Les attaques par corruption de mémoire

Synopsis

Ce document traite des failles permettant des attaques par corruption de mémoire, les attaques par *buffer overflow* (dépassement de buffer) étant les plus connues.

Deux grandes catégories de personnes sont concernées :

- Les développeurs et testeurs qui, dans le cadre du processus de développement d'un logiciel, mettent en œuvre des techniques permettant de détecter des failles dans le code source, failles pouvant être utilisées dans le cadre d'attaques par corruption de mémoire une fois le logiciel en production.
- Les administrateurs et auditeurs réseau qui, dans le cadre d'un audit ou contrôle des ressources du réseau de l'entreprise, mettent en œuvre des techniques permettant de détecter les machines et logiciels vulnérables à des attaques par corruption de mémoire.

Ce document décrit dans un premier temps ce que sont les attaques par corruption de mémoire. Les sous-parties 1.2 et 1.5 sont techniques et permettent d'approfondir la compréhension du problème. Le document décrit ensuite, dans un second temps, l'audit de code pour les développeurs et testeurs et, dans un troisième temps, l'audit de vulnérabilités pour les administrateurs et auditeurs réseau.

1. Introduction aux attaques par corruption de mémoire

1.1. Définitions, vocabulaire

1.1.1. Personnes impliquées

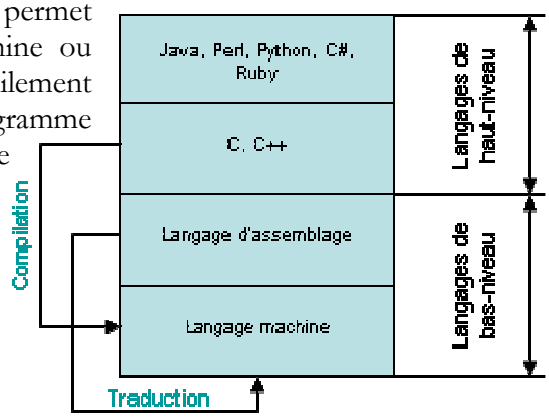
Pirate informatique ou *hacker*	Un hacker désigne un attaquant, c'est-à-dire une personne accédant à un système (distant ou local) sans autorisation, par acte de piratage informatique. Ses motivations peuvent être variées : destruction, curiosité, vol, satisfaction personnelle, etc. Elles ne sont pas toutes nuisibles mais ont toutes pour point commun d'être illégales.
Auditeur	Un auditeur est un professionnel de la sécurité informatique employant les mêmes méthodes que les attaquants pour détecter les vulnérabilités d'un programme ou des ressources d'un réseau. Il se concentre sur la détection de vulnérabilités dans le but d'établir un rapport d'audit destiné à empêcher les hackers d'exploiter ces vulnérabilités. Les auditeurs sont parfois qualifiés de *white-hat hackers* (chapeaux blancs).

1.1.2. Eléments techniques

Programme	Un programme est une suite d'instructions à exécuter par le processeur. Il est obtenu à partir d'un code source (cf. *Langages de programmation*). Le système d'exploitation alloue à chaque programme en cours d'exécution une portion de la mémoire.
Logiciel	Un logiciel est un ensemble de programmes regroupés en un tout cohérent pour atteindre un but précis. Par exemple, le logiciel antivirus Avast Home Edition est composé de plusieurs programmes : programme principal, programme de mise à jour du produit, programme de scan d'emails, programme de rapport de bug, …
Mémoire tampon ou *buffer*	Un programme a besoin de stocker des données temporairement afin de les manipuler par la suite. Ces données sont stockées dans ce que l'on appelle des buffers. En langage C, un buffer fixe destiné à contenir une chaîne de caractères peut se déclarer comme ceci : `char buffer[LONGUEUR]` où `LONGUEUR` est la taille du buffer. Il est également possible de stocker des données dans des variables. Les variables et les buffers sont stockés dans la mémoire allouée au programme par le système d'exploitation.
Corruption de mémoire	La corruption de mémoire consiste à écrire dans une zone mémoire qui ne doit pas être modifiée explicitement. Ceci est la plupart du temps dû à des erreurs de programmation. Le programme peut alors « planter » ou mal se comporter. Si un attaquant utilise cette situation à son avantage, alors il peut exécuter un code pirate et prendre le contrôle total d'un système dont l'accès ne lui est normalement pas autorisé. Les attaques par corruption de mémoire sont principalement constituées de deux sous-ensembles : • Les attaques par dépassement de buffer (*buffer overflow attacks*) ; • Les attaques sur chaînes de formatage (*format string attacks*).
Dépassement de buffer ou *buffer overflow*	Un dépassement de buffer (*buffer overflow*) se produit quand un programme écrit des données au-delà de la limite d'un buffer : cela corrompt la mémoire située au-delà du buffer. Utiliser cette situation pour exécuter un code pirate s'appelle une attaque par dépassement de buffer. Ce type d'attaque est essentiellement composé de deux sous-types : • Attaques par dépassement de pile (*stack overflow attacks*) ; • Attaques par dépassement de tas (*heap overflow attacks*). La pile est la mémoire fixe d'un programme ou d'une fonction tandis que le tas est la mémoire dynamique (par utilisation des fonctions `malloc/free` en C). Il est possible d'attaquer d'autres sections de la mémoire (section `BSS` et section `data`) mais ceci est plus rare (cf. Organisation de la mémoire).
Chaîne de formatage ou	Les chaînes de formatage du langage C (de la famille de `printf`) sont vulnérables à des attaques par corruption de mémoire lorsque les fonctions de la famille de `printf` sont

format string	mal utilisées. Une chaîne de formatage permet de spécifier le format de la sortie. Par exemple dans l'instruction `printf("Nombre : %d", nb)` la chaîne de formatage est `"Nombre : %d"`. Cette instruction affiche le nombre se trouvant dans la variable `nb`.
Shellcode ou code pirate	Un shellcode est un code pirate permettant à un attaquant d'exécuter des actions diverses et agressives : création de compte administrateur avec mot de passe connu, ouverture d'un *shell* (invite de commande) administrateur à distance, etc. Un shellcode est écrit en langage machine. Le mot shellcode est lié au fait que les premiers code pirates consistaient à ouvrir une invite de commande (*shell*) sur le système compromis.
Ver ou *worm*	Un ver (*worm*, à ne pas confondre avec un virus) est un programme qui a la capacité de se reproduire automatiquement et de se propager sur les réseaux. Nombre d'entre eux exploitent des attaques par corruption de mémoire sur des serveurs ou des systèmes d'exploitation vulnérables pour infecter les machines d'un réseau.
Programme SUID	Dans le monde Unix, il est possible d'exécuter un programme avec des privilèges différents du compte qui lance le programme. Par exemple, un utilisateur peut lancer un programme avec des droits administrateur (`root`) si l'administrateur a positionné le bit SUID sur l'exécutable du programme. Si un attaquant compromet un programme SUID `root` alors il peut avoir un accès `root` (administrateur) au système compromis, même si le programme s'exécute sur un compte avec des privilèges moindres. Le programme `ping` est un exemple de programme SUID `root`.

1.1.3. Langages de programmation

Langage haut-niveau	Un langage de programmation haut-niveau est un langage qui apporte une syntaxe confortable pour le programmeur et qui permet au programme d'être porté d'un système à un autre via recompilation ou réinterprétation. Nous avons deux types de langages : • Les langages de type Java, C# ou Python sont interprétés par une machine virtuelle, en passant éventuellement par un code managé ou *byte-code*. Beaucoup de ces langages sont équipés de ramasse-miettes (*garbage collectors*) pour la gestion automatique de la mémoire. Ces solutions sont donc sécurisées puisqu'elles enlèvent les risques de corruption mémoire par accès direct. • Les langages de type C ou C++, compilés, permettent la gestion manuelle de la mémoire. Ces langages sont moins sécurisés que leurs confrères puisqu'une erreur de programmation peut se traduire par une vulnérabilité permettant à un attaquant de corrompre la mémoire et d'utiliser la situation à son avantage.
Langage bas-niveau	Un langage bas-niveau est un langage qui permet d'écrire directement des instructions machine ou des instructions qui peuvent être facilement traduites en instructions machine. Un programme dans ce type de langage n'est pas portable d'un système à un autre. L'exemple typique est le langage d'assemblage (cf. définition du *Langage d'assemblage*). 
Langage d'assemblage	Le langage d'assemblage est un langage bas-niveau. Il représente en réalité le langage machine directement compréhensible par le microprocesseur à la différence près qu'il utilise des mnémoniques. Ces mnémoniques sont simplement des noms lisibles par un humain pour désigner les instructions microprocesseur. Par exemple `jmp` (lisible) en langage d'assemblage correspond au code machine `0xeb` (illisible) C'est le rôle de l'assembleur de traduire le programme du langage d'assemblage en code machine. Le désassembleur effectue l'opération inverse en traduisant du code machine en langage

	d'assemblage. Les shellcodes sont écrits en code machine.

1.1.4. Audit de sécurité

Ressource réseau	Nous entendons par ressource réseau n'importe quelle composante d'un réseau : serveur, poste client, imprimante, élément de connexion, etc. Toutes les ressources exécutent des programmes qui peuvent potentiellement comporter des vulnérabilités.
Vulnérabilité	Une vulnérabilité est une faille issue d'une erreur de programmation, d'une erreur de conception ou d'un problème de configuration et pouvant être utilisée par un attaquant pour compromettre une ressource d'un réseau.
Exploitation	L'exploitation consiste à utiliser une vulnérabilité dans un programme dans le but d'y injecter un code pirate afin de prendre le contrôle de la ressource l'exécutant.
Audit de sécurité	L'audit de sécurité est un acte de contrôle (vérification, analyse) d'une ressource réseau ou du code source d'un programme dans le but d'établir un rapport d'audit exposant les vulnérabilités trouvées sur la ressource ou dans le programme audité.

1.1.5. Audit de code

Audit de code source	L'audit de code est un acte de contrôle du code source d'un programme dans le but d'y trouver des vulnérabilités. Il est exécuté par un développeur, par un testeur ou par un auditeur de code source.
Analyse statique	L'analyse statique est l'analyse du code source d'un programme avant compilation et exécution. L'analyse statique permet de repérer des erreurs de programmation et est réalisée soit visuellement (relecture de code), soit par un outil d'analyse statique. L'analyse statique est l'étape centrale d'un audit de code. Elle est à opposer à l'analyse dynamique qui a lieu à l'exécution du programme.

1.1.6. Audit de vulnérabilités

Audit de vulnérabilités	L'audit de vulnérabilités est un acte de contrôle des ressources du réseau de l'entreprise (local ou non) dans le but d'identifier des vulnérabilités sur des ressources et pouvant être exploitées par un attaquant.
Scanner de vulnérabilités	Un scanner de vulnérabilités est un logiciel d'analyse des ressources d'un réseau dans le but d'identifier des vulnérabilités sur les ressources analysées et pouvant être exploitées par un attaquant. Les vulnérabilités identifiées proviennent d'une base de connaissance : elles sont donc connues et répertoriées, comme par exemple une vulnérabilité sur un serveur HTTP Apache ou Microsoft IIS.

1.2. Eléments techniques sur la mémoire

Cette sous-section permet de mieux comprendre l'origine des attaques par corruption de mémoire et notamment l'organisation de la mémoire lors de l'exécution d'un programme. Elle requiert une connaissance de base du langage C. Sa lecture est nécessaire pour la compréhension des différents types d'attaques par corruption de mémoire de la sous-section Les différents types d'attaques par corruption de mémoire.

1.2.1. Organisation de la mémoire

Le système d'exploitation alloue à chaque programme un espace virtuel découpé en 5 sections lors de son exécution:

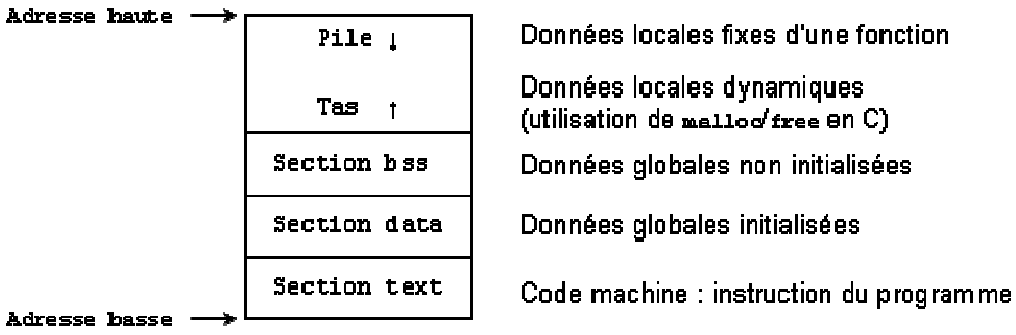


Figure 1 - Organisation de la mémoire d'un programme

- La pile est basée sur une structure de données de type pile. Pour rappel une pile est composée de deux opérations élémentaires :
  - Empiler : `push`,
  - Dépiler : `pop`.
- Le tas correspond à la mémoire utilisée par les fonctions ou opérateurs d'allocation dynamique :
  - `malloc` et `free` en C
  - `new` et `delete` en C++.

Chaque octet de la mémoire est repéré par une adresse. Pour un système 32 bits, il est possible d'avoir 2^{32} adresses (allant de `0x00000000` à `0xffffffff` en hexadécimal, soit de 0 à 4 294 967 295 en décimal) repérant chacune un octet ce qui donne 4 Go adressable[1].

La plupart du temps, on représente la mémoire par une *tour* composée de *cases* de 4 octets (32 bits) pour un système 32 bits, 4 octets étant la taille d'un entier long mais aussi la taille des adresses. Les instructions `push` et `pop` d'un processeur 32 bits travaillent donc, par défaut, sur des valeurs de 4 octets.

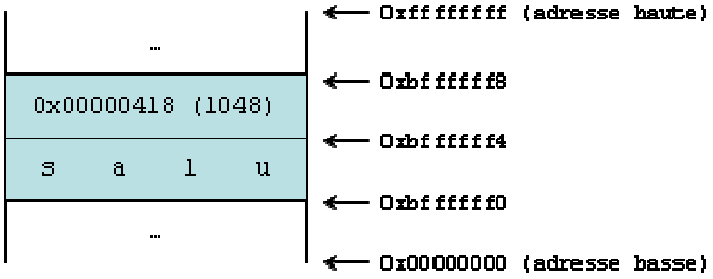


Figure 2 - Adressage de la mémoire

Le tas croît vers les adresses hautes alors que la pile croît vers les adresses basses. Par exemple :

- Soit l'adresse du bas de la pile `@pile = 0xbffffff8`,
- Le micro-processeur exécute l'instruction `push 1048`,
- Ceci place le nombre 1048 au bas de la pile et décrémente `@pile` de 4 octets,
- On obtient `@pile = 0xbffffff4` (`= 0xbffffff8 - 4 octets`).

En réalité, le micro-processeur tient à jour une mémoire qui contient l'adresse du bas de la pile. Cette mémoire est le registre `esp` (cf. section suivante).

On peut d'ores et déjà constater que les attaques par dépassement de pile ont lieu dans la pile et les attaques par dépassement de tas on lieu dans le tas. Les attaques dans les autres sections sont plus rares.

[1] Sur un système 64 bits, 2^{64} bits sont adressables soit des milliards et des milliards de milliards d'octets.

1.2.2. Appel de fonction

En langage C, un programme est constitué de fonctions, la fonction principale étant `main`. Lors de l'exécution d'une fonction, le processeur tient à jour un registre, c'est-à-dire une mémoire interne, contenant l'adresse de l'instruction suivante à exécuter : c'est le registre `eip` (*Instruction Pointer*). Pour rappel, les instructions sont stockées dans la section `text` allouée au programme.

Le processeur tient à jour deux autres registres importants :

- `esp` (*Stack Pointer*) qui contient l'adresse du dernier élément (bas) de la pile ;
- `ebp` (*Base Pointer*) qui est le pointeur de référence pour la fenêtre de la fonction en cours.

La fenêtre d'une fonction est l'espace mémoire qu'elle occupe pour réaliser ses actions.

Soit la fonction C suivante :

```
1   void fonction(int i) {
2      int a = 5;
3   }
```

Lors de l'appel d'une fonction, une nouvelle fenêtre est créée avec ses variables locales. Avant d'exécuter la ligne 2, la pile se trouve dans cet état :

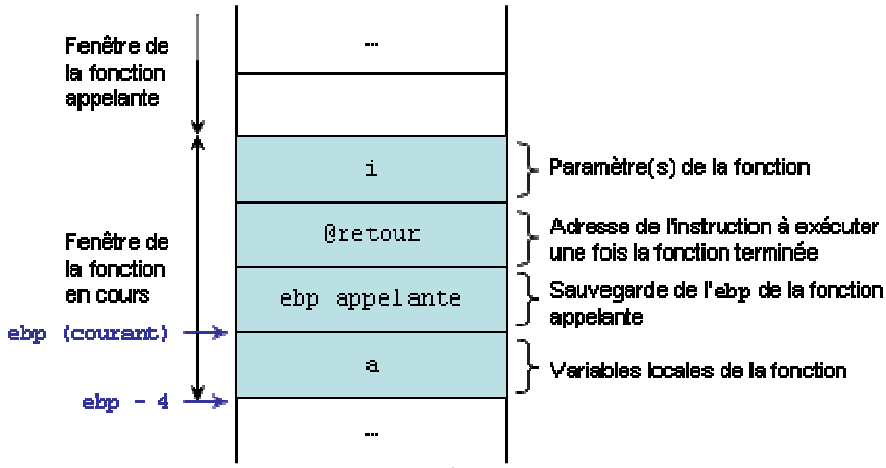


Figure 3 - Pile lors d'un appel de fonction

Pour rappel, la pile croît vers le bas. Sont empilés :

- Les paramètres/arguments de la fonction à exécuter (entier `i` ici),
- L'adresse de l'instruction à exécuter au retour de la fonction,
- La valeur du registre `ebp` (*Base Pointer*) dans la fonction appelante afin de restaurer cette valeur au retour de la fonction.

