Software Security 2
Introduzione alla Binary Exploitation

Nicola VELLA
Università di Pisa

https://cybersecnatlab.it

License & Disclaimer

License Information

This presentation is licensed under the Creative Commons BY-NC License

To view a copy of the license, visit:

http://creativecommons.org/licenses/by-nc/3.0/legalcode

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.

Argomenti

- Introduzione alla categoria ***pwn***
- Corruzione della memoria
- Mitigazioni moderne

Argomenti

- Introduzione alla categoria ***pwn***
- Corruzione della memoria
- Mitigazioni moderne

PWN: Cosa?

Pwnare un binario vuol dire inserire dell'input non previsto che prende il controllo del programma

Normalmente, lo scopo è quello di riuscire a trasformare il comportamento di un binario qualsiasi, a quello di una shell (/bin/sh)

PWN: Perché?

Privilege escalation

- Jailbreak di dispositivi mobile e console

Remote code execution

- Esecuzione di codice in un dispositivo accessibile attraverso la rete, con o senza interazione da parte dell'utente

PWN: Come è possibile?

- Memory (un)safety:
  - Linguaggi come C/C++ lasciano il controllo completo della memoria al programmatore
  - Problema: Tutti fanno errori, soprattutto se è molto facile compierli

Argomenti

Introduzione alla categoria ***pwn***

Corruzione della memoria

Mitigazioni moderne

Memory Corruption: Cosa?

- Scrittura oltre i limiti di un buffer
  - Buffer Overflow Lineare
  - Accessi Out of Bounds

Memory Corruption: Cosa?


```c
#include <string.h>

// [1] Buffer Overflow Lineare
int main(int argc, char *argv[]) {
    char user_input[8];

    for (int i = 0; i < strlen(argv[1]); i++)
        user_input[i] = argv[1][i];

  return 0;
}
```

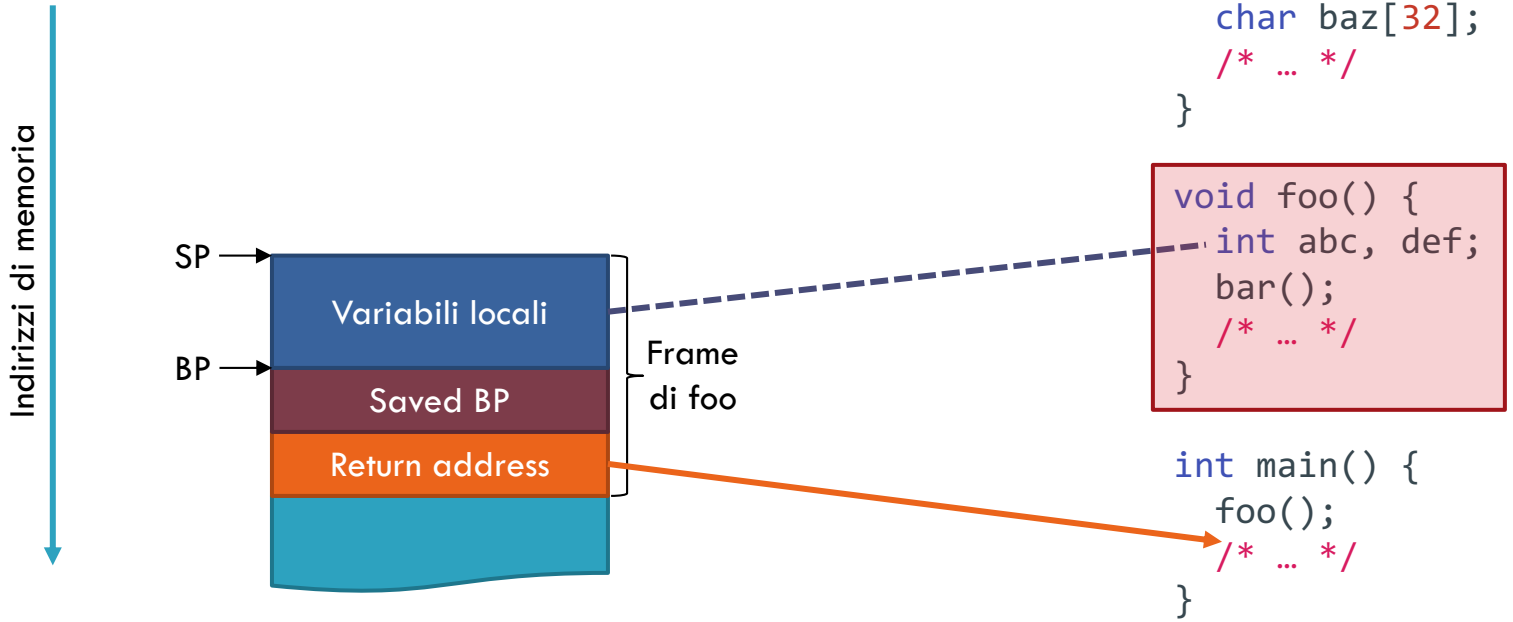
Memory Corruption: Cosa?


```c
#include <string.h>
#include <stdlib.h>

// [2] Accessi Out of Bounds
int main(int argc, char *argv[]) {
    char array[8];

    // argv[i] -> Indice array
    // argv[i+1] -> Valore array
    for (int i = 1; i < argc-1; i+=2) {
        int index = atoi(argv[i]);
        array[index] = atoi(argv[i+1]);
        i++;
    }
  return 0;
}
```

