

CyberChallenge 2020 - Lezione 7

PWN



Samuele Perticarari
samuele.perticarari@gmail.com
s1084178@studenti.univpm.it

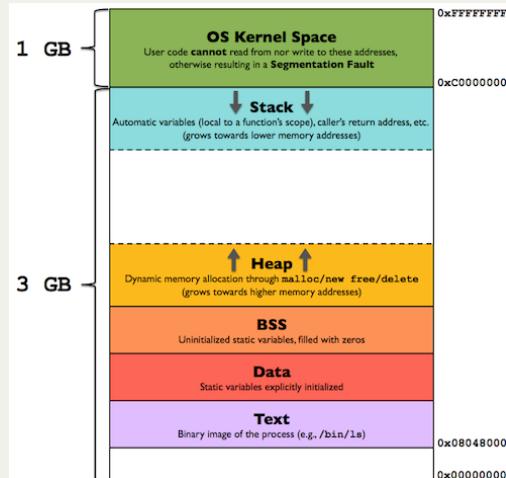


Riassunto della scorsa lezione

Caricamento di un eseguibile in memoria

Text corrisponde alla sezione *.text*

Data e *BSS* corrispondono alle sezioni *.data* e *.rodata*



[program_in_memory.png]

Stack: Contiene lo stack del programma. E' una pila, cioè una struttura LIFO (Last In, First Out)

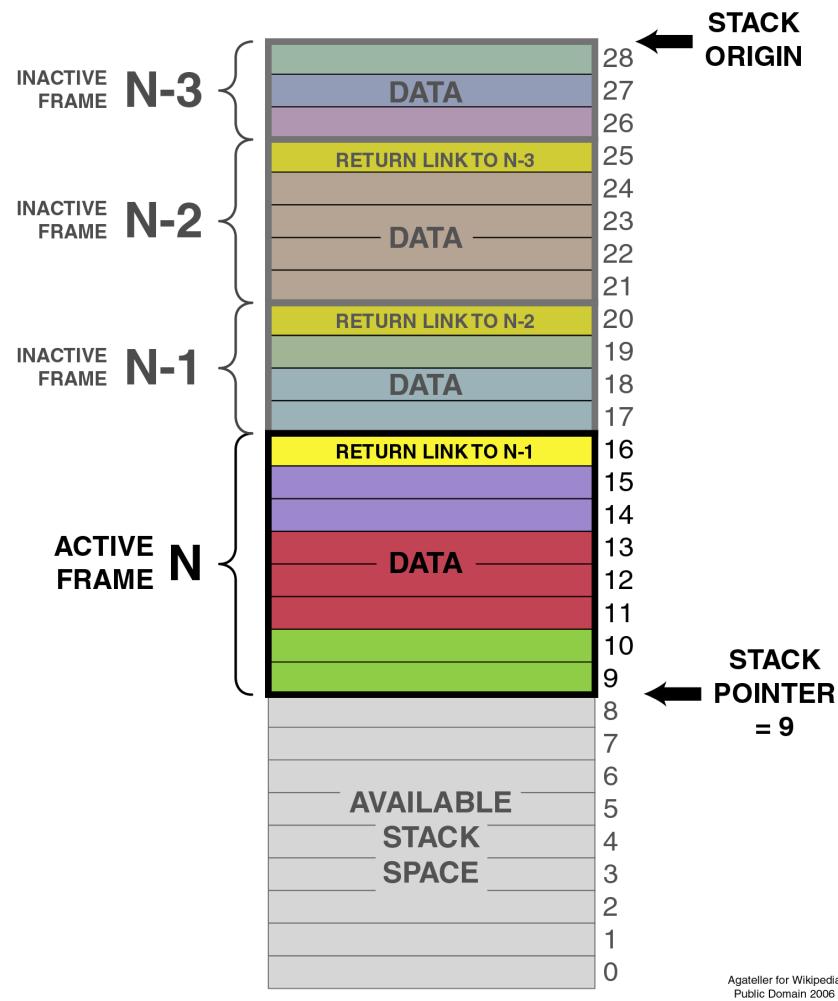
Heap: Contiene dei chunk (segmenti) che possono essere allocati con *malloc* (e *calloc*) e deallocati con *free*.

Nella heap si hanno tutti i blocchi di memoria che vengono allocati dinamicamente dal programma.

