaque en clair connu sur XOR. Ceci lui suffit pour injecter par exemple des trames courtes sur le réseau.

LE CHIFFREMENT

L'initialisation du moteur RC4 se fait par concaténation de la clé partagée (clé WEP) avec un vecteur d'initialisation (IV) de 24 bits. Il est clair que l'espace des clés utilisées est alors nettement trop faible pour garantir une sécurité suffisante. Ensuite, le chiffrement s'appuyant sur XOR, la moindre connaissance des versions claire et chiffrée d'un même message rend immédiatement la séquence RC4 associée au vecteur d'initialisation courant. Enfin, une attaque mathématique, dite de Fluhrer, Mantis et Shamir (les auteurs), exploitant certains vecteurs dits faibles permet de retrouver la clé WEP de manière déterministe. Ce qui est fantastique avec cette attaque, c'est que la quantité de données nécessaire à sa réalisation croît quasi linéairement avec la taille de la clé, la où la recherche exhaustive croît exponentiellement. WEP 128 bits sera à peine plus du double de celui requis pour une clé de 64 bits. Le calcul de la clé sera ensuite effectué hors ligne et ne prendra que peu de temps par rapport à la collecte.

LE CONTRÔLE D'INTÉGRITÉ

Le contrôle d'intégrité s'appuie sur un simple CRC32. La connaissance du format de la trame lié à la linéarité du XOR permet à un attaquant de modifier arbitrairement des trames 802.11 chiffrées tout en étant capable de placer un CRC valide.

MÉLANGER LES INGRÉDIENTS, PASSER AU SHAKER ET SERVIR FRAIS

L'association de toutes ces vulnérabilités permet la mise en place de scénarii d'attaques dont le succès n'est conditionné que par



une collecte suffisante de données, comprendre une simple question de temps. Si les constructeurs ont éliminé les vecteurs d'initialisation faibles de leurs produits, il n'en reste pas moins que les attaques fondées sur la découverte systématique des séquences RC4 associées à tous les vecteurs d'initialisation possibles ne peuvent pas être mises en échec.

Par exemple, le scénario suivant peut être envisagé :

- la génération d'un grand nombre d'authentifications et leur observation permet déjà d'obtenir les 128 premiers octets des séquences associées aux IV utilisés ;
- l'utilisation des ces séquences permet de construire des trames courtes contenant des requêtes applicatives à destination d'une machine sur Internet contrôlée par l'attaquant (requêtes HTTP minimales par exemple) ;
- la récupération des réponses par écoute du réseau (le tri se fait par déchiffrement des 128 premiers octets de chaque trame) permet une attaque en clair connu ;
- construction d'une table associant à chaque IV sa séquence RC4 permettant à terme de déchiffrer n'importe quelle trame.

En résumé, ce n'est pas la joie... Le système ne peut pas être utilisé dans des environnements professionnels réclamant un minimum d'intimité.

L'INTRODUCTION DU 802.1X

Encore à l'état de *draft* l'an dernier, la norme 802.1x a été ratifiée. Depuis, 802.1x fournit un mécanisme d'authentification de niveau 2 particulièrement intéressant dans le monde sans fil fondé sur le protocole EAP, dont vous trouverez une description dans mon article [5] sur le sujet dans MISC 11.

802.1x/EAP permet d'offrir au monde sans fil des méthodes d'authentification diverses, du simple *login/password* à l'authentification forte par certificats x509, réglant d'emblée le problème de l'authentification WEP. Il offre en outre, pour certaines méthodes, la possibilité de dériver la clé de chiffrement des informations échangées pendant la phase d'authentification. Ainsi, sur un réseau sans fil dont l'authentification des stations se ferait par 802.1x, deux stations n'utiliseraient jamais la même clé de chiffrement.

Mieux, une même station n'obtiendrait pas la même clé pour deux authentifications différentes. Le 802.1x apporte donc une solution au problème de l'unicité de la clé partagée. En outre, en configurant l'expiration des sessions EAP, on peut forcer une station à renouveler son authentification et donc à renouveler sa clé de chiffrement. 802.1x devient alors un moyen de forcer le renouvellement des clés de chiffrement.

Selon le système utilisé, le support 802.1x peut ne pas être disponible. Chez Microsoft, le support 802.1x est disponible sous forme de patch [6] pour Windows 2000 SP3 et est fourni sous Windows 2000 SP4 et Windows XP. Ce support se présente sous la forme d'un service appelé "Configuration sans fil" qu'il faudra démarrer. Les autres versions de Windows devront faire appel à un client logiciel tiers. Chez Apple, un client natif est disponible à partir de Mac OS 10.3 Panther.

LES MÉTHODES D'AUTHENTIFICATION LES PLUS COURAMMENT IMPLÉMENTÉES SONT LES SUIVANTES :

EAP-MD5	Permet l'échange d'un login/password sous forme de challenge MD5. Outre le fait que cette méthode ne permette pas la dérivation d'une clé de chiffrement, elle ne permet pas d'authentifier le point d'accès et est vulnérable à des attaques consistant à utiliser un client valide pour obtenir la réponse au challenge. Cette méthode ne devra donc pas être utilisée dans le cadre des réseaux sans fil. Il est à noter que le client natif Microsoft ne propose cette méthode que pour les réseaux filaires, et non pour les liens 802.11.
LEAP	Méthode EAP développement par Cisco, qui fournit une authentification mutuelle par double challenge/réponse ainsi que la dérivation de la clé de chiffrement.
EAP-TLS	Permet l'authentification mutuelle forte par certificats x509 du point d'accès et de la station.
PEAP	Permet d'encapsuler dans une session TLS une deuxième authentification EAP (MSCHAPv2 ou GTC) après validation par la station d'un certificat présenté par le point d'accès.
EAP-TTLS	Offre les mêmes services que PEAP, mais de manière plus générique.

Pour les Unix en général, on pourra utiliser le client Xsupplicant du projet Open1x [7] pour l'authentification comme pour l'attribution dynamique des clés si le *driver* le supporte. D'autres clients propriétaires existent.

Poussée par Microsoft, Cisco et RSA, PEAP tend à s'imposer par rapport à EAP-TLS, qui nécessite la mise en place d'une architecture PKI (et peut être utilisée comme seconde phase de PEAP) et par rapport à EAP-TTLS. LEAP est encore rencontré mais disparaît au profit de PEAP, plus souple et sûre.

LA RÉPONSE DE LA WI-FI ALLIANCE : LE WPA

Les constructeurs devaient réagir et vite. En effet, la mouture nouvelle génération du système de sécurité, 802.11i, était en cours de discussion et son avènement ne semblait pas pouvoir intervenir assez vite (et il n'est toujours pas intervenu à l'heure où j'écris ces lignes). En outre, cette nouvelle norme supposait des modifications matérielles, rendant son déploiement long et coûteux. La Wi-Fi Alliance s'est donc mise au travail pour trouver une solution rapide pouvant fonctionner sur le matériel existant. C'est ainsi que le *Wi-Fi Protected Access* (WPA) a vu le jour [8].

WPA n'est en substance qu'une norme de fait, poussée par les constructeurs, améliorant le WEP en corrigeant une à une ses failles, en utilisant dans la norme 802.11i les solutions transposables au parc matériel existant.



Ainsi, WPA propose :

- un nouveau mécanisme d'authentification par secret partagé ou reposant sur 802.1x/EAP ;
- un nouveau mécanisme de dérivation des clés WEP utilisées pour le chiffrement des trames ;
- un nouveau mécanisme de contrôle d'intégrité des trames.

AUTHENTIFICATION

L'authentification d'une station en WPA peut se faire de deux manières différentes, par secret partagé ou par 802.1x. La Wi-Fi Alliance a en effet souhaité conserver un mode d'authentification simple à mettre en œuvre par rapport à 802.1x. En plus de la validation de l'accès des stations au réseau, la phase d'authentification sert de point de départ du mécanisme de génération des clés de chiffrement en fournissant à chacune des deux parties (station et point d'accès) une clé maîtresse de 256 bits que les intimes appellent PMK (*Pairwise Master Key*).

Lorsque l'authentification se fait en 802.1x, cette PMK est calculée de concert par la station et le serveur d'authentification, ce dernier la fournissant ensuite de manière sécurisée au point d'accès. Dans le cas d'une authentification par secret partagé, dit PSK (*PreShared Key*), la PMK sera dérivée du secret. Ce dernier peut se présenter sous deux formes :

- ♦ une clé de 256 bits fournie sous la forme d'une chaîne de 64 caractères hexadécimaux, qui constituera directement la PMK ;
- ♦ une *passphrase* de 8 à 63 caractères de laquelle sera dérivée la PMK par un rapide calcul normalisé.

Cette PMK servira par la suite à initialiser le mécanisme de génération des clés de chiffrement et de vérification d'intégrité.

La phase d'authentification par secret partagé est évidemment vulnérable aux attaques par dictionnaire ou par force brute [9]. Le choix éclairé d'un secret est donc une étape cruciale dans la mise en place d'un réseau WPA, puisque n'importe qui le connaissant sera capable de retrouver la PMK, puis à partir d'elle de recalculer toutes les clés qui seront utilisées pendant la session. Une clé partagée devra donc comporter au moins 128 bits aléatoires et une *passphrase* devra être longue d'au minimum 20 caractères pour rendre des attaques plus difficiles.

CHIFFREMENT ET INTÉGRITÉ

Par souci de compatibilité avec le matériel existant, le chiffrement s'appuiera sur un moteur WEP. Mais pour l'améliorer, la méthode de gestion des clés utilisées par ce moteur est modifiée. On utilise TKIP (*Temporal Key Integrity Protocol*), qui permet un renouvellement des clés de chiffrement nettement plus robuste dans le temps. WPA fournit en outre un système de contrôle d'intégrité cryptographique appelé Michael. Bref, TKIP est ni plus ni moins qu'une *rustine* pour boucher les trous du WEP, mais une bonne rustine qui atteint parfaitement son but.

Génération des clés

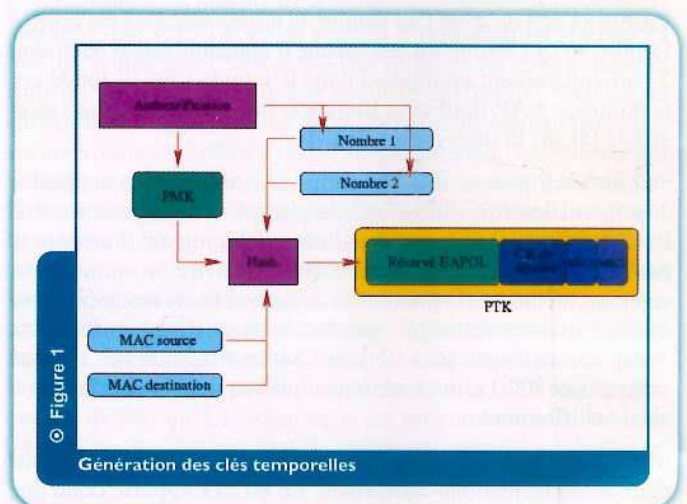
Une fois associé et authentifié auprès du point d'accès, nous sommes en possession de notre clé maîtresse de 256 bits, la PMK.

Comme nous l'avons mentionné plus tôt, cette clé sert de point de départ au mécanisme de génération de toutes les clés qui seront nécessaires pendant toute notre session Wi-Fi. Ce mécanisme de génération sert à générer trois clés appelées Temporal Keys. On *hash* la PMK avec deux nombres aléatoires (transmis en clair) et les adresses MAC des deux parties pour obtenir une clé de 512 bits appelée PTK (*Pairwise Transient Key*) au cours d'un échange appelé *4-Way Handshake*. 4 paquets EAPoL sont échangés entre les deux parties pour dériver la même PTK.

On va ensuite en extraire par découpage les clés temporelles qui nous intéressent (cf **figure 1**) :

- une clé de 128 bits appelée *Key Confirmation Key* (KCK), qui sert à une partie à prouver à l'autre qu'elle possède bien la bonne PMK ;
- une clé de 128 bits appelée *Key Encryption Key* (KEK) servant à assurer la distribution de la clé de groupe ;
- une clé de session de 128 bits qui servira dans le mécanisme de chiffrement ;
- deux clés de 64 bits dédiées au contrôle d'intégrité, une pour chaque sens de communication.