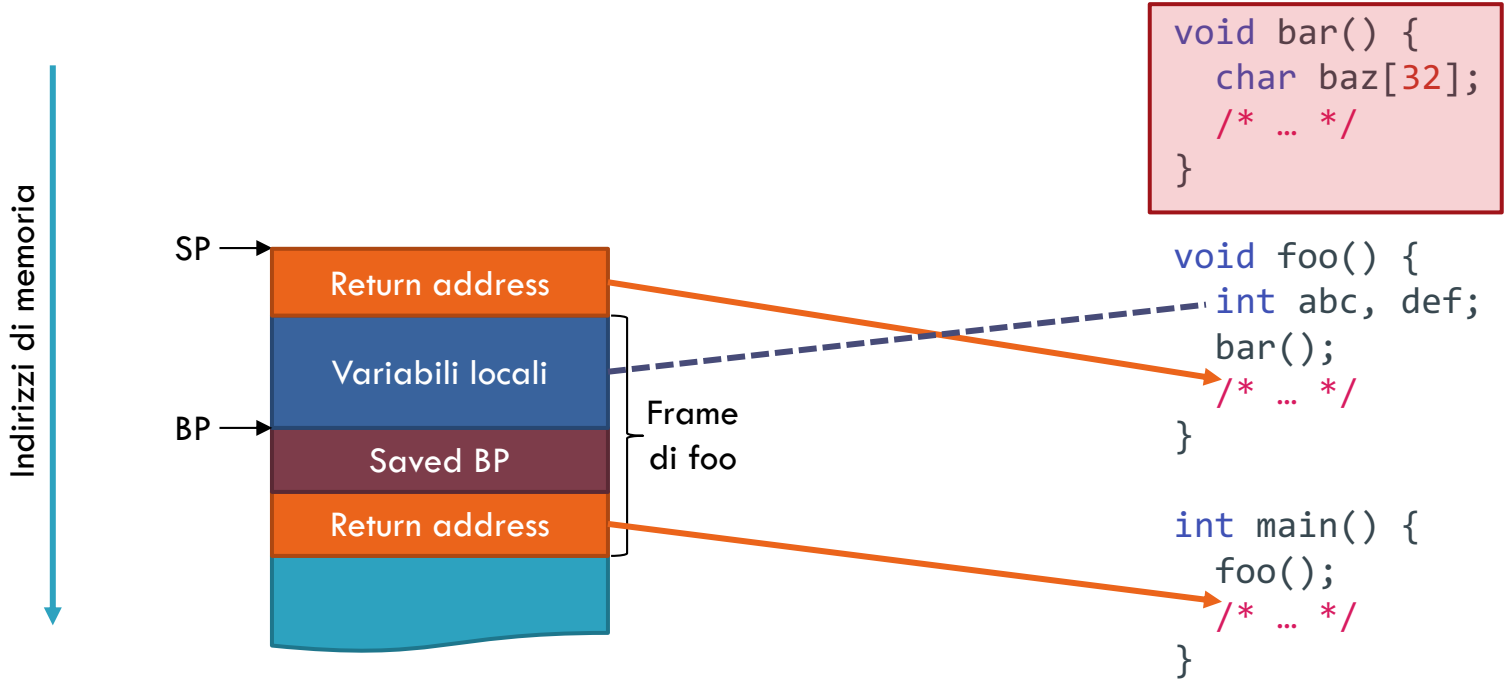
Buffer overflows su stack

- 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!

