Università degli Studi di Milano
Facoltà di Scienze e Tecnologie

Stack-based buffer overflow

Andrea Lanzi

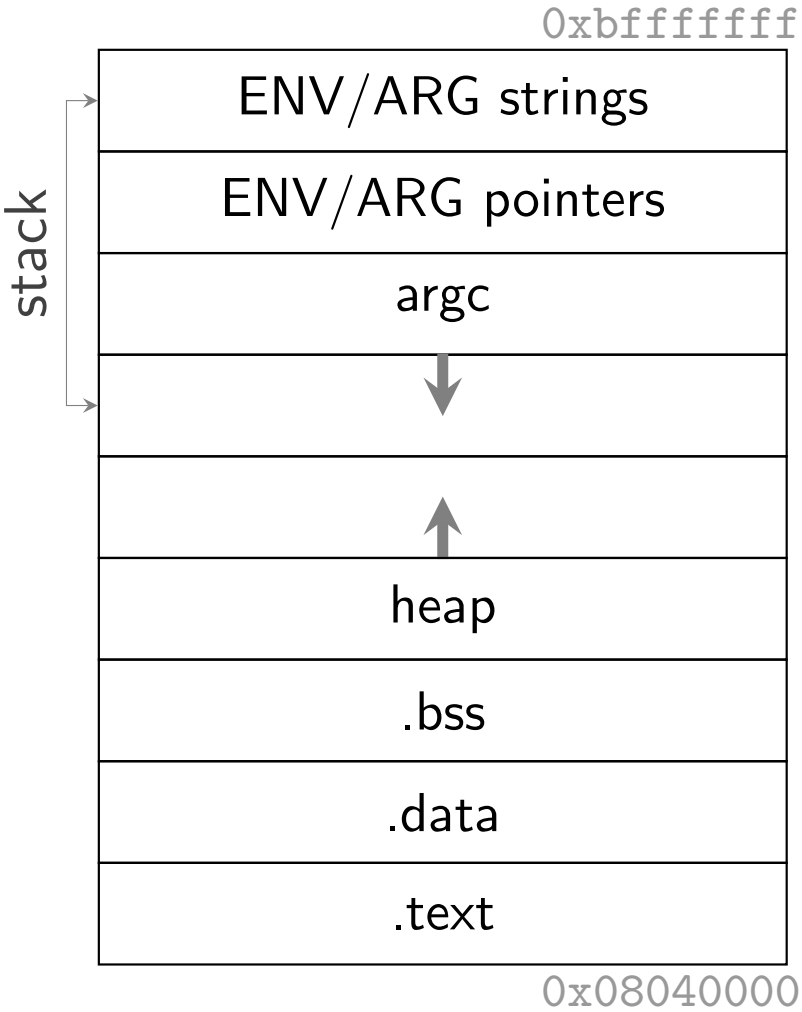
A.A. 2015–2016

Classificazione attacchi

Attacchi di tipo overflow

- Stack overflow
  - **return-into-stack**
  - **return-into-libc**
  - off-by-one
- Heap overflow
- Integer overflow

Layout memoria di un processo

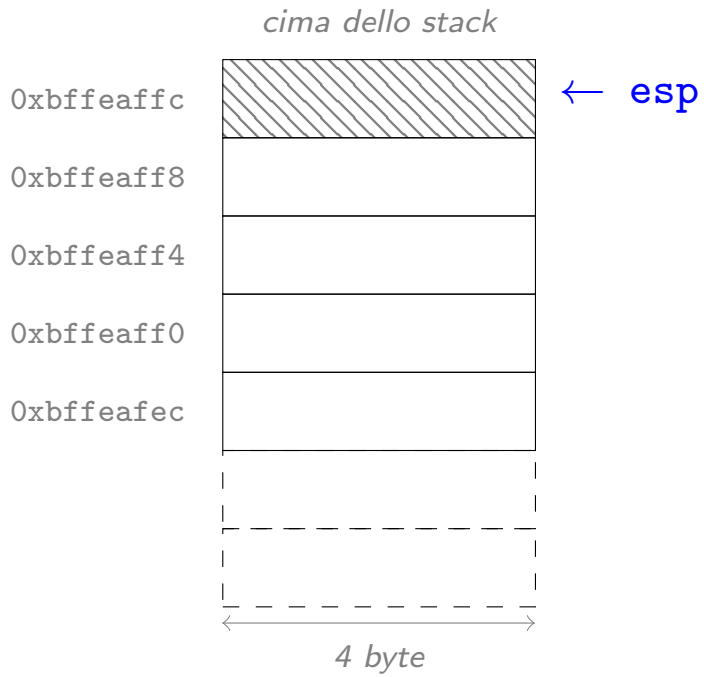


Stack layout

Caratteristiche stack Intel x86

- struttura LIFO (*last-in first-out*)
- "cresce" da indirizzi alti verso indirizzi bassi
- due registri per la gestione dello stack:
  - *frame pointer* (`ebp`): punta al record di attivazione della procedura corrente
  - *stack pointer* (`esp`): punta all'ultimo elemento inserito sullo stack
- istruzioni assembly Intel per manipolare lo stack:
  - `push`: *decrementa* `esp` e inserisce l'elemento in cima allo stack, nella locazione puntata da `esp`
  - `pop`: elimina l'elemento dalla cima dello stack, *incrementando* `esp`