Sont également réservés :

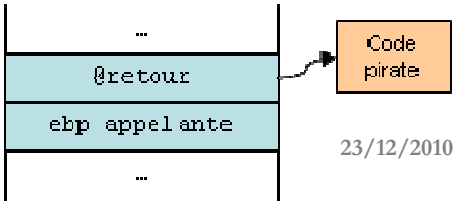
- Espace pour les variables local (entier a ici, soit 4 octets).

Le registre `ebp` contient l'adresse de base (*Base Pointer*) de la fenêtre de la fonction en cours, cette adresse étant toujours celle juste en dessous de la sauvegarde du registre `ebp` de la fonction appelante. Les variables locales sont repérées par rapport à cette adresse : par exemple, a est à l'adresse `ebp - 4`.

1.2.3. Attaque par corruption de mémoire

Une attaque par corruption de mémoire peut principalement prendre deux formes :

- Corrompre (écraser) l'adresse de retour d'une fonction pour la faire pointer sur un code pirate (qui peut être stocké



ailleurs dans la mémoire, comme dans un buffer). Ce code pirate peut prendre diverses formes, comme l'ouverture un *remote shell* (invite de commande à distance).

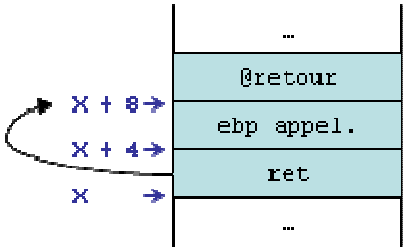
- Corrompre (écraser) le contenu d'une variable que l'utilisateur n'est pas censé modifier explicitement.

1.2.4. Exemple de corruption de mémoire

Soit le programme C suivant :

```
1   char shellcode[] =
2      "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
3      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
4      "\x80\xe8\xdc\xff\xff\xff/bin/sh"; /* Aleph One Shellcode */
5   void main() {
6      int *ret;
7      ret = (int *) &ret + 2;
8      (*ret) = (int) shellcode;
9   }
```

La fonction `main` étant une fonction comme une autre, l'explication de la sous-section Appel de fonction permet de comprendre ce qui se passe dans ce programme :



- Ligne 6 : le pointeur `ret` est à l'adresse `X` et pointe sur une zone totalement quelconque (non initialisé).
- Ligne 7 : `ret` pointe désormais sur l'adresse `X + 8` car on a ajouté 2 (mots de 4 octets) à `X`.
- Ligne 8 : modifier le contenu de `ret` revient à modifier `@retour` car `ret` pointe sur l'adresse `X + 8`. `@retour` est donc écrasé avec l'adresse du shellcode.

Par conséquent, au retour de la fonction `main` le micro-processeur exécute le code `\xeb\x1f` …. Ce code machine, que nous ne détaillons pas, ouvre un *shell* (invite de commande) sous Linux.

1.3. Histoire des attaques par corruption de mémoire

1.3.1. Origines de ce type d'attaque

C'est en 1988 qu'est apparue l'une des premières attaques par corruption de mémoire : le ver Morris utilisait un dépassement de buffer dans le programme Unix `fingerd` pour se propager de manière autonome à travers l'Internet causant ainsi la paralysie de 6000 ordinateurs du réseau.

Plus tard, en 1995, T. Lopatic redécouvrait les dépassements de buffers et divulgua ses trouvailles sur la liste de diffusion Bugtraq. Mais ce n'est qu'un an plus tard, en 1996, qu'Aleph One publia l'article *Smashing the Stack for Fun and Profit* dans le magazine électronique Phrack[1] mettant ainsi au grand jour le fonctionnement des attaques par dépassement de pile (*buffer overflow*). Les attaques par dépassement de pile sont les attaques par corruption de mémoire les plus connues et également celles qui sont apparues en premier : on les appelle les attaques de première génération.

Entre 1996 et 1998 sont apparues les attaques de seconde génération qui sont composées essentiellement des dépassements de tas (*heap overflow*) et des attaques par dépassement de pile plus poussées : écrasement de pointeur, dépassement de buffer d'un seul octet (*off-by-one*). Une faille connue exploitant un dépassement de tas est la vulnérabilité Microsoft JPEG GDI+ découverte et corrigée en 2004.

Le ver Code Red, en 2001, s'est attaqué à Microsoft Internet Information Services 5.0 (IIS), le serveur HTTP de Microsoft. En envoyant des requêtes spéciales à IIS, il était capable de déclencher un dépassement de buffer menant le ver à des droits administrateur sur le serveur compromis. Une fois ces

[1] Smashing The Stack For Fun And Profit – Elias Levy alias Aleph One (1996) : http://www.phrack.com, issue 49.

droits acquis, il remplaçait le site Web hébergé par un autre message et était capable de se diffuser à d'autres serveurs, infectant ainsi 359 000 machines à travers le monde.

Le ver SQL Slammer, en 2003, a exploité une faille de type dépassement de buffer dans Microsoft SQL Server 2000 (MS-SQL), pour infecter pas moins de 75 000 serveurs en 10 minutes à travers le monde. La particularité de ce ver est que, lors de sa diffusion, un patch corrigeant la faille existait depuis 6 mois déjà : cela montre à quel point il est important d'avoir des systèmes et des serveurs à jour, point sur lequel nous reviendrons dans les sections suivantes.

1.3.2. Les attaques par corruption de mémoire aujourd'hui

Durant la période 2002 – 2005, plusieurs vers utilisant des attaques par corruption de mémoire se sont propagés sur tout l'Internet : Blaster, Sasser, Zotob en sont les principaux exemples. Depuis 2005, le nombre de vulnérabilités permettant à des vers de se propager massivement sur Internet tend à diminuer, d'après une étude statistique du SANS Institute[1]. Cependant, il ne faut pas négliger pour autant les attaques par corruption de mémoire qui existent bel et bien.

SecurityFocus est un site répertoriant des vulnérabilités logicielles. La capture d'écran suivante, prise le 24/09/2008, prouve que les attaques par corruption de mémoire sont toujours d'actualité :

Figure 4 - Vulnérabilités logicielles récentes sur le site SecurityFocus (24/09/2008)

1.3.3. Profil des attaquants

Les attaquants exploitant les attaques par corruption de mémoire ont des motivations variées.

[1] SANS Top-20 2007 Security Risks : http://www.sans.org/top20.

L'employé d'une entreprise peut par exemple utiliser une attaque de ce type pour prendre le contrôle à distance de la machine du collègue du bureau d'à-côté à des fins d'espionnage, pourvu que la machine de ce collègue présente une vulnérabilité non corrigée. Notons que dans cet exemple l'attaque se produit sur le réseau local d'une entreprise.

Des hackers peuvent propager des vers à travers tout l'Internet créant ainsi un réseau de zombies (appelé botnet ou réseau de bots) prêts à recevoir les ordres d'un ou plusieurs ordinateurs maîtres. Un exemple typique est la propagation d'un ver grâce à une vulnérabilité de type dépassement de pile. Lorsque cette vulnérabilité est exploitée par le ver, la machine infectée exécute un shellcode qui se connecte à un serveur IRC et attend ses ordres sur un *channel* IRC, comme le montre la figure suivante :

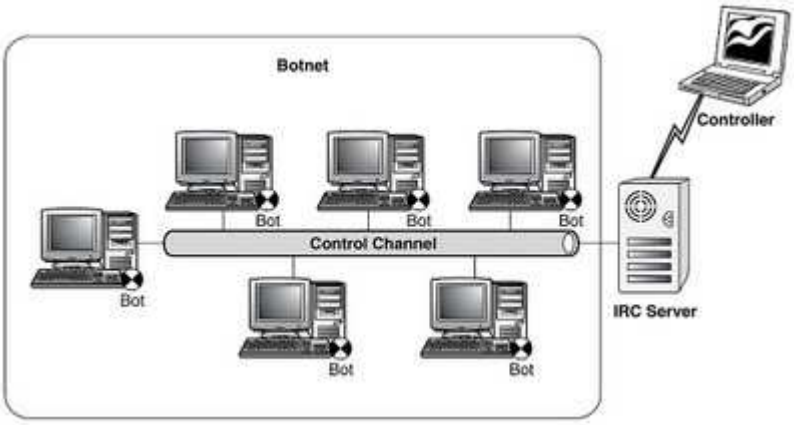


Figure 5 - Schéma d'un réseau de zombies (bots) avec connexion des zombies à un serveur IRC

Ces réseaux de zombies peuvent ensuite être revendus à des organisations, le plus souvent criminelles. Celles-ci peuvent alors orchestrer des attaques par déni de service en demandant par exemple à tous les zombies d'envoyer une requête sur un serveur précis à un instant t. Elles peuvent également utiliser les bots pour diffuser du spam plus facilement sans être détectées.

1.3.4. Personnes et ressources visées

Comme nous l'avons montré, l'exploitation d'une attaque par corruption de mémoire peut entraîner espionnage ou utilisation de machines compromises à des fins criminelles.

Par conséquent, toutes les ressources du réseau d'une entreprise, que celles-ci soient sur le réseau local ou sur l'Internet sont menacées par les attaques par corruption de mémoire.

Il est important pour l'auditeur de comprendre et repérer les vulnérabilités permettant ces attaques afin qu'elles soient ensuite corrigées.

1.4. Mise en œuvre d'une attaque par corruption de mémoire

1.4.1. Analyse et détection de failles

Tout d'abord il faut détecter les failles potentielles d'un logiciel. Par logiciel, nous entendons système d'exploitation, serveur (HTTP, FTP, SMTP, DNS, …), etc.

Il existe deux manières de détecter les failles.

Méthode	Acteur	But
Analyse ou audit de code	Développeur, testeur, responsable qualité du	L'audit de code consiste détecter les erreurs de programmation pouvant mener à des trous de sécurités dans les logiciels

	logiciel, attaquant	produits par l'entreprise. Le but est de corriger ces erreurs pour fournir aux clients de l'entreprise des logiciels finaux de qualité, fiables, sécurisés et exempts de failles.
Analyse ou audit de vulnérabilités	Administrateur, auditeur réseau, attaquant	L'audit de vulnérabilités consiste à analyser les ressources du réseau (local ou non) à la recherche de vulnérabilités, répertoriées, exploitables par un attaquant. Le but est de corriger ces failles afin que le réseau et les ressources de l'entreprise soient sécurisés au maximum.

L'audit de code permet de détecter des failles inconnues sur les logiciels produits alors que l'audit de vulnérabilités permet de détecter des failles connues et répertoriées sur les logiciels utilisés.

Les hackers peuvent utiliser les deux méthodes selon le but qu'ils cherchent à atteindre. Un hacker voulant prouver ses capacités intellectuelles et techniques voudra mettre au grand jour une vulnérabilité inconnue jusqu'à présent. Un hacker souhaitant détruire quelque chose le plus vite possible tentera plutôt d'utiliser une faille répertoriée sur un logiciel vulnérable de l'entreprise.

1.4.2. Exploitation

Un auditeur se contente de détecter les failles et d'écrire un rapport d'audit destiné à des personnes supposées corriger les failles détectées. Il peut éventuellement, dans le cadre d'un *Proof of Concept*, démontrer qu'une faille est exploitable, c'est-à-dire utilisable. Le hacker, quant à lui, cherche à utiliser la faille à ses propres fins, c'est-à-dire l'exécution d'un code pirate : c'est l'exploitation.

L'exploitation d'un programme se déroule en trois étapes :

- Corruption d'une portion de mémoire : le plus souvent, ceci consiste à envoyer une requête spéciale à un programme vulnérable, programme identifié lors de l'étape précédente d'analyse ;
- Redirection de l'exécution : le but de la corruption de mémoire est de rediriger l'exécution du programme compromis vers un shellcode (code pirate) ;
- Exécution du shellcode (code pirate) : le shellcode peut permettre diverses choses agressives (cf. définition de *shellcode*) notamment l'accès non autorisé et total à une machine distante.

Le mini-programme servant à exploiter un programme vulnérable s'appelle un exploit. Celui-ci peut-être écrit dans un langage de script de type Perl mais aussi en C. Toutes les valeurs nécessaires à son élaboration (adresse de retour, position du shellcode en mémoire, …) sont déterminées soit par tâtonnement soit avec l'aide d'un débogueur : `gdb` est un exemple de débogueur Linux.

L'élaboration d'exploits est parfois considérée comme un art. Le livre *The Art of Exploitation* explique en détail comment élaborer ses propres exploits[1].

L'utilisation d'exploits tout prêts n'est toutefois pas considérée comme un art puisqu'il s'agit simplement d'utiliser un programme trouvé sur l'Internet ou dans le Metasploit Framework, logiciel d'exploitation détaillé dans la sous-section Présentation du Metasploit Framework..

1.4.3. Installation de rootkits ou de backdoors

Une fois l'accès à une ressource non autorisée gagné, l'attaquant peut installer un rootkit ou une backdoor pour revenir ultérieurement et facilement sur cette ressource. Ceci sort du cadre de l'audit et un auditeur n'effectue que très rarement cette troisième étape.

[1] Hacking: The Art of Exploitation, 2nd Edition – John Erickson (2008).

1.5. Les différents types d'attaques par corruption de mémoire

Cette sous-section présente différentes attaques par corruption de mémoire possibles. De nouvelles attaques étant régulièrement inventées, il se peut que cette liste ne soit pas totalement exhaustive. Elle représente tout de même un très bon aperçu des différentes possibilités.

Tous les exemples sont basés sur un système Linux et sur la suite de compilation GNU (`gcc`, `gdb`). Une connaissance des bases du langage C est requise pour comprendre la présentation des attaques, et la lecture de la sous-section Eléments techniques sur la mémoire est nécessaire pour comprendre les explications techniques des attaques.

Toutes les explications restent théoriques. Seule la lecture d'un ouvrage comme *The Art of Exploitation* peut permettre de mettre en œuvre soi-même les attaques.

1.5.1. Dépassement de pile simple

Présentation de l'attaque

La fonction C suivante est vulnérable à une attaque par dépassement de pile simple, également appelée *stack smashing attack* :

```
1   void fonction(char * chaine) {
2       char buffer[128];       // Déclare un buffer de 128 caractères
3       strcpy(buffer, chaine); // Copie chaine dans buffer
4   }
```

A la ligne 3, `chaine` est copiée dans `buffer` sans contrôle de la longueur. Si chaine fait plus de 128 caractères, il est possible de corrompre la mémoire au-delà de `buffer` puisqu'aucun contrôle de longueur n'est réalisé et ainsi réaliser un dépassement de pile simple.

Catégorie

Attaque par dépassement de buffer / Dépassement de pile.

Explication technique

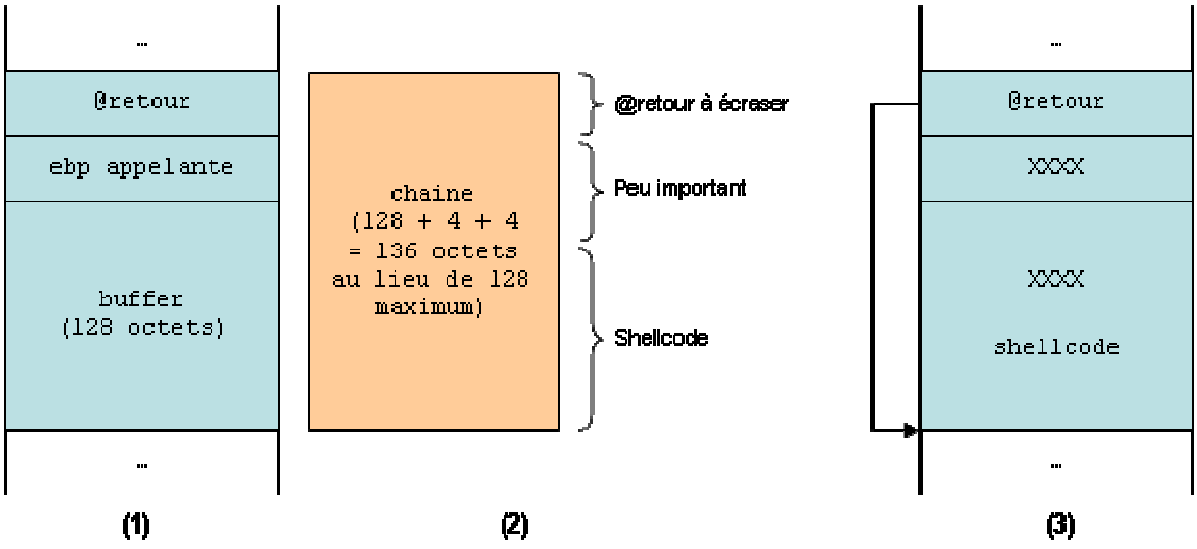


Figure 6 - Etapes d'une attaque par dépassement de pile simple

Avant d'exécuter la ligne 3, la pile se trouve dans l'état (1), sur la figure ci-dessus. Pour rappel, sont empilés : l'adresse de l'exécution à exécuter au retour de la fonction (`@retour`), la sauvegarde de la valeur du registre `ebp` dans la fonction appelante et enfin l'espace pour les variables locales (le buffer, ici).