La clé de groupe, appelée *Group Transient Key* (GTK), sert à chiffrer les trafics *multicast* et de *broadcast* pour les sécuriser. Elle est évidemment la même pour tous les participants du réseau. Chacun d'entre eux la recevra chiffrée par sa KEK au cours d'un *Group Key Handshake*. Cette clé de 256 bits est la concaténation d'une clé de chiffrement et de deux clés de contrôle d'intégrité.



Chiffrement des données

Armés de nos clés temporelles, nous allons nous lancer dans TKIP et en particulier dans la génération de la clé qui sera fournie au moteur WEP de notre équipement Wi-Fi pour chiffrer la trame.

Cette clé est générée en deux phases (*mixing phases*), à partir des trois éléments suivants :

- ♦ la clé de session de 128 bits ;
- ♦ un vecteur d'initialisation (IV) de 48 bits appelé Extended IV ;
- ♦ l'adresse MAC de l'émetteur (48 bits).



La somme de contrôle d'intégrité

L'algorithme Michael permet de calculer une somme cryptographique permettant au destinataire de s'assurer de l'authenticité de la trame reçue. Cette somme est un bloc de données appelé MIC (*Message Integrity Control*) qui est calculé sur la trame claire complète (contrairement au CRC de WEP qui ne concernait que les données de la trame) à l'aide d'une des deux clés temporelles de 64 bits dérivées à cet effet de la PTK. Le MIC est alors placé à la suite du bloc de données de la trame. Ce bloc final (données + MIC) est envoyé pour chiffrement au moteur WEP qui, comme à son habitude, en calculera le CRC, le placera à la suite et chiffrera le tout à l'aide de la PPK. Lors de la réception de la trame par son destinataire, le bloc sera déchiffré, son CRC vérifié et le MIC extrait. On reconstruira alors la trame originale dont on pourra vérifier l'authenticité en recalculant un MIC et en le comparant à celui reçu.

Différentes études sur Michael s'accordent sur sa solidité. Il ne serait pas possible de créer un message valide de toute pièce sans être en possession de la clé adéquate, ce qui est une bonne chose. Cela nous assure une bonne protection contre la modification des trames. Il est par contre possible de générer des dénis de service. En utilisation normale, la vérification du MIC ne devrait jamais échouer. En effet, les détériorations accidentelles de trames devraient être détectées lors de la vérification du CRC par le moteur WEP déchiffrant le bloc de données. Si cette dernière échoue, la trame est détruite et le MIC ne sera pas examiné. Lorsqu'on détecte un MIC erroné, cela implique que le CRC était bon. Or, on sait que la probabilité pour qu'une erreur dans la trame ne puisse pas être détectée est faible et que cette éventualité, si elle se reproduit trop souvent, résulte sûrement d'une attaque. La norme considère qu'au dessus d'une erreur de vérification de MIC par tranche de 60 secondes, on a affaire à une attaque et que les parties doivent renégocier l'association pour changer les clés.

Un attaquant pourra donc mettre cette protection à profit pour générer des désassociations en injectant des trames avec un CRC correct mais un faux MIC. Pour ce faire, il capturera une trame et la modifiera exactement comme il le faisait en WEP classique, mais sans toucher au MIC. La trame passera donc le déchiffrement WEP, puis la vérification du CRC, mais pas celle du MIC comme il le souhaitait.

Ce n'est cependant pas si simple puisque le jeu n'est pas possible : la réutilisation d'un IV met la trame hors séquence et entraîne sa destruction. L'attaquant devra donc empêcher la trame originale de parvenir à sa destination, ce qui implique la réalisation d'une redirection de trafic.

Notons que les trafics multicast et de broadcast sont traités de la même manière. Cependant, les clés servant à initialiser TKIP (clé de session) et à construire le MIC seront extraites de la GTK comme vu précédemment.

CONCLUSION

En résumé, WPA consiste à apporter au système existant une couche logicielle de sécurité en attendant la nouvelle génération de matériel qui supportera 802.11i.

Les éléments de sécurité ajoutés font d'ailleurs partie de cette dernière norme. WPA peut être décrit simplement par cette simple formule : $WPA = 802.1x/EAP + TKIP + MIC + WEP$

Le niveau de sécurité apporté par WPA est fort, mais pas complètement suffisant. En effet, il ne s'intéresse qu'au trafic de données, laissant de côté, à l'instar de WEP, le trafic de contrôle. WPA ne protège donc pas des dénis de service s'appuyant sur ce trafic, comme l'émission de trames de désassociation qui ont été décrites dans MISC 6 [10].

802.11i, LA NORME TANT ATTENDUE

La norme 802.11i, qui sera aussi appelée WPA2, doit apporter une sécurité optimale aux réseaux Wi-Fi. Dans la mesure où WPA reprend de nombreux éléments et parce ce standard est encore à l'état de draft, nous n'allons pas entrer dans une description détaillée, mais vous en brosser un rapide tableau indicatif.

Les éléments de 802.11i repris par WPA sont les suivants :

- ♦ utilisation de 802.1x ou de secrets partagés pour l'authentification ;
- ♦ système de génération de clés (PMK, PTK, et clé de session) ;
- ♦ utilisation possible de TKIP et de Michael (un équipement WPA2 sera interopérable avec un équipement WPA).

802.11i va cependant plus loin et introduit un certain nombre de concepts dont les deux majeurs sont la pré-authentification et l'utilisation de AES.

PRÉ-AUTHENTIFICATION

À l'heure actuelle, le trafic de contrôle n'est pas protégé, ni par WEP, ni par WPA. De fait, il est assez simple pour un attaquant d'injecter des trames pour, par exemple, provoquer des désassociations. Nous l'avons vu auparavant, de tels dénis de service, au-delà de leur côté agaçant pour la victime, servent à provoquer de nouvelles authentifications susceptibles d'apporter à l'intrus de précieuses informations.

La méthode retenue pour pallier à ce problème est de faire intervenir l'authentification de la station avant l'association au point d'accès. Ainsi, des clés pourront être dérivées pour protéger le trafic de contrôle et éviter son usurpation. Une telle méthode est lourde de conséquence, particulièrement dans la manière d'intégrer 802.1x dans ce schéma, dans la mesure où, actuellement, sa mise en œuvre nécessite un lien de niveau 2 pour transporter EAP, lien qui n'est actif qu'une fois la station associée. 802.11i répondra à cela.

CHIFFREMENT ET INTÉGRITÉ PAR AES

Le chiffrement et le contrôle d'intégrité seront fournis par des algorithmes utilisant AES. L'ensemble s'appelle CCMP pour *Counter Mode CBC-MAC Protocol*. Le chiffrement sera assuré



par AES dans un mode rétroactif en utilisant la clé de session dérivée de la PTK. La tambouille effectuée par TKIP pour dériver la PPK de cette clé n'est plus nécessaire. AES est en effet un algorithme de chiffrement par bloc qui utilise la même clé tout au long de la session. Le MIC sera calculé toujours en utilisant AES CBC-MAC, initialisé par l'IV de 48 bits, sur l'ensemble de la trame en utilisant aussi la même clé que pour le chiffrement. De fait, la PTK pour CCMP ne comporte que 3 clés de 128 bits. Ce MIC de 64 bits est alors ajouté à la suite des données et est chiffré avec. De même, la GTK pour CCMP est longue de 128 bits et ne comporte qu'une seule clé.

L'utilisation du moteur WEP est donc complètement supprimée, d'où la nécessité pour les constructeurs d'implémenter un moteur AES dans leur matériel.

ET ENCORE ?

Quelques fonctionnalités seront ajoutées, en particulier sur la sécurité du mode Ad Hoc (IBSS) qui n'est aujourd'hui pas couverte par WPA. Ces réseaux pourront alors profiter des apports de 802.1x, du 4-Way Handshake et d'une négociation de clés de groupe. Les mécanismes sont les mêmes qu'en mode Infrastructure (BSS) mais en prenant en compte les spécificités d'un IBSS où chaque station est l'égale des autres. Un autre algorithme de chiffrement, appelé WRAP (*Wireless Robust Authenticated Protocol*), utilise AES en mode OCB. En fait, il s'agit du premier mécanisme proposé pour le chiffrement AES de 802.11i et qui a été remplacé par CCMP. Mais comme certains constructeurs l'avaient déjà implémenté, il est inclus comme composant optionnel dans la norme. Ceci étant, il serait moins sûr que CCMP, d'où son abandon. On ne l'utilisera donc que si on n'a pas le choix, auquel cas l'équipement ne serait pas 802.11i compliant.

Il est à noter que le driver HostAP [11] inclut dans sa version de développement (voir le CVS) des routines CCMP entièrement implémentées logiciellement. Le driver HostAP parvient en effet à reprendre à son compte la génération des données des trames qu'il envoie, sans passer par les routines dédiées du *firmware*, ce qui n'est pas forcément possible avec toutes les cartes Wi-Fi.

Ce qui m'amène à faire un rapide coup de pub pour la *libwlan* [12] qui permet l'injection de trames 802.11 de manière plus aisée que le driver AirJack [13].

Pour plus de détail, je vous renvoie à une présentation technique et assez complète de 802.11i [14].

GUIDE PRATIQUE

Tous les points d'accès récents disponibles sur le marché supportent WPA avec authentification par secret partagé au minimum. Les points d'accès plus professionnels auront le support 802.1x en sus. Concernant les systèmes d'exploitation, seuls Windows XP et MacOS 10.3 Panther disposent d'un client WPA natif. Les autres systèmes devront utiliser soit les interfaces fournies avec les drivers, soit un client tierce partie. Sous GNU/Linux, les drivers HostAP et Prism54 [15] supportent d'ores et déjà WPA.

Le **tableau 1** page suivante résume les fonctionnalités des solutions disponibles aujourd'hui, en comparaison avec ce qu'apportera 802.11i.

LES RECOMMANDATIONS SUIVANTES POURRONT ÊTRE APPLIQUÉES

→ WEP avec clé statique

Réservé à un usage domestique restreint en 128 bits sur du matériel ne supportant pas le WPA (et ne pouvant pas être mis à jour). Migrer dès que possible. Essayer de maîtriser la couverture hertzienne des points d'accès.

→ WEP avec 802.1x

Niveau de sécurité moyen. Penser à configurer l'expiration des sessions EAP au niveau du serveur d'authentification pour forcer le renouvellement régulier de la clé de chiffrement (1h à 2h selon la charge du réseau). Vérifier si votre matériel ne peut pas être mis à jour (*update* de *firmware*) pour supporter WPA.

→ WPA/WPA2 avec secret partagé

Parfait pour l'usage domestique. Utiliser un secret partagé unique par adresse MAC si possible. Choisir un secret partagé fort.

→ WPA/WPA2 avec 802.1x

Configuration professionnelle recommandée pour une sécurité optimale. Bien choisir ses éléments d'identification et sa méthode EAP.

Dans tous les cas, on mettra à profit le filtrage d'adresses MAC sur le point d'accès. Même si cette mesure peut être facilement contournée, elle permettra de décourager les opportunistes peu motivés. La maîtrise du rayonnement de ses points d'accès, lorsqu'on utilise WPA, n'est plus une obligation et constitue un apport de sécurité secondaire, au même titre que le filtrage d'adresses MAC, qui ne devra pas nuire à la bonne couverture de la zone de réception.

Certains produits vous permettront d'utiliser WEP et WPA en même temps, dans le but avoué de rendre les migrations du premier vers le second plus faciles ou encore de ne pas priver les stations ne supportant pas WPA d'accès Wi-Fi. Une telle configuration ne doit évidemment pas être utilisée en production, puisque sa sécurité se résume à celle du WEP qui sera mis en place. Et pourquoi ne pas laisser un WEP 40bits pour les vieilles cartes à côté du WPA/802.1X/PEAP tant qu'on y est ?

Par contre, sur les produits haut de gamme qui supportent la création de VLAN et leur transport via un lien 802.1q, il est possible de définir plusieurs réseaux (ESSID) de configuration de sécurité différente affectés chacun à un VLAN distinct.

