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;
}
```

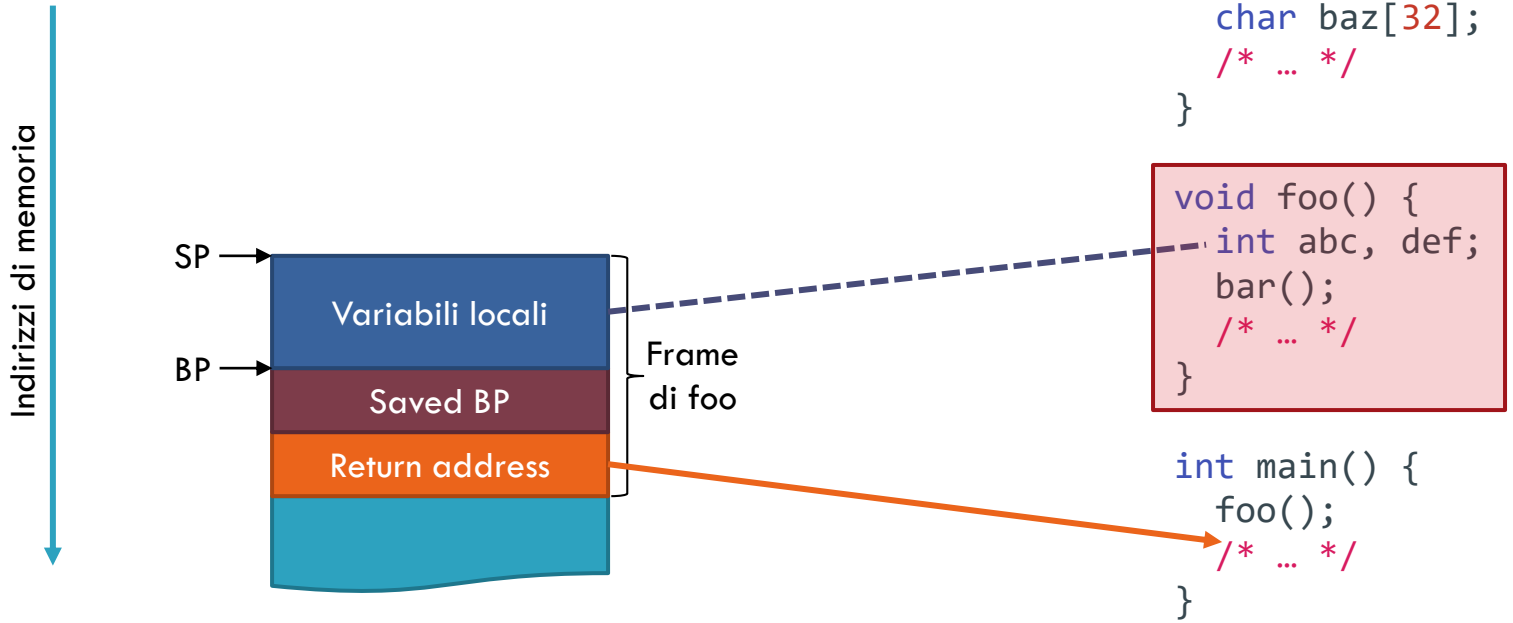
Memory Corruption: Cosa?