Caricamento di un eseguibile in memoria

```
gef> vmmmap
[ Legend:  Code | Heap | Stack ]
Start      End          Offset      Perm Path
0x08048000 0x08049000 0x00000000 r-x /tmp/hidden_flag_function
0x08049000 0x0804a000 0x00000000 r-- /tmp/hidden_flag_function
0x0804a000 0x0804b000 0x00001000 rw- /tmp/hidden_flag_function
0x0804b000 0x0806d000 0x00000000 rw- [heap]
0xf7dad000 0xf7dca000 0x00000000 r-- /usr/lib32/libc-2.31.so
0xf7dca000 0xf7f21000 0x0001d000 r-x /usr/lib32/libc-2.31.so
0xf7f21000 0xf7f90000 0x00174000 r-- /usr/lib32/libc-2.31.so
0xf7f90000 0xf7f92000 0x001e2000 r-- /usr/lib32/libc-2.31.so
0xf7f92000 0xf7f94000 0x001e4000 rw- /usr/lib32/libc-2.31.so
0xf7f94000 0xf7f98000 0x00000000 rw-
0xf7fdc000 0xf7fd0000 0x00000000 r-- [vvar]
0xf7fd0000 0xf7fd1000 0x00000000 r-x [vdso]
0xf7fd1000 0xf7fd2000 0x00000000 r-- /usr/lib32/ld-2.31.so
0xf7fd2000 0xf7ff0000 0x00001000 r-x /usr/lib32/ld-2.31.so
0xf7ff0000 0xf7ffb000 0x0001f000 r-- /usr/lib32/ld-2.31.so
0xf7ffc000 0xf7ffd000 0x0002a000 r-- /usr/lib32/ld-2.31.so
0xf7ffd000 0xf7ffe000 0x0002b000 rw- /usr/lib32/ld-2.31.so
0xffffdd000 0xfffffe000 0x00000000 rw- [stack]
gef>
```

Stack



La stack supporta operazioni di push e di pop.

Per allocare spazio nella stack, viene decrementato il puntatore ad essa.

Esempio:

Se devo allocare un intero e un array di 16 caratteri.

Il compilatore fa scorrere il puntatore alla stack più in basso.

```
; 4 byte per l'intero  
; 16 byte per l'array  
sub esp, 20
```

Va bene, ma in pratica come funziona il tutto?

Nel momento in cui si richiama una funzione:

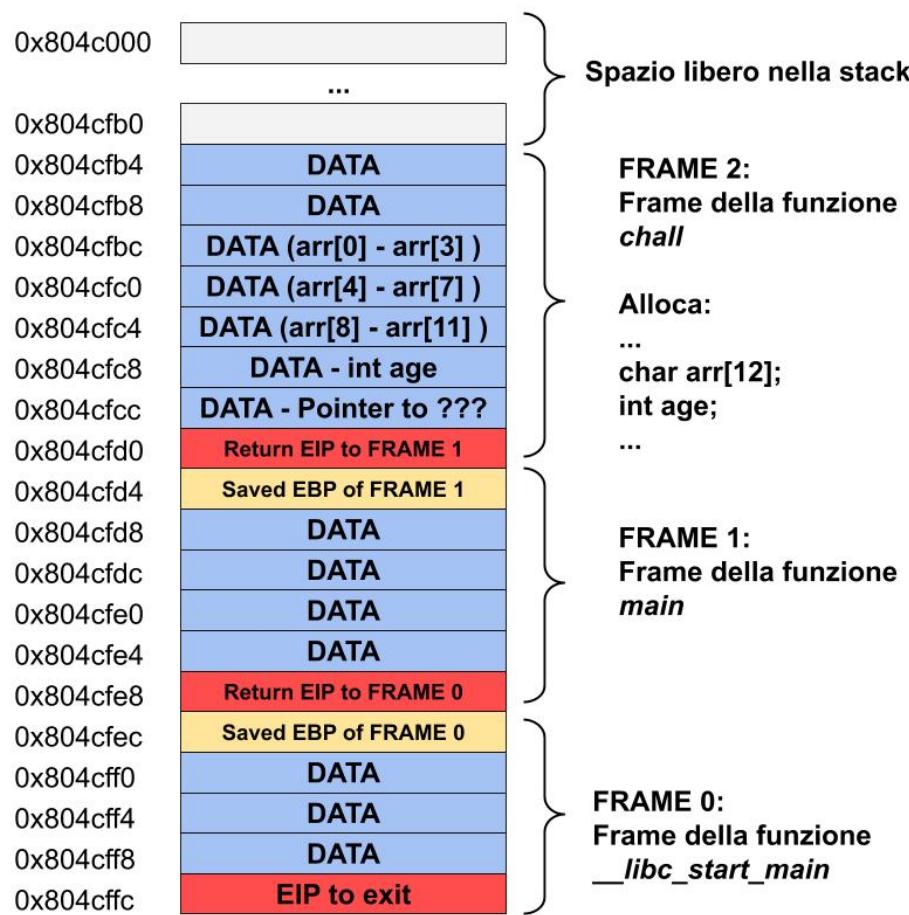
1. *Viene fatto il push dell' istruzione successiva alla chiamata della funzione.*
2. Viene fatto il push di EBP. (salvo il puntatore alla stack per il ritorno nella funzione padre)
3. Viene copiato ESP in EBP
4. Si esegue il codice della funzione