On pourra par exemple avoir :

- un VLAN ouvert pour un accès Internet de type *hotspot* ;
- un VLAN avec WEP statique permettant un accès à quelques ressources non critiques en consultation et Internet ;



Fonctionnalités	WEP	WEP+802.1x	WPA+PSK	WPA+802.1	802.11i/CCMP
Authentification	Clé partagée	EAP	Clé partagée	EAP	Pré-Auth EAP
Chiffrement	RC4	RC4	RC4	RC4	AES
Distribution de clés	-	EAP	-	EAP	EAP
Taille de clé	64/128	64/128	128	128	128
Variation des clés	WEP	WEP	TKIP	TKIP	Inutile
Élément variable	IV	EAP+IV	ExtIV	EAP+ExtIV	Inutile
Intégrité données	CRC	CRC	Michael	Michael	CCM
Intégrité entête	-	-	Michael	Michael	CCM
Anti-rejeu	-	-	Ext.IV	Ext.IV	Ext.IV

Tableau 1 : fonctionnalités des solutions disponibles

- un VLAN avec WEP statique permettant un accès complet via IPSEC ;
- un VLAN avec accès WPA/802.1x permettant un accès complet ;
- etc.

Sur ces mêmes produits, il sera possible avec l'authentification 802.1x d'affecter un VLAN à une station en fonction des éléments d'authentification.

Des configurations tout aussi intéressantes que la précédente pourront être mises en place :

- un VLAN pour un accès Internet de type hotspot pour les utilisateurs non authentifiés ;
- un VLAN pour les utilisateurs normaux ;
- un VLAN pour les administrateurs ;
- etc.

On peut donc mettre en place des architectures suffisamment souples pour accueillir des systèmes divers aux fonctionnalités différentes et des profils d'utilisateur variés sans pour autant mettre en péril la sécurité de l'ensemble du réseau.

CONCLUSION

Après une période difficile qui a considérablement freiné son développement et marqué les esprits assez négativement, le Wi-Fi se refait une image dans le monde de la sécurité avec le WPA. Il est aujourd'hui tout à fait possible de mettre en œuvre un réseau sans fil 802.11 dans de bonnes conditions, sans porter atteinte à l'intégrité de l'ensemble de son système d'information. Cependant, WPA reste une norme de transition.

802.11i entraînera de profonds changements et le renouvellement du parc matériel en cas d'utilisation d'AES pour le chiffrement. Le coût d'une telle migration associé à des reports successifs de la publication définitive de la norme ne laissent malheureusement pas présager le déploiement généralisé de cette solution. Mais qui sait, dans un an, bien des choses auront changé.

RÉFÉRENCES

- [1] Dossier (In)Sécurité du wireless, MISC 6, mars/avril 2003.
- [2] Wi-Fi Alliance, <http://www.wi-fi.org/>
- [3] Éric Filiol et Frédéric Raynal, *La sécurité du WEP*, MISC 6.
- [4] Simon Guillem-Lessard, Tutoriel sur la cryptographie - RC4, http://www.uqtr.ca/~delisle/Crypto/prives/flux_rc4.php
- [5] Cédric Blancher, *EAP*, MISC 11.
- [6] *Using 802.1x Authentication on Computers Running Windows 2000*, <http://support.microsoft.com/default.aspx?scid=kb;en-us;313664>
- [7] *Open1x, Open Source implementation of IEEE 802.1x*, <http://www.open1x.org/>
- [8] *Wi-Fi Protected Access*, http://www.wifialliance.com/OpenSection/Protected_Access.asp
- [9] Robert Moskowitz, *Weakness in Passphrase Choice in WPA Interface*, <http://wifinetnews.com/archives/002452.html>
- [10] Daniel Polombo et Cédric Blancher, *Attaques sur le 802.11b*, MISC 6.
- [11] *Host AP driver for Intersil Prism2/2.5/3*, <http://hostap.epitest.fi/>
- [12] Libwlan, <http://libwlan.tuxfamily.org/>
- [13] AirJack, <http://802.11ninja.net/airjack/>
- [14] N. Cam-Winget, T. Moore, D. Stanley et J. Walker, *IEEE 802.11i Overview*, http://csrc.nist.gov/wireless/S10_802.11i%20Overview-jw1.pdf
- [15] *Linux Driver for the 802.11g Prism GT / Prism Duette / Prism Indigo Chipsets*, <http://www.prism54.org/>

Cédric Blancher

Consultant sécurité, Arche groupe Omnetica

cedric.blancher@arche.fr



Du miel en pot virtuel avec VMware

INTRODUCTION

Comparé à un simple *honeypot*, la mise en place d'un *honeynet* augmente les ressources nécessaires : si le réseau devant passer sur l'autel des offrandes doit contenir X hôtes, il faudrait théoriquement X ordinateurs physiques. VMware [1] est un logiciel permettant d'avoir un *honeynet* complet sur le même ordinateur physique. Pour cela, il émule le matériel sensé composer le système physique. En soit, ses capacités sont similaires à celles de User Mode Linux. L'inconvénient de VMware par rapport à ce dernier vient du fait qu'il s'agit d'un logiciel commercial - les sources ne sont pas disponibles, ce qui peut limiter certaines techniques permettant d'observer l'activité du pirate sur le système pris pour cible. Son avantage de taille est qu'il permet d'utiliser quasiment tous les systèmes d'exploitations disponibles sur architecture Intel : des différentes versions de Windows à Linux, en passant par les *BSD ou encore Solaris pour i386.

INSTALLATION ET CONFIGURATION

Nous n'allons pas traiter dans cette partie du cheminement complet de l'installation de VMware ni de toutes ses options. Nous examinerons plutôt les points importants de l'installation et de la création des hôtes clients (les systèmes fonctionnant sur le matériel émulé) ayant une incidence directe sur la mise en place de honeypots avec VMware. La version utilisée de VMware est VMware Workstation 4 sous Linux ; ceux qui souhaitent tester la mise en place de honeypots ou de honeynets avec VMware peuvent télécharger [2] une version d'évaluation (complète mais limitée à 1 mois). VMware fonctionne aussi bien sous Windows que sous Linux.

L'installation en elle-même ne devrait pas poser de problème. Des packages *RPM* ou des archives *TGZ* sont disponibles. Lors de la configuration, il est nécessaire d'avoir les *headers* du noyau utilisé. Lors de la configuration avec *vmware-config.pl*, différentes interfaces réseaux virtuelles sont créées pour connecter les hôtes clients suivant ces 3 modes :

→ **Bridged networking** : les hôtes clients utilisant ce mode ont un accès direct sur le réseau. Un mécanisme de bridge est utilisé au niveau du système de base [3]. Chaque hôte client dispose alors d'une adresse MAC et d'une adresse IP sur le réseau sur lequel il est connecté ;

→ **Host-only networking** : les hôtes clients sont contenus dans un réseau virtuel privé pour communiquer uniquement avec le système de base et entre eux. Les hôtes clients n'ont donc pas d'accès direct au réseau sur lequel est connecté le système de base ;

→ **NAT networking** : les hôtes clients ont accès au réseau via un système de translation d'adresse (NAT).

Le choix d'un de ces 3 modes dépend essentiellement de la topologie du réseau sur lequel est implanté le honeynet, de la politique de capture de données (sniffers, IDS) et de confinement (utilisation de limites de trafic au niveau d'un *firewall* ou de *Snort Inline* par exemple). Le choix de Host-only networking permet d'avoir un plus grand contrôle au niveau du système de base qui sert alors également de passerelle. Il est toutefois possible d'avoir un second système physique pour la capture des données et le confinement des honeypots, auquel cas l'utilisation du Bridged networking pourra être envisagée.

Lors de la création des hôtes clients, le choix du type de disque est important. VMware offre deux possibilités :

→ **Emploi d'un disque physique** : dans ce cas, un disque dur physique ou une de ses partitions est utilisé pour installer l'hôte client. Il est fortement conseillé de prendre un disque dur différent de celui occupé par le système de base. L'avantage de ce choix réside dans la possibilité de se servir d'outils de forensic comme par exemple *The Coroner's Toolkit* [4]. Comme pour un système normal, on pourra cloner les installations avec *dd* ou encore *g4u* ;

→ **Emploi d'un disque virtuel** : ce cas correspond à l'option par défaut de la dernière version de VMware. Un disque virtuel consiste en un fichier contenant la ou les partitions de l'hôte client. Les avantages sont multiples. Tout d'abord, l'installation est plus simple. Pour mettre en place des honeynets à plusieurs endroits, il suffit de recopier ces fichiers contenant l'installation des hôtes clients. De même, une fois compromis, nous avons la capacité de copier les fichiers des hôtes clients sur une autre station pour analyse et remettre le honeynet en production en restaurant les fichiers de base préalablement sauvegardés. Un autre avantage est l'espace disque nécessaire sur le système de base : on peut en effet définir une partition du disque virtuel de 20 Go ; celle-ci ne prend comme taille que ce qui est



réellement occupé sur l'hôte client (quelques centaines de Mo). Le fichier contenant le disque virtuel grandit seulement si l'espace utilisé sur l'hôte client croît.

Une fois les disques créés, l'hôte client a la capacité de s'en servir de différentes manières :

→ **mode persistant** : si cette option est choisie, les modifications faites sur le disque dur sont écrites directement comme dans le cas d'un système ordinaire ;

→ **mode non-persistant** : celui-ci permet de ne jamais modifier le système lancé sous VMware. Une fois le système installé, si on place le disque dur en mode non-persistant, aucune modification n'y sera apportée et le système sera identique à chaque redémarrage ;

→ **mode normal** : ce mode est de loin le plus intéressant. Dans les versions de VMware précédant la 4, il correspond au mode **Undoable**. Après la mise en service de l'hôte client, aucune modification n'est apportée au fichier contenant le disque dur virtuel ou sur le disque dur physique utilisé. Toutes les modifications apportées sont stockées dans un fichier **.REDO**. Celui-ci peut être fusionné au disque utilisé lors de l'arrêt de l'hôte client ou si l'on choisit de faire un *snapshot*. Avec cette option de *snapshot*, VMware prend une sorte d'image du système à un instant *t*. Dans le cas où un pirate est venu butiner dans notre honeypot, il nous sera utile de prendre une image du système alors qu'il y est encore connecté et copier cette image sur un autre système pour analyse.

SURVEILLER LE HONEYPOT

Comme vous l'avez sûrement pressenti dans la partie précédente, le fait que VMware puisse enregistrer dans un fichier séparé tous les changements ayant pu être faits sur le disque dur apporte un réel plus pour surveiller le honeypot. Cela permet de scruter tous les changements sans même que le pirate ne puisse s'en rendre compte, ce qui reste bien plus pratique et discret qu'un *kernel rootkit*. Ceux-ci permettent, grâce à un module du kernel, de journaliser l'activité du système (processus, système de fichier, *tty*, etc.) de manière furtive ou de masquer l'activité de programmes qui serviront à espionner le pirate [5]. Ces rootkits font en général partie de la trousse à outils des attaquants qui s'en servent pour masquer leurs activités, mais ils peuvent être aussi très pratiques pour la création de honeypots.

Un des principes des honeypots ou honeynets est que le moindre trafic réseau qui lui est dirigé est suspicieux. De même, la moindre modification des données sur le disque dur l'est. La surveillance des logs d'un IDS couplée à celle du fichier **.REDO** de VMware est donc très efficace pour savoir ce qui se passe sur le honeypot. De plus, ce fichier permettra d'avoir un historique de ce qui s'est passé et pas seulement un état des lieux une fois que le vandale les a quittés. Si en délaissant la scène du crime, le pirate fait un `rm -rf /` ou encore plus un `dd if=/dev/zero of=/dev/hda` on aura toujours les données nécessaires à l'analyse de ce qui s'est réellement passé.