A la ligne 3, `chaine` est copiée dans `buffer` sans aucun contrôle de longueur. Il est possible d'écraser la mémoire au-delà de `buffer` si `chaine` fait plus de 128 caractères.

Admettons que `chaine` provienne d'une entrée utilisateur et prenne la forme présentée ci-dessus (2). Elle est composée d'un shellcode, puis d'un contenu qui importe peu et enfin d'une adresse retour qui est en réalité l'adresse du shellcode.

Après la copie la pile se retrouve dans l'état (3).

Par conséquent, au retour de la fonction, le shellcode est exécuté. Si ce shellcode ouvre un *remote shell* (invite de commande à distance) et que le programme utilisant cette fonction est un serveur SUID `root`, alors l'attaquant obtient à un accès administrateur à la ressource compromise.

Solution/parade

Ce type d'attaque est possible quand la longueur des entrées utilisateur n'est pas contrôlée avant d'être copiée dans un buffer. Il faut donc prêter une attention particulière aux fonctions C suivantes :

- `strcat`, `strcpy`,
- `strecpy`, `streadd`, `strtrns`,
- `getwd`, `gets`,
- `sprintf`, `vsprintf`,
- `scanf`, `fscanf`, `sscanf`, `vscanf`, `vsscanf`, `vfscanf`,
- `syslog`,
- `realpath`, `getopt`.

La fonction vulnérable du paragraphe précédent peut être corrigée comme ceci :

```
1   void fonction(char * chaine) {
2       char buffer[128]; int i;     // Déclare un buffer de 128 caractères
3       for (i = 0 ; i < 127 ; i++)  // Copie 127 caractères maximum
4           buffer[i] = chaine[i];
5       buffer[127] = '\0';          // Place le terminateur en 128e
6   }
```

En effet, le buffer peut contenir 128 caractères. Or, il convient de rappeler qu'en C une chaîne de caractère se termine par un *NULL byte*, également appelé terminateur. Il faut donc veiller à :

- Copier au maximum 127 caractères dans le buffer de 128 caractères ;
- Vérifier que le dernier octet est bien le terminateur.

De même, il faut faire attention aux indices : les indices pour un buffer de 128 caractères vont de 0 à 127, le dernier octet étant donc repéré par l'indice numéro 127 et non 128.

Afin de contrôler la longueur des entrées, il est également possible de remplacer :

- `strcpy` par `strncpy`,
- `strcat` par `strncat`,
- `gets` par `fgets`,
- `sprintf` par `snprintf`.

L'autre correction possible pour la fonction vulnérable est :

```
1   void fonction(char * chaine) {
2      char buffer[128];             // Déclare un buffer de 128 caractères
3      strncpy(buffer, chaine, 127); // Copie 127 caracterès maximum
4      buffer[127] = '\0';           // Place le terminateur en 128e
5   }
```

Une bonne pratique pour les buffers de caractères est la suivante :

```
#define n 128              // Taille du buffer
char buffer[n];
buffer[n - 1] = '\0';      // nieme et dernier caractere : terminateur
strncpy(buffer, …, n – 1); // Copie n – 1 caracteres maximum
```

1.5.2. Dépassement un-octet

Présentation de l'attaque

La fonction C suivante est vulnérable à une attaque par dépassement un-octet, également appelée *off-by-one overflow attack* :

```
1   void fonction(char * chaine) {
2      char buffer[128]; int i;
3      for (i = 0 ; i <= 128 ; i++) // Copie 129 caractères
4         buffer[i] = chaine[i];
5   }
```

En effet, la boucle copie 129 caractères (de 0 à 128 inclus) de `chaine` dans `buffer`. Or ce dernier a une capacité de seulement 128. Il est par conséquent possible d'effectuer un dépassement un-octet.

Il s'agit ici d'un dépassement un-octet sur la pile car le buffer se trouve sur la pile (variable locale de la fonction `fonction`). Il est possible de trouver des dépassements un-octet sur le tas ou en section data/BSS selon l'endroit où le buffer est déclaré.

Catégorie

Attaque par dépassement de buffer / Dépassement de {pile, tas, section data/BSS}.

Explication technique

Soit la fonction appelante et la fonction principale `main` suivantes :

```
1   void appelante(void) {
2      fonction("test");
3   }
4
5   int main() {
6      appelante();
7      return 0;
8   }
```

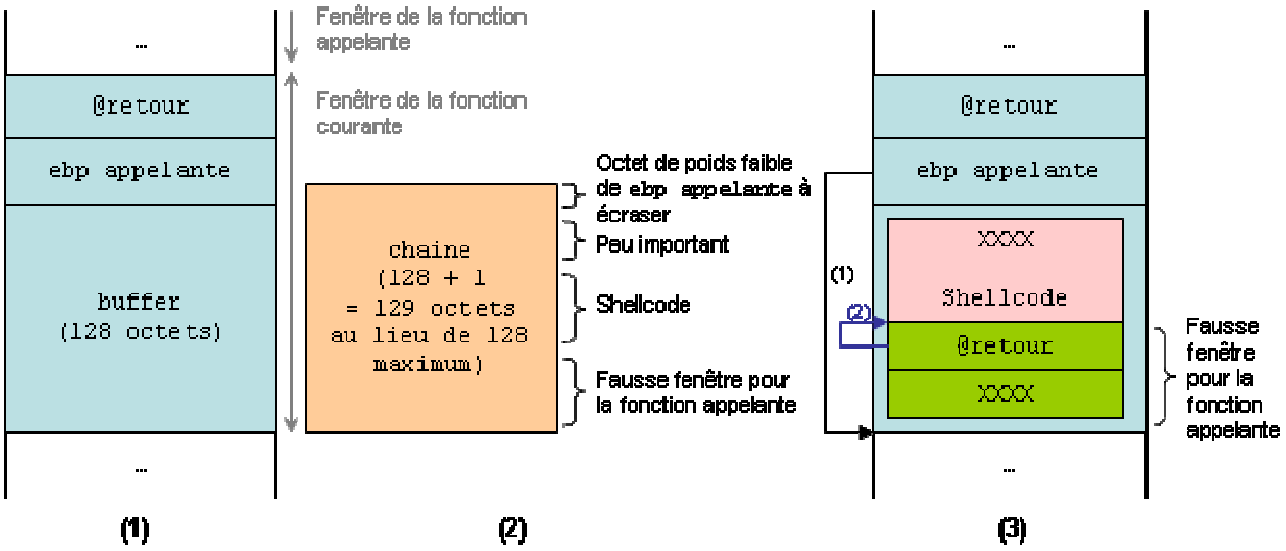


Figure 7 - Etapes d'une attaque par dépassement un-octet

Avant d'exécuter la ligne 3 de `fonction` (boucle), la pile se trouve dans l'état (1), ce qui est exactement la même chose que dans la sous-section précédente (Dépassement de pile simple).

Aux lignes 3 et 4, `chaine` est copiée dans `buffer` mais un octet de trop est copié. Il est ainsi possible d'écraser l'octet de poids faible de `ebp appelante` qui est la sauvegarde de la valeur du registre `ebp` dans la fonction appelante. Pour rappel le registre `ebp` contient l'adresse de référence de la fenêtre de la fonction courante. Les variables locales sont repérées par rapport à la valeur dans `ebp`.

Le but est de faire pointer `ebp appelante` sur une fausse fenêtre dans le buffer pirate.

Admettons que `chaine` provienne d'une entrée utilisateur et prenne la forme présentée ci-dessus (2). Elle est composée d'une fausse fenêtre pour la fonction appelante, d'un shellcode, puis d'un contenu qui importe peu et enfin de quoi écraser l'octet de poids faible de la sauvegarde `ebp`.

Après la copie la pile se retrouve dans l'état (3).

Par conséquent, au retour de la fonction, `ebp` est restauré avec `ebp appelante` qui contient la valeur pirate, c'est-à-dire celle dont l'octet de poids faible a été modifié. La fenêtre de la fonction appelante devient donc notre fausse fenêtre (en kaki sur le schéma), fenêtre dans laquelle nous avons pris soin d'insérer une adresse retour qui pointe sur le shellcode. C'est pourquoi le shellcode est exécuté non pas au retour de la fonction `fonction` mais au retour de la fonction `appelante`.

Solution/parade

Il faut retenir la règle : un buffer de n caractères peut contenir au maximum n - 1 caractères et le terminateur en position n (indice n – 1 en langage C).

Toutes les solutions énoncées pour les attaques Dépassement de pile simple sont valables.

Toutefois, les attaques par dépassement un-octet deviennent plus difficiles car les compilateurs ajoutent désormais du *padding* lors de la déclaration de buffers, c'est-à-dire qu'ils réservent plus d'octets que nécessaire pour les buffers : par conséquent, un dépassement d'un seul octet est sans conséquence. Cependant, l'oubli du terminateur (*NULL byte*) peut être lourd de conséquences.

1.5.3. Dépassement de type

Présentation de l'attaque

Le programme C suivant est vulnérable à une attaque par dépassement de type, ce qui mène par la suite à un dépassement de pile simple :

```
1   void fonction(char * chaine, short n) {
2      char buffer[128]; int longueur_max = 128;
3      if (n < longueur_max)
4         strcpy(buffer, chaine);
5   }
6
7   int main(int argc, char * argv[]) {
8      fonction(argv[1], strlen(argv[1]));
9      return 0;
10  }
```

La fonction `main` passe à la fonction `fonction` l'entrée utilisateur `argv[1]` et la longueur de cette entrée `strlen(argv[1])`. Appelons cette longueur `n`. Dans `fonction`, si `n` est inférieur à la taille du buffer alors la copie (ligne 4) est exécutée sinon elle ne l'est pas. Ce programme semble donc correct.

Mais, `n` est de type `short` et les `shorts` sont limités à 32 767. Si l'on place 40 000 dans un `short` on obtient en réalité un nombre négatif, -25 536 dans ce cas. Par conséquent, si la longueur de l'entrée est supérieure à 32 767, alors `n` devient négatif. `n` est alors clairement inférieur à la longueur maximum autorisée. La copie (ligne 4) est ainsi exécutée alors qu'elle ne devrait pas : nous nous retrouvons dans la situation de dépassement de pile simple avec un `strcpy` sans contrôle de longueur.

Outre le programme vulnérable présenté précédemment, il faut faire attention avec les fonctions `strncpy` et `strncat`. Leurs prototypes sont les suivants :

- `char *strncpy(char *dest, const char *src, size_t n);`
- `char *strncat(char *dest, const char *src, size_t n).`

`n` est de type `size_t`, c'est-à-dire entier long non-signé (positif). Par conséquent, la conversion d'un entier négatif en entier positif (`size_t`) le convertit en un nombre très grand. Par exemple, la conversion de -1 en `size_t` donne 4 294 967 295, ce qui est un nombre très grand. Comme dans l'exemple précédent, indiquer -1 comme longueur à `strncpy` revient à copier un maximum de 4 294 967 295 caractères.

Catégorie

Attaque par dépassement de buffer / Dépassement de pile.

Explication technique

Cette attaque est globalement basée sur le même principe que celui détaillé dans la sous-section Dépassement de pile simple. La différence tient au fait que la chaîne pirate passée à la fonction `fonction` doit contenir des milliers de caractères de remplissage (par exemple 40 000 `A`) après l'adresse de retour afin de duper la vérification de la longueur de l'entrée utilisateur (ligne 3).

Solution/parade

Il convient de porter la plus grande attention aux conversions de types.

Il est recommandé, dans la compilation de programmes avec les compilateurs `gcc` ou `g++`, d'utiliser les options `-Wall –Wextra -Wconversion`.

1.5.4. Mauvaise utilisation des chaînes de formatage

Présentation de l'attaque

Le programme C suivant est vulnérable à une attaque sur chaîne de formatage, ou *format string attack* :

```
1   void fonction(char * chaine) {
2      printf(chaine);
3   }
4
5   int main(int argc, char * argv[]) {
6      fonction(argv[1]);
7      return 0;
8   }
```

Rappels sur `printf` et les fonctions d'affichage/formatage
Le prototype de `printf` est : `int printf(const char *format, ...)`. Le premier paramètre est la chaîne de formatage. Lorsque `printf` rencontre un modificateur (également appelé marqueur) dans la chaîne de formatage, elle les remplace par les paramètres suivants (`...`). Par exemple : `int nb = 10; printf("nb = %d en decimal, %x en hexadecimal", nb, nb);` Affiche : `nb = 10 en decimal, A en hexadecimal` (`%d` et `%x` sont remplacé par `nb` et `nb`). Parmi les modificateurs `%n` permet d'écrire le nombre de caractères affichés jusqu'à présent. Par exemple : `int nb; printf("abcd%n", &nb); printf(" - nb = %d", nb);` Affiche : `abcd - nb = 4` (`nb` contient 4 car 4 caractères ont été imprimés avant `%n` : `a`, `b`, `c` et `d`).

Imaginons qu'un utilisateur ou un attaquant fournisse l'entrée : `%08x|%08x|%08x|%08x|%08x`.

Cela n'affiche pas `%08x|%08x|%08x|%08x|%08x` mais des valeurs hexadécimales qui semblent aléatoires : elles ne le sont en réalité pas. En effet, il s'agit de valeurs présentes sur la pile. Dans la fonction `fonction`, `printf` est mal utilisée car `chaine` est utilisée comme chaîne de formatage. Par conséquent, si elle comporte des modificateurs, ceux-ci sont interprétés.

Que se passe-t-il dans le cas où `chaine` contient le modificateur `%n` ? Il est possible d'écrire dans la mémoire, d'où une attaque possible par corruption de mémoire.

Toutes les fonctions utilisant les chaînes de formatage sont vulnérables : `printf`, `fprintf`, `sprintf`, `snprintf`, `vprintf`, `vfprintf`, `vsprintf`, `vsnprintf`.

Catégorie

Attaque sur chaînes formatage.

Explication technique

L'objectif final est le même que pour les autres attaques : faire pointer l'adresse de retour de la fonction `fonction` sur un shellcode afin d'exécuter un code pirate.

Nous avons vu qu'avec le modificateur `%n` il est possible d'écrire des nombres dans la mémoire. Les adresses mémoires étant représentées par des nombres il est donc possible d'écraser l'adresse retour d'une fonction pour la faire pointer sur un code pirate.

Par conséquent, en utilisant judicieusement les modificateurs `%x` et `%n`, il est possible, dans la fonction précédente, de faire pointer l'adresse de retour de `fonction` sur un shellcode.

Solution/parade

La solution est très simple. Au lieu de :

```
printf(chaine);
```

Il faut écrire :

```
printf("%s", chaine);
```

Pour aider le programmeur à détecter ces erreurs il est recommandé, dans la compilation de programmes avec les compilateurs `gcc` ou `g++`, d'utiliser les options `-Wall –Wextra -Wformat=2`.

1.5.5. Ecrasement de pointeur de fonctions

Présentation de l'attaque

Le programme C suivant est vulnérable à une attaque par dépassement de buffer dans la section BSS (lieu de stockage des données globales non initialisées) :

```
1   void fonction() {
2      printf("void fonction()\n");
3   }
4
5   int main (int argc, char * argv[]) {
6      static char buffer[128];
7      static void (*ptrfct)(); // Pointeur de fonction
8
9      ptrfct = fonction;
10
11     strcpy(buffer, argv[1]); // Copie sans vérification de longueur
12     (void) (*ptrfct)();
13
14     return 0;
15  }
```

En effet, aux lignes 6 et 7, les pointeurs sont déclarés avec le mot clé `static` et les données statiques sont des données globales (non initialisées dans ce cas). Ce type d'attaque peut être perpétré sur la pile, sur le tas ou en section data/BSS. Nous l'illustrons pour la section BSS afin de varier les exemples.

A la ligne 11, l'entrée utilisateur `argv[1]` est copiée dans le buffer `buffer` sans contrôle de la longueur. On peut donc faire déborder le buffer `buffer`. Or le pointeur de fonction `ptrfct` est stocké juste au dessus du buffer `buffer`. Fournir une entrée grande permet donc d'écraser le pointeur `ptrfct` et de potentiellement le faire pointer sur un code pirate. Ainsi, à la ligne 12, le code pirate serait exécuté.

Catégorie

Attaque par dépassement de buffer / Dépassement de {pile, tas, section data/BSS}.

Explication technique

Cette attaque est basée sur le même principe que celui détaillé dans la sous-section Dépassement de pile simple. La différence tient au fait que l'on écrase un pointeur de fonction au lieu d'une adresse de retour.

