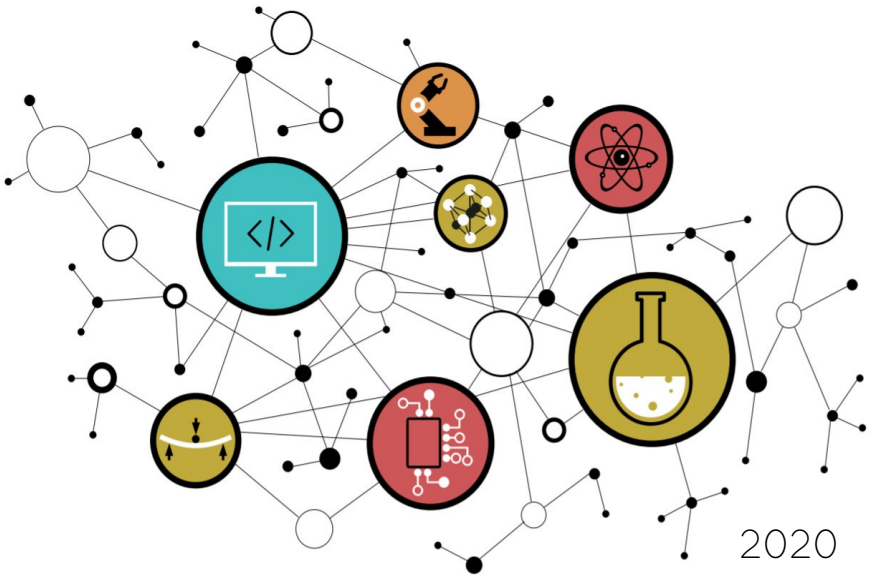
TRAVAUX PRATIQUES

HACKING



2020

SOMMAIRE

8. HACKING

8.1. Exécution d'un shellcode sur la pile
8.2. Extraction et édition d'un shellcode
8.3. Édition d'un exploit
8.4. White Hat

8. HACKING

Le terme *hacker* a été maintenant depuis bien des années détourné par les médias pour faire référence à la notion de pirate informatique. Cependant, à la racine, derrière ce mot se cache des utilisateurs développeurs fouineurs poussés par la passion, le jeu, le plaisir, l'échange et le partage. Des passionnés devenus experts dans la compréhension et la maîtrise des systèmes numériques d'information. Depuis la naissance d'internet, dans les années 90, le concept de cybercriminalité a émergé. 2 familles de hackers apparaissent, les *black hat* , exerçant des activités qualifiées de criminelles, et les *white hat* d'une vision éthique, cherchant à aider à la sécurité des systèmes d'informations.

Un code éthique du hacker a d'ailleurs été formalisé au MIT et publié dans un livre de Steven Levy en 1984 (*Hackers : Heros of the Computer Revolution*). Il s'agit d'un ensemble de concepts et de visions provenant de la relation symbiotique entre hackers et machines. Celui-ci comprend 6 règles :

- L'accès aux ordinateurs - ainsi que tout ce qui peut permettre de comprendre comment le monde fonctionne - doit être universel et sans limitations. Il ne faut pas hésiter à se retrousser les manches pour surmonter les difficultés.
- Toute information doit être libre.
- Se méfier de l'autorité - encourager la décentralisation.
- Les hackers doivent être jugés selon leurs *hacks*, et non selon de faux critères comme les diplômes, l'âge, l'origine ethnique ou le rang social.
- On peut créer l'art et le beau à l'aide d'un ordinateur.
- Les ordinateurs peuvent améliorer notre vie.

Durant cet exercice, nous allons humblement chercher à entrouvrir le domaine du *hacking* et de l'expertise système. Nous allons explorer des solutions simples d'injection de code sur la pile par *shellcode*. Nous vous invitons bien entendu à explorer le sujet, en regardant par exemple des solutions plus avancées comme la ROP chain (Return-Oriented Programming) et bien d'autres. Vous trouverez d'ailleurs de nombreuses exploitations de vulnérabilités sur internet, notamment sur des applications ayant des failles connues publiées sur le site CVE (Common vulnerabilities and Exposures, https://cve.mitre.org/index.html).