```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;
}
```

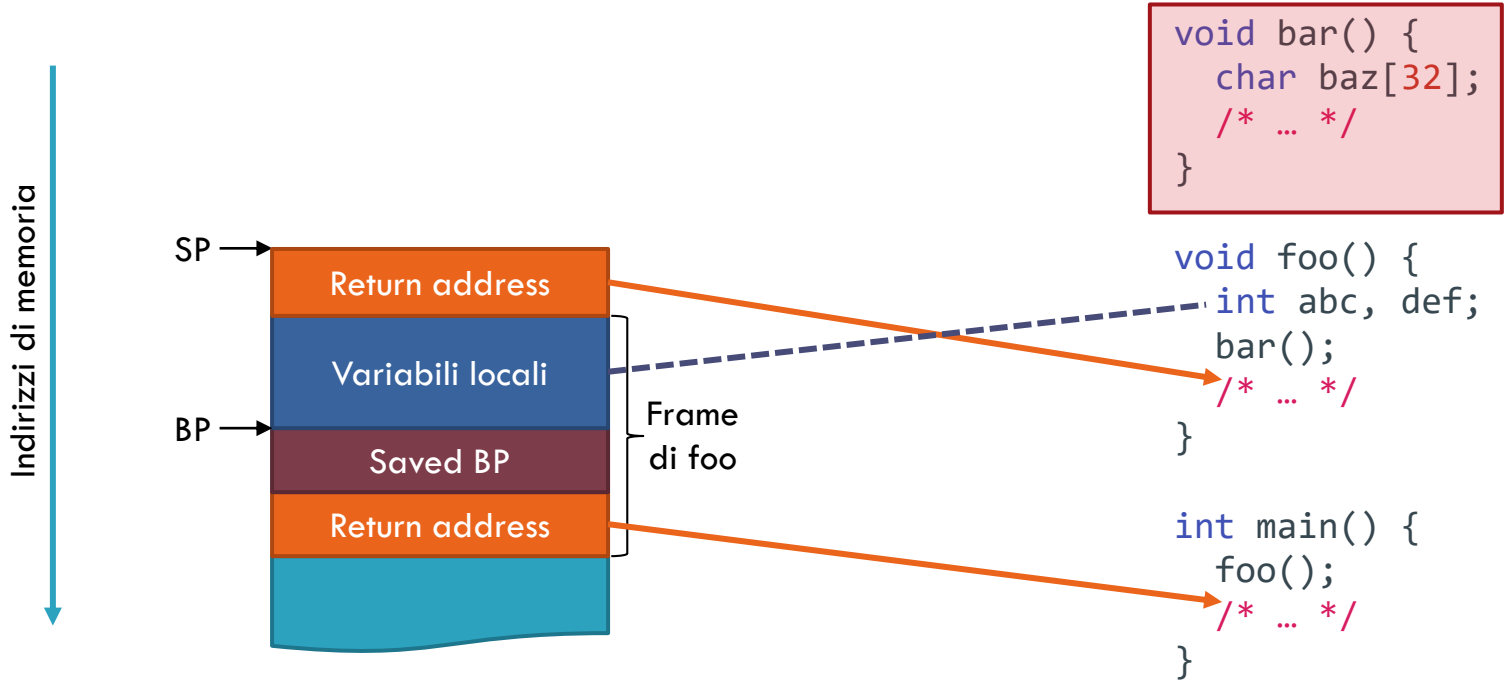
Buffer overflows su stack