Bien sûr, le fichier **.REDO** contient beaucoup de données difficilement traitables de façon simple mais nous limiterons les

informations aux plus intéressantes en utilisant **xtail** (qui comme **tail** scrute les données écrites à un fichier) et **strings** (qui isole les chaînes de caractères) :

```
ifm@hp:~/vmware/rh72$ xtail *REDO* | strings -a > honeypot.log
```

Par exemple, pour un attaquant ayant extrait d'une archive **.tgz** les sources d'adore, on aura :

```
[...coupé...]
.rodata
.data
.eh_frame
.dynamic
.ctors
.dtors
.got
.bss
.comment
.note
adore.c
Makefile.~
ava.c
LICENSE
README
/** (C) 1999/2000 by Stealth - http://www.scorpions.net/~stealth
***                                     http://teso.scene.at
***
***
*** (C)'ed Under a BSDish license. Please look at LICENSE-file.
*** SO YOU USE THIS AT YOUR OWN RISK!
*** YOU ARE ONLY ALLOWED TO USE THIS IN LEGAL MANNERS.
*** !!! FOR EDUCATIONAL PURPOSES ONLY !!!
***
*** Some of the source has been taken from heroin.c (by R. Jensen) and
*** knark (by Creed @ sekure.net).
***
*** Major advantages of adore V 0.14:
***
```

[...coupé...]

```
sys_call_table[SYS_setuid] = o_setuid;
/* ditto */
unlock_kernel();
return 0;

CC=cc
CFLAGS=-O2 -Wall -I/usr/src/linux/include -DBEHAVE_STEALTH -DHIDDEN_SER-
VICE="" -D31337 -DDELITE_CMD=50000 #DMODVERSIONS
all:  adore.c ava.c
      $(CC) -c $(CFLAGS) adore.c -o adore.o
      $(CC) $(CFLAGS) -O2 -Wall ava.c -o ava
```

[...etc...]

Nous remarquons dans la première partie jusqu'à **.note** les indices qui permettent de déduire l'utilisation de **tar**. La commande `strings -a /bin/tar` nous donnerait la même chose. Ensuite, les 5 lignes de **adore.c** à **README** nous révèlent le contenu de l'archive. Nous avons enfin les fichiers les uns à la suite des autres. La fin de **adore.c** (`return 0;`) est tout de suite suivie par le début du **Makefile** (`CC=cc`). Le fichier log conserve donc une trace de chaque fichier écrit sur le disque dur, par exemple un **.c**



que le pirate voudra compiler sur l'hôte client ou encore toutes les données contenues en mémoire *swap*. Pour limiter les données à analyser par ce biais, il est très utile de supprimer les tâches des *crontab* et autres scripts lancés automatiquement sur le système. Il est parfois conseillé également de limiter la taille de mémoire vive allouée à l'hôte client sous VMware pour que la *swap* soit utilisée le plus possible et ainsi récupérer des informations supplémentaires.

Un autre moyen permettant de surveiller le honeypot est de monter le disque virtuel à l'aide du *Virtual Disk Driver* [6]. Ceci permet de récupérer les fichiers du pirate et d'avoir des données sous une forme plus facilement analysable par rapport à un fichier *REDO* brut.

Bien entendu, la surveillance du honeypot par les méthodes exposées ici n'empêche en rien la mise en place de moyens de surveillance additionnels disponibles avec les autres types de honeypots.

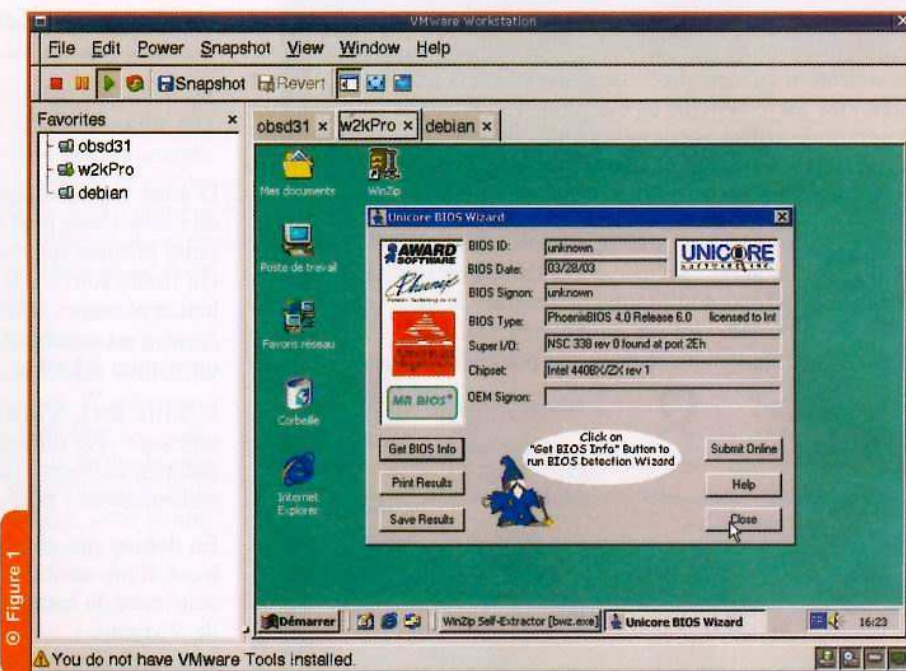


Figure 1

LE PROBLÈME DE FURTIVITÉ

Un gros problème de l'usage de VMware dans le domaine des honeypots reste son manque de discrétion. Par défaut, le pirate dispose – s'il veut bien s'en donner la peine – de nombreux moyens de détecter l'usage de VMware. Nous allons maintenant exposer certaines de ces faiblesses et donner les solutions quand elles existent.

Tout d'abord, il peut paraître suspicieux au pirate de voir que *VMware tools* est installé sur l'hôte client. *VMware tools* permet entre autre d'avoir un meilleur affichage sous Xwindow ou sous Windows. La question de son installation n'est toutefois pas si évidente. Il est en effet possible de vouloir se faire passer pour un vrai serveur de test qui utiliserait VMware. Après tout, VMware ne sert pas qu'à piéger les pirates dans un honeypot... Si on considère qu'il existe bien d'autres façons d'identifier l'utilisation de VMware, cette stratégie pourrait être valable. De plus, si on se plaçait dans ce cas, il serait possible de voir si l'attaquant n'a pas un exploit encore non publié visant VMware pour atteindre le système réel. On peut d'ailleurs noter qu'un tel exploit avait été publié pour prendre la main sur VMware version GSX il y a quelques temps. Le système sur lequel tourne le honeypot virtuel deviendrait alors lui-même un honeypot. Il pourrait alors être valable de ne mettre en place aucun mécanisme de supervision directement visible sur ce système. Les esprits les plus critiques diront sûrement que l'idée est à la fois poussée et tordue et ils auront probablement raison !

Pour résumer, soit vous décidez de cacher le fait que vous utilisez VMware et vous n'installez pas *VMware tools*, soit vous souhaitez faire passer votre honeypot pour un serveur de test utilisant VMware et dans ce cas vous pouvez l'installer, éviter de passer

du temps à cacher l'emploi de VMware et mettre l'accent sur le côté psychologique qui permettra à l'attaquant de penser qu'il s'agit effectivement d'un serveur de test (par exemple en ayant des données caractéristiques d'un serveur de test, un nom d'hôte parlant comme *serv-test-03*).

Le second point réside dans les caractéristiques des périphériques virtuels de VMware. Par exemple, si le pirate, pour connaître un peu mieux son environnement, jette un oeil à *dmesg*, il trouvera entre autre :

```
virtualix:~# dmesg | grep VMware
hda: VMware Virtual IDE Hard Drive, ATA DISK drive
hdb: VMware Virtual IDE CDROM Drive, ATAPI CDROM drive
hda: VMware Virtual IDE Hard Drive, 2048MB w/32kb Cache, CHS=520/128/63
virtualix:~#
```

Il est évident pour lui qu'il se trouve sur un système utilisant VMware. Il en sera de même s'il utilise *lspci* ou s'il regarde les données contenues dans */proc*, par exemple */proc/ide/ide0/hda/model*. La solution à ce problème est la même que concernant la mise en place d'honeypots avec UML.

Un autre problème vient des adresses MAC des cartes réseaux virtuelles, qui commencent toutes de la même façon (la première moitié de l'adresse identifie le constructeur de la carte réseau) [7]. Il est possible de modifier l'adresse MAC de l'hôte client en éditant le fichier *honeypot.vmx* et en changeant la ligne :

```
ethernet0.generatedAddress = "00:0c:29:b2:f7:ee"
```

Depuis la version 4 de *VMware Workstation*, il est toutefois impossible de modifier la première moitié de l'adresse. Il est alors nécessaire de s'en tenir aux restrictions détaillées dans le manuel [8], sous peine de voir un message d'erreur au lancement de la machine virtuelle. Il devient donc impossible de camoufler l'adresse MAC spécifique à VMware. C'est ici une limitation par rapport à UML ou à d'autres équivalents Open Source !



VMware utilise des I/O particulières sur le système physique pour communiquer entre l'hôte client et le logiciel [9]. Une méthode de détection encore plus vicieuse consiste à tester si elles sont présentes sur le système en question. Voici dans le **Tableau 1** un programme utilisé par certains pirates venant de compromettre un système qui souhaitaient savoir s'ils se trouvent sur un honeypot :

Pour en finir avec les caractéristiques matérielles, il faut aussi souligner que le type de BIOS reste caractéristique de VMware. Il existe en effet des utilitaires permettant de faire une copie du BIOS ou d'en extraire les propriétés mais il est impossible d'empêcher facilement ceci. Voyez par exemple sous Windows la **figure 1**.

Tableau 1

```
#include <stdio.h>
#include <sys/signal.h>

#if __INTSIZE == 2 /* 16 bit environment */
typedef unsigned int uint16;
typedef unsigned long uint32;
#else /* 32 bit environment */
typedef unsigned short uint16;
typedef unsigned int uint32;
#endif /* __INTSIZE */

void segfault(){
    printf("Not running inside VMware.\n");
    exit(1);
}

int main(){
    uint32 verMajor, verMinor, magic, dout;

    signal(SIGSEGV, segfault);

    __asm__ __volatile__ (
        mov $0x56405868, %%eax; /* magic number */
        mov $0x3c6cf712, %%ebx; /* random number */
        mov $0x0000000A, %%ecx; /* specifies command */
        mov $0x5658, %%edx; /* VMware I/O port */

        in %%dx, %%eax;

        mov %%eax, %0;
        mov %%ebx, %1;
        mov %%ecx, %2;
        mov %%edx, %3;
        : "=r"(verMajor), "=r"(magic), "=r"(verMinor),
        "=r"(dout)
    );

    if (magic == 0x56405868) {
        printf("Running inside VMware. ");
        printf("(Version %1u,%1u)\n", verMajor,
        verMinor);
    }

    return 0;
}
```

On retrouve en général :

BIOS ID: unknown
BIOS Signon: unknown
BIOS Type: PhoenixBIOS 4.0 Release 6.0 licensed to Intel
Chipset: Intel 440BX/ZX rev 1

D'autre part, l'adéquation entre les différents aspects matériels de l'hôte client peut mettre la puce à l'oreille de l'attaquant. Un point critique repose sur le rapport mémoire vive / processeur. On limite souvent la mémoire vive de l'hôte client utilisé pour lancer plusieurs hôtes clients depuis le même système physique. Ainsi, il est possible d'avoir un hôte client ayant comme processeur un Athlon à 2 GHz mais seulement 64 Mo de mémoire vive...

D'autre part, VMware autorise l'allocation de quantités de mémoire vive différentes de 64, 128, 256 ou 512 Mo, ce qui en réalité n'existe pas ou peu souvent. Il faut donc s'assurer que les ressources de l'hôte client sont réalistes.

En dehors des considérations précédentes à propos d'un hôte isolé, il me semble utile de donner un avis tout personnel sur la cohérence du honeynet dans son ensemble. J'ai lu dans un tutoriel de VMware [10] la proposition tentante de mettre en place 5 systèmes d'exploitation totalement différents sur le même honeynet (Linux Red Hat 8.0, Solaris 8 x86, OpenBSD 3.1, Windows 2000 et Windows XP).

Comme VMware permet de faire tourner un grand nombre de systèmes d'exploitation différents, proposer autant de cibles d'attaques semble donc alléchant. Il peut toutefois paraître douteux à l'attaquant d'arriver dans un milieu aussi hétérogène, qu'on ne trouverait sans doute pas dans un environnement de production réel.