Quando finisce la funzione:

1. Viene messo in EAX il valore di ritorno della funzione
2. Viene ripristinato lo stato dei registri, in particolare viene ripristinato ESP. (ripristino il puntatore alla stack per il ritorno nella funzione padre)
3. *Viene fatto il pop di EIP con l'istruzione ret.*
4. Continua l'esecuzione nell'indirizzo in cui punta EIP

Stack call di un eseguibile

Stack allocata in memoria dall'indirizzo 0x804c000 al 0x804d000



Stack Overflow

```
int main(int argc, char** argv)
{
    chall();
    return;
}
```

```
void chall()
{
    char arr[12];
    int age;
    void* pointer;

    // ...

    scanf("%s", arr);

    // Codice sicuro:
    // scanf("%11s", arr);

    return;
}
```

Input: ABCDEFGHIJKLMNOPQRSTUVWXYZ+

0x804c000				
0x804cfb0	...			
0x804cfb4	DATA			
0x804cfb8	DATA			
0x804cfbc	DATA (arr[0] - arr[3])			
0x804fcf0	DATA (arr[4] - arr[7])			
0x804fcf4	DATA (arr[8] - arr[11])			
0x804fcf8	DATA - int age			
0x804fcfc	DATA - Pointer to ???			
0x804cf0	Return EIP to FRAME 1			
0x804cf04	Saved EBP of FRAME 1			
0x804cf08	DATA			
0x804cf0c	DATA			
0x804cf0e	DATA			
0x804cf04	DATA			
0x804cf08	Return EIP to FRAME 0			
0x804cf0c	Saved EBP of FRAME 0			
0x804cf0e	DATA			
0x804cf04	DATA			
0x804cf08	DATA			
0x804cf0c	DATA			
0x804cf0e	EIP to exit			
...	...			
...	...			



Intro PWN

Intro PWN

- *Address Space Layout Randomization (ASLR)*
- *Buffer overflow / Stack overflow*
- *Global Offset Table*
- *Stack canary*
- *Return Oriented Programming (ROP)*
- *Tecniche di attacco per BOF*
- *Format string exploit*

Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR)

L'ASLR è una misura di protezione contro buffer overflow e exploit, la quale consiste nel rendere (parzialmente) casuale l'indirizzo delle funzioni di libreria e delle più importanti aree di memoria.

Ciò limita l'attaccante, rendendo più difficile l'attacco.

Vedremo successivamente come bypassare questa protezione (quando possibile).

Address Space Layout Randomization (ASLR)

```
gef> checksec
[+] checksec for '/home/samuele/Scriva
Canary : ✗
NX      : ✓
PIE     : ✓
Fortify: ✗
RelRO   : Full
```

In questo caso l'ASLR è attivo in questo eseguibile.