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

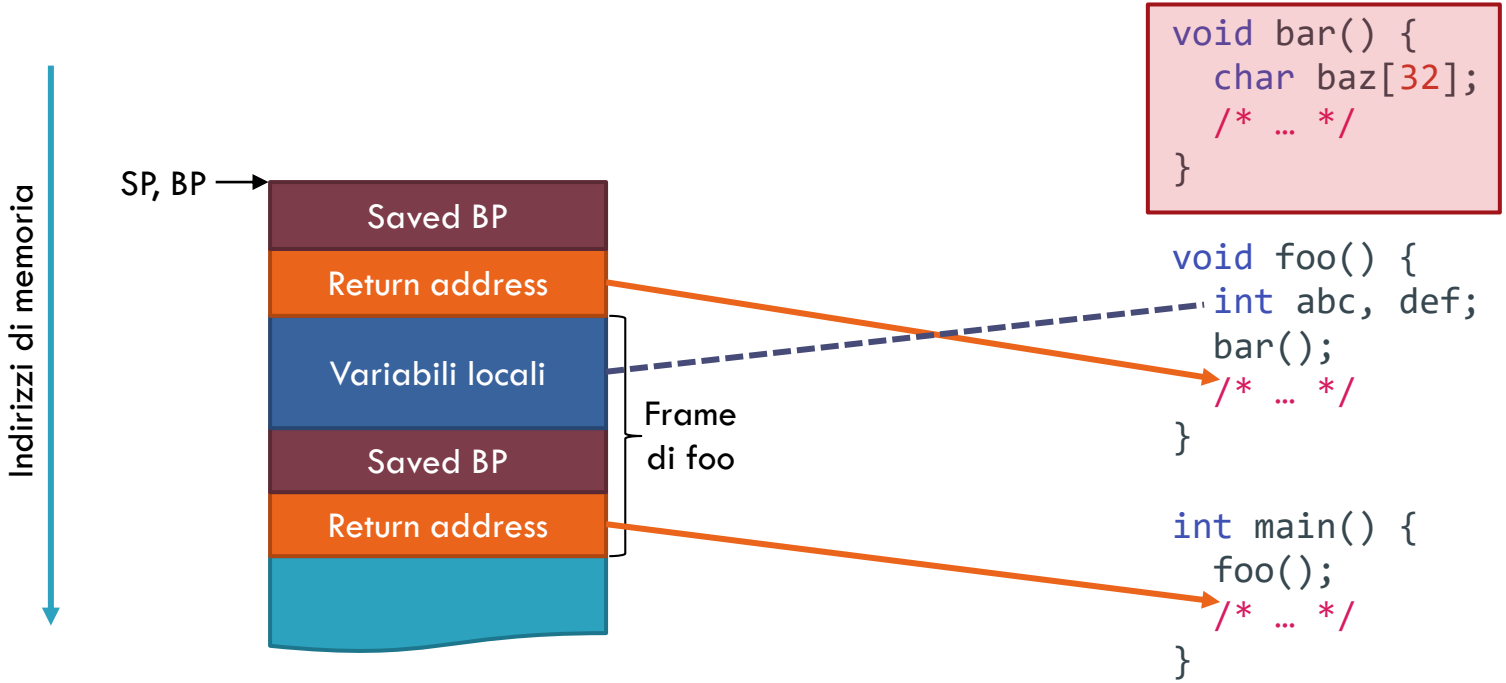
Lo stack x86



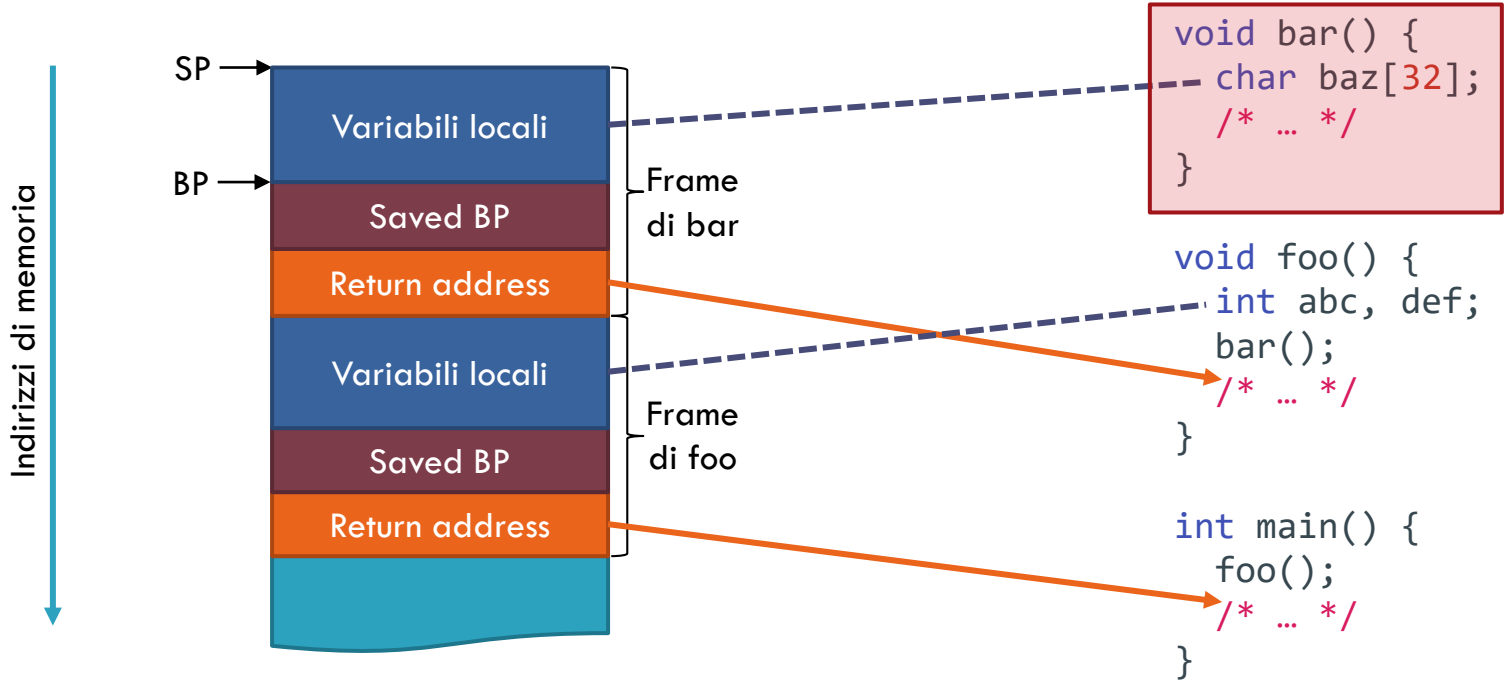
Lo stack x86