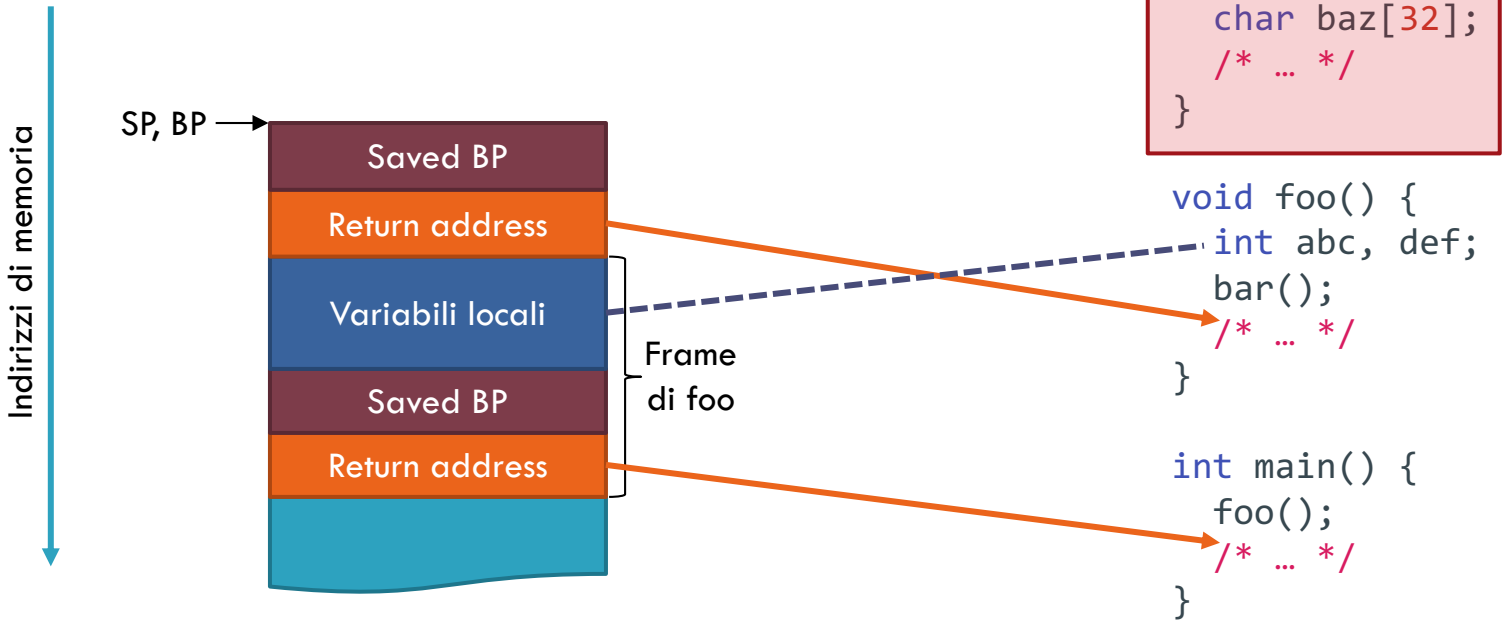
Lo stack x86



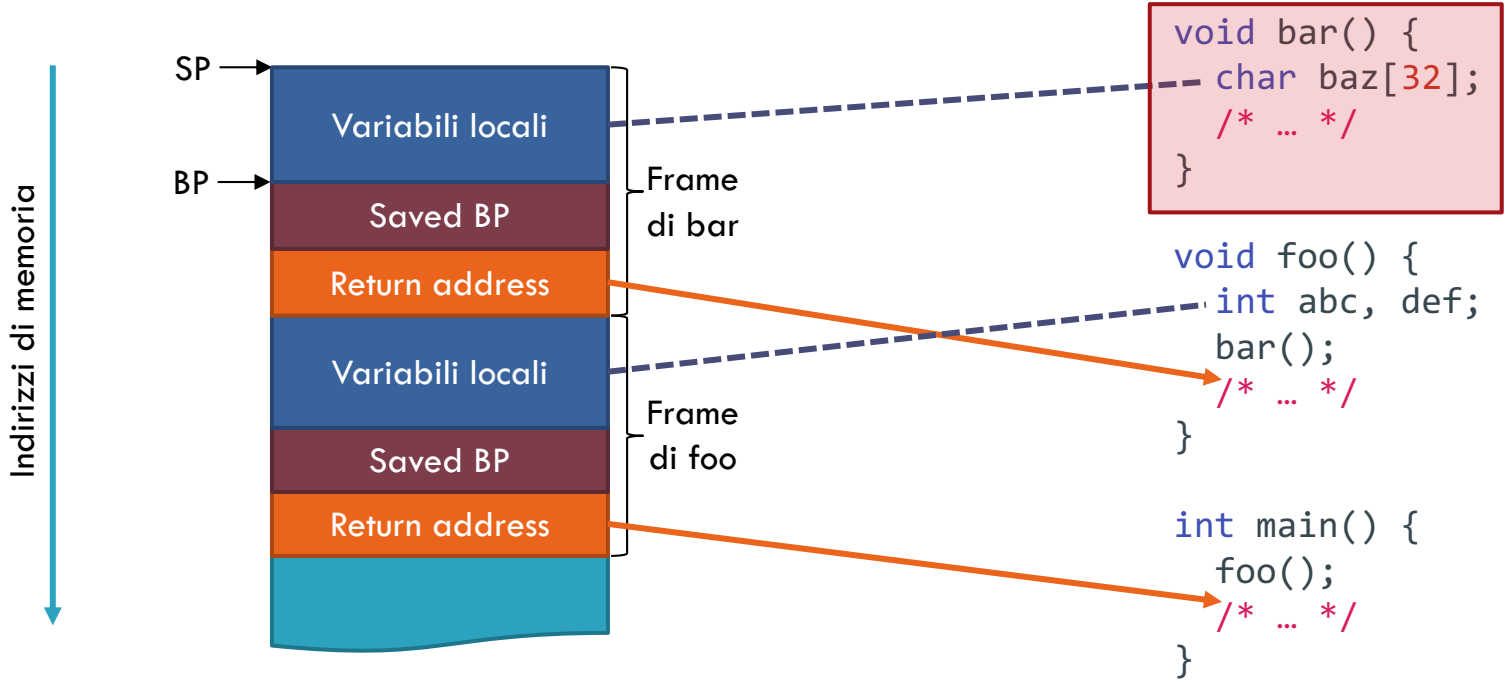