LE CHAT ET LA SOURIS

Relativement récemment, Kostya Kortchinsky, un membre du French HoneyNet Project, a dévoilé sur la liste de discussion Honeypots de SecurityFocus [11] un patch permettant de :

- modifier l'identifiant des disques durs et CD-ROM IDE ou SCSI;
- changer les propriétés de la carte graphique émulée par VMware;
- déplacer le numéro de la porte dérobée vue plus haut.

Peu de temps ensuite, dans un faux numéro du magazine en ligne Phrack [12], plusieurs méthodes ont été mises en avant pour contourner cette obfuscation :

- d'abord, contourner le masquage des disques en utilisant hdparm;
- ensuite, rechercher le numéro de la porte dérobée de VMware en balayant toute une plage d'adresses.

Tout ceci illustre très bien que même si des techniques sont mises en avant pour masquer l'usage de VMware, les attaquants recherchent continuellement des méthodes pour les contourner. Ce petit jeu du chat et de la souris risque bien de continuer longtemps !

CONCLUSION

VMware est un outil intéressant pour la mise en place de honeypots ou de honeynets, à la condition qu'on n'oublie pas ses faiblesses. Un pirate paranoïaque pourra se rendre compte qu'il se trouve sur un système émulé sous VMware et s'il est prudent, changera bien vite de terrain de jeu. Le grand nombre de systèmes d'exploitation gérés, la facilité de mise en place, de réplication et de surveillance de honeypot sous VMware est donc à relativiser car ce talon d'Achille est de taille.

Pour ceux qui souhaitent mettre en place des honeypots offrant une proie sous Linux préféreront sûrement UML, qui offre beaucoup plus de possibilités de personnalisation. D'autre part, la surveillance du honeypot en se basant sur le fichier `.REDO` est très pratique, mais il sera certainement préférable de garder des moyens de surveillance supplémentaires directement depuis l'hôte client, avec des rootkits kernel notamment, pour garder une trace de ce qui se passe uniquement en mémoire vive.

Victor Vuillard
victor.vuillard@utbm.fr

RÉFÉRENCES

- [1] <http://www.vmware.com> - Site de la société VMware.
- [2] <http://www.vmware.com/download/>
Page de téléchargement de la version d'évaluation de VMware.
- [3] Un bridge "firewallant", la solution discrète - Loïc Minier - Linux Magazine Hors-Série n°13.
- [4] <http://www.porcupine.org/forensics/tct.html>
The Coroner's Toolkit
- [5] *Roots-kits et intégrité* - Frédéric Raynal - Linux Magazine Hors-Série n°8 / MISC 0 (nostalgie, quand tu nous tiens...)
- [6] <http://chitchat.at.infoseek.co.jp/vmware/vdk.html>
Virtual Disk Driver
- [7] http://coffer.com/mac_find/
Trouver la marque de la carte réseau grâce à l'adresse MAC
- [8] http://www.vmware.com/support/ws4/doc/network_macaddr_ws.html
Changement de MAC address dans VMware - limitations
- [9] <http://chitchat.tripod.co.jp/vmware/backdoor.html>
The VMware backdoor I/O port
- [10] <http://www.honeynet.org/papers/vmware/>
Know your enemy: Learning with VMware - The Honeynet Project
- [11] <http://cert.uni-stuttgart.de/archive/honeypots/2004/01/msg00014.html>
Counter measures to VMware fingerprinting
- [12] <http://www.phrack.org/fakes/p63/p63-0x09.txt>
Fake Phrack 63 - 5. Encore du VMware



La cryptographie quantique : à la conquête des photons



La cryptographie quantique apporte une sécurité accrue par rapport aux systèmes de cryptographie classique. Des particules de lumière sont utilisées pour transmettre de l'information. Contrairement aux cryptosystèmes de la cryptographie classique, la sécurité de la cryptographie quantique ne dépend pas de la puissance de calcul de l'ennemi. Elle offre une sécurité inviolable qui repose sur les lois de la physique quantique.

LA CRYPTOGRAPHIE CLASSIQUE

LE PROBLÈME DE L'ÉCHANGE DE LA CLÉ

La cryptographie symétrique est la plus ancienne forme de chiffrement pratiquée. On sait aussi qu'il existe un chiffrement symétrique parfaitement sûr. Il s'agit du chiffre de Vernam (le *one-time pad* ou masque jetable), qui s'appuie sur une clé aléatoire, et à usage unique, de même taille que le message. Pour produire le message chiffré, on combine par XOR le message à chiffrer et la clé. Shannon a démontré que, sans connaissance préalable de la clé, un espion ne peut obtenir aucune information sur le message chiffré. L'histoire de la cryptographie pourrait s'arrêter là si on n'avait pas omis un détail important dans les explications qui viennent d'être données : le problème de l'échange de la clé. On a admis que nos deux interlocuteurs, Alice et Bob, partagent une clé secrète. Dans la pratique, Alice et Bob doivent d'abord s'échanger cette clé de façon sécurisée. Sinon, un espion Eve, qui intercepte la clé, pourrait déchiffrer tous les messages entre Alice et Bob. A la fin des années 1970, Whitfield Diffie et Martin Hellman inventent la cryptographie à clé publique. Une des grandes avancées théoriques et pratiques due à la cryptographie à clé publique est qu'elle résout le problème de l'échange de la clé. En effet, Alice et Bob possèdent chacun une clé de chiffrement et une clé de déchiffrement, et il n'y a donc tout simplement plus besoin d'échanger de clé secrète !

LA SÉCURITÉ DE LA CRYPTOGRAPHIE À CLÉ PUBLIQUE

L'algorithme phare en cryptographie à clé publique se nomme RSA. Une grande partie de la sécurité présumée de RSA est fondée sur la difficulté de la factorisation des nombres. Si un attaquant avait la possibilité de déduire facilement les facteurs

premiers d'un nombre entier, il pourrait alors déduire la clé privée de la clé publique. La difficulté de la factorisation n'est malheureusement qu'une hypothèse. Une avancée théorique majeure dans le domaine n'est pas exclue, et il est tout à fait imaginable (même si les spécialistes tendent à dire que cela est peu probable) que le problème de la factorisation se révèle être un problème facile. On dispose d'ailleurs actuellement de meilleurs algorithmes de factorisation qu'il y a quelques années.

De plus, la factorisation est devenue beaucoup plus facile ces quinze dernières années grâce à l'augmentation de la puissance des ordinateurs et leur nombre qui a explosé (plusieurs algorithmes de factorisation sont facilement parallélisables sur des architectures distribuées). Il est donc essentiel de continuellement augmenter les tailles de clé RSA pour compenser l'augmentation de la puissance de calcul d'un éventuel ennemi. Ce gain en puissance de calcul a aussi un effet pervers supplémentaire : si un individu peut factoriser en 2019 les tailles de clé actuelles, il pourra alors déchiffrer tout le trafic généré en 2004 !

Pour finir, la future arrivée d'un ordinateur quantique (voir l'encadré sur l'informatique quantique) rendra complètement caduque l'utilisation de RSA. Grâce à la découverte d'algorithmes quantiques de factorisation rapide (algorithme de Shor, 1994), on sait que la factorisation est un problème facile sur un ordinateur quantique.

Évidemment, RSA, même s'il est le cryptosystème le plus largement utilisé, n'est pas le seul algorithme à clé publique qui existe. Cependant, ces autres cryptosystèmes souffrent tous pour l'instant des mêmes vulnérabilités. Il n'existe pas de cryptographie à clé publique sûre.

La cryptographie quantique permet quant à elle d'assurer un échange de clés parfaitement sécurisé quelle que soit la puissance de calcul de l'ennemi (même une puissance de calcul infinie !). On admet en effet, en cryptographie quantique, qu'un attaquant n'est limité que par les lois de la physique. On a aussi expliqué plus haut qu'il existait un chiffrement symétrique parfait, le chiffre



de Vernam. Ainsi la distribution quantique de clé associée avec un chiffrement de Vernam est le premier cryptosystème dont la sécurité est absolue et dont la preuve de sécurité a été apportée.

LA DISTRIBUTION QUANTIQUE DE CLÉ

Comme son nom l'indique, la cryptographie quantique exploite les principes de la physique quantique pour permettre de garantir la confidentialité de communication. Les champs d'application de la physique classique et de la physique quantique sont très différents. La physique classique décrit les objets dits macroscopiques. C'est elle qui, par exemple, permet d'analyser la chute d'une pomme. Elle fut développée progressivement au cours des deux derniers millénaires. A la fin du 19^e siècle, un certain nombre de situations, pour lesquelles la description proposée par cette théorie n'était pas adaptée, furent mises en évidence. Des physiciens comme Max Planck et Albert Einstein entreprirent donc de développer un nouvel ensemble de théories, connu sous le nom de physique quantique. Celle-ci décrit les objets du monde microscopique, comme les molécules, les atomes ou les particules élémentaires. Chacune ayant son champ d'application, physique quantique et classique sont complémentaires. Néanmoins, leurs prédictions diffèrent de façon radicale. Par exemple, alors que la physique classique est intrinsèquement déterministe, la physique quantique prévoit que certains phénomènes sont fondamentalement aléatoires (voir encadré). Cette dernière impose aussi des limitations à la précision avec laquelle la mesure d'une propriété d'un système peut être effectuée (conséquence du principe d'incertitude d'Heisenberg). La cryptographie quantique exploite ces limitations.

Le principe qui sous-tend la cryptographie quantique est relativement simple. Elle exploite le fait que, selon la physique quantique, il n'est pas possible d'observer un système sans le perturber de façon irrémédiable. Cette interaction entre l'observateur et l'objet observé est fondamentale dans le monde microscopique. Ainsi par exemple, lorsqu'un lecteur parcourt cet article, la feuille de papier sur laquelle le texte est imprimé doit être éclairée. L'impact des particules de lumière, les photons, sur la feuille aura pour effet d'augmenter, très légèrement certes, sa température. Il modifie donc la feuille de papier. Cet effet est très faible et n'est en pratique pas perceptible avec un objet macroscopique comme la feuille de papier. La situation est toutefois drastiquement différente dans le cas d'un objet microscopique. De façon similaire, si un objet décrit par la physique quantique est utilisé dans un système de télécommunication comme support permettant de transmettre de l'information, son interception se traduira forcément par des perturbations. Un espion est en effet forcé de l'observer. Cette perturbation, synonyme d'erreurs dans la transmission, révélera de façon péremptoire la présence de l'espion.

Un système de communication conventionnel peut être comparé à deux personnes jouant au tennis. L'émetteur prend une par une des balles de tennis, y inscrit l'information qu'il souhaite transmettre, et les envoie à son partenaire. Celui-ci les attrape et

QUELQUES DATES

QUELQUES DATES IMPORTANTES DANS LA CRYPTOGRAPHIE QUANTIQUE

> 1927

Principe d'incertitude d'Heisenberg. Il stipule que l'observation d'un état quantique le perturbe.

> 1970

Stephen Wiesner écrit "Conjugate Coding".

> 1982

Théorème de non-clonage quantique de Wootters et Zurek. Ils démontrent qu'il n'existe pas de machine permettant de cloner un état quantique en un autre état quantique identique.

> 1984

Bennett et Brassard décrivent BB84, le premier protocole de cryptographie quantique.

> 1991

Ekert décrit un protocole basé sur le principe EPR (intrication quantique).

> 1992

Bennett invente le protocole B92 qui ne requiert que deux états pour encoder l'information.

> 1993

Un groupe de six scientifiques (dont Bennett, Brassard et Wootters) proposent le principe de la téléportation quantique.

LA CRYPTOGRAPHIE QUANTIQUE DANS LE MONDE RÉEL

> 1989

Bennett et Brassard testent BB84 sur 32 cm.

> 1995

L'université de Genève effectue une distribution quantique de clés sous le lac Léman sur une distance de 23 km.

> 1998

L'équipe de Richard Hughes à Los Alamos effectue un échange de clés sur 48 km. Elle effectue aussi une expérience à l'air libre sur 1 km de nuit (0.5 km de jour).

> 2001

L'équipe de Los Alamos pousse leur record à l'air libre à 10 km (de jour comme de nuit) dans le Nouveau Mexique.

> 2002

La société id Quantique et l'université de Genève effectuent un échange de clés sur 67 km entre Lausanne et Genève.