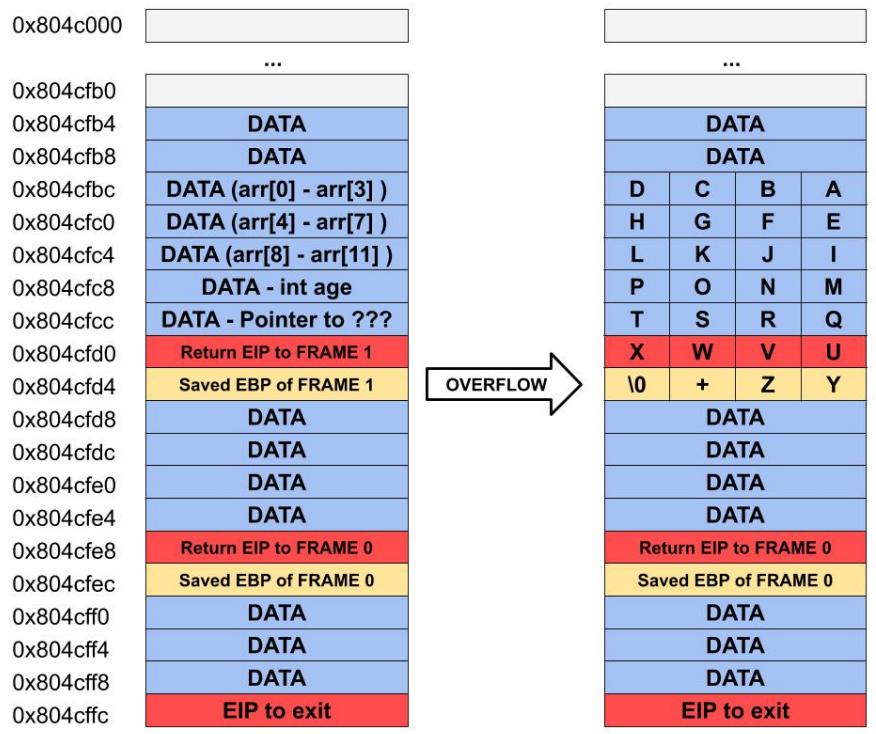
Buffer overflow

Buffer overflow

Il buffer overflow è una condizione di errore che si verifica a runtime quando in un buffer di una data dimensione vengono scritti dati di dimensioni maggiori.

Buffer overflow

Input: ABCDEFGHIJKLMNOPQRSTUVWXYZ+



Scrivendo nel buffer un numero maggiore di caratteri possiamo sovrascrivere il contenuto posteriore al buffer, potendo controllare il flusso del programma.

Buffer overflow

main:

```
....  
0x08048597 <+11>: mov    ebp,esp  
0x08048599 <+13>: push   ecx  
0x0804859a <+14>: sub    esp,0x4
```

```
0x0804859d <+17>: call   0x0804851f <setup>
```

setup:

```
0x0804851f <+0>: push   ebp  
0x08048520 <+1>: mov    ebp,esp  
.....  
0x0804854d <+46>: nop  
0x0804854e <+47>: leave  
0x0804854f <+48>: ret
```

```
→ 0x080485a2 <+22>: call   0x08048550 <chall>
```

chall:

```
0x08048550 <+0>: push   ebp  
0x08048551 <+1>: mov    ebp,esp  
.....  
0x08048589 <+57>: nop  
0x0804858a <+58>: leave  
0x0804858b <+59>: ret
```

```
→ 0x080485a7 <+27>: mov    eax,0x0  
0x080485ac <+32>: add    esp,0x4  
0x080485af <+35>: pop    ecx  
0x080485b0 <+36>: pop    ebp  
0x080485b1 <+37>: lea    esp,[ecx-0x4]  
0x080485b4 <+40>: ret
```

Buffer overflow

main:

```
....  
0x08048597 <+11>: mov    ebp,esp  
0x08048599 <+13>: push   ecx  
0x0804859a <+14>: sub    esp,0x4
```

```
0x0804859d <+17>: call   0x804851f <setup>
```

setup:

```
0x0804851f <+0>: push   ebp  
0x08048520 <+1>: mov    ebp,esp  
.....  
0x0804854d <+46>: nop  
0x0804854e <+47>: leave  
0x0804854f <+48>: ret
```

```
→ 0x080485a2 <+22>: call   0x8048550 <chall>
```

chall:

```
0x08048550 <+0>: push   ebp  
0x08048551 <+1>: mov    ebp,esp  
.....  
0x08048589 <+57>: nop  
0x0804858a <+58>: leave  
0x0804858b <+59>: ret
```

```
→ 0x080485a7 <+27>: mov    eax,0x0  
0x080485ac <+32>: add    esp,0x4  
0x080485af <+35>: pop    ecx  
0x080485b0 <+36>: pop    ebp  
0x080485b1 <+37>: lea    esp,[ecx-0x4]  
0x080485b4 <+40>: ret
```

get_shell:

```
0x08048506 <+0>: push   ebp  
0x08048507 <+1>: mov    ebp,esp  
.....  
0x08048514 <+14>: call   0x80483b0 <system@plt>  
.....
```