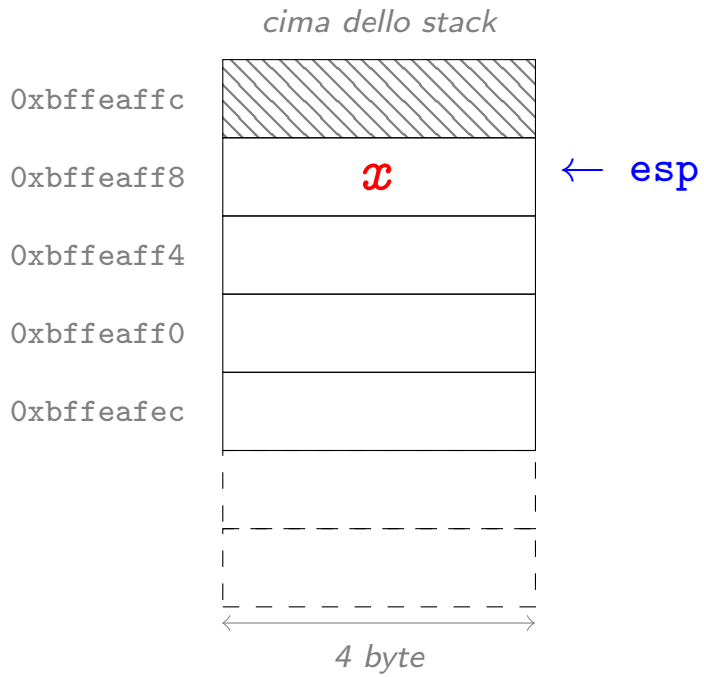
Stack layout
Esempio




```
push %eax
push %ebx
push %ecx
pop %eax
pop %ecx
pop %ebx
```

eax = x
ebx = y
ecx = z

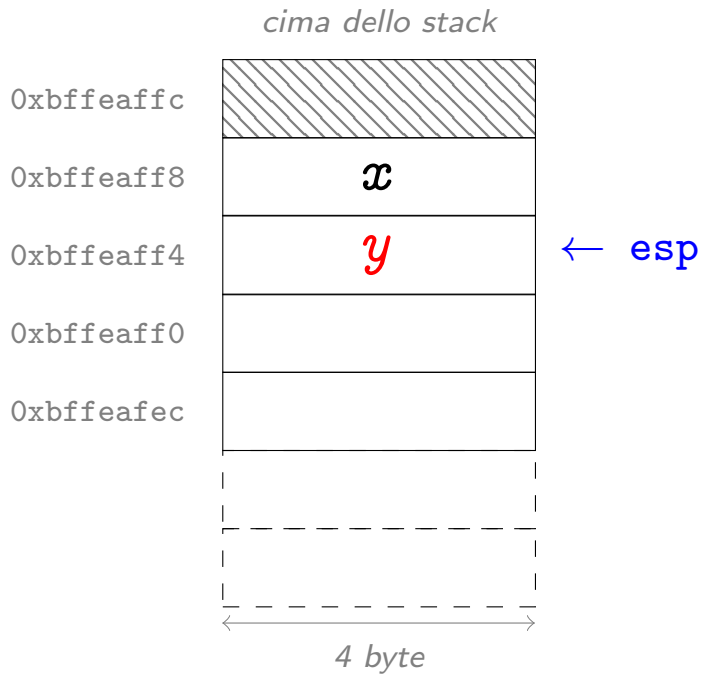
Stack layout
Esempio




```
push %eax
push %ebx
push %ecx
pop %eax
pop %ecx
pop %ebx
```

eax = x
ebx = y
ecx = z

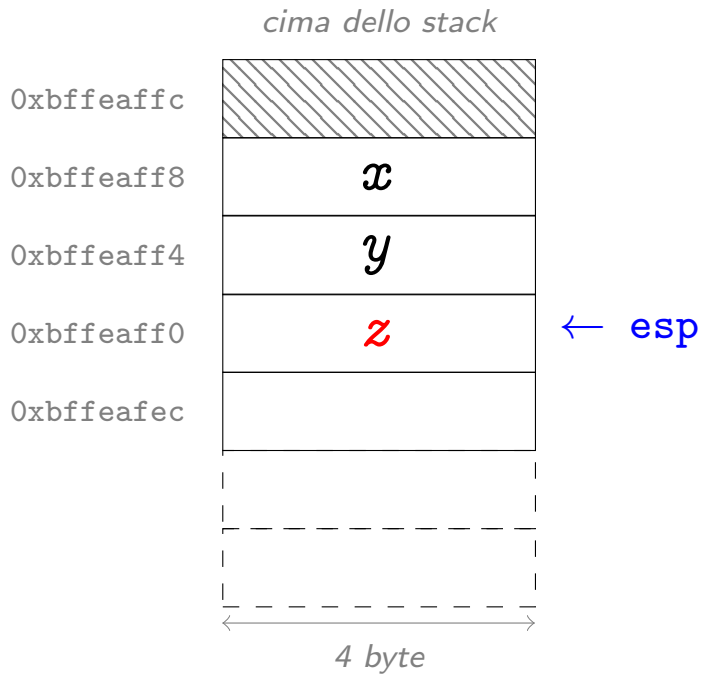
Stack layout
Esempio




```
push %eax
push %ebx
push %ecx
pop %eax
pop %ecx
pop %ebx
```