A la ligne 9 du programme vulnérable, le pointeur de fonction `ptrfct` pointe sur la fonction `fonction`. Avant d'exécuter la ligne 11, la pile se trouve dans l'état (1) :

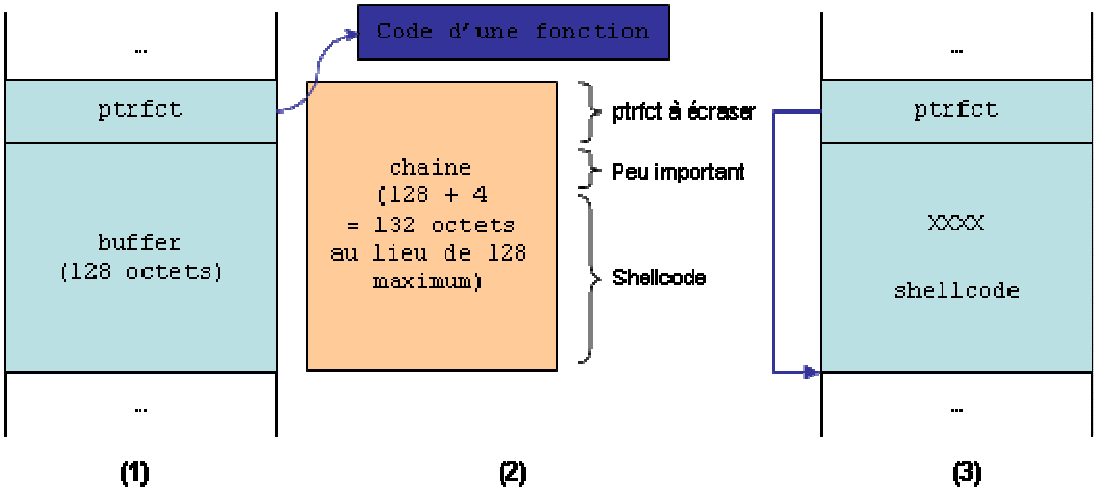


Figure 8 - Etapes d'une attaque par écrasement de pointeur de fonction

A la ligne 11, l'entrée utilisateur `argv[1]` est copiée dans `buffer` sans aucun contrôle de longueur. Il est possible d'écraser la mémoire au-delà de `buffer` si l'entrée utilisateur trop longue.

Le but est de faire pointer `ptrfct` sur l'adresse de `buffer`, c'est-à-dire là où le shellcode va être stocké.

Admettons que l'entrée utilisateur `argv[1]` prenne la forme présentée ci-dessus (2). Elle est composée d'un shellcode, puis d'un contenu qui importe peu et enfin d'une adresse qui est en réalité l'adresse de `buffer`, c'est-à-dire l'adresse du shellcode.

Après la copie la pile se retrouve dans l'état (3) : `ptrfct` est écrasée avec l'adresse du shellcode.

Par conséquent, lors de l'utilisation du pointeur de fonction à la ligne 12, le shellcode est exécuté à la place de la fonction `fonction`.

Solution/parade

Les solutions et parades sont identiques à celles décrites dans Dépassement de pile simple : il faut prêter la plus grande attention à la taille des buffers et contrôler la longueur des entrées.

1.5.6. Ecrasement de pointeur de données

Présentation de l'attaque

La fonction C suivante est vulnérable à une attaque par dépassement de tas :

```
1   typedef struct _structure {
2      char buffer[128];
3      int * p;
4      int a;
5   } structure;
6
7   void fonction(char * chaine) {
8      structure * x;
9      x = (structure *) malloc(sizeof(structure));
10     x->a = 0;
11     x->p = &(x->a); // p pointe sur a
12     strcpy(x->buffer, chaine);
13     *(x->p) = x->a;
14     printf("x->a = %08x\n", x->a);
15     free(x);
16  }
```

En effet, à la ligne 9, une structure de nom `structure` est allouée sur le tas avec la fonction `malloc` et le dépassement se produit dans le buffer `buffer` de cette structure lors de la copie à la ligne 12, copie de

chaine dans le buffer sans contrôle de la longueur. Ceci permet d'écrire au-delà du buffer et de corrompre le pointeur `p`, stocké juste après `buffer`.

Ce type d'attaque peut également être perpétré sur la pile ou en section data/BSS. Nous l'illustrons pour le tas car c'est dans cette portion de la mémoire que cette dernière révèle toute sa puissance. Les fonctions `malloc` et `free` manipulent des structures complexes et effectuent beaucoup d'opérations sur les pointeurs. Il est possible de réaliser des attaques similaires à celle exposée ici en cas de mauvaise utilisation de `malloc` et `free`. C'est en particulier le cas, pour certaines versions, où il est possible d'exécuter un code pirate après un *double-free*, c'est-à-dire si un même pointeur est libéré deux fois.

Catégorie

Attaque par dépassement de buffer / Dépassement de {pile, tas, section BSS}.

Explication technique

Dans cette attaque, il est possible d'écraser une adresse de retour comme dans le cas d'un dépassement de buffer simple, cependant nous préférons nous concentrer sur une technique où ni adresse de retour, ni pointeur de fonction n'est écrasé. Or, pour exécuter un code pirate, il faut obligatoirement que l'on modifie le cours de l'exécution du programme vers ce code pirate.

Ceci est possible grâce à la GOT (*Global Offset Table*).

Lorsqu'un programme est compilé, le compilateur/*linker* ne sait pas à quelle adresse les fonctions des bibliothèques dynamiques seront chargées lors de l'exécution. Il utilise donc une indirection : la *Global Offset Table*. Cette table permet à partir d'une adresse fixe fixée lors de la compilation de faire référence à une adresse variable lors de l'exécution. Cette adresse variable est maintenue à jour dynamiquement par le système en fonction de l'endroit où les bibliothèques dynamiques sont chargées.

Par exemple, lors d'un appel à `printf`, le programme consulte l'adresse fixe de `printf` dans la GOT pour connaître son adresse réelle dynamique et ainsi l'exécuter. Pour rappel, `printf` fait partie de la bibliothèque dynamique `libc` et son emplacement réel n'est donc connu qu'à l'exécution.

On remarque que `printf` est utilisée dans le programme vulnérable. Le but de l'attaque est donc de faire correspondre l'adresse fixe de `printf` dans la GOT, non pas avec l'adresse réelle de `printf` mais avec un shellcode. Ainsi le programme, en croyant exécuter `printf` à la ligne 14, exécutera en réalité le shellcode.

Avant d'exécuter la ligne 12, le tas se trouve dans l'état (1) :

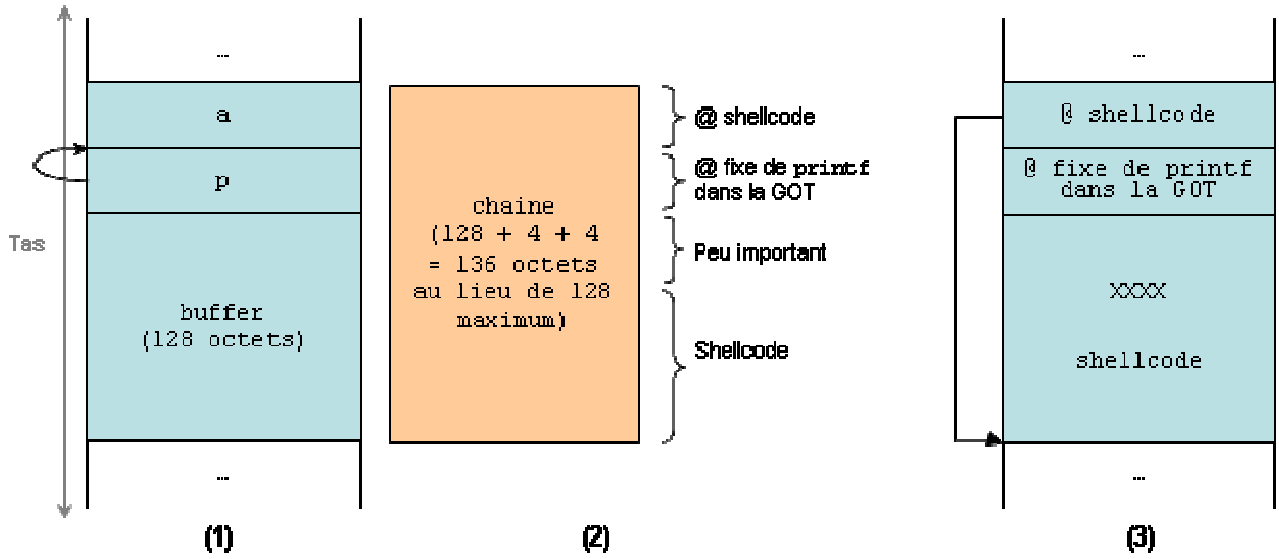


Figure 9 - Etapes d'une attaque par dépassement de tas/écrasement de pointeur

A la ligne 12, la chaîne passée en paramètre est copiée dans le buffer `buffer` de la structure `x` sans aucun contrôle de longueur. Il est possible d'écraser la mémoire au-delà de `buffer` si l'entrée utilisateur est trop longue, c'est-à-dire d'écraser le pointeur `p` et l'entieer `a`.

Admettons que `chaine` provienne de l'entrée utilisateur et prenne la forme présentée ci-dessus (2). Elle est composée d'un shellcode, puis d'un contenu qui importe peu et enfin de l'adresse fixe de `printf` dans le GOT et de l'adresse du shellcode (adresse de `buffer`).

Après la copie le tas se retrouve dans l'état (3) : `p` et `a` sont respectivement écrasés avec l'adresse fixe de `printf` dans la GOT et avec l'adresse du shellcode.

Lors de l'exécution de la ligne 13, on déréférence `p` et on y copie `a`. Etant donné l'état du tas lors de cette action, déréférencer `p` revient à accéder à l'adresse dynamique de `printf`. Cela revient donc à dire : *écraser l'adresse dynamique de `printf` dans la GOT avec l'adresse du shellcode.* Ainsi, lors de l'appel de `printf` à la ligne 14, le shellcode est exécuté en lieu et place de `printf`.

L'adresse fixe d'une fonction dans un programme compilé est déterminée avec `objdump` sous Linux :

```
~/dunod# objdump -R pointeur | grep printf
08049704 R_386_JUMP_SLOT   printf
~/dunod#
```

Dans l'exemple ci-dessus `0x08049704` est l'adresse fixe de `printf` dans la GOT.

Les attaques par dépassement de tas font, la très grande majorité du temps, intervenir des pointeurs.

Solution/parade

Les solutions et parades sont identiques à celles décrites dans Dépassement de pile simple : il faut prêter la plus grande attention à la taille des buffers et contrôler la longueur des entrées.

De plus, pour se protéger des vulnérabilités *double-free* il est intéressant de programmer une macro à utiliser à la place de la fonction `free` :

```
#define myFree(x) if (x) { free(x); x = 0; }
```

Cette macro permet de fixer le pointeur libéré à 0. Par conséquent, libérer deux fois un pointeur `p` revient à faire `free(p)` puis `free(0)`. Or, un `free(0)` est totalement inoffensif et ne fait rien.

Il est possible de créer la même macro pour `delete` et `delete []` en C++ :

```
#define myDelete (x) if (x) { delete    x; x = 0; } // Libère un pointeur simple
#define myDeletea(x) if (x) { delete [] x; x = 0; } // Libère un tableau
```

1.5.7. Return-to-libc

Présentation de l'attaque

La fonction C suivante est vulnérable à une attaque par dépassement de pile simple, également appelée *stack smashing attack* :

```
1   void fonction(char * chaine) {
2       char buffer[128];
3       strcpy(buffer, chaine);
4   }
```

Elle est en tout point identique à la fonction vulnérable de la sous-section Dépassement de pile simple. La même attaque que précédemment peut donc être perpétrée.

Nous nous intéressons dans cette sous section à une façon différente d'aborder et d'exécuter l'attaque. Le résultat produit est le même. L'explication technique, ci-dessous, est différente.

Catégorie

Attaque par dépassement de buffer / Dépassement de pile.

Explication technique

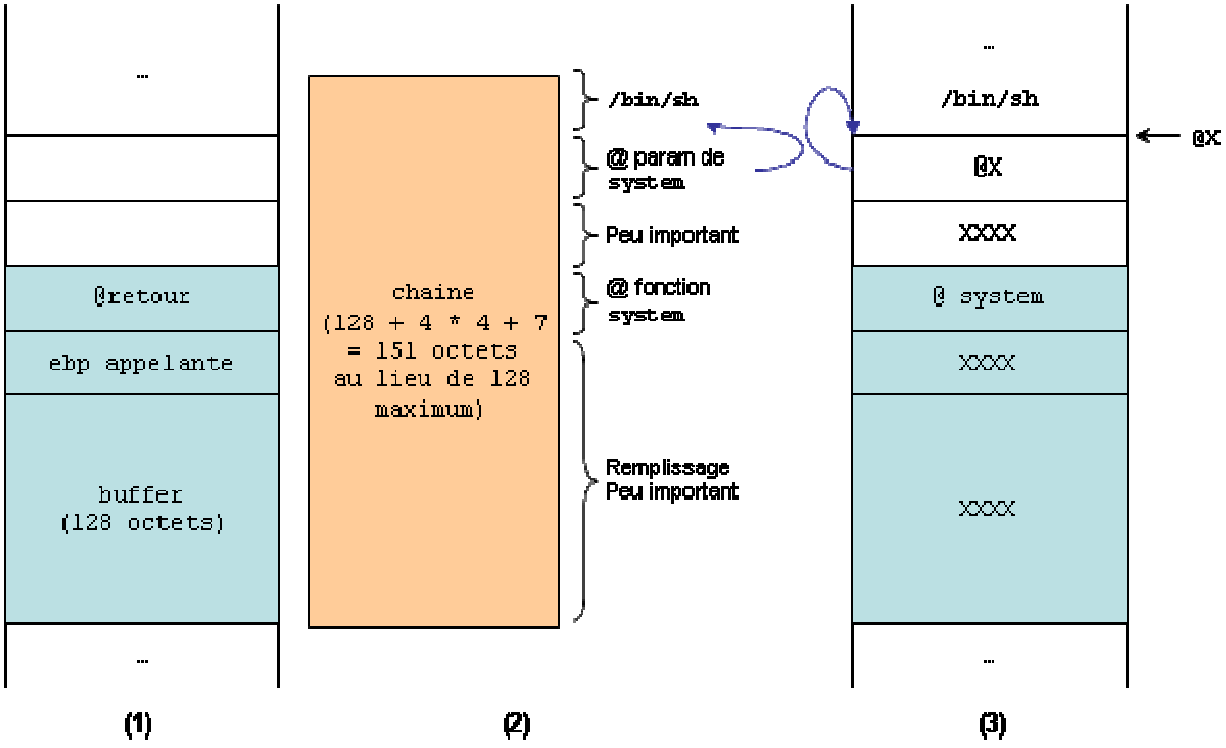


Figure 10 - Etapes d'une attaque return-to-libc

Avant d'exécuter la ligne 3, la pile se trouve dans l'état (1), sur la figure ci-dessus. A la ligne 3, `chaine` est copiée dans `buffer` sans aucun contrôle de longueur. Il est possible d'écraser la mémoire au-delà de `buffer` si `chaine` fait plus de 128 caractères.

On cherche à faire exécuter la commande suivante : `system("/bin/sh")`. Un shellcode pourrait convenir mais la fonction `system` est directement présente dans la `libc`. Utilisons-la. En effet, une des protections contre les attaques par corruption de mémoire consiste à empêcher l'exécution de code dans la pile (cf. Pages non exécutables). Il serait donc impossible dans ce cas d'exécuter un shellcode stocké dans `buffer` (sur la pile). Mais il est toujours possible d'exécuter une fonction de la `libc`.

Admettons que `chaine` provienne d'une entrée utilisateur et prenne la forme présentée ci-dessus (2). Elle est composée de caractères de remplissage, de l'adresse de la fonction `system`, de remplissage, du paramètre de `system` (c'est-à-dire l'adresse de la chaîne `/bin/sh` dans cet exemple) puis enfin de la chaîne `/bin/sh`. Après la copie la pile se retrouve dans l'état (3). L'adresse de retour `@retour` de la fonction `fonction` est remplacée par l'adresse de la fonction `system`.

Les 4 octets de remplissage entre l'adresse de la fonction `system` et son paramètre sont en réalité l'adresse de la fonction à exécuter au retour de `system`. Ceci importe peu dans cet exemple.

L'adresse de la fonction `system` en mémoire peut être déterminée avec le débogueur `gdb`.

Par conséquent, au retour de la fonction, la fonction `system("/bin/sh")` est exécutée, ce qui ouvre un *shell* (invite de commande). Si le programme utilisant cette fonction est un serveur SUID `root`, alors l'attaquant obtient à un accès complet à la ressource compromise.

Solution/parade

Les solutions et parades sont identiques à celles décrites dans Dépassement de pile simple : il faut prêter la plus grande attention à la taille des buffers et contrôler la longueur des entrées.

1.5.8. Ecrasement de la section .dtors

Présentation de l'attaque

Sur certains systèmes, il est possible de faire déborder un buffer dans la section `data`. La fonction C suivante peut être vulnérable à une attaque par dépassement de section `data` :

```
1   char buffer[128] = {1};
2
3   void fonction(char * chaine) {
4      strcpy(buffer, chaine);
5   }
```

Elle est identique à la fonction vulnérable de la sous-section Dépassement de pile simple à la différence près que `buffer` est une variable globale initialisée et se situe donc dans la section `data` et non pas dans la pile (cf. Organisation de la mémoire).

L'explication technique permet de mieux comprendre comment réaliser cette attaque. La finalité de l'attaque est la même que pour Dépassement de pile simple : exécuter un code pirate.

Catégorie

Attaque par dépassement de buffer / Dépassement de section `data`.

Explication technique

Sur certains systèmes, la section `.dtors` suit la section `.data` d'un programme. Ces systèmes sont vulnérables. Qu'est-ce que la section `.dtors` ? C'est une liste d'adresses qui correspondent à l'adresse des fonctions *destructors* à lancer une fois le programme terminé, après le dernier *exit*.