> 2002

Des chercheurs de l'université de Munich et de Qinetiq en Grande-Bretagne effectuent un échange quantique de clés à l'air libre sur 23.4 km entre deux montagnes.

> 2003

Toshiba dépasse les 100 km en cryptographie quantique et sera suivi de peu par NEC.



lit l'information inscrite. Malheureusement, il est toujours possible pour une personne mal intentionnée de se placer entre les deux joueurs avec un filet à papillons pour intercepter les balles de façon à lire l'information avant de réexpédier les balles de tennis au destinataire. Si l'espion est un tant soit peu prudent, celui-ci ne se rendra compte de rien. Si, au lieu d'utiliser des balles de tennis, l'émetteur utilise des bulles de savon comme support de l'information, la situation est complètement différente. Les bulles de savon étant extrêmement fragiles, un espion qui essaierait de les intercepter les ferait éclater, l'empêchant de passer inaperçu.

En utilisant des objets quantiques pour le transport d'information, la cryptographie quantique permet de mettre en évidence l'interception de communication. Celle-ci introduit en effet des erreurs dans le message. Il est toutefois clair que si cette approche est utilisée pour échanger des messages confidentiels, il est possible de se rendre compte de leur interception, mais seulement *a posteriori*, ce qui est d'un intérêt somme toute assez limité. Il est donc plus judicieux de transmettre par ce biais des séquences de bits aléatoires, de vérifier leur intégrité de façon à pouvoir mettre en évidence une éventuelle interception, puis, une fois seulement que leur confidentialité a été établie, de les utiliser pour chiffrer des messages au moyen d'un algorithme symétrique. La cryptographie quantique permet donc, grâce aux lois de la physique quantique, de distribuer des séquences de bits aléatoires et de s'assurer de leur confidentialité, résolvant ainsi le problème de l'échange de la clé en cryptographie. C'est pourquoi le terme de "distribution quantique de clé" constitue une appellation plus appropriée pour cette technique.

Il est important de noter que, lors des analyses de sécurité de la cryptographie quantique, la seule contrainte en général imposée à un espion est qu'il suive les lois de la physique. Il a donc à sa disposition toutes les technologies, présentes et futures, respectant le cadre scientifique. Malgré l'extrême liberté accordée à l'espion, il est possible de démontrer la sécurité absolue de la cryptographie quantique. C'est du fait de cette caractéristique que cette nouvelle technique constitue une véritable révolution.

EN PRATIQUE

Bien que de nos jours l'information soit généralement traitée au moyen de systèmes électroniques (ordinateurs, téléphones), elle est le plus souvent transmise au moyen de réseaux optiques. Pour chaque bit, une impulsion lumineuse est envoyée le long d'une fibre optique de l'émetteur au récepteur. Ce dernier la détecte et la transforme en signal électronique. Ces impulsions sont typiquement constituées de millions de particules de lumière (les physiciens parlent de photons). En cryptographie quantique, on suit le même principe, mais avec des impulsions constituées d'un unique photon (correspondant à la bulle de savon). Un photon représente une quantité d'énergie minuscule. Il constitue un système quantique élémentaire, ce qui signifie qu'il n'est pas possible de le casser en deux.

Ainsi, un espion souhaitant intercepter une communication ne peut prendre un demi-photon, tout en laissant l'autre moitié poursuivre sa route. S'il veut intercepter le bit, il lui faut mesurer

l'état du photon, et donc interrompre la communication. Il est clair que dans ce cas, rien ne l'empêche de préparer un nouveau photon selon le résultat qu'il a obtenu pour l'envoyer au destinataire. Toutefois, en cryptographie quantique, Alice et Bob coopèrent pour empêcher Eve de suivre cette stratégie, en s'assurant qu'elle ne pourra le faire sans introduire des erreurs.

LE PROTOCOLE BB84

Le premier protocole de cryptographie quantique, BB84, a été présenté en 1984 par Charles Bennett d'IBM et Gilles Brassard de l'Université de Montréal. Il repose sur une idée contenue dans la thèse (début des années 1970) de Stephen Wiesner. Celle-ci est passée complètement inaperçue à l'époque. Wiesner y proposait de coder de l'information sur les états quantiques de particules pour fabriquer une monnaie infalsifiable. Un état quantique est un ensemble de caractéristiques d'un objet dans le domaine quantique (par exemple, sa position, son spin, sa polarisation...).

Dans le protocole BB84, Alice et Bob utilisent la polarisation des photons comme support pour transporter de l'information. La lumière est une onde électromagnétique. La polarisation d'un photon est une grandeur qui correspond à la direction d'oscillation de son champ électrique et qui caractérise un photon dans le monde quantique. Alice envoie à Bob des photons un par un. Sur chaque photon elle a pris soin d'encoder de l'information (un 0 ou un 1) par sa polarisation. Cela signifie que la direction de polarisation du photon va indiquer si le photon est porteur d'un 0 ou d'un 1. Ces bits d'information véhiculés sur un support quantique sont parfois appelés "qubits" (pour *quantum bits*). Si un espion tente de mesurer l'information transportée par le photon, le principe d'incertitude garantit que l'information est perturbée de façon irrémédiable. Détaillons maintenant le fonctionnement du protocole BB84.

Déroulement du protocole

1. Alice envoie à Bob des 0 et des 1 encodés sur la polarisation des photons.

Pour encoder ces 0 et ces 1, Alice utilise en fait deux bases non orthogonales : par exemple la base horizontale/verticale et la base diagonale. Cela signifie qu'Alice a deux façons d'encoder un 0, et deux façons d'encoder un 1, selon qu'elle utilise la première ou la seconde base. On parle alors d'un système à 4 états. Pour chaque photon qu'Alice envoie, elle choisit aléatoirement une des deux bases pour encoder l'information.

2. Bob reçoit chaque photon et le mesure aléatoirement dans une des deux bases d'Alice.

Bob ne connaît pas la base dans laquelle l'information a été encodée et il mesure donc chaque photon aléatoirement dans une des deux bases au moyen d'un filtre appelé séparateur de polarisation.

→ Lorsque Bob mesure un photon dans la même base qu'Alice, il apprend alors l'information correcte transportée par le photon ;



→ Lorsque Bob mesure un photon dans la mauvaise base, l'information est modifiée aléatoirement : 50% des fois sur un 0 et 50% sur un 1. (Bob a 50% de chances de connaître la bonne valeur du bit).

Après la transmission, Bob aura donc utilisé 50% du temps le bon filtre et 50% du temps le mauvais filtre. Alice et Bob partagent à ce moment une clé brute dont 75% des entrées sont semblables, soit avec 25% d'erreurs.

Le protocole BB84 ne s'arrête pas là. Sur un autre canal (classique celui-là), Bob communique à Alice les bases qu'il a utilisées tout en gardant secrets les bits qu'il a mesurés. Alice lui répond en lui signalant les mesures qu'il a effectuées dans des bases correctes. Alice et Bob, en éliminant de leur clé les entrées où les bases ne coïncident pas, obtiennent donc une clé identique ! Cette clé, appelée clé tamisée, est de la moitié de la taille de leur clé brute initiale.

Scénario de l'espion

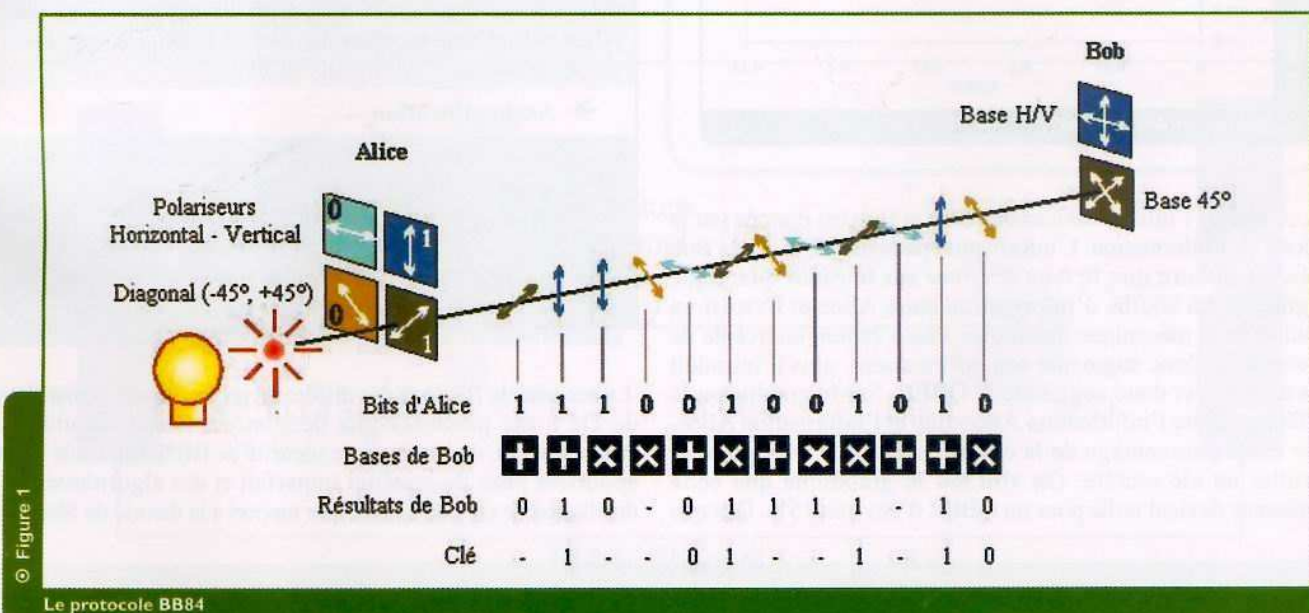
Que peut faire l'espionne Eve pour intercepter la clé ? La meilleure stratégie est d'effectuer une attaque appelée "intercept-and-resend". Eve se place entre Alice et Bob. Elle intercepte chaque photon, le mesure aléatoirement dans une des deux bases (celles utilisées par Alice et Bob) et le renvoie en direction de Bob.

Dans la moitié des cas, Eve choisit la même base qu'Alice. Elle apprend donc l'information sans la perturber et renvoie le photon dans le même état qu'Alice l'a envoyé. Dans l'autre moitié des cas, Eve choisit la mauvaise base et a 50% de chances de renvoyer le photon dans un état différent de celui d'Alice. Donc, si Eve mesure tous les photons qui transitent entre Alice et Bob, elle connaîtra la valeur de 50% des bits de la clé tamisée. Cela correspond à une information de Shannon de 0.5 bit par bit de clé. Cependant, sa présence ne passe pas inaperçue. Elle introduit

INFORMATIQUE QUANTIQUE

Bien que la physique quantique eut une forte influence sur le développement technologique au 20^e siècle (invention du laser ou du transistor), son impact sur le traitement de l'information commence seulement à apparaître. La théorie quantique de l'information constitue un champ de recherche nouveau et dynamique au carrefour de l'informatique et de la physique. Il s'intéresse aux conséquences de l'utilisation d'un système quantique comme support d'information logique (bit). Au-delà de considérations pratiques, le fait qu'un bit soit écrit sur un morceau de papier, mémorisé dans un transistor ou inscrit sur l'état d'un atome (un système quantique) a des conséquences importantes sur les opérations de traitement pouvant lui être appliqué. Avec un système quantique, les propriétés et les possibilités de traitement de l'information sont révolutionnaires et sans équivalent en théorie classique de l'information. Pour bien insister sur cette différence, on parle, dans le cadre de la théorie quantique de l'information, de bit quantique ou qubit. Le but ultime de ce nouveau domaine est le développement d'un ordinateur quantique traitant un ensemble de qubits. Les travaux de recherche ont ainsi déjà démontré de façon théorique qu'un tel système, quand il verra le jour, possèdera une puissance de calcul sans commune mesure avec celle des ordinateurs conventionnels, grâce à un parallélisme massif. Ces travaux sont particulièrement intéressants si l'on considère que la puissance des ordinateurs actuels croît grâce à la miniaturisation des transistors permettant d'en multiplier le nombre dans un microprocesseur. Dans un futur proche, cette miniaturisation se heurtera à des barrières physiques. Les spécialistes estiment ainsi que d'ici dix à quinze ans, la taille de ces transistors sera tellement petite que leur comportement sera influencé par les lois de la physique quantique.