Buffer overflow

```
0x8048566 <chall+22>      sub    esp, 0xc
0x8048569 <chall+25>      lea    eax, [ebp-0x28]
0x804856c <chall+28>      push   eax
→ 0x804856d <chall+29>      call   0x80483a0 <gets@plt>
↳ 0x80483a0 <gets@plt+0>    jmp    DWORD PTR ds:0x804a010
0x80483a6 <gets@plt+6>      push   0x8
0x80483ab <gets@plt+11>     jmp    0x8048380
0x80483b0 <system@plt+0>   jmp    DWORD PTR ds:0x804a014
0x80483b6 <system@plt+6>   push   0x10
0x80483bb <system@plt+11>  jmp    0x8048380

gets@plt (
    [sp + 0x0] = 0xff890e40 → 0xf7ec6000 → 0x001d4d6c,
    [sp + 0x4] = 0xf7f05da0 →  pop edx
)

[#0] Id 1, Name: "vuln", stopped 0x804856d in chall (), reason: SINGLE STEP
[#0] 0x804856d → chall()
[#1] 0x80485a7 → main()
```

Buffer overflow

```
gef> bt
#0 0x0804856d in chall ()
#1 0x080485a7 in main ()

gef> telescope
0xff890e30 +0x0000: 0xff890e40 → 0xf7ec6000 → 0x001d4d6c ← $esp
0xff890e34 +0x0004: 0xf7f05da0 → pop edx
0xff890e38 +0x0008: 0xf7d58acb → <setvbuf+11> add edi, 0x16d535
0xff890e3c +0x000c: 0x00000000
0xff890e40 +0x0010: 0xf7ec6000 ← BUFFER
0xff890e44 +0x0014: 0x00000000
0xff890e48 +0x0018: 0xff890e68 → 0xff890e78 → 0x00000000
0xff890e4c +0x001c: 0x0804854a → <setup+43> add esp, 0x10
0xff890e50 +0x0020: 0xf7ec6ce0 → 0xfbcd2087
0xff890e54 +0x0024: 0x00000000

gef>
0xff890e58 +0x0028: 0x00000002
0xff890e5c +0x002c: 0x00000000
0xff890e60 +0x0030: 0x00000001
0xff890e64 +0x0034: 0xff890f24 → 0xff891187 → "./vuln"
0xff890e68 +0x0038: 0xff890e78 → 0x00000000 ← $ebp
0xff890e6c +0x003c: 0x080485a7 → <main+27> mov eax, 0x0
0xff890e70 +0x0040: 0xffffffff → push ebp
0xff890e74 +0x0044: 0xff890e90 → 0x00000001
0xff890e78 +0x0048: 0x00000000
0xff890e7c +0x004c: 0xf7d09e81 → <__libc_start_main+241> add esp, 0x10
```



Buffer overflow

```
gef> telescope
0xff890e30  +0x0000: 0xff890e40    →  0x78787878      ← $esp
0xff890e34  +0x0004: 0xf7f05da0    →  pop edx
0xff890e38  +0x0008: 0xf7d58acb    →  <setvbuf+11> add edi, 0x16d535
0xff890e3c  +0x000c: 0x00000000
0xff890e40  +0x0010: 0x78787878
0xff890e44  +0x0014: 0x78787878
0xff890e48  +0x0018: 0x78787878
0xff890e4c  +0x001c: 0x78787878
0xff890e50  +0x0020: 0x78787878
0xff890e54  +0x0024: 0x78787878
gef>
0xff890e58  +0x0028: 0x78787878
0xff890e5c  +0x002c: 0x78787878
0xff890e60  +0x0030: 0x78787878
0xff890e64  +0x0034: 0x78787878
0xff890e68  +0x0038: 0x78787878
0xff890e6c  +0x003c: 0x08048506    →  ← $ebp
0xff890e70  +0x0040: 0x17100900    →  add dh, BYTE PTR [ebp+0x6e]
0xff890e74  +0x0044: 0xff890e90    →  0x00000001
0xff890e78  +0x0048: 0x00000000
0xff890e7c  +0x004c: 0xf7d09e81    →  <_libc_start_main+241> add esp,
```

Buffer overflow:
Sovrascrivo da 0xFF890E40
a 0xFF890E6F

0xFF890E6C contiene l'RIP
di ritorno!