Lo stack x86



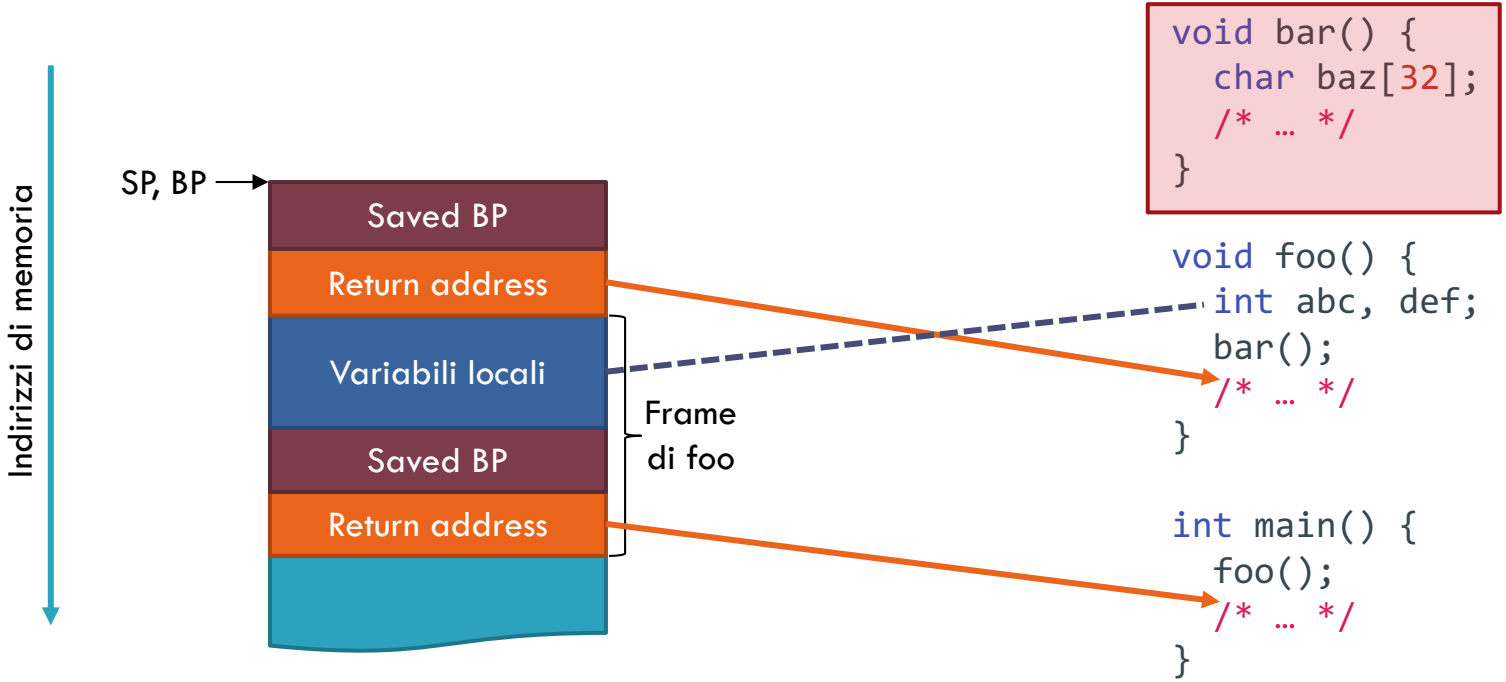
Lo stack x86



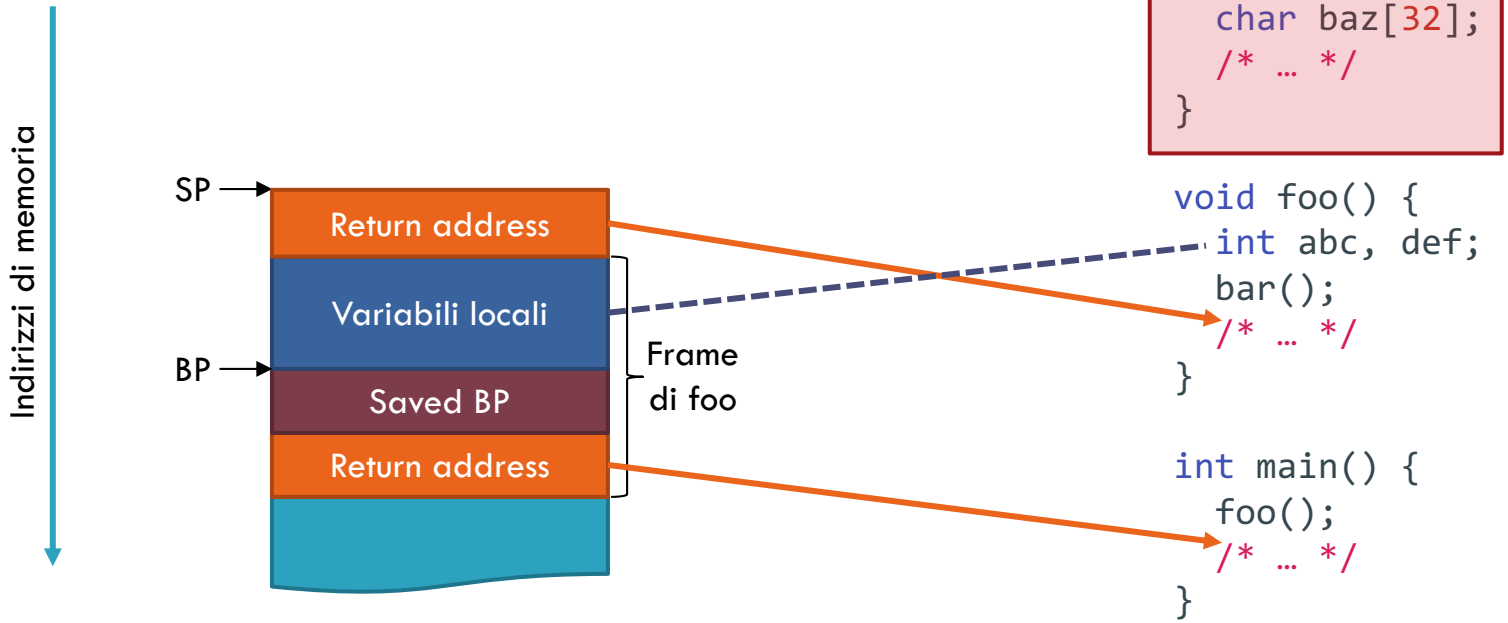