eax = x
ebx = y
ecx = z

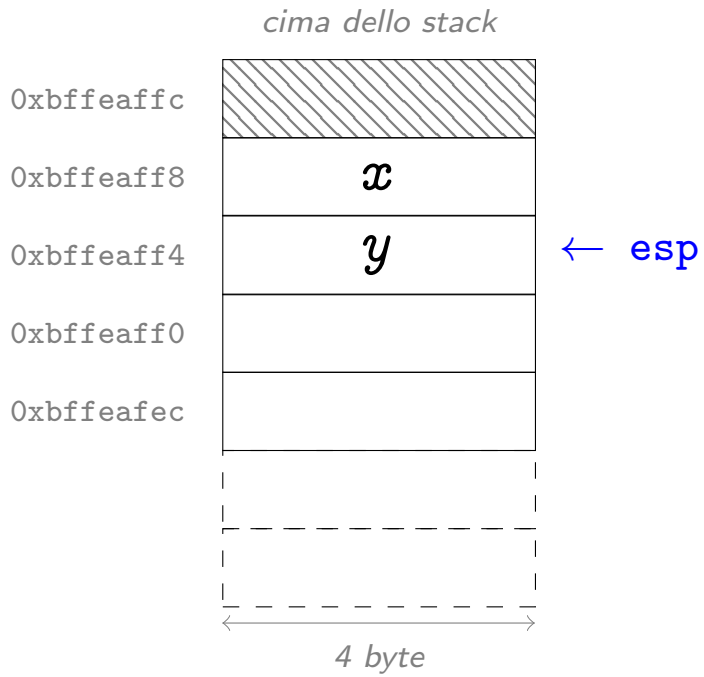
Stack layout
Esempio




```
push %eax
push %ebx
push %ecx
pop %eax
pop %ecx
pop %ebx
```

eax = x
ebx = y
ecx = z

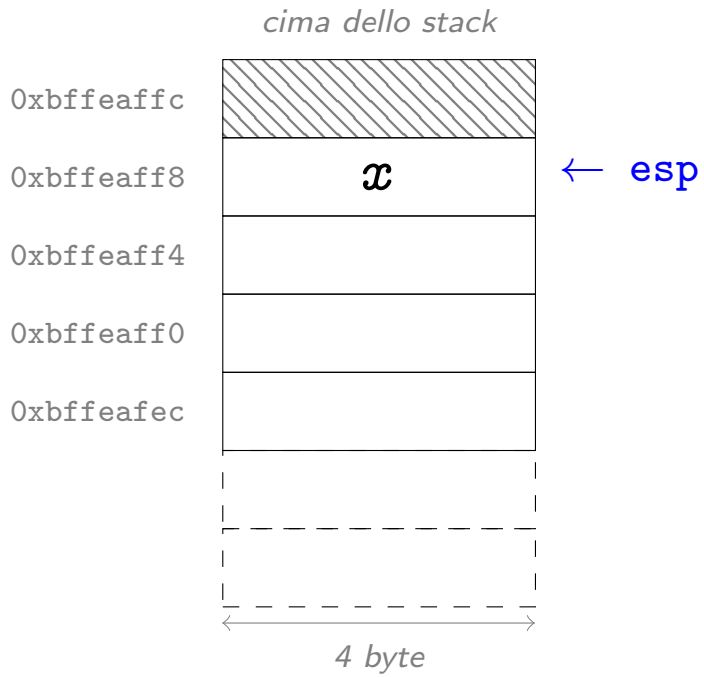
Stack layout
Esempio




```
push %eax
push %ebx
push %ecx
pop %eax
pop %ecx
pop %ebx
```

eax = *z*
ebx = *y*
ecx = *z*

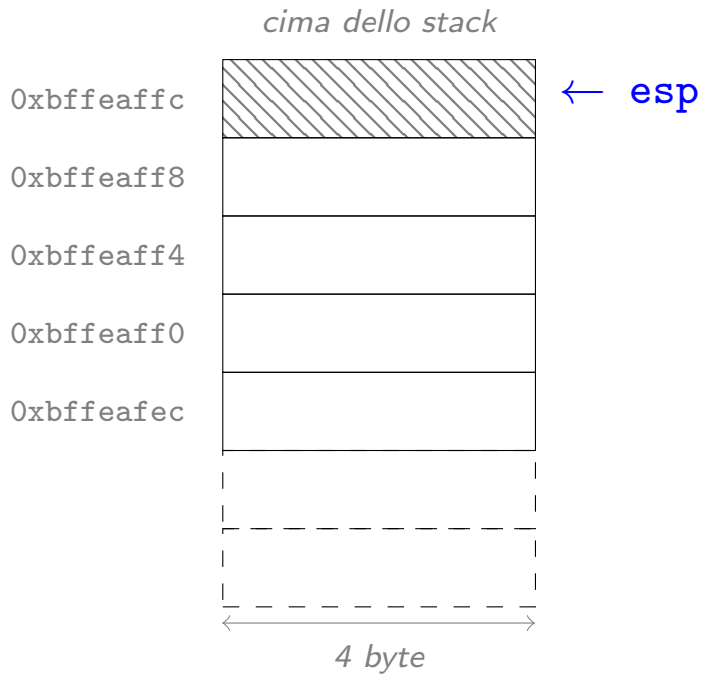
Stack layout
Esempio




```
push %eax
push %ebx
push %ecx
pop %eax
pop %ecx
pop %ebx
```

```
eax = z
ebx = y
ecx = y
```