Global Offset Table

Global Offset Table (GOT)

- o Procedure Linkage Table (PLT)

La Global Offset Table (GOT) viene utilizzata per chiamare procedure / funzioni esterne al programma, il cui indirizzo non è noto al momento del linking.

Al momento dell'esecuzione il linker dinamico popola la GOT con gli indirizzi delle funzioni di cui necessita.

Global Offset Table (GOT)

o Procedure Linkage Table (PLT)

```
0x8048553 <chall+3>          sub    esp, 0x28
0x8048556 <chall+6>          sub    esp, 0xc
0x8048559 <chall+9>          push   0x8048648
→ 0x804855e <chall+14>         call   0x8048390 <printf@plt>
↳ 0x8048390 <printf@plt+0>    jmp    DWORD PTR ds:0x804a00c
    0x8048396 <printf@plt+6>    push   0x0
    0x804839b <printf@plt+11>   jmp    0x8048380
    0x80483a0 <gets@plt+0>     jmp    DWORD PTR ds:0x804a010
    0x80483a6 <gets@plt+6>     push   0x8
    0x80483ab <gets@plt+11>   jmp    0x8048380

printf@plt (
    [sp + 0x0] = 0x8048648 → "Hi, what's your name?\n",
    [sp + 0x4] = 0xf7feada0 → pop edx
)
```

```
gef> x/3i 0x8048390
=> 0x8048390 <printf@plt>:      jmp    DWORD PTR ds:0x804a00c
    0x8048396 <printf@plt+6>:    push   0x0
    0x804839b <printf@plt+11>:   jmp    0x8048380
gef> x/x 0x804a00c             Override *(0x804A00C)
0x804a00c: 0x08048396          to hijack printf function
```

Stack canary

Stack canary / Stack cookie

Le stack canary sono una delle protezioni utilizzate per evitare gli attacchi al software.

Le canary sono valori noti che vengono inseriti tra un buffer e l'indirizzo di ritorno, servono a monitorare gli overflow del buffer.

Stack canary / Stack cookie

Quando il buffer è stato riempito, l'ultimo dato che verrà corrotto prima dell' Instruction Pointer di ritorno sarà la stack canary.

Se la stack canary iniziale è diversa dalla canary attuale, si è verificato uno stack overflow.

Esso può quindi essere gestito, ad esempio, terminando il programma.

Stack canary / Stack cookie

```
void chall(void)
{
    int in_GS_OFFSET;
    char local_30 [32];
    int STACK_CANARY;

    STACK_CANARY = *(int *) (in_GS_OFFSET + 0x14);
    printf("Hi, what's your name?\n> ");
    gets(local_30);
    printf("Hello %s!\n", local_30);
    if (STACK_CANARY != *(int *) (in_GS_OFFSET + 0x14)) {
        __stack_chk_fail_local();
    }
    return;
}
```

Stack canary / Stack cookie

```
gef> telescope
0xfffffce00 +0x0000: 0x08048731 → "Hello %s!\n" ← $esp
0xfffffce04 +0x0004: 0xfffffce1c → "mario rossi"
0xfffffce08 +0x0008: 0xf7fab000 → 0x001d4d6c
0xfffffce0c +0x000c: 0x080485d3 → <chall+12> add ebx, 0x1a2d
0xfffffce10 +0x0010: 0xfffffce48 → 0xfffffce58 → 0x00000000
0xfffffce14 +0x0014: 0xf7feada0 → pop edx
0xfffffce18 +0x0018: 0xf7e3dacb → <setvbuf+11> add edi, 0x16d535
0xfffffce1c +0x001c: "mario rossi"
0xfffffce20 +0x0020: "o rossi"
0xfffffce24 +0x0024: 0x00697373 ("ssi"?)  

gef>
0xfffffce28 +0x0028: 0xfffffce48 → 0xfffffce58 → 0x00000000
0xfffffce2c +0x002c: 0x080485be → <setup+61> add esp, 0x10
0xfffffce30 +0x0030: 0xf7fabce0 → 0xfbdb2087
0xfffffce34 +0x0034: 0x00000000
0xfffffce38 +0x0038: 0x00000002
0xfffffce3c +0x003c: 0x88e21f00
0xfffffce40 +0x0040: 0x00000001
0xfffffce44 +0x0044: 0x00000000
0xfffffce48 +0x0048: 0xfffffce58 → 0x00000000 ← $ebp
0xfffffce4c +0x004c: 0x08048657 → <main+37> mov eax, 0x0
```

Return Oriented Programming

Return Oriented Programming (ROP)

La programmazione orientata al ritorno (ROP) è una tecnica di exploit che consente di eseguire codice in presenza di difese di sicurezza. Per esempio ASLR.