Bien que le développement de l'ordinateur quantique constitue un objectif relativement éloigné, deux applications de la théorie quantique de l'information permettent déjà aujourd'hui d'améliorer de façon significative la sécurité des échanges d'information. Il s'agit de la génération de nombres aléatoires et de la distribution quantique de clé cryptographique.





de cette façon 25% d'erreurs sur la clé tamisée. Ce taux d'erreur sur la ligne quantique est fréquemment appelé QBER (pour *Quantum Bit Error Rate*).

Eve peut cependant appliquer cette attaque sur une fraction des photons seulement. Par exemple, si Eve mesure 10% des photons seulement, le taux d'erreur introduit sera de 2.5% pour une information obtenue de 5%. En définitive, le taux d'erreur trahit la présence de l'espion Eve aux yeux d'Alice et Bob.

Cette approche du problème correspond cependant à un système idéal. Dans un système réel de cryptographie quantique, du fait de l'imperfection du matériel, le bruit présent sur le canal quantique (le QBER) n'est pas nul, même en l'absence d'un espion. Alice et Bob ne peuvent pas distinguer les erreurs introduites par Eve et les erreurs expérimentales dues au bruit. Toutes les erreurs sont donc à attribuer à Eve.

Même si le canal quantique est bruité, Alice et Bob ont la possibilité de produire une clé secrète à partir de leur clé tamisée. La condition pour qu'Alice et Bob puissent déduire une clé secrète malgré des erreurs sur la ligne est que l'information sur la clé partagée par Alice et Bob soit plus grande que l'information sur la clé partagée par Alice et Eve. La **figure 2** nous montre l'évolution de ces deux informations (au sens d'information mutuelle de Shannon par bit de clé) en fonction du QBER.

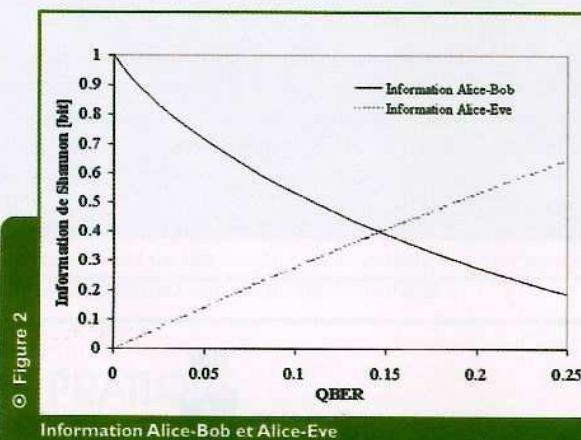


Figure 2

Information Alice-Bob et Alice-Eve

La courbe de l'information entre Alice et Bob est donnée par la théorie de l'information. L'information partagée par Alice et Bob baisse à mesure que le taux d'erreur sur le canal quantique augmente. La courbe d'information entre Alice et Eve est un résultat de la mécanique quantique. Plus l'espion intercepte de photons - et donc augmente son information - plus il introduit des erreurs - et donc augmente le QBER. Sur le graphique, la différence entre l'information Alice-Bob et l'information Alice-Eve est le pourcentage de la clé tamisée qu'ils vont pouvoir distiller en clé secrète. On voit sur le graphique que cette différence devient nulle pour un QBER d'environ 15%. Dès que

le QBER dépasse ce seuil, l'information entre Alice et Bob cesse d'être supérieure à l'information entre Alice et Eve. Il n'est alors plus possible pour Alice et Bob de déduire une clé secrète à partir de leur clé tamisée.

La procédure permettant de passer d'une clé brute à une clé secrète lorsque la ligne est bruitée (ou, et c'est équivalent, lorsqu'il y a un espion) s'appelle la distillation de clé. Il s'agit d'une suite de différents protocoles. A la fin de celle-ci, Alice et Bob ont la certitude de partager une clé parfaitement secrète qu'ils vont pouvoir utiliser pour chiffrer et déchiffrer leurs messages. Le protocole de distillation de clé commence avec le tamisage, c'est-à-dire l'échange des bases entre Alice et Bob après la transmission quantique. Il se déroule sur un second canal entre Alice et Bob.

Voici, sans entrer dans le détail, comment il se décompose (voir aussi **figure 3**) :

→ Tamisage

Alice et Bob comparent leurs bases et éliminent les bits de leur clé brute qui correspondent à des choix de bases incorrects.

→ Estimation d'Erreur

Alice et Bob mesurent l'erreur produite par le canal quantique (le QBER).

→ Correction d'Erreur

Comme Alice et Bob partagent une clé qui contient des erreurs, ils utilisent un algorithme de correction d'erreur pour les corriger. Il est essentiel que celui-ci divulgue le moins d'information possible.

→ Amplification de Confidentialité

Alice et Bob partagent maintenant une clé identique. Cependant, en attaquant le canal quantique et en écoutant le canal classique, Eve a pu obtenir de l'information sur la clé qu'ils partagent. A l'aide d'un algorithme approprié, Alice et Bob vont sacrifier des bits de leurs clés pour faire chuter le niveau d'information d'Eve.

→ Authentification

Le canal classique utilisé pour les communications de la distillation de clé est public. S'il n'a pas besoin d'être confidentiel, il doit cependant être authentique pour assurer l'intégrité des données. L'authentification des communications garantit à Alice qu'elle parle bien à Bob et à Bob qu'il parle bien à Alice.

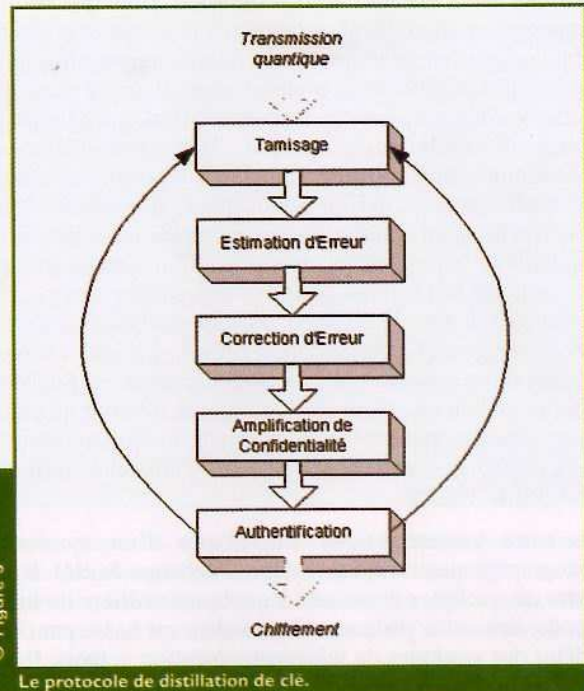
La sécurité de BB84 et des différents protocoles de la distillation de clé a été plusieurs fois démontrée. Il faut ajouter que, étonnamment, des preuves de sécurité de BB84 ont aussi pu être apportées pour du matériel imparfait et des algorithmes de la distillation de clé non idéaux (par rapport à la théorie de Shannon).

¹ Le canal quantique n'a par contre pas besoin d'être authentique. Après la correction d'erreurs, Alice et Bob prennent connaissance du taux d'erreur exact qui sépare leurs deux clés. Si Eve effectue une attaque de type Man-In-The-Middle sur le canal quantique, Alice et Bob s'aperçoivent alors qu'ils partagent une clé non corrélée.

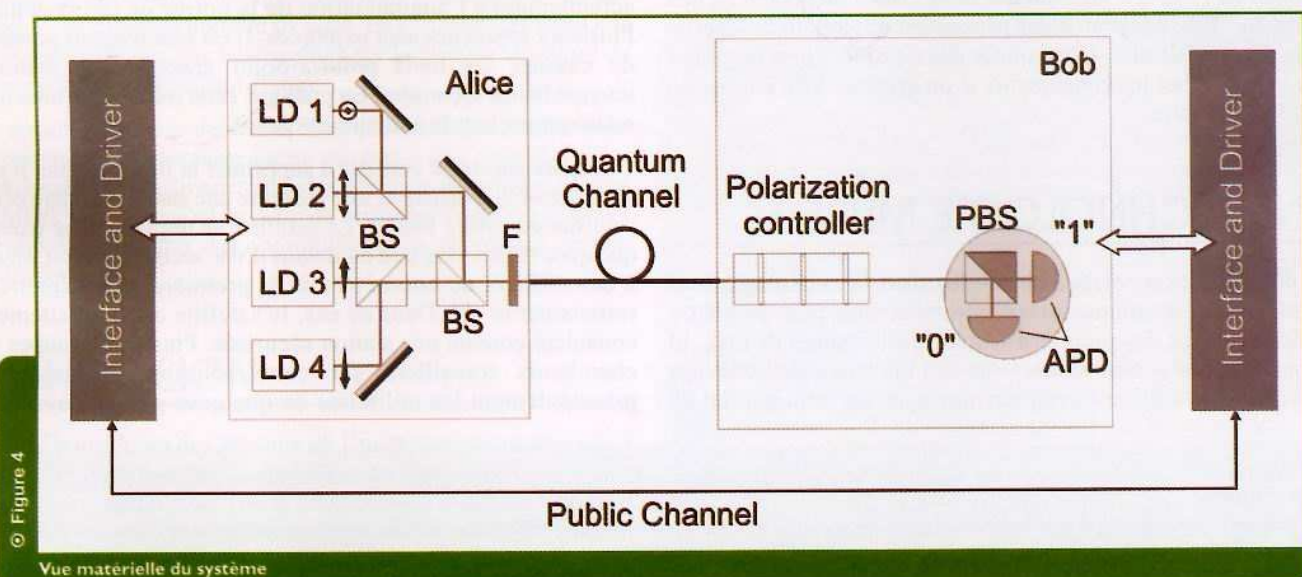


ET L'ÉQUIPEMENT ?

A quoi peut bien ressembler un système de cryptographie quantique ? L'émetteur doit pouvoir émettre des photons à la demande dans quatre états de polarisation. En dépit d'efforts importants des physiciens, les sources à photons uniques sont encore peu pratiques (elles remplissent un laboratoire entier et requièrent un fonctionnement à basse température) et peu efficaces. Les scientifiques travaillant sur les premiers systèmes de cryptographie quantique préfèrent donc utiliser des lasers conventionnels produisant des impulsions lumineuses intenses qu'ils atténuent de façon à ce qu'elles ne contiennent qu'un seul photon. L'approche la plus simple consiste à utiliser quatre lasers, un pour chaque état de polarisation (voir **figure 4**). L'émetteur est connecté à une fibre optique qui le relie au récepteur. Le récepteur consiste en un séparateur de polarisation, permettant de séparer les photons à polarisation verticale de ceux à polarisation horizontale par réflexion/transmission, suivi de deux compteurs de photons. Les compteurs de photons sont des détecteurs suffisamment sensibles pour détecter un photon à la fois. Comme les photons peuvent appartenir à deux bases différentes (polarisation horizontale/verticale ou polarisation diagonale), on rajoute devant le séparateur de polarisation un élément actif permettant de tourner sur demande la polarisation des photons incidents de 45° . Le récepteur utilise cet élément pour choisir sa base de mesure et l'active de façon aléatoire. L'émetteur et le récepteur sont en outre dotés chacun d'un ordinateur leur permettant d'enregistrer la valeur des bits et de mener à bien la procédure de distillation. Pour que le système fonctionne, il est important que l'émetteur et le récepteur soient alignés. Il est en effet important que, si l'émetteur envoie un photon polarisé verticalement par exemple, ce photon arrive dans un état identique chez le récepteur, en dépit des perturbations qu'il aura subi lors de son voyage le long de la fibre optique. C'est pour cette raison qu'il est nécessaire d'ajouter au niveau du récepteur un dispositif permettant de suivre les modifications



des états de polarisation et de réaligner les systèmes. En pratique, ce suivi est difficile à réaliser. Les chercheurs ont donc exploré d'autres voies. Ils ont ainsi proposé d'utiliser d'autres propriétés du photon comme support de l'information. La plupart des systèmes actuels de cryptographie quantique utilisent ainsi la phase des photons, plutôt que leur polarisation. Toutefois, leur fonctionnement repose sur des principes similaires à ceux discutés ci-dessus.