8.1. Exécution d'un shellcode sur la pile

Se placer dans le répertoire */disco/hack/* afin d'appliquer la totalité des commandes qui suivent. Nous allons chercher à invoquer une console système d'interface avec le noyau (*shell*) en injectant puis en exécutant du code depuis la pile d'un applicatif. Le programme standard permettant d'invoquer un *shell* (*/bin/sh*) utilise la fonction *execve* et réalise un appel système au noyau Linux (instruction *syscall* en 64bits et *int 0x80* en 32bits). Observons quelques informations concernant cette fonction et son implémentation en *user space* (distribution GNU\Linux Debian-like comme Ubuntu) et *kernel space* (Linux) :

- Manuel Ubuntu pour la fonction *execve* (user space) :

https://manpages.ubuntu.com/manpages/precise/fr/man2/execve.2.html

- Source code officiel Linux, fichier */fs/execve.c* (kernel space) :

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/exec.c?h=v5.4-rc7

- Vulnérabilités connues sur le fichier du noyau Linux */fs/execve.c* :

https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=fs%2Fexec.c

- Ouvrir puis compiler le fichier *shellcode.c*. Exécuter le programme.

```
gcc shellcode.c -o bin/shellcode

./bin/shellcode
```

- Que constatons-nous ?

- Observer l'en-tête de programme du fichier ELF exécutable de sortie, et préciser les droits (rwx ou read write executable) sur le futur segment de pile qui sera associé à notre programme à l'exécution ?

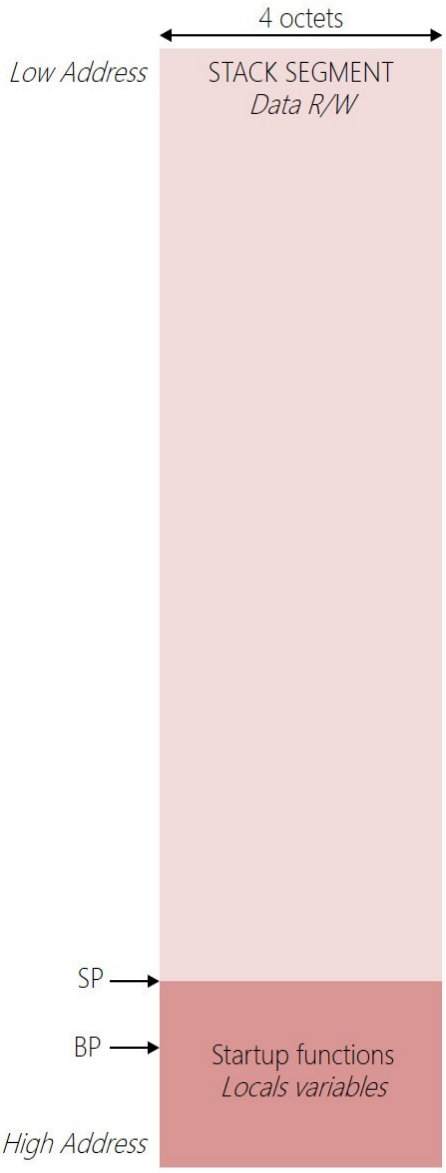
```
objdump -fp bin/shellcode
```

- Durant l'édition des liens, le *linker* a la capacité de préparer les droits associés aux futurs segments mémoire de l'application, notamment le segment de pile. Ce travail est réalisé à la construction du fichier ELF (en-tête du programme). Rendre la pile exécutable, vérifier les droits sur le segment de pile et exécuter le programme. Que constatons-nous ? *Entrer exit pour quitter le shell*

```
gcc -fno-stack-protector -z execstack shellcode.c -o bin/shellcode

./bin/shellcode

objdump -fp bin/shellcode
```

- Compiler et analyser le programme assembleur. Analyser ensuite l'exécution du programme en ouvrant une session de *debug* avec *gdb*. Décrire le fonctionnement du programme en proposant un schéma commenté ! .

```
gcc -S -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-protection=none -z execstack shellcode.c -o misc/shellcode.s

gcc -g -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-protection=none -z execstack shellcode.c -o bin/shellcode

gdb ./bin/shellcode
(gdb) la a
(gdb) b main
(gdb) r
(gdb) s
(gdb) ni
...etc
(gdb) ni
$
$ exit
(gdb) q
```



- Après analyse du script assembleur, représenter ci-contre le contenu de la pile avant exécution de l'instruction *call* *%rdx implémentant *return(0)* de la fonction main. S'aider des outils et des commandes suivantes.

```
gcc -c -fno-stack-protector -fno-asynchronous-unwind-tables -fno-pie -fcf-protection=none -z execstack shellcode.c -o obj/shellcode.o

objdump -S ./obj/shellcode.o

objdump -s ./obj/shellcode.o
```

8.2. Extraction et édition d'un shellcode

```
int main(void)
{
  char *argv[] = { "/bin/sh" , NULL};

  execve(argv[0], argv, NULL);

  exit(EXIT_FAILURE);
}
```

Nous allons maintenant nous intéresser à une méthodologie afin d'identifier et d'éditer un *shellcode*. Pour cela, nous allons partir de la solution générée par défaut par *gcc* à partir d'un appel standard d'un programme depuis un applicatif (cf. ci-dessus, appel de */bin/sh*). La compilation et l'édition des liens se feront en statique (option -*static*) de façon à pouvoir observer l'implémentation binaire et assembleur (après désassemblage) de l'appel de la fonction système *execve*.