Alla base del ROP c'è un tipo di programmazione basata sul return. Cioè vengono impostati i valori dei registri secondo operazioni di *pop* dalla stack

Return Oriented Programming (ROP)

```
code:x86:32
0x80485e7 <main+181>      mov    DWORD PTR [esp], 0x1
0x80485ee <main+188>      call   0x80483d0 <exit@plt>
0x80485f3 <main+193>      leave 
→ 0x80485f4 <main+194>      ret
↳ 0x804851d <topo+0>      pop    eax
0x804851e <topo+1>      ret
0x804851f <macellaio+0>    mov    ecx, eax
0x8048521 <macellaio+2>    ret
0x8048522 <fuoco+0>       pop    edi
0x8048523 <fuoco+1>       ret
threads
[#0] Id 1, Name: "fiera_dell_est", stopped 0x80485f4 in main (), reason: SING
trace
[#0] 0x80485f4 → main()
[...]
0x080485f4 in main ()
gef> telescope
0xff95e12c +0x0000: 0x0804851d → <topo+0> pop eax      ← $esp
0xff95e130 +0x0004: 0x0000000b
0xff95e134 +0x0008: 0x08048526 → <gatto+0> pop ebx
0xff95e138 +0x000c: 0x0804a02c → "/bin/sh"
0xff95e13c +0x0010: 0x08048528 → <cane+0> pop edx
0xff95e140 +0x0014: 0x00000000
0xff95e144 +0x0018: 0x08048530 → <bastone+0> pop ecx
0xff95e148 +0x001c: 0x00000000
0xff95e14c +0x0020: 0x0804852d → <angelo_della_morte+0> int 0x80
0xff95e150 +0x0024: 0xf7f50000 → 0x00026f34
```

Tecniche di attacco per BOF

Tecniche di attacco per BOF

Stack overflow - Stack RWX

ASLR OFF:

1. Inserisco la shellcode nella stack.
2. Setto l'istruzione pointer all'inizio della shellcode.

ASLR ON:

1. Trovo l'indirizzo base della stack.
2. Calcolo l'indirizzo del buffer dove andrò ad inserire la shellcode.
3. Inserisco la shellcode nella stack.
4. Setto l'istruzione pointer all'inizio della shellcode.

Tecniche di attacco per BOF

Stack overflow - Stack RW- (Senza ROP) (Senza GOT overwrite)

ASLR OFF:

1. Setto l'istruzione pointer alla funzione system.
2. Dopo l'istruzione pointer inserisco un puntatore a '/bin/sh'

Tecniche di attacco per BOF

Stack overflow - Stack RW- (Senza ROP) (Senza GOT overwrite)

Questo attacco è conosciuto come ret-to-libc

ASLR ON:

1. Trovo l'indirizzo base della libreria libc.
2. Calcolo l'indirizzo della funzione system.
3. Setto l'istruzione pointer alla funzione system.
4. Dopo l'istruzione pointer inserisco un puntatore a '/bin/sh'

Tecniche di attacco per BOF

Stack overflow con stack canary

1. Devo fare il leak del valore della stack canary oppure fare un bruteforce di essa. (ove possibile)
2. Faccio buffer overflow inserendo alla giusta posizione la stack canary.
3. Ora la situazione è riconducibile a BOF con stack RW- oppure al seguente caso.

Tecniche di attacco per BOF

Stack overflow - ROP

Si utilizza questa tecnica di exploit quando non ci sono altre vie per poter ottenere l'accesso remoto, oppure è più semplice trovare una ROP chain che seguire altre vie.

1. Faccio buffer overflow inserendo nell'istruzione pointer la rop chain.
2. Le operazioni di pop faranno il resto del lavoro.

Presentazione di pwntools per il debug e per la scrittura di exploit

Nella prossima lezione affronteremo più nel dettaglio:

- *Format string exploit*
- *Ripasso su Address Space Layout Randomization (ASLR)*
- *Ripasso su Global Offset Table (GOT)*
- *Ripasso su Buffer Overflow (BOF)*
- *Ripasso su Stack Canary*
- *Ripasso su Return Oriented Programming (ROP)*
- *Ripasso su tecniche di attacco*
- *Vostre richieste... (Se ci sono)*