Stack layout
Esempio




```
push %eax
push %ebx
push %ecx
pop %eax
pop %ecx
pop %ebx
```

eax = z
ebx = x
ecx = y

Chiamata a funzione

In genere, 3 passi principali:

1. il *chiamante* inserisce i parametri della funzione sullo stack
2. il *chiamante* trasferisce il controllo alla funzione (e.g. `call <strcpy>`), salvando prima sullo stack il *return address*
3. il *chiamato* esegue alcune operazioni preliminari (*prologo*)
   - salvataggio (`push`) del registro `ebp`
   - `ebp` = `esp`
   - allocazione delle variabili locali alla funzione chiamata

Terminazione di una funzione

Epilogo

1. `esp` = `ebp` (ripristino spazio nello stack adibito alle variabili locali della funzione)
2. ripristino `ebp` relativo al record d'attivazione precedente
3. ripristino *return address* (i.e., indirizzo del chiamante da cui riprendere l'esecuzione)

Chiamata e terminazione di una funzione

Esempio

```
1   /* hello.c */
2
3   void foo(int n)
4   {
5     printf("numero: %d;\n", n);
6   }
7
8   int main(int argc, char **argv)
9   {
10    foo(10);
11    return 0;
12  }
```

```
1   push %ebp
2   mov %esp,%ebp
3   sub $0x8,%esp
4   mov 0x8(%ebp),%eax
5   mov %eax,0x4(%esp)
6   movl $0x8048480,(%esp)
7   call 0x80482d8 <printf@plt>
8   leave
9   ret
```

`foo()`

```
1   push %ebp
2   mov %esp,%ebp
3   sub $0x4,%esp
4   movl $0xa,(%esp)
5   call 0x8048374 <foo>
6   mov $0x0,%eax
7   leave
8   ret
```

`main()`

Buffer overflow

Buffer

Insieme di locazioni di memoria *contigue*, contenente più istanze di uno stesso tipo di dato (e.g. `int temperature[10];`).

Overflow

Cosa succede se cerchiamo di inserire in un buffer più dati di quanti ne possa contenere?

- il sistema rileva una condizione anomala e interrompere l'operazione (e.g. Java), *oppure*
- il sistema *non può* rilevare l'anomalia, quindi l'operazione viene effettuata ugualmente (e.g. C)

. . . ma dove finiscono i dati "in più"?

Buffer overflow

Buffer

Insieme di locazioni di memoria *contigue*, contenente più istanze di uno stesso tipo di dato (e.g. `int temperature[10];`).

Overflow

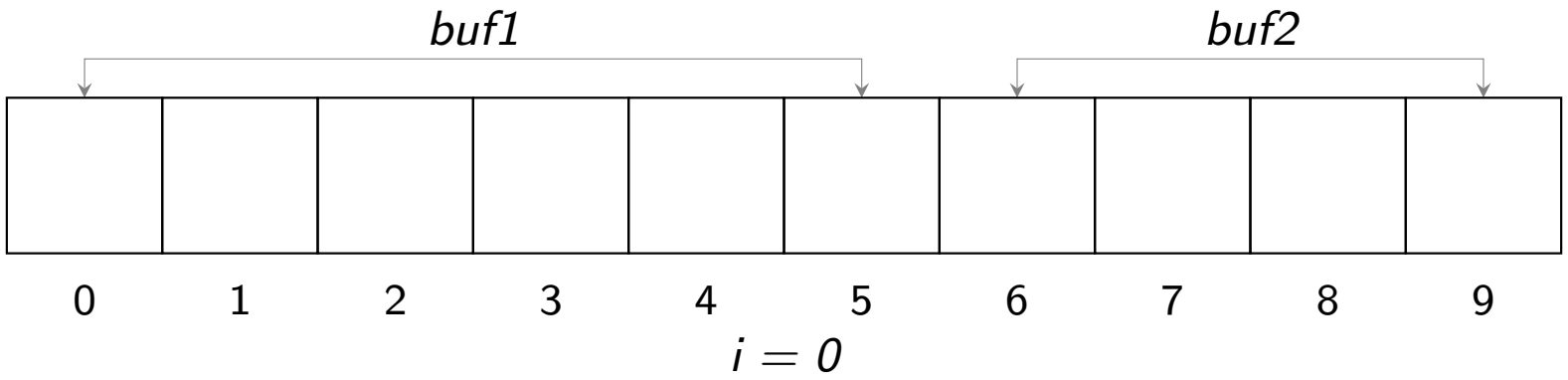
Cosa succede se cerchiamo di inserire in un buffer più dati di quanti ne possa contenere?