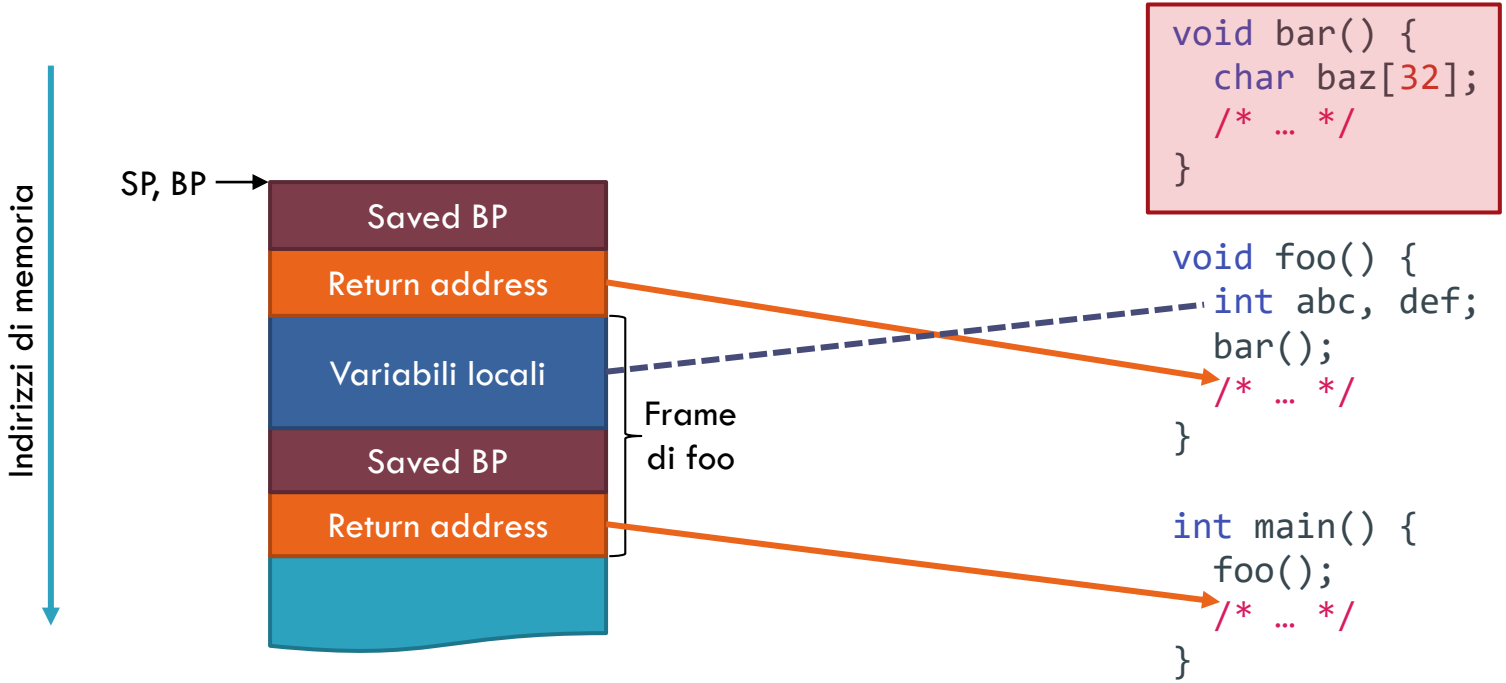
Lo stack x86



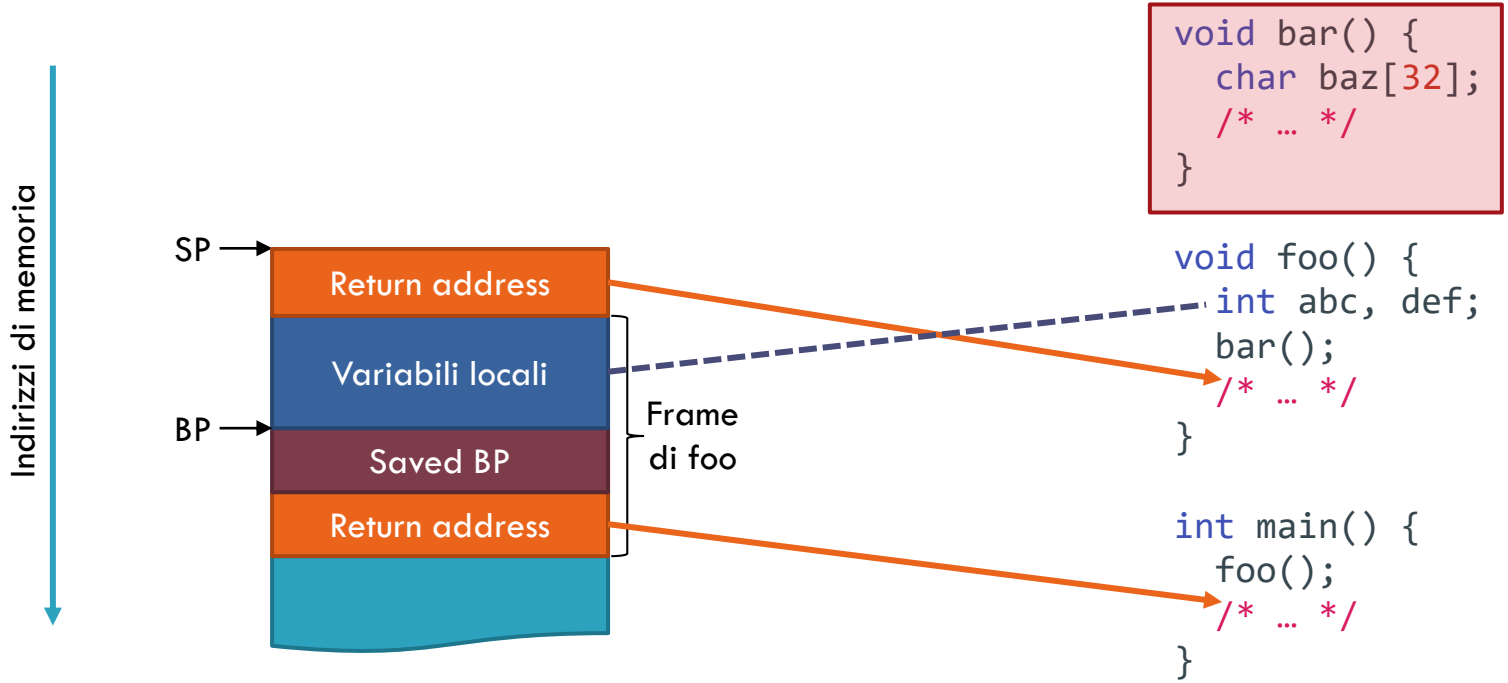
Lo stack x86



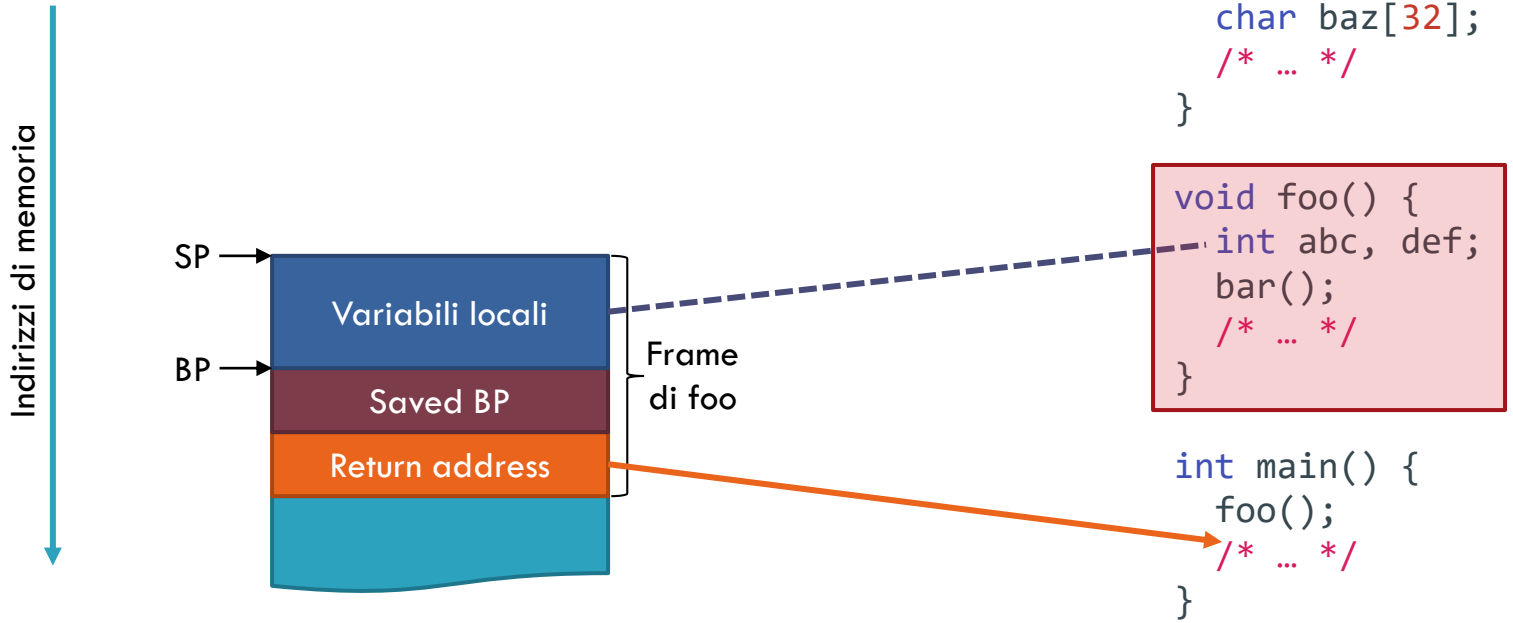
Lo stack x86