Lo stack x86



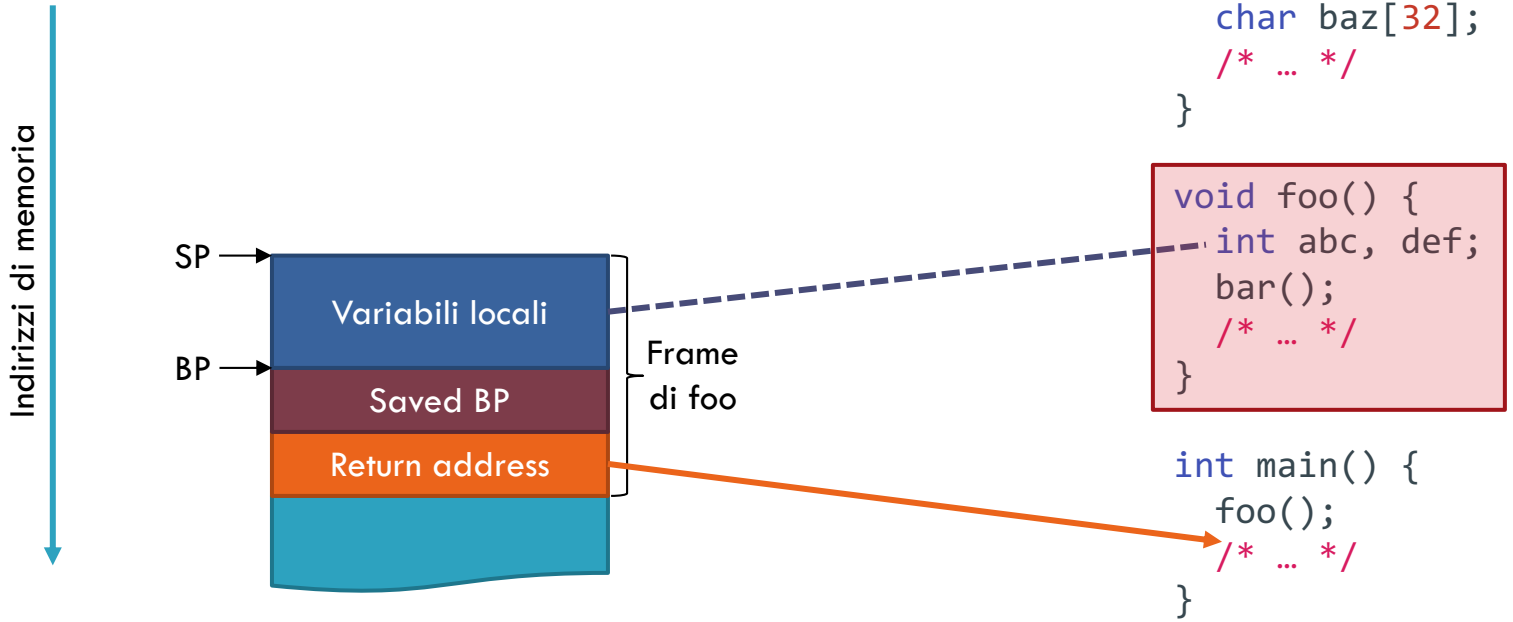
Lo stack x86



Lo stack x86