- il sistema rileva una condizione anomala e interrompere l'operazione (e.g. Java), *oppure*
- il sistema *non può* rilevare l'anomalia, quindi l'operazione viene effettuata ugualmente (e.g. C)

. . . ma dove finiscono i dati "in più"?

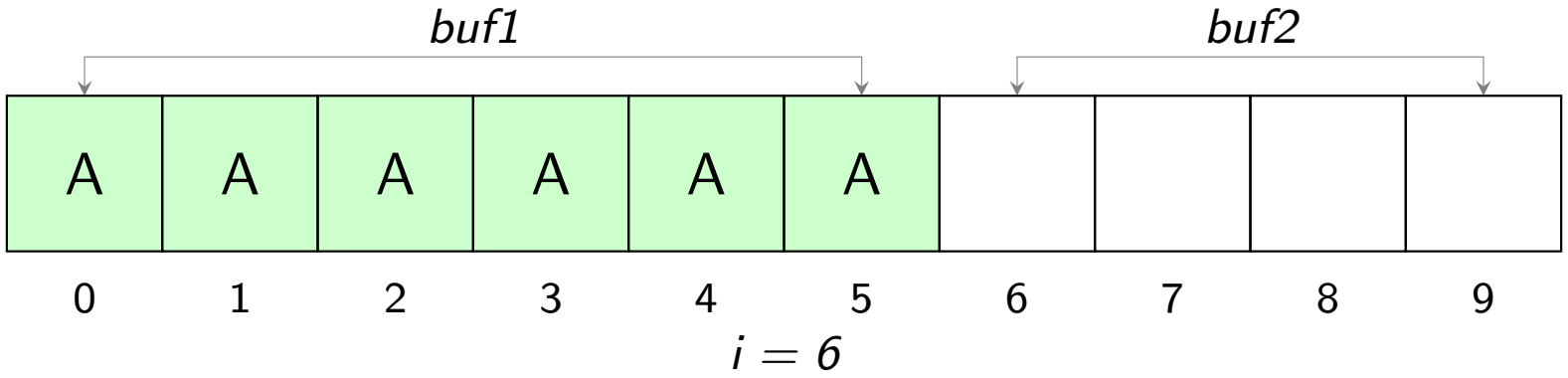
Esempio di *overflow*

```
1   /* overflow.c */
2
3   int main()
4   {
5     int i;
6     char buf2[4];
7     char buf1[6];
8
9     for(i=0; i<10; i++)
10      buf1[i] = 'A';
11
12    return 0;
13  }
```



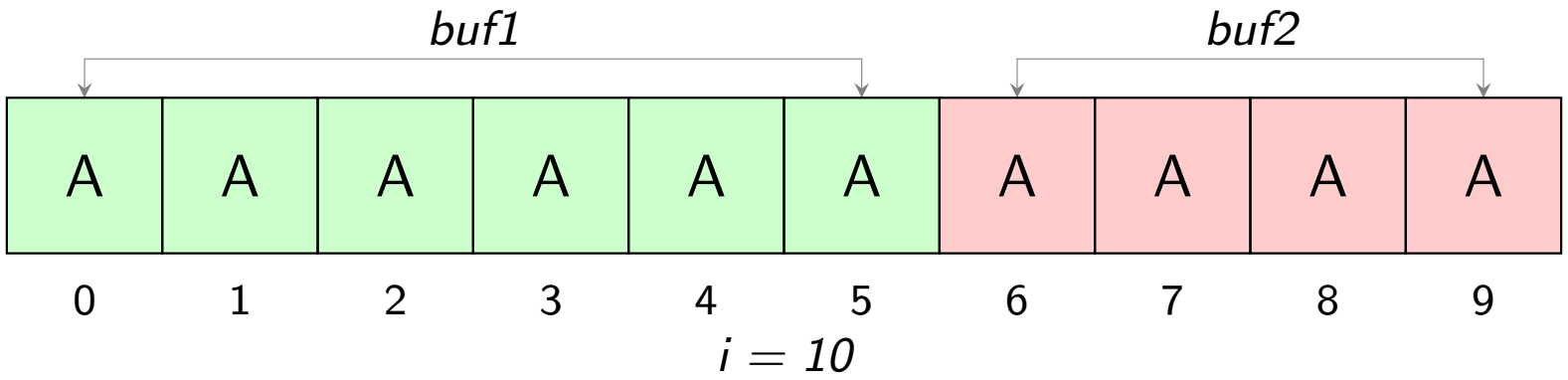
Esempio di *overflow*

```
1   /* overflow.c */
2
3   int main()
4   {
5     int i;
6     char buf2[4];
7     char buf1[6];
8
9     for(i=0; i<10; i++)
10      buf1[i] = 'A';
11
12    return 0;
13  }
```



Esempio di *overflow*

```
1   /* overflow.c */
2
3   int main()
4   {
5     int i;
6     char buf2[4];
7     char buf1[6];
8
9     for(i=0; i<10; i++)
10      buf1[i] = 'A';
11
12    return 0;
13  }
```



Perché succede?