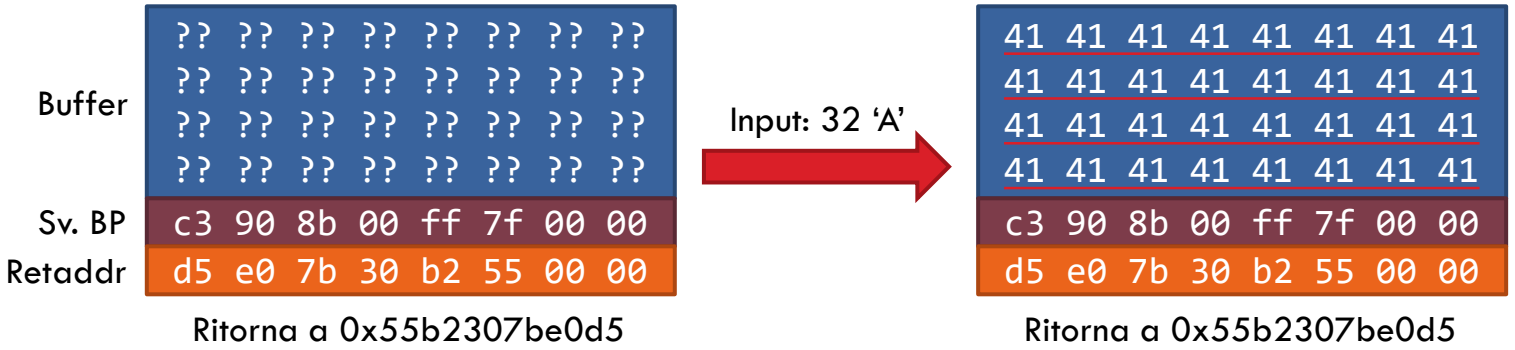
Lo stack x86



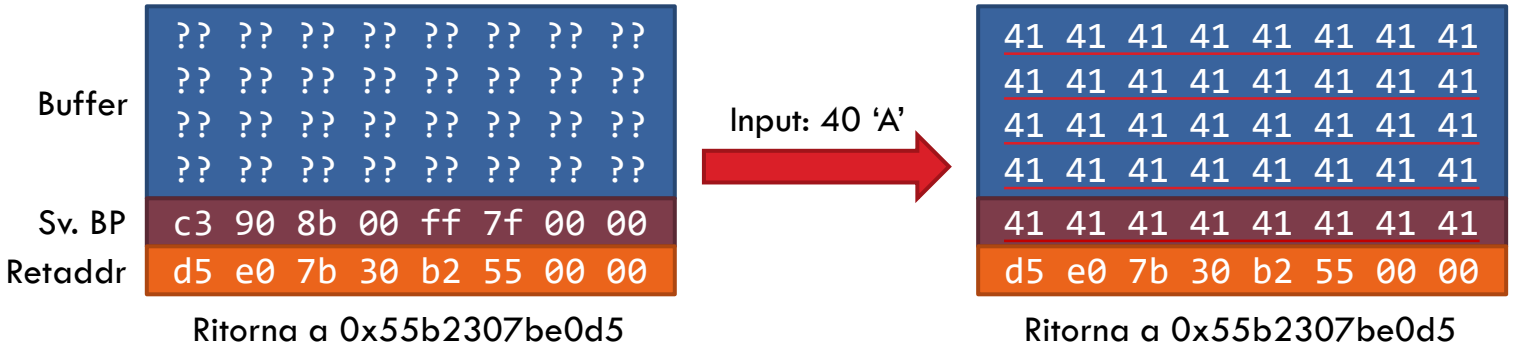
Lo stack x86



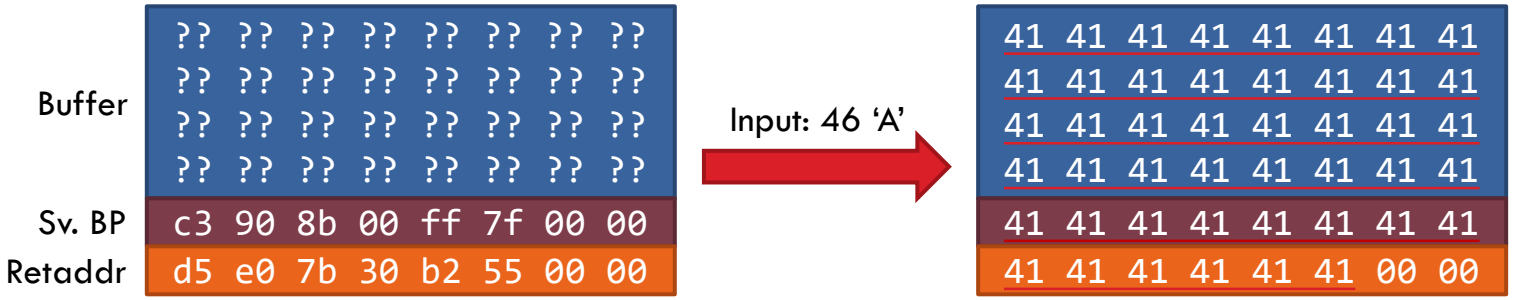
Stack overflow



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 difficile 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!

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

Software Security 2
Introduzione alla Binary Exploitation

Nicola VELLA
Università di Pisa

https://cybersecnatlab.it