- Ouvrir puis compiler le fichier *shell.c*. Exécuter le programme.

```
gcc -g -fno-stack-protector -fno-asynchronous-unwind-tables -fno-
pie -fcf-protection=none -static shell.c -o bin/shell

./bin/shell
```

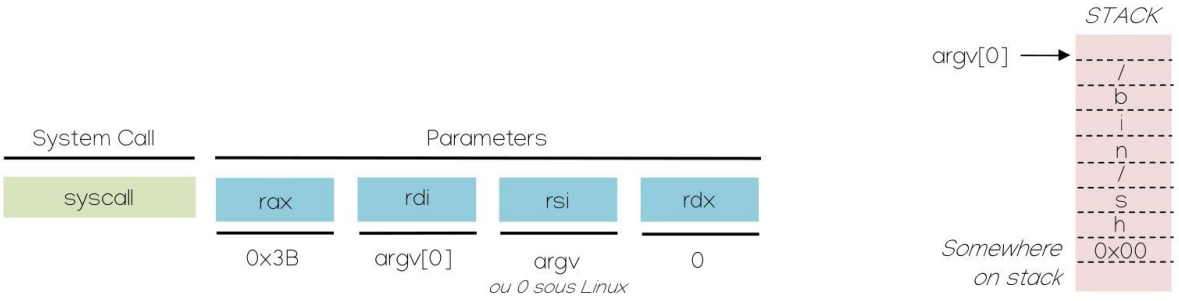
- Que constatons-nous ?

- Analyser le programme assembleur ([CTRL] + [f] puis rechercher *execve*) et comprendre l'implémentation réalisée par *gcc*, notamment les passages de paramètres par registres. Nous allons nous efforcer d'en reproduire une version allégée en assembleur puis en binaire.

```
objdump -S bin/shell > misc/shell_disassembly.md

gdb ./bin/shell
(gdb) la a
(gdb) b main
(gdb) r
(gdb) b execve
(gdb) c
(gdb) ni
...etc
(gdb) ni
$
$ exit
(gdb) q
```

- Que réalise l'instruction *syscall* ?

- Préciser ci-dessous les 4 registres utilisés par le compilateur pour passer des arguments à l'appel système *syscall* ?



Après analyse, nous pouvons en déduire les registres utilisés et les valeurs des arguments à passer avant l'appel système. Pour information, seulement sous Linux, le troisième argument passé peut être nul. Attention, cette implémentation ne respecte pas le standard Unix. Nous allons faire ceci afin d'alléger l'implémentation de notre *shellcode* (instruction XOR pour une mise à zéro rapide de registre et une empreinte minimale du code). Nous allons chercher à réaliser l'implémentation minimale proposée graphiquement ci-dessous.

- Observer le contenu du fichier *shell_asm.s* afin d'implémenter la solution illustrée ci-dessus. Valider l'assemblage et l'exécution du programme

```
as shell_asm.s -o obj/shell_asm.o

ld obj/shell_asm.o -o bin/shell_asm

./bin/shell_asm
```

- Identifier et extraire le shellcode en observant l'implémentation binaire du programme. Analyser à nouveau le *shellcode* binaire du fichier *shellcode.c*. Il est possible d'en trouver une grande quantité déjà édités sur internet (http://shell-storm.org/shellcode/). Écrire le *shellcode* identifié ci-dessous (24 octets)

```
objdump -S obj/shell_asm.o
```

8.3. Édition d'un exploit

Nous allons dans cette dernière partie donner quelques briques et techniques permettant l'ouverture au développement d'un *exploit* (exploitation d'une vulnérabilité). Une technique classique consiste à remplacer l'adresse de retour d'une fonction par celle d'un code malveillant (*shellcode*) caché sur le pile. Rappelons que l'adresse de retour d'une fonction est sauvée par défaut sur la pile durant l'appel de la fonction (*call*). Ainsi, lorsque la fonction courante souhaitera se terminer en exécutant l'instruction *ret* , qui dépile l'adresse de retour de la fonction appelante depuis la pile pour la placer dans le registre CPU d'instruction *rip* , la fonction la remplacera par l'adresse du code malveillant à exécuter. Le tableau ci-dessous présente une écriture en pseudo-code des instructions *call* et *ret*.