Lo stack x86



Lo stack visto da GDB

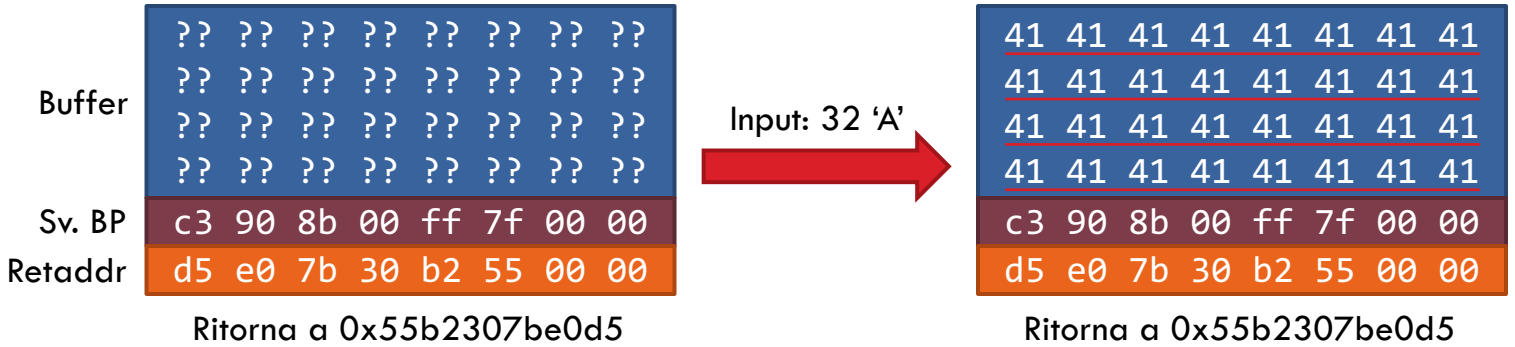

```c
void f3(void) {
    puts("Hello from f3!");
}

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 ()
```