Due ragioni:

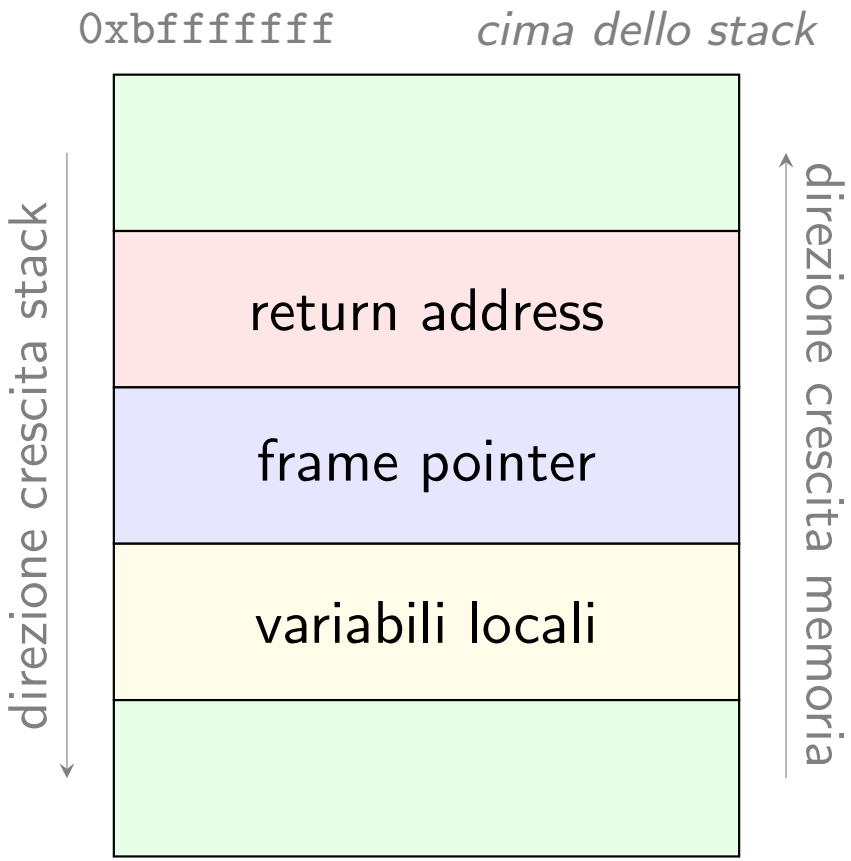
- in alcuni linguaggi (e.g. C) mancano *boundary check*
- *channeling problem*: informazioni di controllo e dati nello stesso canale.

Perché succede?

Negli stack-based buffer overflow si ha:

- *canale*: stack.
- *dati*: parametri e variabili locali delle funzioni
- *informazioni di controllo*: code pointer (e.g., *return address*) o frame pointer

Layout stack



Programma vulnerabile

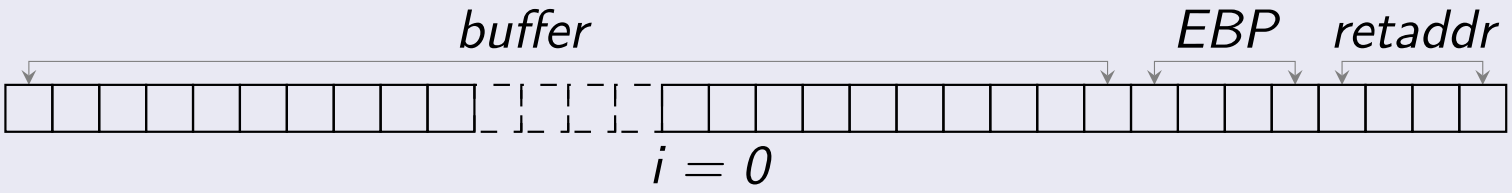
```
1   /* vuln.c */
2
3   void foobar(char *str)
4   {
5     char buffer[100];
6     strcpy(buffer, str);
7   }
8
9   int main(int argc, char **argv)
10  {
11    if(!argv[1]) return -1;
12    foobar(argv[1]);
13    exit(0);
14  }
```

Effetti sullo stack di `strcpy(buffer, str)`, supponendo che `argv[1]` sia formato da 108 caratteri `'A'`:

Programma vulnerabile

```
1   /* vuln.c */
2
3   void foobar(char *str)
4   {
5     char buffer[100];
6     strcpy(buffer, str);
7   }
8
9   int main(int argc, char **argv)
10  {
11    if(!argv[1]) return -1;
12    foobar(argv[1]);
13    exit(0);
14  }
```