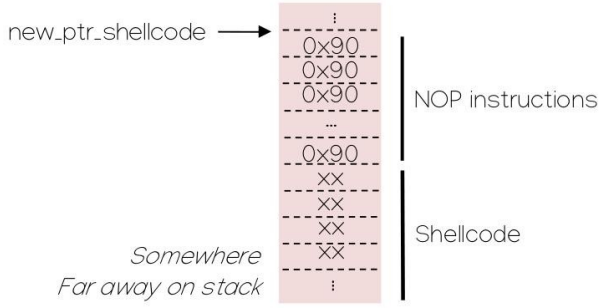
Instruction	CALL *function_address*	RET	
Pseudo-code	RSP ← RSP – 8 *(RSP) ← RIP RIP ← *function_address*	*tmp* ← *(RSP) RSP ← RSP + 8 RIP ← *tmp*	*<--- shellcode address here !* *<--- goto shellcode !*

Nous allons développer ces deux actions sur un programme simple, soit changer l'adresse de retour d'une fonction puis déplacer un *shellcode* sur la pile. Ces techniques permettent d'ouvrir à une infinité d'autres *exploit* possibles. Mais encore faut-il trouver des failles, s'assurer que les bonnes conditions soient réunies et enfin réussir à l'exploiter !

- Modifier le premier commentaire *TODO* dans le fichier *disco/hack/exploit.c* par une ligne de code permettant de remplacer l'adresse de retour de la fonction appelante par l'adresse du *shellcode* sur la pile. Valider la bonne exécution du programme. Ne pas hésiter à activer la macro PRINT_DEBUG durant vos tests, mais penser à la désactiver afin de simplifier l'analyse de l'exploit et de valider son bon fonctionnement.

```
gcc -fno-stack-protector -z execstack exploit.c -o bin/exploit

./bin/exploit

objdump -S bin/exploit
```

- Modifier le second commentaire *TODO* afin de déplacer le *shellcode* sur la pile. Nous placerons des instructions NOP (No Operation, opcode binaire 0x90) avant le *shellcode*. Cette technique est couramment utilisée afin de faciliter la recherche d'un *shellcode* sur la pile par un *exploit*. Notamment lorsque l'adresse exacte du code malveillant n'est pas précisément connue. Valider la bonne exécution du programme.



- Les solutions sont cachées dans le répertoire *hack/misc* !

8.4. White Hat

L'exercice de travaux pratiques s'arrête ici. J'offre avec plaisir un café et la possibilité d'insérer son code dans les ressources de TP à tout étudiante ou étudiant réussissant à développer un *exploit* permettant d'ouvrir une console *root* sur sa machine personnelle (sans avoir à user des droits *root*). Voici sinon les livrables attendus :

- Code source de l'exploit avec Makefile propre à la racine
- Conditions et procédure de test et validation (fichier texte README.md)
- Démonstration sur machine personnelle (contre café à mes frais)
- Contre-mesure et solution (montée de version, patch, etc) pour bloquer cette faille avec procédure de déploiement sur le système

Notamment depuis l'arrivée d'internet, les systèmes numériques d'information se sont grandement complexifiés, mais également durcis. Bien des vulnérabilités ont déjà été explorées et exploitées, et les solutions déjà déployées. Mais comme une recherche inextinguible de trésor, bien des failles restent encore à trouver. Que ce soit au niveau matériel (NX bit ou No eXecute bit dans la table de translation d'adresses utilisée par la MMU,), ou au niveau kernel Linux avec par exemple une gestion aléatoire des mapping mémoire de certains segments applicatifs (pile, tas, bibliothèques partagées et vDSO), plusieurs contre-mesures sont déjà à l'œuvre afin de contraindre le travail des *hackers* dans leur volonté de pénétrer les systèmes.

Prenons l'exemple de la fonction *execve*. comme précisé dans la documentation officielle, cette fonction travaille par remplacement de segments mémoire de la fonction appelante (.text, .bss, .data et pile). De même, il existe un jeu de conditions possibles et documentées permettant à l'appelant d'hériter des privilèges d'exécution de l'appelé. Je vous laisse donc mûrir et entrouvrir le spectre du possible, si l'applicatif appelé et le système de fichiers sous-jacent n'a pas été pensé contre !

Voilà, et sinon, un peu de musique avec votre café avant d'attaquer le *hack* ...

https://www.youtube.com/watch?v=_AdBdCz3tVs


notes