La commande Linux `objdump -s -j .dtors programme` permet d'explorer la section `.dtors` d'un exécutable qui s'appelle `programme`. La plupart du temps elle a le contenu suivant : `ffffffff 00000000`. En présence d'un destructeur explicite dans le programme il y aurait l'adresse de ce destructeur entre `ffffffff` et `00000000`, pour donner par exemple `ffffffff 0804fff0 00000000`.

Le but de l'attaque est de remplacer `00000000` par l'adresse d'un shellcode. Ainsi, lorsque le programme se termine, celui-ci exécute le shellcode.

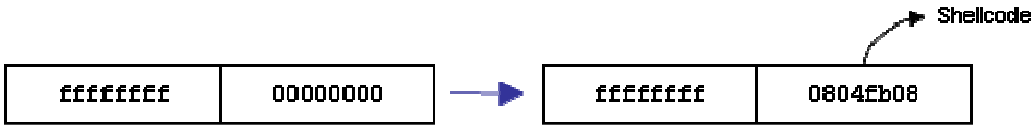


Figure 11 - Ecrasement de la section .dtors pour redirection sur code pirate

Les étapes de l'attaque sont présentées dans la figure ci-dessous :

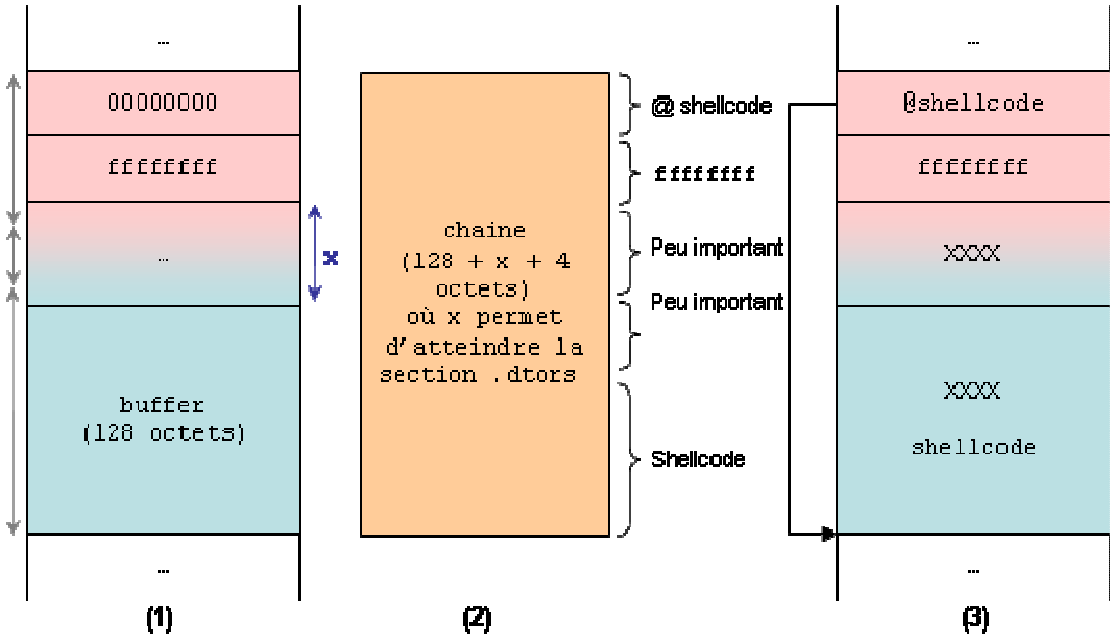


Figure 12 - Etape d'une attaque par dépassement de section data

Renseigner l'adresse du shellcode, c'est-à-dire l'adresse de `buffer`, à la place du destructeur permet d'exécuter le shellcode à la sortie du programme.

On peut noter qu'écraser la section `.dtors` est également possible dans une attaque par écrasement de pointeur. Dans la sous-section Ecrasement de pointeur de données, nous avons expliqué comment écraser l'entrée de `printf` dans la GOT : il est tout à fait possible d'écraser la `.dtors` à la place. Mais dans ce cas il faudrait attendre la fin du programme pour que le shellcode s'exécute.

Solution/parade

Les solutions et parades sont identiques à celles décrites dans Dépassement de pile simple : il faut prêter la plus grande attention à la taille des buffers et contrôler la longueur des entrées.

De plus, avoir recours à un système plaçant la section `.dtors` avant la section `data` permet de se prémunir du type d'attaque présenté ici : la plupart des systèmes Linux récents placent la section `.dtors` avant la section `data`, ce qui empêche de corrompre `.dtors` lors d'un dépassement en section `data`.

2. Audit de code

2.1. Contexte de réalisation d'un audit de code

2.1.1. Présentation des deux types de tests

Lorsqu'une entreprise développe un programme ou un logiciel, elle suit un cycle de développement. Quel que soit ce cycle de développement, il y a toutes les chances d'y retrouver diverses phases de tests. En effet, on dit que l'erreur est humaine. Or les développeurs sont des humains. Il est donc probable que le code développé par les développeurs de l'entreprise contienne des erreurs. Les phases de tests servent donc à et à corriger les erreurs dans le code source.

Cependant, il peut y avoir deux types d'erreurs qui nécessitent toutes deux des outils et des méthodes parfois différents :

- Les erreurs fonctionnelles ;
- Les erreurs impactant sur la sécurité du logiciel final.

Il est donc nécessaire, à chaque phase de test, de procéder en deux temps :

- Un temps pour les tests fonctionnels ;
- Un temps pour les tests relatifs à la sécurité.

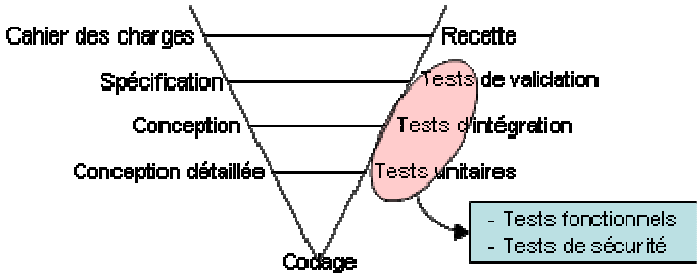


Figure 13 – Les deux types de tests sur un exemple de cycle de développement en V

2.1.2. Les tests fonctionnels

Les tests fonctionnels permettent de vérifier qu'une fonction ou qu'un programme résout correctement le problème qu'il est censé résoudre, c'est-à-dire qu'il fournit les sorties attendues en fonction d'entrées fixées. Prenons, par exemple, la fonction `f(x) = √x`. Un test fonctionnel de type test unitaire pour cette fonction est (en pseudo-langage) : `assert f(36) = 6` ou `assert f(4) = 2`, c'est-à-dire que l'on attend le résultat 6 pour l'entrée 36 et le résultat 2 pour l'entrée 4.

2.1.3. Les tests de sécurité

Les tests de sécurité permettent de s'assurer que le code produit répond à des critères de qualité précis et qu'il ne comporte pas d'erreur pouvant mener à des failles de sécurité. Par exemple, une fonction faisant mauvaise utilisation des chaînes de formatage en langage C (`printf`, …) est une erreur pouvant conduire à de graves problèmes de sécurité si elle est exploitée judicieusement par un attaquant.

Le schéma ci-dessous illustre comment une entreprise peut *constituer* ces tests de sécurité.

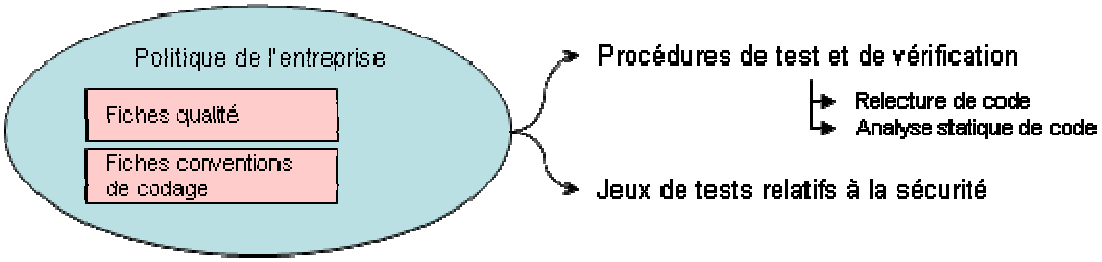


Figure 14 - Mise en œuvre d'une politique qualité/sécurité du logiciel

Pour chaque entreprise ou projet dans l'entreprise il existe des impératifs de sécurité, des standards de codage à respecter, etc. Ceux-ci sont centralisés dans ce que l'on nomme ici fiches qualité et fiches conventions de codage. Ces fiches permettent ensuite d'écrire deux types de choses :

- Jeux de tests relatifs à la sécurité

Au même titre que les jeux de tests fonctionnels qui testent si une fonction ou un programme résout correctement le problème qu'il est censé résoudre, il faut écrire des jeux de tests relatifs à la sécurité. Ceux-ci vérifient la robustesse de cette fonction ou de ce programme face à des entrées inattendues. Pour l'exemple de la fonction `f(x) = √x`, un test unitaire de sécurité consiste à vérifier que la fonction enregistre une erreur dans le cas d'une entrée négative (car le calcul de la racine carrée est impossible). Pour l'exemple d'un programme serveur http, un test d'intégration de sécurité consiste à vérifier que le serveur http renvoie une erreur 404 dans le cas d'une requête composée de milliers de caractères spéciaux (la page doit être inexistante) et qu'il ne plante pas. Les jeux de tests relatifs à la sécurité peuvent avoir lieu à tous les niveaux : unitaire, intégration, validation. L'important ici est de souligner la nécessité de tester la robustesse du code face à une utilisation *abusive*.

- Procédures de test et de vérification

Admettons que l'entreprise interdise l'utilisation de certaines fonctions du langage Java dans un projet Java. Ceci figure dans les fiches qualité ou conventions de codage. Il est ensuite nécessaire de rédiger une procédure d'audit qui va exposer au développeur, au testeur ou à l'auditeur la marche à suivre pour contrôler le code source et vérifier que ces fonctions interdites n'ont pas été utilisées. C'est ce que nous appelons ici la vérification ou l'audit de code.

L'audit de code s'appuie donc sur des procédures d'audit écrites à partir d'impératifs qualité et de conventions de codage. *Audit* signifie donc *vérification*. Par conséquent, la plus simple des vérifications est de relire le code écrit : cela peut aussi bien être l'affaire d'un développeur, que d'un testeur ou d'un auditeur contrôlant le code source d'un logiciel dans son ensemble. Heureusement, il existe un moyen d'automatiser en partie ce processus de relecture : l'analyse statique.

2.2. Analyse et détection de failles

2.2.1. Introduction à l'analyse statique de code

Origines de l'analyse statique

Le premier constat est qu'il est donc probable que les logiciels qu'une entreprise de *software* produit contiennent des erreurs. Le rôle des tests (fonctionnels ou sécurité) et de la relecture de code sont de repérer et de corriger ces erreurs.

Le deuxième constat est qu'une erreur, aussi minime soit-elle, peut avoir des conséquences graves sur la sécurité d'un logiciel : c'est ce que nous avons montré avec les attaques par corruption de mémoire dans la section Introduction aux attaques par corruption de mémoire.

Enfin, le troisième constat est que les développeurs ont tendance à commettre les mêmes erreurs dans le temps : les débutants commettent les erreurs que leurs aînés commettaient avant, et les expérimentés ne sont jamais à l'abri d'une étourderie.

Il est donc possible d'automatiser un minimum la détection de ces erreurs, parfois graves et récurrentes, grâce à l'analyse statique.

Fonctionnement de l'analyse statique

Les outils d'analyse statique se chargent de parcourir le code source d'un programme à la recherche d'erreurs : c'est donc une relecture de code (*code review*) automatisée. Une analyse statique vérifie le code source avant sa compilation, a contrario d'une analyse dynamique qui vérifie le comportement du programme à l'exécution (*run-time checking*).

Les erreurs détectées dépendent de l'outil, mais le plus souvent celles-ci sont :

- Fonctions présentant un risque de dépassement de buffer si elles sont mal utilisées (`strcpy`, `strcat`, etc. pour le langage C) ;
- Mauvaise utilisation de chaîne de formatage (`printf`, …) en C ;
- *Race conditions*, c'est-à-dire un double blocage de thread ou de fichier ;
- Vérification de propriétés spécifiées par un langage de spécification (`JML` pour Java, …) …

Ce qui nous intéresse dans la section présente sont les deux premiers points, c'est-à-dire la capacité de ces outils à détecter des fonctions mal utilisées ou des constructions pouvant mener à des attaques par corruption de mémoire. Nombre des erreurs à éviter ont été exposées dans la section précédente, dans la sous-section Les différents types d'attaques par corruption de mémoire.

Il est cependant nécessaire de noter qu'un outil d'analyse statique reste un outil d'aide pour la sécurité et ne remplace en aucun cas l'œil humain. Il ne peut pas trouver toutes les erreurs et il peut trouver des erreurs qui n'en sont pas réellement. Une erreur non trouvée est un *false negative* et une erreur trouvée qui n'en est pas réellement une est un *false positive*. Par conséquent l'inspection manuelle du code est toujours nécessaire mais grandement facilitée par ce genre d'outils. Le tableau suivant récapitule leurs avantages et leurs inconvénients :

Avantages	Inconvénients
• Examine tous les chemins dans un programme (*test coverage* de 100%) • Nettement plus rapide que l'œil humain • Analyse basée sur une base de données recensant diverses erreurs répétées par les développeurs à travers le temps	• N'interprète pas l'architecture du logiciel • N'interprète pas la sémantique d'une fonction • Nécessite un œil expert pour corriger les erreurs potentielles trouvées

Un outil d'analyse statique produit donc une liste de vulnérabilités potentielles. Celles-ci doivent être étudiées par un œil expert et corrigées si besoin est. Pour la plupart des outils il est possible de placer une annotation dans le code source, sous forme de commentaire, afin d'éviter de signaler un *false positive*.

2.2.2. Outils d'analyse statique de code

Compilateur avec niveau d'avertissement maximum

L'outil de base d'analyse statique est le compilateur lui-même. Il convient d'activer le niveau d'alerte au maximum. Par exemple, pour le compilateur GCC cela consiste à utiliser les options `-Wall -Wextra`. On peut ajouter à cela les options `-Wformat=2` pour les alertes sur une mauvaise utilisation des chaînes de

formatage et `-Wconversion` pour les alertes sur les conversions suspectes de types. Par conséquent, pour simplement compiler et *linker* un fichier, GCC devrait systématiquement être utilisé comme ceci :

```
gcc –Wall –Wextra -Wformat=2 -Wconversion programme.c –o programme
```

Outils spécialisés

Les outils d'analyses statiques peuvent être classés selon le langage de programmation qu'ils analysent et selon qu'ils sont en licence Open Source, freeware ou commerciale.

Le tableau suivant donne le nom de quelques outils :

Licence	Langage	Noms d'outils
Open Source	C/C++	Flawfinder, RATS (*Rough Auditing Tool for Security*), ITS4, Splint, CQual/CQual++, Sparse, Eau Claire
	Java	ESC/Java, PMD, FindBugs
Freeware	C/C++	Microsoft PREfast
Commerciale	C/C++	Parasoft C++test, Coverity Prevent for C/C++, GrammaTech CodeSonar
	Java	Coverity Prevent for Java, Klocwork Developer for Java
	Divers	Ounce Labs - Ounce 6, Fortify 360, Klocwork Insight

Globalement, les logiciels commerciaux sont plus complets que les outils Open Source. Les logiciels commerciaux ont la plupart du temps une interface graphique et proposent de nombreuses fonctionnalités, ce qui en font des outils utiles pour un audit de code au niveau intégration ou validation. Les outils Open Source sont pour la plupart accessibles en ligne de commande et sont très simples d'utilisation. En contrepartie, leurs fonctionnalités sont moins nombreuses. Ils sont plutôt destinés à analyser le code lors des phases de tests unitaires.

Il convient de préciser que l'entreprise peut également développer en interne ses propres outils d'analyse statique. L'un des principes de l'analyse statique est de détecter des constructions qui peuvent mener à des failles de sécurité. Il existe plusieurs méthodes de détection :

- Utilisation d'expressions régulières, l'outil de base étant la commande `grep` sous Unix ;
- Création d'un *parser* avec par exemple JavaCC pour Java ou Flex/Bison pour C/C++.

2.2.3. Exemple de processus d'analyse statique

Soit le programme C `vuln.c` présenté en annexe. Il comporte des erreurs de programmation exposées dans la section précédente dans la sous-section Les différents types d'attaques par corruption de mémoire. L'exemple de processus ci-dessous est réalisé sous Linux Slackware 12.0.

Première étape : compilation

On compile le programme (GCC version 4.2.3) avec la commande suivante :

```
gcc -Wall -Wextra -Wformat=2 -Wconversion vuln.c -o vuln
```

GCC effectue un premier niveau d'analyse et repère quelques erreurs :

- Mauvaise utilisation potentielle des chaînes de formatage à la ligne 27,

- Types différents dans la comparaison à la ligne 14 (`short`, `int`) et conversion de `size_t` (`unsigned int`) vers `int` à la ligne 82 lors de l'appel de `fonction3`.

Ces deux erreurs s'avèrent être bien des erreurs pouvant mener à des failles de sécurité.