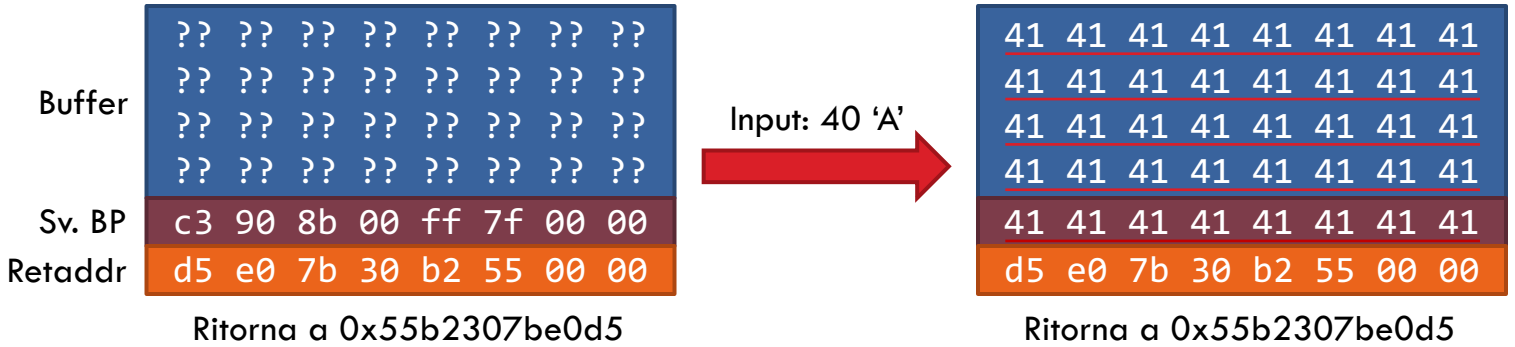
```
pwndbg> backtrace
#0  0x000000000040113b in f2 ()
#1  0x000000000040115b in f1 ()
#2  0x000000000040117b in main ()
#3  0x00007ffff7de7f43 in __libc_start_main () from /lib64/libc.so.6
#4  0x000000000040106e in _start ()
pwndbg> x/8gx $rsp
0x7fffffffced0: 0x00007fffffffcee0      0x000000000040115b
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.
```

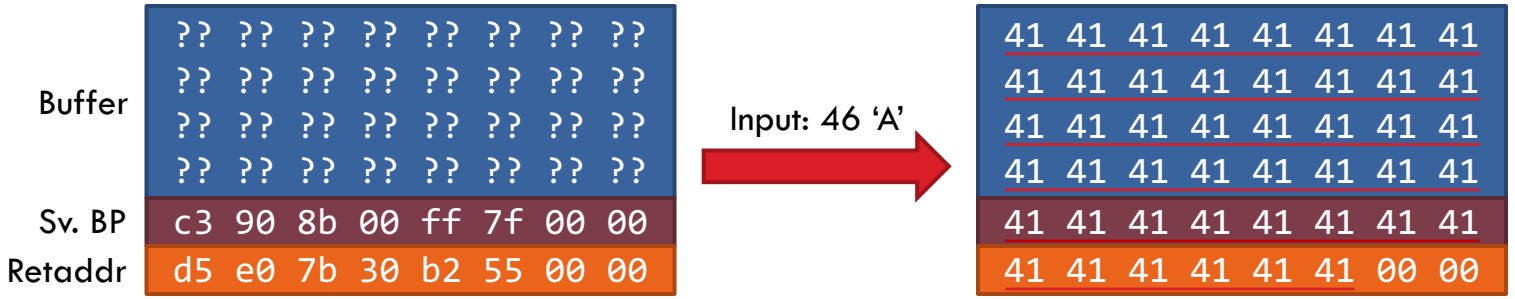
Stack overflow



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

Argomenti

Introduzione alla categoria ***pwn***

Corruzione della memoria

Mitigazioni moderne

Come ci si difende?

Non scrivendo codice con bug

Integrando mitigazioni per rendere più difficile lo sviluppo di un exploit

Utilizzando linguaggi Memory Safe

Come ci si difende?

Non scrivendo codice con bug

Integrando mitigazioni per rendere più difficle lo sviluppo di un exploit

Utilizzando linguaggi Memory Safe

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
- 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 si hanno vulnerabilità subito sfruttabili
- 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 un puntatore a cui legge/scrive dati utente, c'è un overflow su buffer prima del puntatore:
  - Usiamo overflow per sovrascrivere il puntatore
  - 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
Introduzione alla Binary Exploitation

Nicola VELLA
Università di Pisa

https://cybersecnatlab.it