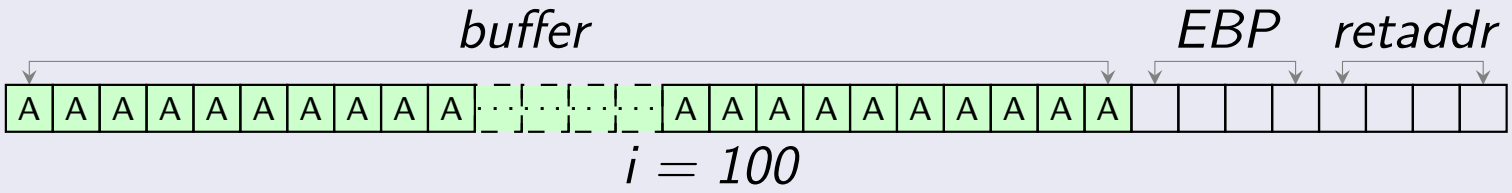
Effetti sullo stack di `strcpy(buffer, str)`, supponendo che `argv[1]` sia formato da 108 caratteri `'A'`:



Programma vulnerabile

```
1   /* vuln.c */
2
3   void foobar(char *str)
4   {
5     char buffer[100];
6     strcpy(buffer, str);
7   }
8
9   int main(int argc, char **argv)
10  {
11    if(!argv[1]) return -1;
12    foobar(argv[1]);
13    exit(0);
14  }
```

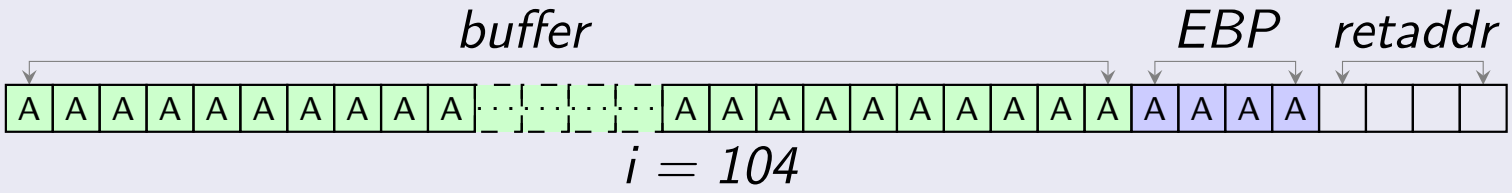
Effetti sullo stack di `strcpy(buffer, str)`, supponendo che `argv[1]` sia formato da 108 caratteri `'A'`:



Programma vulnerabile

```
1   /* vuln.c */
2
3   void foobar(char *str)
4   {
5     char buffer[100];
6     strcpy(buffer, str);
7   }
8
9   int main(int argc, char **argv)
10  {
11    if(!argv[1]) return -1;
12    foobar(argv[1]);
13    exit(0);
14  }
```

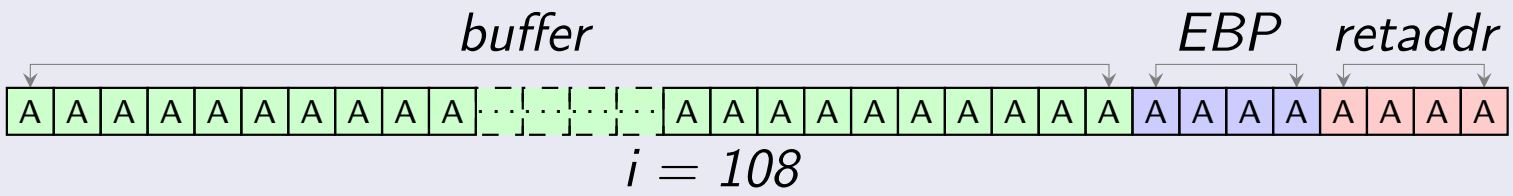
Effetti sullo stack di `strcpy(buffer, str)`, supponendo che `argv[1]` sia formato da 108 caratteri `'A'`:



Programma vulnerabile

```
1   /* vuln.c */
2
3   void foobar(char *str)
4   {
5     char buffer[100];
6     strcpy(buffer, str);
7   }
8
9   int main(int argc, char **argv)
10  {
11    if(!argv[1]) return -1;
12    foobar(argv[1]);
13    exit(0);
14  }
```

Effetti sullo stack di `strcpy(buffer, str)`, supponendo che `argv[1]` sia formato da 108 caratteri `'A'`:

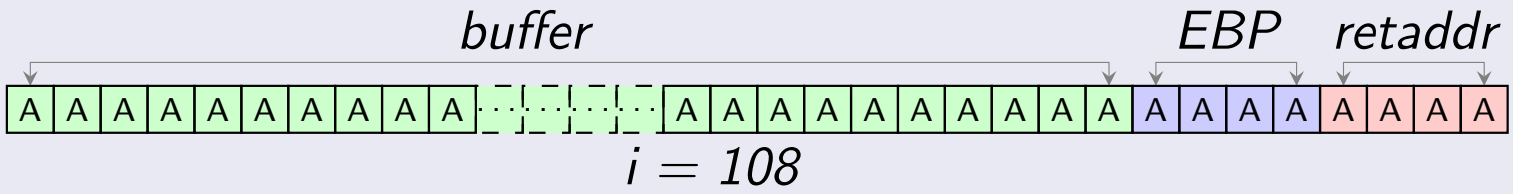


Programma vulnerabile

```
1   /* vuln.c */
2
3   void foobar(char *str)
4   {
5     char buffer[100];
6     strcpy(buffer, str);
7   }
8
9   int main(int argc, char **argv)
10  {
11    if(!argv[1]) return -1;
12    foobar(argv[1]);
13    exit(0);
14  }
```

Dove ritornerà il processo al termine dell'esecuzione della funzione `foobar()`?