Deuxième étape : Flawfinder

On teste le code source avec la commande Flawfinder (version 1.27) suivante :

```
flawfinder vuln.c
```

Cette commande permet de trouver 15 erreurs potentielles. Flawfinder renvoie :

- Une alerte de niveau élevé pour chaque utilisation de la fonction `strcpy` (5),
- Une alerte de niveau élevé pour mauvaise utilisation de `printf`,
- Une alerte de niveau moyen pour chaque déclaration de tableau de caractères de taille fixe (5),
- Une alerte de niveau moyen pour dépassement potentiel d'entier lors de l'utilisation de `atoi` à la ligne 70 (`atoi` convertit une chaîne de caractère en un entier),
- Une alerte de niveau faible pour chaque utilisation de la fonction `strlen` (2).

On remarque que cette analyse ne trouve pas la faille liée à une conversion de type (ligne 82) et la faille liée à un mauvais indice de boucle (ligne 14/15). En effet, Flawfinder ne gère pas les types de données des paramètres de fonctions et le *control flow* ou le *data flow* du programme. D'autres outils, comme Splint, gèrent cela. Le rôle de Flawfinder est donc de fournir des informations basiques et immédiates.

Troisième étape : Splint

Comme spécifié dans le paragraphe précédent, Splint est plus évolué que Flawfinder. Sa sortie est donc plus compliquée et le développeur ou testeur peut être perdu dans la masse d'informations.

On utilise la commande Splint (version 3.1.2) suivante :

```
splint -strict vuln.c
```

L'option `-strict` permet d'activer le niveau de détail le plus élevé. Cela fournit un nombre très important d'alertes (84 au total) mais permet d'afficher les deux failles non trouvées par Flawfinder :

- Types d'opérandes incompatibles (`short`, `int`) à la ligne 21,
- Ecriture hors-frontière (*out-of-bounds store*) à la ligne 15.

Quatrième étape : conclusion

La masse d'informations fournies par ces outils peut être très importante. Il peut donc être intéressant de configurer ces outils ou de créer des scripts filtrant quelques sorties intéressantes.

De plus, on constate que tous les outils d'analyse statique ne sont pas équivalents : il est donc nécessaire de les tester afin de se faire une idée précise de leurs fonctionnalités et de voir si oui ou non ils peuvent se révéler utiles dans le processus qualité du logiciel.

Enfin, les outils présentés dans cet exemple sont des outils qui peuvent aussi bien être utilisés en phase de test unitaire par des développeurs qui souhaitent tester ou auditer leur code immédiatement, que par une équipe de test à un niveau plus haut (phase de test d'intégration ou de validation) souhaitant auditer un programme ou un logiciel dans son ensemble.

2.2.4. Inspection manuelle de code

Comme spécifié en introduction de l'audit de code, un outil d'analyse statique est un outil d'aide. Il ne remplace pas l'œil humain. Certaines procédures de vérification peuvent nécessiter l'inspection manuelle de code. De plus, l'audit de l'architecture applicative générale d'un programme ou d'un logiciel ne peut se faire avec un quelconque outil présenté dans Outils d'analyse statique de code. C'est pourquoi l'inspection manuelle reste de rigueur dans certains cas.

2.3. Parades et protections

2.3.1. Correction des programmes ou logiciels développés

L'analyse statique permet de révéler des constructions pouvant mener à des erreurs affectant la sécurité d'une fonction, d'un programme ou d'un logiciel. Il faut par conséquent prendre en compte les remarques faites par ces outils et corriger les erreurs si erreur il y a, c'est-à-dire s'il ne s'agit pas d'un *false positive*.

A la suite d'une inspection manuelle, il peut également être nécessaire d'apporter des corrections au code source, de même qu'à la suite de tests de sécurité non concluants.

Les corrections apportées au code source des programmes ou logiciels développés font donc immédiatement suite à l'audit de code, quelque-soit le niveau auquel il est réalisé.

2.3.2. Utilisation d'un langage de haut-niveau

C et C++ sont certes des langages de haut-niveau mais la gestion de la mémoire est loin d'être automatique dans ces langages. Au contraire, il existe d'autres langages haut-niveau avec éventuellement gestion automatique de la mémoire via un ramasse-miettes (*garbage collector*). Ces derniers sont le plus souvent interprétés soit directement soit en passant par un code intermédiaire (*byte-code*). La sécurité de ces langages est supérieure et dépend surtout de la machine virtuelle qui les interprète. C'est le cas de Java, PHP, … Tous ces langages sont interprétés par des machines virtuelles et ont, dans leur histoire, connu des failles liées à des vulnérabilités dans la machine virtuelle.

2.3.3. Protection de la pile par utilisation de canaries

Présentation de la protection

La protection de la pile par utilisation de canaries n'est pas une réponse à un audit de code à proprement parler. C'est une option de compilation insérant un code supplémentaire au programme permettant de prévenir les attaques par dépassement de buffer dans le cas où une vulnérabilité logicielle aurait échappé au testeur ou à l'auditeur et se verrait exploitée par un hacker.

Explication technique

Le principe de protection de la pile par utilisation de canaries est le suivant :

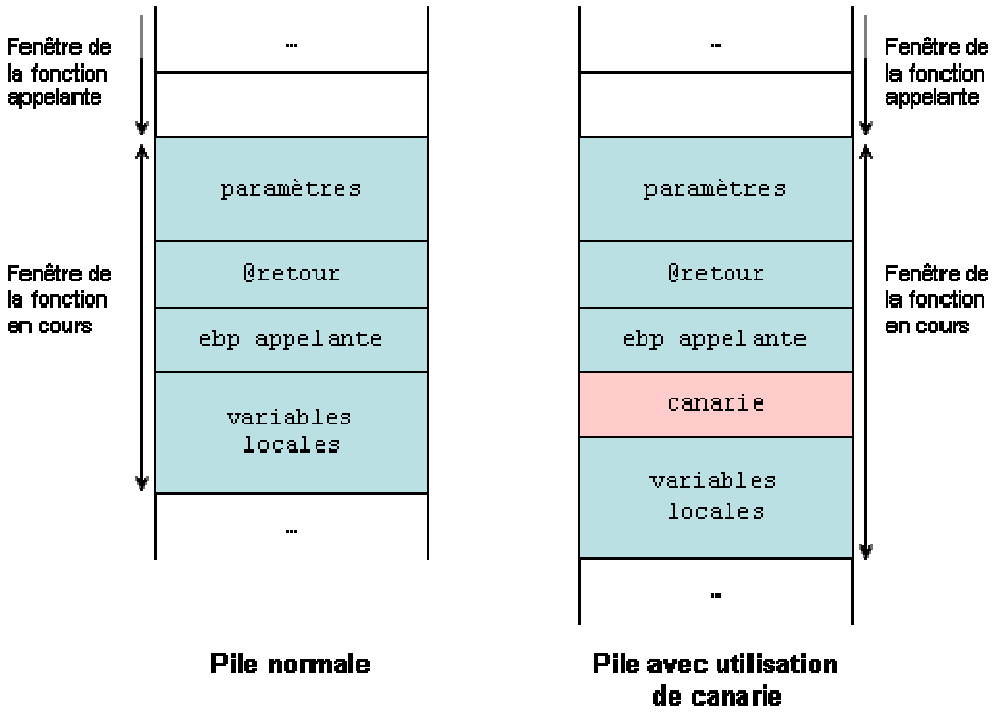


Figure 15 - Principe de protection de la pile par utilisation de canaries

A gauche nous avons l'état de la pile lors d'un appel de fonction normal, sans protection par canarie. A droite nous avons la même chose mais avec insertion d'un canarie.

Le plus souvent le canarie est une chaîne aléatoire calculée à l'initialisation du programme et insérée à chaque appel de fonction entre la sauvegarde du registre `ebp` et les variables locales. Au retour de la fonction, si le canarie est corrompu alors le programme se termine automatiquement. Par conséquent si un buffer (variable locale) écrase `@retour`, il écrase et corrompt aussi le canarie, ce qui permet de détecter l'attaque par dépassement de buffer et ainsi y mettre un terme.

Le canarie opérant sur la pile, il ne protège que des attaques par dépassement de buffer sur la pile.

L'ajout d'un canarie a une contrepartie. La vérification de ce dernier à chaque retour de fonction a un impact sur les performances. Par exemple, la perte de performance d'Apache a été quantifiée à 8%.

Mise en œuvre

GNU GCC C Compiler

En 1997, StackGuard a été le premier à implémenter cette technologie. Il s'agissait d'une extension pour le compilateur GCC. On utilise désormais SSP/ProPolice (*Stack-Smashing Protection*) qui est une extension incluse dans GCC. Il faut pour cela utiliser l'option `-fstack-protector-all`. Outre l'insertion de canaries, SSP réorganise aussi les variables locales en mémoire en mettant les buffers au-dessus des autres variables : ceci permet de se protéger contre les attaques par écrasement de pointeurs de fonctions sur la pile et les attaques modifiant le contenu d'autres variables.

La bonne utilisation du compilateur GCC sur le programme vulnérable `vuln.c` est :

```
gcc -Wall -Wextra -Wformat=2 -Wconversion -fstack-protector-all vuln.c -o vuln
```

Microsoft Visual Studio

Avec Microsoft Visual Studio il est possible d'atteindre le même but, c'est-à-dire d'insérer un canarie lors d'un appel de fonction et de le vérifier à son retour, avec l'option de compilation `/GS`.

3. Audit de vulnérabilités

3.1. Contexte de réalisation d'un audit de vulnérabilités

L'expression exacte est en réalité *audit de vulnérabilités des ressources d'un réseau*. Il consiste à tester, de manière périodique, la fiabilité et la sécurité des ressources (serveurs, postes client, imprimantes, etc.) du réseau (local ou non) d'une entreprise afin de rechercher et corriger des failles connues. Par *connues* il faut comprendre que ce sont des failles portant sur des logiciels du marché, par exemple le système d'exploitation Windows, et qu'elles sont répertoriées dans une base de données de failles.

Une grande partie des failles pouvant être décelées sont des failles permettant de réaliser des attaques par corruption de mémoire. C'est pourquoi nous présentons l'audit de vulnérabilités.

Bien entendu, l'audit de vulnérabilités regroupe un ensemble plus large de tests : par exemple, un serveur accessible avec le nom d'utilisateur `root` et un mot de passe vide ou trivial est considéré comme une vulnérabilité puisqu'un attaquant peut facilement y accéder en devinant le mot de passe.

3.1.1. Cartographie du réseau

Afin d'effectuer un audit de vulnérabilité il est, dans un premier temps, nécessaire de cartographier le réseau à auditer. Ceci sort du cadre de ce chapitre. Mais il est nécessaire d'avoir une vue d'ensemble des ressources du réseau afin de répondre à la question : *quoi auditer ?*

3.1.2. Scanners de sécurité

Ensuite il faut utiliser des scanners de sécurité afin de scanner et auditer les ressources voulues. Il existe plusieurs types de scanners :

- Les scanners passifs se contentent d'écouter le réseau (*sniffing*) afin de détecter des services ou des versions de logiciel tournant sur des serveurs du réseau ;
- Les scanners actifs envoient de nombreuses trames sur le réseau afin d'auditer les ressources en profondeur et de recueillir le maximum d'informations ;
- Les scanners semi-actifs et semi-passifs combinent les deux techniques.

Les scanners de sécurité que nous présenterons par la suite sont des scanners actifs. En effet, le but de ces outils est d'être exhaustif sur le système audité. En effet, l'audit est réalisé par un auditeur, c'est-à-dire quelqu'un dont le travail est de détecter les failles sur les ressources du réseau. Il lui importe donc peu d'être détecté et tracé. Le hacker préfèrera quant à lui les scanners passifs afin de ne pas être détecté lors de sa récolte d'informations.

3.1.3. Planification

Trois règles sont à retenir :

- Les ressources doivent être auditées par un auditeur ou un administrateur réseau avant d'être placées sur le réseau de l'entreprise, et plus particulièrement les serveurs fournissant des services sensibles (authentification, financiers, etc.).
- Les ressources doivent être contrôlées à intervalle de temps régulier par un administrateur réseau. Bien entendu, les scanners de sécurité utilisés doivent être à jour lors de chaque audit.
- Les ressources doivent être audités occasionnellement et en profondeur par une équipe spécialisée dans l'audit des réseaux informatiques. Elle peut pour cela tenter de simuler les actions d'un hacker en utilisant notamment un ou des scanners passifs.

Ces trois règles doivent mener l'entreprise à établir un planning précis, des procédures et des tableaux de bord afin de savoir quand et quoi auditer, comment auditer et les résultats des audits.

3.1.4. Tableaux de bord

Nous entendons par tableau de bord le résultat d'un audit de vulnérabilités. Celui-ci peut être organisé de la façon dont l'entreprise le souhaite. Nous exposons ici un exemple de découpage en trois parties, avec un tableau de bord par ressource auditée :

- Identification de la ressource avec son nom, son adresse IP, la date d'audit, synthèse et statistiques sur le nombre de vulnérabilités trouvées (cf. deuxième point), etc.,
- Liste des vulnérabilités (description et référence) trouvées avec degré de dangerosité et suggestion de solution(s) à implémenter pour les corriger,
- Liste de recommandations pouvant être mises en œuvre pour renforcer la sécurité d'une ressource (par exemple, une politique de mot de passe trop flexible peut être signalée mais ne constitue pas en soit une vulnérabilité, car ceci reste subjectif),

Comme nous le verrons dans la sous-section suivante, la plupart des scanners de sécurité sont dotées de fonction de *reporting* et permettent de générer des rapports, que l'on appelle ici tableaux de bord.

3.1.5. Boîte noire, boîte blanche (credentials)

La plupart des scanners que nous présenterons dans la sous-section suivante permettent d'effectuer des analyses sur des ressources distantes en utilisant ou non des *credentials* (couples login / mot de passe).

Si les *credentials* sont fournis alors le scanner de vulnérabilités s'authentifie sur la ressource distante et réalise ainsi un test en profondeur. Par exemple, pour détecter tous les patchs manquants sur un système Windows distant, il est nécessaire de fournir les *credentials* administrateur. Si les *credentials* ne sont pas fournis alors le scanner de vulnérabilités réalise un test moins complet. Mais cela permet de constater ce qu'un hacker, sans connaissance préalable des ressources, peut découvrir et exploiter.

Les deux types de tests avec ou sans *credentials*, c'est-à-dire avec ou sans authentification sur la ressource distante auditée, sont donc importants. Les *credentials* peuvent être découverts lors d'un test d'intrusion : on parle alors d'audit boîte noire, c'est-à-dire d'un audit sans connaissance préalable sur les ressources du réseau. Si les *credentials* sont connus (par exemple l'administrateur réseau connaît tous les couples login / mot de passe des postes du réseau pour les opérations de maintenance) alors on parle d'audit boîte blanche, c'est-à-dire d'un audit avec connaissance préalable sur les ressources du réseau. Il est important d'effectuer les deux types d'audit. Dans la sous section Planification, nous parlions de trois règles de planification : le contrôle à intervalle de temps régulier peut être de type boîte blanche car récurrent et à effectuer rapidement, alors que le contrôle ponctuel peut être de type boîte noire et boîte blanche afin d'être le plus complet possible.

3.2. Analyse et détection de failles

3.2.1. Présentation des scanners de sécurité

Nous présentons ici des scanners de sécurité polyvalents, c'est-à-dire qu'ils permettent de scanner aussi bien des serveurs que des postes clients. D'autres scanners sont par exemple spécialisés pour les serveurs Web, les serveurs de base de données ou les accès Wifi.

Les critères du tableau ci-dessous sont les suivants :

- Compatibilité PCI : spécifie si le scanner permet de générer des rapports au standard PCI DSS – PCI DSS (*Payment Card Industry Data Security Standard*) est un standard de référence, développé par l'industrie bancaire (Mastercard, Visa, etc.), exposant les principes de sécurité à respecter pour les entreprise manipulant des systèmes de paiement par cartes ;
- Reporting / Solutions : spécifie si le scanner est capable de générer des rapports d'audit spécifiques dans des formats variés (XML, HTML, etc.) et s'il est capable de générer un *remediation report* (suggestions pour corriger les ressources vulnérables) ;

- Scheduler : spécifie si le scanner peut être programmé pour lancer une analyse à une date précise et de manière périodique ;
- Audits personnalisés : spécifie s'il est possible de lancer des alertes sur des audits personnalisés par l'auditeur (par exemple lancer une alerte si les mots de passe sur la ressource auditée n'expirent jamais ou si le compte n'est pas bloqué au bout de trois tentatives de login infructueuses) ;
- Découverte du réseau : spécifie si le scanner est capable d'énumérer les ressources présentes sur une plage d'adresses IP ;
- Déploiement de patchs : spécifie si le scanner est capable de déployer des patchs manquants sur une ressource distante automatiquement (les *credentials* de la ressource distante sont nécessaires).