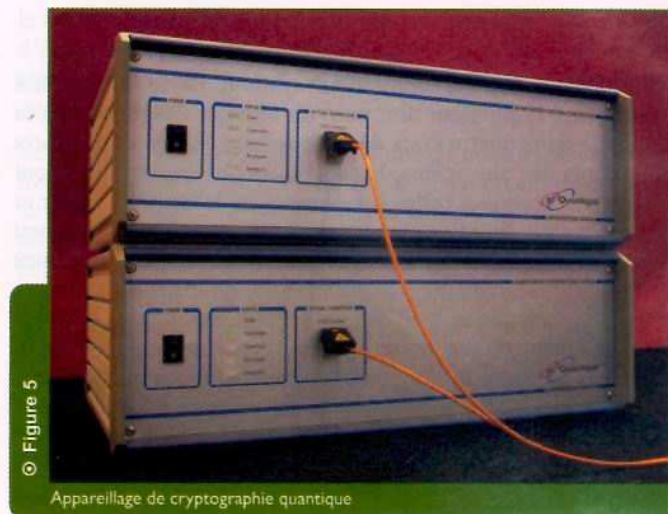
Une caractéristique importante d'un système de cryptographie quantique est la distance de transmission. Bien que les fibres optiques transportant les photons entre l'émetteur et le récepteur soient constituées de verre de très haute qualité, elles ne sont toutefois pas parfaitement transparentes. Il arrive ainsi qu'un photon soit absorbé lors de sa propagation et n'atteigne pas l'extrémité de la fibre optique. Dans les systèmes de télécommunication conventionnels, des répéteurs sont utilisés pour régénérer le signal. Ils sont espacés d'environ 80 km et amplifient le signal optique. En cryptographie quantique, il n'est pas possible d'utiliser de tels répéteurs. Tout comme un espion, ils corrompent la transmission et introduisent des erreurs². Ainsi, le débit décroît avec la distance, puisque de moins en moins de photons atteignent l'extrémité de la fibre. Les photons perdus ne sont pas pris en compte pour la constitution de la clé. Finalement, quand la distance devient trop grande, le nombre de photons transmis devient trop faible pour permettre l'établissement d'une clé. La technologie actuelle permet ainsi d'atteindre une distance de l'ordre de 100 km.

Une autre caractéristique importante d'un système de cryptographie quantique est le débit d'échange de clés. Il est de l'ordre de quelques centaines à quelques milliers de bits par seconde, suivant la distance. Cette valeur est basse par rapport au débit des systèmes de télécommunication actuels. Il s'agit toutefois du prix à payer en échange d'une sécurité absolue garantie par les lois de la physique quantique. Cette limitation n'est d'ailleurs pas aussi critique qu'il y paraît. Il faut se rappeler que le système n'est utilisé que pour échanger une clé. Les données chiffrées peuvent ensuite transiter par un canal à haut débit. Ainsi, une clé de 256 bits peut être changée plusieurs fois par seconde. Il faut insister sur le fait que, grâce à cette technique, une clé peut être générée juste avant d'être utilisée, simplifiant ainsi sa gestion et rendant son stockage superflu.

Il faut noter que BB84 n'est pas le seul protocole de cryptographie quantique. Pour n'en citer que quelques-uns, on peut mentionner le protocole dit de Ekert, exploitant l'intrication³ quantique entre deux photons d'une paire ou le protocole B92, inventé par Charles Bennett en 1992 et n'utilisant que deux états d'un photon au lieu de quatre. Tous ces protocoles permettent de garantir la sécurité d'un échange de clés. C'est plutôt des considérations pratiques qui vont amener les concepteurs d'un système à en choisir un plutôt que l'autre.

UNE TECHNOLOGIE MÛRE !

En dépit d'une connotation un peu futuriste, la technologie de la cryptographie quantique est suffisamment mûre pour permettre le déploiement des premiers systèmes d'échange de clés. id Quantique, une société suisse issue de l'université de Genève, a ainsi été la première à proposer un système commercial de



© Figure 5

Appareillage de cryptographie quantique

cryptographie quantique. Elle a été récemment rejointe par une société américaine, du nom de Magiq Technologies, basée à New York.

Ces systèmes permettent de chiffrer les données entre deux sites reliés par une fibre optique et distants de quelques dizaines de kilomètres. Le chiffrement est effectué suivant le débit désiré, soit avec un one-time pad, soit au moyen de l'algorithme AES et de clés de 256 bits rafraîchies plusieurs fois par seconde de façon automatique.

Ces systèmes intéressent tout particulièrement les organisations souhaitant garantir la confidentialité à long terme des informations qu'elles traitent. On peut par exemple penser aux communications entre un centre de services bancaires et un centre d'archivage. Les gouvernements et les administrations publiques, ainsi que les militaires, portent aussi un intérêt tout particulier à la cryptographie quantique.

Parallèlement aux premiers déploiements de cette technologie, la développement se poursuit. Les ingénieurs travaillent actuellement à l'augmentation de la portée de ces systèmes. Plusieurs approches sont explorées. Il est tout d'abord possible de chaîner les liens point-à-point grâce à des stations intermédiaires sécurisées. En pratique, cette solution est toutefois relativement lourde à mettre en œuvre.

Une autre approche consiste à supprimer la fibre optique. Il est ainsi possible d'échanger une clé entre une station terrestre et un satellite en orbite basse⁴. Le satellite se déplace et se trouve quelques heures plus tard au-dessus d'une seconde station, située à des milliers de kilomètres de la première, et à laquelle il retransmet la clé. Dans ce cas, le satellite est implicitement considéré comme une station sécurisée. Plusieurs groupes de chercheurs travaillent sur cette solution qui intéresse principalement les militaires de quelques grands pays. Des

² Le fait que ces répéteurs ne puissent pas être utilisés n'est pas lié à une quelconque imperfection technique, mais plutôt à l'impossibilité physique d'amplifier un signal quantique sans le perturber.

³ L'intrication quantique désigne un phénomène de corrélations entre deux particules. Ainsi, dans une paire de photons intriqués, lorsqu'une modification est apportée sur l'un d'eux, l'autre subit automatiquement le même changement, quelle que soit la distance qui les sépare.

⁴ L'absorption a lieu uniquement dans les premiers kilomètres. Elle peut être très faible, pourvu qu'une longueur d'onde adéquate soit sélectionnée... et qu'il fasse beau.



échanges de clés à travers un canal aérien entre deux sites distants de dizaines de kilomètres ont déjà été réalisés de façon à étudier les problèmes pratiques. Aucun échange satellitaire n'a toutefois encore eu lieu.

Des chercheurs tentent aussi de réaliser des répéteurs quantiques relayant des qubits, sans les observer et donc sans les perturber, grâce au phénomène d'intrication. Bien que ces travaux n'en soient qu'à leurs balbutiements, ils sont extrêmement prometteurs. Ces répéteurs devraient permettre d'atteindre des distances arbitrairement grandes et constitueraient des ordinateurs quantiques rudimentaires.

Finalement, une extension de portée pourrait aussi venir de progrès dans le domaine de la fabrication des fibres optiques. Depuis quelques années, des chercheurs fabriquent des fibres optiques dites à cristal photonique. Contrairement aux fibres conventionnelles, qui guident la lumière dans du verre, un matériau pouvant contenir des impuretés, ces nouvelles fibres la guident dans de l'air. Les chercheurs prédisent que ces nouvelles fibres permettront de multiplier la distance par dix. Actuellement, elles sont toutefois encore moins performantes que les fibres conventionnelles. Ces dernières ayant bénéficié de trente ans de recherche, ce n'est pas tout à fait étonnant. Il n'en reste pas moins que de nettes améliorations sont à attendre à l'horizon de cinq à dix ans.

Les ingénieurs travaillent aussi à l'augmentation du débit des systèmes de cryptographie quantique. Leur objectif est de permettre une utilisation à haut débit du chiffrement par masque jetable. Ces travaux consistent à améliorer les composants utilisés dans les systèmes, de façon à leur permettre de fonctionner à haute vitesse. Les compteurs de photons sont les principaux composants à poser problème. Des améliorations pourraient venir de l'utilisation de détecteurs à base de matériaux supraconducteurs.

Dans cet article, il a jusqu'à maintenant uniquement été question de liaison point-à-point. Bien que ce type de liaison soit adapté à certains cas, les ingénieurs travaillent aussi à la généralisation du concept de cryptographie quantique dans le contexte de réseaux impliquant un grand nombre de sites. Ils s'intéressent aussi à la constitution de réseaux hétérogènes sur la base de systèmes de cryptographie quantique différents.

En résumé, dans une décennie, il est probable que les systèmes de cryptographie quantique auront une portée de l'ordre du millier de kilomètre et fonctionneront à des débits de plusieurs centaines de Mbits/s. Ils formeront en outre de véritables réseaux quantiques permettant d'assurer des communications sécurisées vers de multiples sites.

CONCLUSION

David Deutsch, un des artisans de l'informatique quantique, a utilisé ces mots pour parler de la cryptographie quantique : "Le modèle théorique d'Alan Turing est la base de tous les ordinateurs. Maintenant, pour la première fois, ses capacités ont été dépassées." La cryptographie quantique possède cependant aussi ses

GÉNÉRATION DE NOMBRES ALÉATOIRES

La génération de nombres aléatoires est une primitive essentielle dans le domaine de la cryptographie. On peut par exemple songer à la génération d'une clé de cryptage, qui consiste en une séquence de bits aléatoires. La production d'une telle séquence n'est pas triviale. Un ordinateur conventionnel ne peut produire que des nombres pseudo-aléatoires, puisqu'il est régi par la physique classique et qu'il est, à ce titre, déterministe. Les nombres pseudo-aléatoires n'étant pas adéquats pour les besoins de la cryptographie, une source physique de hasard doit être utilisée. Comme la physique quantique est la seule théorie qui prévoit des phénomènes aléatoires, il est naturel de l'exploiter dans ce but. On peut par exemple créer un générateur basé sur le choix que fait un photon sur un miroir semi-réfléchissant. Rappelons qu'un photon est une particule élémentaire insécable et que, selon les lois de la physique quantique, le fait qu'il soit réfléchi ou transmis est fondamentalement aléatoire. Une valeur de bit est associée à chaque cas. Une séquence est générée en répétant ce processus.

détructeurs parmi les chercheurs en cryptographie. Sans remettre en doute les prouesses de la cryptographie quantique, ceux-ci argumentent généralement que la sécurité des algorithmes actuels est suffisante et qu'une sécurité plus grande n'est tout simplement pas nécessaire.

Aujourd'hui, les applications de la cryptographie quantique sont bien particulières. Elles sont réservées aux domaines où la sécurité est tout simplement essentielle. Dans le futur, il y a fort à parier que cette technologie se généralisera. Comme le disait Isaac Asimov, "toute technologie est d'abord rejetée puis acceptée". Il se peut même qu'un jour, suite à des progrès théoriques en mathématiques ou au développement du premier ordinateur quantique, la cryptographie quantique constitue l'unique solution de garantir la confidentialité des communications.

Olivier Gay - olivier.gay@epfl.ch

Grégoire Ribordy - gregoire.ribordy@idquantique.com

RÉFÉRENCES

- [1] Gisin, N., G. Ribordy, W. Tittel and H. Zbinden, 2002, "Quantum cryptography," *Reviews of Modern Physics*, Vol 74, 145, <http://www.gapoptic.unige.ch/Publications/Pdf/QC.pdf>.
- [2] Singh S., *The Code Book: The Secret History of Codes and Code-breaking*.
- [3] Bennett, C.H., and G. Brassard, 1984, in *Proceedings of the IEEE International Conference on Computer, Systems and Signal processing*, Bangalore, India, pp.175-179.
- [4] Bennett, C.H., 1992, "Quantum cryptography using any two nonorthogonal states," *Physical Review Letter*, vol. 68, 3121-3124.
- [5] Bennett, C.H., F. Bessette, G. Brassard, L. Salvail and J. Smolin, 1992, "Experimental quantum cryptography," *J. Cryptology*, 5, 3-28.
- [6] Ekert, A. K., 1991, "Quantum cryptography based on Bell's theorem," *Physical Review Letters*, vol.67, pp.661 - 663.