Effetti sullo stack di `strcpy(buffer, str)`, supponendo che `argv[1]` sia formato da 108 caratteri `'A'`:



Esempio

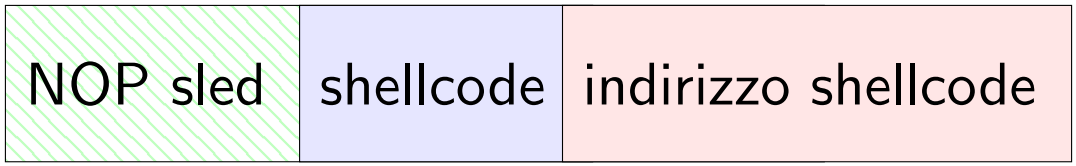
```
laser@laser:~/esempi$ gcc -o vuln vuln.c
laser@laser:~/esempi$ objdump -d vuln

vuln:     file format elf32-i386

Disassembly of section .init:
...
08048394 <foobar>:
 8048394:       55                      push   %ebp
 8048395:       89 e5                   mov    %esp,%ebp
 8048397:       83 ec 78                sub    $0x78,%esp
 804839a:       8b 45 08                mov    0x8(%ebp),%eax
 804839d:       89 44 24 04             mov    %eax,0x4(%esp)
 80483a1:       8d 45 9c                lea    0xffffff9c(%ebp),%eax
 80483a4:       89 04 24                mov    %eax,(%esp)
 80483a7:       e8 28 ff ff ff          call   80482d4 <strcpy@plt>
 80483ac:       c9                      leave
 80483ad:       c3                      ret
...
laser@laser:~/esempi$ ./vuln $(python -c 'print "X"*100 + "Y"*4 + "\x01\x02\x03\x04"')
Segmentation fault
...@laser:~/...$ strace -i ./vuln $(python -c 'print "X"*100 + "Y"*4  + "\x01\x02\x03\x04";')
...
[04030201] --- SIGSEGV (Segmentation fault) @ 0 (0) ---
```

Obiettivo dell'attacco

Dirottare il flusso di esecuzione verso codice "iniettato" dall'attaccante. Per fare questo bisogna:

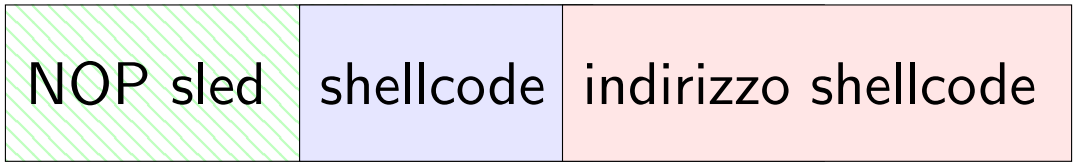
- iniettare il codice da eseguire (*shellcode*) all'interno di zone di memoria scrivibili del processo (stack, `.data`, heap, ...)
- modificare un *code pointer* del processo (e.g., *return address*) in modo da poter dirottare il flusso del programma verso il codice iniettato

Componenti *injection vector*



- *shellcode*: insieme di istruzioni che rappresentano il codice da eseguire (e.g. `execve("/bin/sh")`)
- *indirizzo dello shellcode*: indirizzo del buffer in memoria (e.g.. sullo stack) contenente lo *shellcode*
- *NOP sled* (opzionale): istruzioni (`nop`) che "non fanno nulla", ma aumentano la probabilità di guessing dell'indirizzo del buffer

Componenti *injection vector*



- *shellcode*: insieme di istruzioni che rappresentano il codice da eseguire (e.g. `execve("/bin/sh")`)
- *indirizzo dello shellcode*: indirizzo del buffer in memoria (e.g.. sullo stack) contenente lo *shellcode*
- *NOP sled* (opzionale): istruzioni (`nop`) che "non fanno nulla", ma aumentano la probabilità di guessing dell'indirizzo del buffer

In che formato sono le istruzioni?

Esercizi

0x{01,02,03,04,05}

Fare in modo che l'input fornito ai seguenti programmi faccia stampare la stringa "you win!"

`http://security.di.unimi.it/sicurezza1516/slides/stack1.c`

Memorandum

```
studente@sec1:~$ sudo echo 0 > /proc/sys/kernel/randomize_va_space
-----------------------------------------------------------

studente@sec1:~$ gcc -m32 -fno-stack-protector -zexecstack  -o stack1 stack1.c
-----------------------------------------------------------
studente@sec1:~$ objdump -d stack1 | less
...
-----------------------------------------------------------
studente@sec1:~$ perl -e 'print "\x41\x41\x41\x41" . "\x42"x4'
AAAABBBB
-----------------------------------------------------------
studente@sec1:~$ strace -fi ./stack1
...
```

Bibliografia

- Smashing The Stack For Fun And Profit (Aleph One) – *Phrack Magazine*
- "PC Assembly Language", Paul a Carter, `http://www.drpaulcarter.com/pcasm/pcasm-book.pdf.gz`