	AdventNet Security Plus Manager 5.2	**eEye Retina Network Security Scanner 5.10.4**	**GFI LANguard 9.0-beta**	**MaxPatrol Security Scanner 7.5**
Prix	€1360 / 200 IP Gratuit / 5 IP	$1650 / 256 IP	$1981 / 256 IP	$9200 / 256 IP
Architecture	Client, serveur	Autonome	Autonome	Autonome
Systèmes hôte	Windows, Linux	Windows	Windows	Windows
Compatibilité PCI	X	X		
Reporting / Solutions	X	X	Add-on gratuit	X
Scheduler	X	X	X	X
Audits personnalisés		X		X
Découverte du réseau	X	X	X	
Déploiement de patchs	X		X	

	Microsoft Baseline Security Analyzer	**Qualys QualysGuard**	**Safety-Lab Shadow Security Scanner 7.147**	**Tenable Nessus 3.2.1**
Prix	Gratuit avec Windows	Cf. éditeur	$999 / 256 IP	$1200
Architecture	Autonome	En ligne	Autonome	Client, serveur
Systèmes hôte	Windows	Tous	Windows	Windows, Linux, Solaris, FreeBSD, MacOS
Compatibilité PCI		X		
Reporting / Solutions	X	X	X	X
Scheduler	Planificateur de tâches	X	X	Via ligne de commande
Audits personnalisés		X	X	
Découverte du réseau		X		
Déploiement de patchs				

On peut également citer d'autres scanners de sécurités :

- SARA (Security Auditor's Research Assistant, ex SATAN) – Open Source,
- IBM Internet Security Systems,
- CoreImpact Pro,
- SAINT (Security Administrator's Integrated Network Tool) …

Le meilleur produit est celui qui répond le mieux au besoin des administrateurs ou auditeurs. On peut s'aider des ressources Internet suivantes pour établir un choix :

- Top 10 Vulnerability Scanners : http://sectools.org/vuln-scanners.html,
- Security Scanners Chart : http://en.hakin9.org/attachments/consumers_test.pdf.

La sélection doit s'appuyer sur les fonctionnalités (déploiement de patchs, découverte, etc.) et sur les performances du produit. La méthodologie de test de performance est :

- Préparer des machines de tests avec des vulnérabilités connues,
- Utiliser le scanner de sécurité en mode profond (scan le plus complet et le plus lent),
- Comparer les résultats attendus avec les résultats obtenus.

3.2.2. Présentation du site SecurityFocus

SecurityFocus (http://www.securityfocus.com) est un site, propriété de Symantec, portant sur la sécurité informatique. C'est une source d'informations importante sur les vulnérabilités logicielles.

Il héberge la très célèbre liste de diffusion Bugtraq qui est le système de suivi des bugs (et donc en particuliers des failles permettant de réaliser des attaques par corruption de mémoire) des applications commerciales et Open Sources qui peuvent être utilisées sur un réseau (Windows, Linux, Apache, IIS, ftpd, PHP, machine virtuelle Sun Java, etc.).

Les vulnérabilités sont centralisées dans la section *Vulnerabilities* sur site. Elles sont identifiées :

- Soit par un identifiant CVE (*Common Vulnerabilities and Exposures*) fixé par le MITRE (organisme américain indépendant intervenant en autre sur les technologies de l'information),
- Soit par un identifiant CAN (*Candidate Number*) qui est un identifiant CVE temporaire en instance de validation par le MITRE
- Soit par un Bugtraq ID qui est la référence de la vulnérabilité sur le site SecurityFocus.

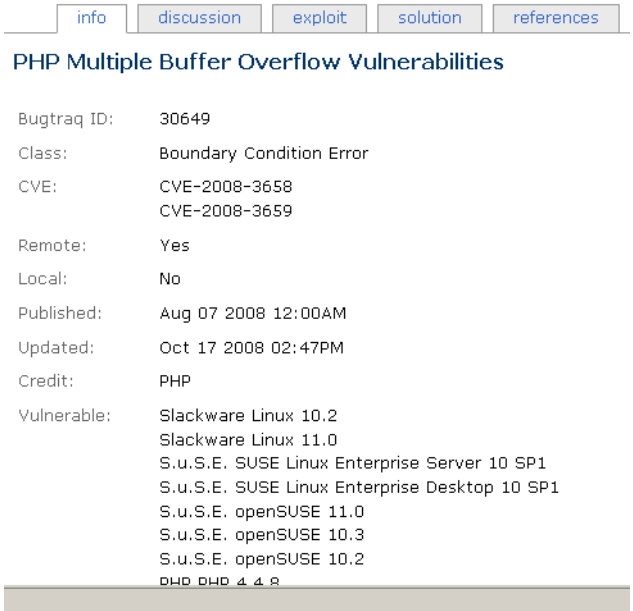


Figure 16 - Page exposant une vulnérabilité sur SecurityFocus

Pour chaque vulnérabilité il existe une page sur SecurityFocus. Les informations que l'on retrouve sont :

- Une page *Info* avec identification de la vulnérabilité (CVE, CAN, Bugtraq ID), l'application concernée et les systèmes vulnérables,
- Une page *Discussion* avec une description de la vulnérabilité,
- Une page *Exploit* avec parfois un script, un morceau de code ou un programme permettant d'exploiter la vulnérabilité à des fins de *Proof of Concept*,
- Une page *Solution* avec la plupart du temps un lien vers le correctif de la vulnérabilité (patch),
- Une page *References* avec des liens vers des ressources annexes concernant la vulnérabilité.

3.2.3. Présentation du Metasploit Framework

Le Metasploit Framework (http://www.metasploit.com/framework) est une plate-forme Open Source de création et d'utilisation d'outils de sécurité et d'exploits. Pour rappel, exploit est le terme utilisé pour parler d'un mini-programme permettant de compromettre une ressource vulnérable à une attaque, par exemple, de type corruption de mémoire.

Chaque exploit enregistré dans le Framwork est un module. Il est possible d'ajouter ses propres modules et donc exploits. Le Framework fournit de nombreux utilitaires et un cadre pour développer ses propres exploits, le tout étant programmé en langage Ruby. Il est également possible d'utiliser les exploits enregistrés. Pour exemple, le Metasploit Framework 3.1 comporte 262 exploits et 46 utilitaires directement utilisables. Il est disponible pour Windows et pour Linux/Unix. Le Metasploit Framework peut être utilisé en ligne de commande, via une interface Web ou via une interface graphique.

Il est simple à installer et simple à utiliser, particulièrement via l'interface graphique. Par conséquent il se révèle d'une grande utilité pour les auditeurs qui veulent tenter de pénétrer leurs propres systèmes. Mais il est également redoutable puisqu'il permet à des non-connaisseurs de compromettre des ressources du réseau de l'entreprise très simplement.

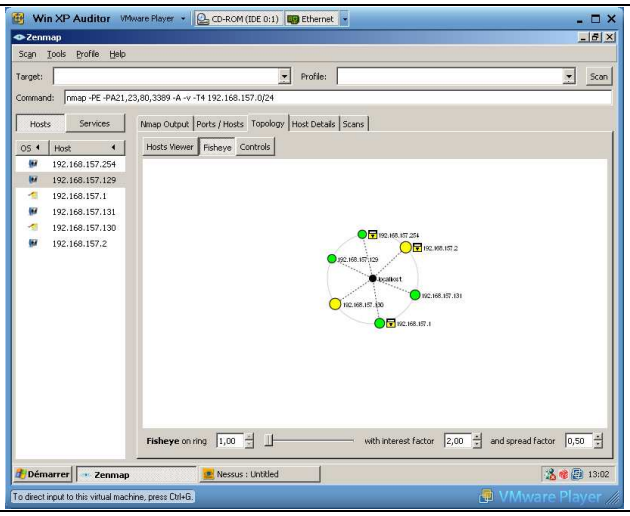
La sous-section suivante montre un exemple de détection de vulnérabilité sur une machine et ensuite l'aisance avec laquelle il est possible de compromettre cette machine en utilisant le Metasploit Framework.

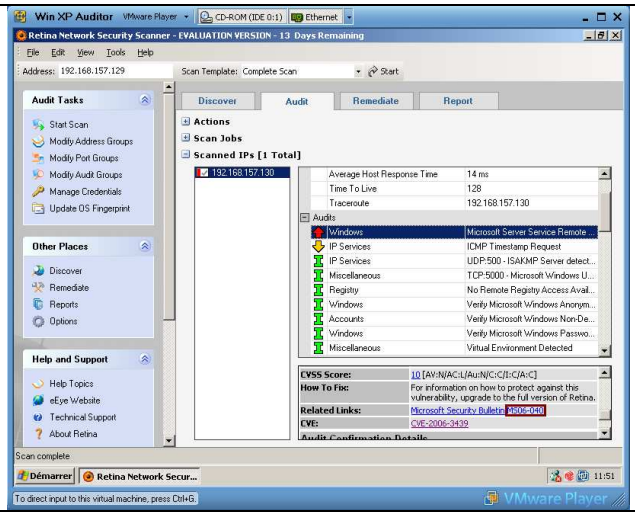
3.2.4. Exemple d'exploitation d'une machine

Dans cette sous-section, nous montrons comment une vulnérabilité peut être exploitée simplement par un attaquant, sous Windows, pour compromettre totalement une machine. La machine auditée et exploitée est une machine Windows XP SP0, connectée au réseau local avec l'adresse IP 192.168.157.130.

Cet exemple fait intervenir les outils :

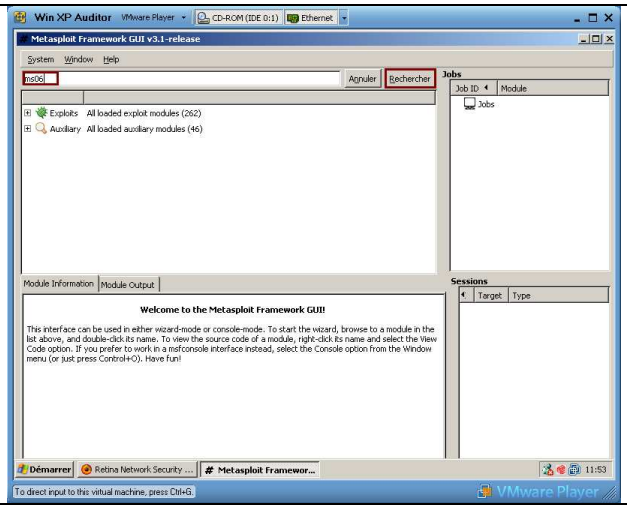
- Le scanner `nmap` pour la cartographie et l'identification des machines du réseau local,
- Le scanner Retina pour la détection de vulnérabilités,
- Le Metasploit Framework pour l'exploitation (compromission).

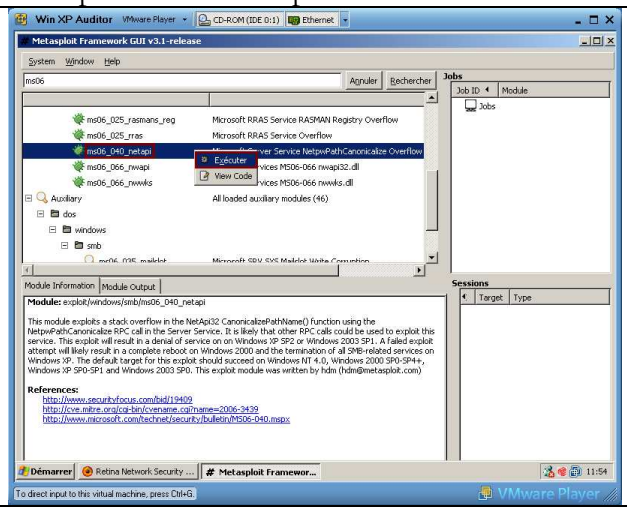




Nous avons un réseau local simple avec une machine Windows (XP SP0) en 192.168.157.130.

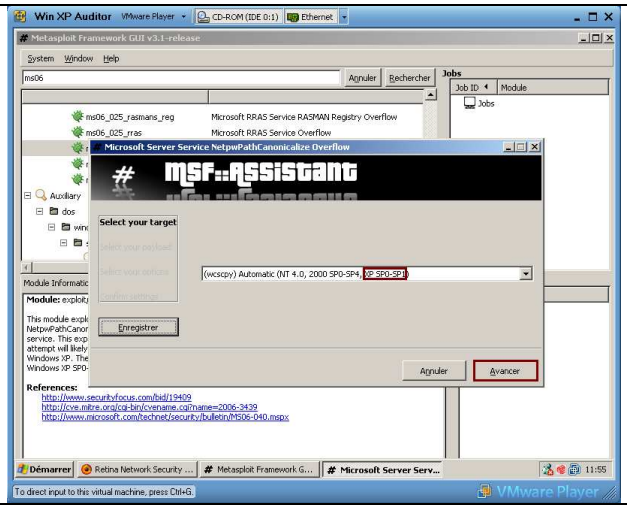
Retina (version de démonstration) est lancé sur l'adresse IP 192.168.157.130 sans utilisation de *credentials* (analyse boîte noire). La machine analysée possède la vulnérabilité CVE-2006-3439 (patch MS06-040 chez Microsoft) qui est une vulnérabilité permettant un dépassement de buffer.

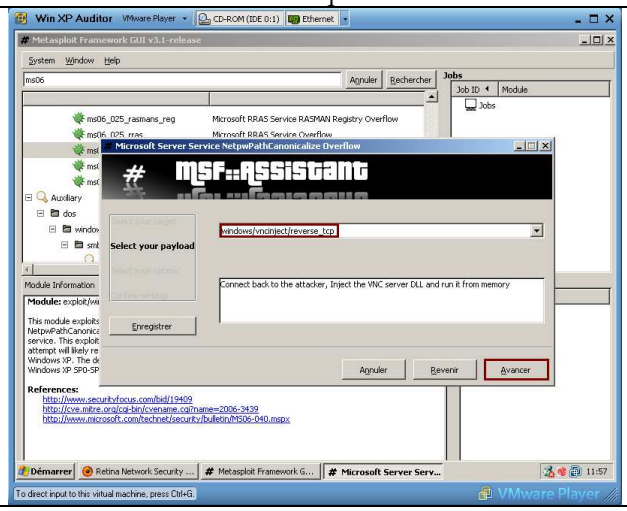


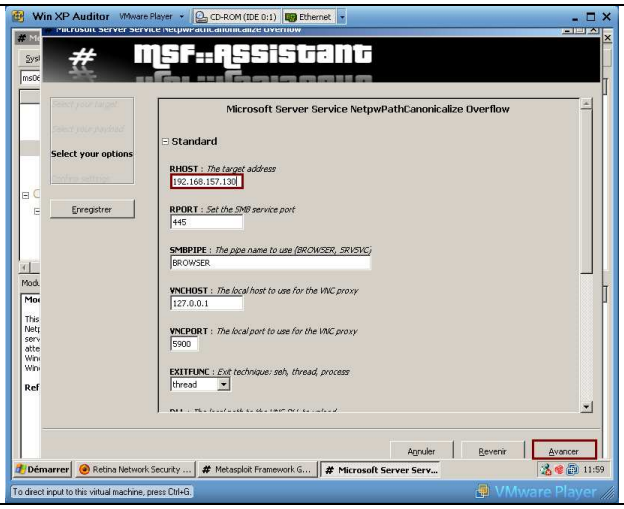
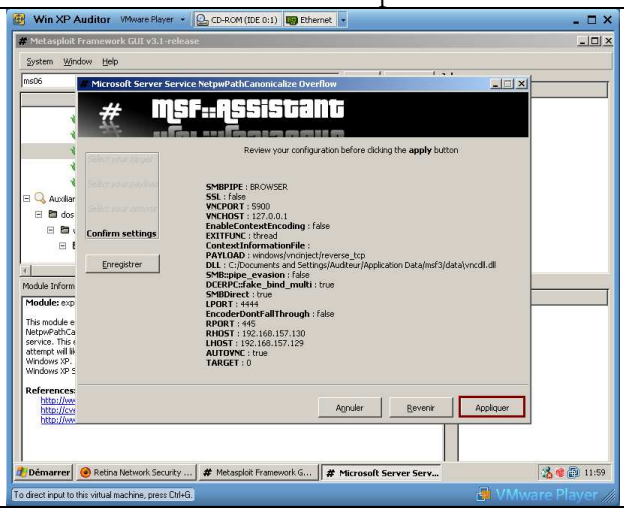
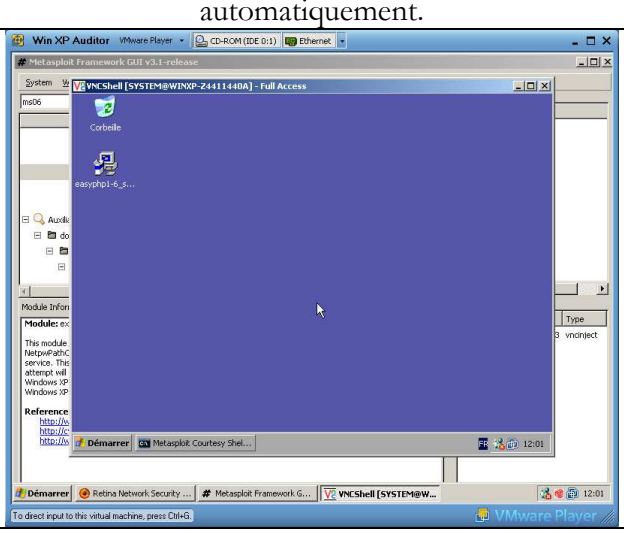


Le Metasploit Framework (interface graphique) est lancé et on recherche la vulnérabilité MS06-040.

Elle existe dans le Metasploit Framework ! Un clic droit dessus puis `Exécuter` permet de paramétrer l'attaque.





La première chose est de choisir la cible. XP SP0 figure dans la liste.	La deuxième étape est de choisir un shellcode ou payload, c'est-à-dire le code pirate à injecter. Nous choisissons VNC Reverse TCP qui permet d'avoir une interface graphique d'administration à distance sur la machine à compromettre.
	
Il suffit de renseigner l'adresse IP de la cible uniquement, tout le reste étant rempli automatiquement.	Un récapitulatif s'affiche et il reste à cliquer sur `Appliquer` pour lancer l'attaque.
	
Nous accédons à une interface graphique qui correspond au bureau Windows du système compromis : le succès est total.	

Nous avons montré dans cette sous-section avec quelle aisance il est possible d'accéder à un système distant si celui-ci est vulnérable à une attaque par corruption de mémoire.

3.3. Parades et protections

3.3.1. Correction des bugs et failles détectées par scanners

Que cela soit le ver Code Red ou le ver SQL Slammer (cf. Histoire des attaques par corruption de mémoire), les vulnérabilités avaient été détectées plusieurs jours ou mois auparavant. Des correctifs avaient même été diffusés par Microsoft. Pourtant de nombreux systèmes ont été compromis car ils n'étaient pas à jour en termes de correctifs (patchs).

C'est pourquoi, la règle suivante est fondamentale :

Les ressources d'une entreprise doivent être mises à jour aussi souvent que possible.

Pour cela il existe plusieurs possibilités :

- Paramétrage de mises à jour automatiques, comme les mises à jour automatiques de Windows ;
- Veille technologique sur les logiciels utilisés (serveurs, systèmes d'exploitation, etc.) afin de détecter quand une nouvelle version d'un produit est disponible ;
- Audit et déploiement automatique de patchs sur un réseau.

Pour le déploiement automatique de patchs nous pouvons citer :

- LANguard qui permet d'auditer les patchs manquants et de déployer les correctifs Microsoft sur les machines Windows d'un réseau local ;
- Microsoft WSUS (Windows Server Update Services) ou Microsoft SMS (Systems Management Server) pour l'installation de correctifs sous Windows ;
- De nombreux autres éditeurs fournissant des solutions de gestion de correctifs, une recherche sur *patch management* dans Google affichant de nombreux résultats.

Un système à jour ne présentant pas de faille connue est bien moins vulnérable.

3.3.2. Filtrage des connexions par firewalls

Dans la sous-section Exemple d'exploitation d'une machine nous avons utilisé un code pirate (shellcode) intitulé VNC Reverse TCP. Nous aurions pu utiliser un VNC TCP car notre réseau de test ne comportait pas de firewall. La remarque est la même pour un *Bind Shell TCP* (invite de commande à distance) et un *Bind Shell Reverse TCP* (qualifié aussi de *Connect-Back Shell*).

La différence est la suivante :

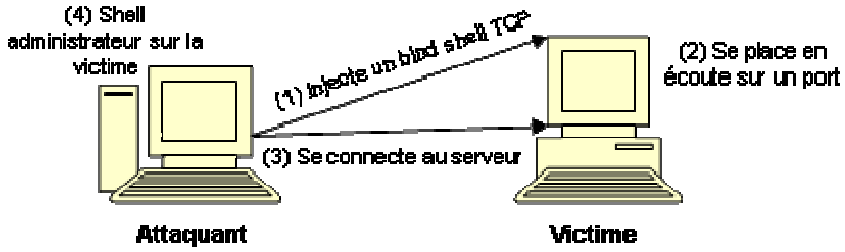


Figure 17 - Fonctionnement d'un Bind Shell TCP

Dans un *Bind Shell TCP* la victime fait office de serveur et l'attaquant s'y connecte. La connexion de la victime est donc entrante.

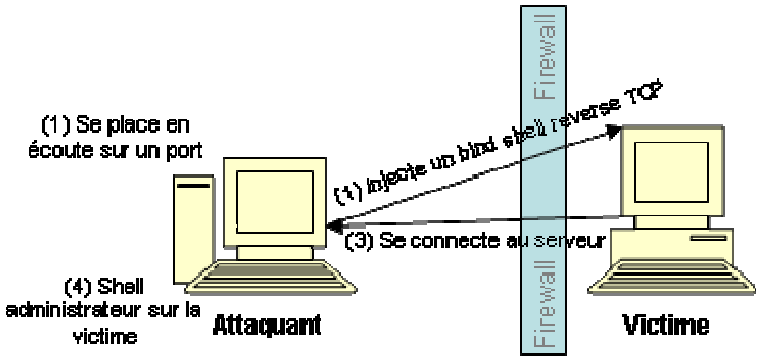


Figure 18 - Fonctionnement d'un Bind Shell Reverse TCP

Dans un *Bind Shell Reverse TCP* l'attaquant fait office de serveur et la victime s'y connecte. La connexion de la victime est donc sortante.

La règle d'or est donc : toute connexion entrante doit être interdite. Ceci interdit l'utilisation des shellcode qui ne sont pas *connect-back* ou *reverse*. De plus il faut restreindre autant que possible le nombre de ports ouverts en connexion sortante afin de limiter l'utilisation de shellcodes *connect-back* ou *reverse*.

3.3.3. Utilisation d'un IDS

Un IDS (*Intrusion Detection System*) peut être utilisé pour détecter des tentatives d'attaques par corruption de mémoire. Pour rappel, dans l'étape d'exploitation (cf. Mise en œuvre d'une attaque par corruption de mémoire), une attaque par corruption de mémoire passe par l'envoie d'un shellcode sur le réseau pour la compromission d'une machine distante. Le but est de détecter ce shellcode grâce à un IDS.

Les règles que nous énonçons ci-dessous peuvent être disponibles sous forme de signature pour un IDS ou entrées manuellement dans l'IDS par une personne compétente.

Il est possible de fixer une règle détectant chaque shellcode disponible dans le Metasploit Framework (117 shellcode pour la version 3.1), c'est-à-dire la grande majorité des shellcodes utilisés par les non-connaisseurs voulant compromettre une ressource rapidement.

Si un shellcode est légèrement modifié, les règles précédentes ne fonctionnent pas. Il est donc nécessaire de détecter des portions de shellcodes :

- Appel de la fonction système `system` ou `execve`,
- Utilisation de la chaîne `/bin`,
- Détournement de `stdout` (sortie standard) ou `stdin` (entrée standard) sur un socket (réseau).

Concernant les attaques sur chaînes de formatage il convient de détecter ces chaînes de formatage : une utilisation normale d'un programme serveur a très peu de chances de voir transiter des chaînes de formatage (contenant des séquences comme `%x` ou `%n`).

3.3.4. Maintien à jour des antivirus

Au même titre que les règles d'un IDS peuvent permettre de détecter des shellcodes, un antivirus à jour peut également détecter un shellcode grâce à ses signatures.

Par exemple, dans la démonstration précédente (cf. Exemple d'exploitation d'une machine), si Avast Home Edition avait été installé sur la victime et à jour, le shellcode aurait été intercepté et l'attaque serait devenue impossible.

Ci-dessous, on peut voir ce qui se passe lors de l'exécution du shellcode le plus basique, shellcode qui ouvre une invite de commande administrateur en local :

Figure 19 - Alerte d'Avast sur détection d'un shellcode

Encore une fois, la détection de shellcodes personnalisés est plus difficile et non systématique.

3.3.5. Protections diverses au niveau système

Remplacement de la bibliothèque standard : LibSafe

LibSafe est une solution pour Linux qui intercepte certains appels à des fonctions potentiellement vulnérables de la bibliothèque `libc` et utilise sa propre implémentation sécurisée à la place. LibSafe utilise notamment des versions sécurisées de `strcpy`, `strcat`, `printf`, etc. LibSafe protège donc à la fois des débordements de pile et des attaques sur chaînes de formatage.

L'utilisation de LibSafe ne nécessite aucune recompilation d'un quelconque programme. Il suffit simplement de l'installer et de l'activer. Cependant :

- LibSafe ne fonctionne que sur les exécutables faisant un appel dynamique à la `libc`, c'est-à-dire les exécutables qui ne sont pas compilés en statique.
- Les programmes compilés de façon à ne pas utiliser le pointeur de base `ebp` (`gcc -fomit-frame-pointer`) ne fonctionnent pas avec LibSafe car LibSafe se sert de `ebp` pour calculer la taille d'un buffer avant d'y copier quelque chose dedans.
- LibSafe n'intercepte que les débordements de pile et les attaques sur chaîne de formatage, un débordement de tas n'est pas intercepté, de même qu'un écrasement de pointeur.

Installer LibSafe sur un serveur critique est un acte à double tranchant : cela peut occasionner des problèmes sur un logiciel précis mais peut en contrepartie protéger d'attaques par corruption de mémoire encore non connues à ce jour.

Pages non exécutables

Certains micro-processeurs ont une fonctionnalité appelée bit NX (*No eXecute*) ou XD (*eXecute Disabled*). Quand cette fonctionnalité est utilisée conjointement avec un système d'exploitation qui la supporte il est

possible d'empêcher l'exécution de code sur la pile ou sur le tas, faisant échouer toute tentative d'exécution d'un shellcode injecté sur la pile.

Ces systèmes d'exploitations marquent des pages mémoires comme non-exécutables grâce au bit NX ou XD, empêchant ainsi le micro-processeur qui le supporte d'exécuter du code dans ces pages mémoires.

Cette fonctionnalité s'appelle DEP (*Data Execution Prevention*) dans Windows XP et supérieur. Pour Linux, le noyau actuel supporte cette fonctionnalité. Il existe certains systèmes ou certains patchs permettant d'émuler les pages non exécutables pour les micro-processeurs qui n'ont pas la fonctionnalité bit NX/XD ? Nous pouvons citer PaX, Exec Shield, Openwall.

Ce type de protection ne protège pas des attaques *return-to-libc* (cf. Return-to-libc) qui peuvent permettre de désactiver cette fonctionnalité avant de mener une attaque plus importante avec un shellcode puissant (par exemple un shellcode VNC Reverse TCP).

Randomisation de l'espace d'adressage

A partir du noyau Linux 2.6.12 et dans Windows Vista ou Server 2008, l'espace d'adressage d'un programme est aléatoire. Avant, lorsqu'un programme s'exécutait il se voyait souvent affecté le même espace mémoire. Un buffer précis se retrouvait donc au même endroit, c'est-à-dire à la même adresse en mémoire, sur des machines différentes pourvus que celles-ci exécutent le même système d'exploitation. Ainsi savait-on précisément l'emplacement en mémoire d'un shellcode injecté.

Avec un espace d'adressage aléatoire cela est plus difficile car les cases mémoires allouées à un programme sont différentes à chaque exécution. Par conséquent il est impossible de savoir à l'avance l'adresse d'un buffer et donc d'un shellcode injecté.

De plus, tout est alloué de manière aléatoire : pile, tas, bibliothèques dynamiques… Les attaques par corruption de mémoire sont donc bien plus difficiles, quelles qu'elles soient.

Dans Linux, pour un noyau >= 2.6.12, il est possible d'activer ou de désactiver l'ASLR (*Address Space Layout Randomization*). Pour l'activer :

```
echo 1 > /proc/sys/kernel/randomize_va_space
```

Pour le désactiver :

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

4. Conclusion

Nous avons vu que le site SecurityFocus recense les failles dans les logiciels diffusés librement et dans les logiciels commerciaux. Il ne se passe pas une semaine sans qu'une faille de type *buffer overflow* apparaisse et ne soit répertoriée sur ce site. Les failles de ce type permettent ensuite de réaliser des attaques par corruption de mémoire, attaques qui peuvent laisser un attaquant prendre le contrôle total d'une ressource distante. Deux types d'acteurs peuvent intervenir afin de régler ces problèmes :

- Les développeurs et testeurs qui, dans le cadre du processus de développement d'un logiciel, mettent en œuvre des techniques permettant de détecter des failles dans le code source.
- Les administrateurs et auditeurs réseau qui, dans le cadre d'un audit ou contrôle des ressources du réseau de l'entreprise, mettent en œuvre des techniques permettant de détecter les machines et logiciels vulnérables à des attaques par corruption de mémoire.

Les développeurs et testeurs doivent appliquer l'audit de code avec :

- L'analyse statique de code grâce à des logiciels d'analyse statique qui scannent et détectent des vulnérabilités potentielles dans le code source, de façon automatique ;
- La relecture du code afin de vérifier que les procédures et conventions de codage sont respectées et que le code source est exempt de failles qui seraient passées outre les mailles du filet lors de l'analyse statique et automatique.

Afin de se prémunir de possibles attaques par corruption de mémoire, les développeurs et testeurs peuvent corriger les failles identifiées pendant l'audit de code mais aussi :

- Faire le choix d'un langage de haut-niveau comme Java où la gestion de la mémoire est automatisée et donc plus sécurisée et moins sujette à des attaques de type *buffer overflow* ;
- Utiliser certaines options de compilation (mise en place d'un canarie) pour imposer au logiciel compilé et exécuté d'effectuer des vérifications supplémentaires afin qu'il détecte lui-même s'il est train de subir une attaque par corruption de mémoire.

Les administrateurs et auditeurs réseau doivent appliquer l'audit de vulnérabilités avec :

- L'analyse et la détection de failles sur les ressources du réseau grâce à l'utilisation de scanners de vulnérabilités (Nessus, Retina, LANguard, etc.) ;
- La veille technologique sur un site comme SecurityFocus afin de contrôler régulièrement si une faille a été découverte sur un logiciel important et utilisé sur le réseau.

Afin de se prémunir de possible attaques par corruption de mémoire, les administrateurs et auditeurs réseau peuvent patcher les ressources vulnérables identifiées pendant l'audit de vulnérabilité mais aussi :

- Filtrer par firewall les connexions entrantes et sortantes afin d'empêcher au maximum les codes pirates de se connecter à la machine d'un attaquant ou à un attaquant de se connecter à une machine vulnérable sur le réseau ;
- Utiliser un IDS (Intrusion Detection System) pour détecter des portions de codes pirates transitant sur le réseau et ainsi voir leur origine et destination ;
- Maintenir à jour les antivirus qui sont capables, au même titre que les IDS, de détecter des portions de codes pirates et empêcher leur exécution.
- Utiliser des patchs ou des systèmes supportant des fonctions permettant d'empêcher ou de limiter l'exécution de code pirates :
    - Bibliothèque LibSafe pour Linux redéfinissant des fonctions C vulnérables ;
    - Pages non exécutables (DEP depuis Windows XP SP2) ;
    - Randomisation de l'espace d'adressage (Linux Kernel >= 2.6.12, Windows Vista).

5. Annexe

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <string.h>
4
5   // Dépassement de pile simple
6   void fonction1(char * chaine) {
7      char buffer[128];
8      strcpy(buffer, chaine); /* Vulnerabilite */
9   }
10
11  // Dépassement un-octet
12  void fonction2(char * chaine) {
13     char buffer[128]; int i;
14     for (i = 0 ; i <= 128 && i < strlen(chaine); i++)  /* Vulnerabilite */
15        buffer[i] = chaine[i];
16  }
17
18  // Dépassement de type
19  void fonction3(char * chaine, short n) {
20     char buffer[128]; int longueur_max = 128;
21     if (n < longueur_max) /* Vulnerabilite */
22        strcpy(buffer, chaine);  /* Vulnerabilite */
23  }
24
25  // Chaîne de formatage
26  void fonction4(char * chaine) {
27     printf(chaine);  /* Vulnerabilite */
28  }
29
30  // Ecrasement de pointeur de fonction
31  void fonction5() {
32     printf("void fonction5()\n");
33  }
34
35  // Ecrasement de pointeur (1)
36  typedef struct _structure {
37     char buffer[128];
38     int * p;
39     int a;
40  } structure;
41
42  // Ecrasement de pointeur (2)
43  void fonction6(char * chaine) {
44     structure * x;
45     x = (structure *) malloc(sizeof(structure));
46     x->a = 0;
47     x->p = &(x->a);
48     strcpy(x->buffer, chaine);  /* Vulnerabilite */
49     *(x->p) = x->a;
50     printf("x->a = %08x\n", x->a);
51     free(x);
52  }
53
54  // Dépassement de section .data (1)
55  char global_buffer[128] = {1};
56
57  // Dépassement de section .data (2)
58  void fonction7(char * chaine) {
59     strcpy(global_buffer, chaine);  /* Vulnerabilite */
60  }
61
62  int main (int argc, char * argv[]) {
63     static char buffer[128];
64     static void (*ptrfct)() = fonction5;
65     int choix;
66
67     if (argc != 3) {
68        printf("Usage: %s <choix> <parametre>", argv[0]);
69     } else {
70        choix = atoi(argv[1]);  /* Vulnerabilite */
71
72        switch (choix) {
73        case 1: // Dépassement de pile simple
74           fonction1(argv[2]);
```

```
75           printf("Dépassement de pile simple");
76           break;
77        case 2: // Dépassement un-octet
78           fonction2(argv[2]);
79           printf("Dépassement un-octet");
80           break;
81        case 3: // Dépassement de type
82           fonction3(argv[2], strlen(argv[2]));
83           printf("Dépassement de type");
84           break;
85        case 4: // Chaîne de formatage
86           fonction4(argv[2]);
87           printf("\nChaîne de formatage");
88           break;
89        case 5: // Ecrasement de pointeur de fonction
90           strcpy(buffer, argv[2]);  /* Vulnerabilite */
91           (void) (*ptrfct)();
92           printf("Ecrasement de pointeur de fonction");
93           break;
94        case 6: // Ecrasement de pointeur
95           fonction6(argv[2]);
96           printf("Ecrasement de pointeur");
97           break;
98        case 7: // Dépassement de section .data
99           fonction7(argv[2]);
100          printf("Dépassement de section .data");
101          break;
102       }
103    }
104
105    printf("\n");
106
107    return 0;
108 }
```