Programmazione

Chiarimenti Elaborato 7

Andrea Piroddi

Dipartimento di Informatica, Scienza e Ingegneria

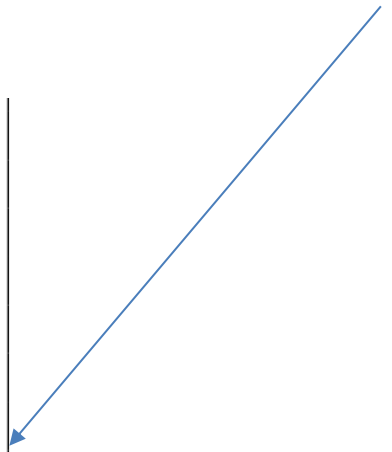
Chiarimenti Elaborato 7

```
#define IS_MINE(board,i,j) (board[i][j] == UNKN_MINE || board[i][j] == FLAG_MINE || board[i][j] == MINE)
#define IS_FLAG(board,i,j) (board[i][j] == FLAG_MINE || board[i][j] == FLAG_FREE)
#define IS_UNKNOW(board,i,j) (board[i][j] == UNKN_MINE || board[i][j] == UNKN_FREE)

#define SET_LIMIT int i0 = (i == 0) ? 0 : (i - 1); int j0 = (j == 0) ? 0 : (j - 1); int i1 = (i == (rows - 1)) ? i : (i + 1); int j1 = (j == (cols - 1)) ? j : (j + 1);


#define ABS(x,y) ((x) > (y) ? (x)-(y) : (y)-(x))

static int setup_unknown(int board[][GAME_COLS], unsigned int rows, unsigned int cols, int i, int j);
static int count_flagged(int board[][GAME_COLS], unsigned int rows, unsigned int cols, int i, int j);
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Chiarimenti Elaborato 7

Nel file HEADER



```
enum field {
  C0,           // The cell is sorrounded by 0 mines
  C1,           // The cell is sorrounded by 1 mine
  C2,           // The cell is sorrounded by 2 mines
  C3,           // The cell is sorrounded by 3 mines
  C4,           // The cell is sorrounded by 4 mines
  C5,           // The cell is sorrounded by 5 mines
  C6,           // The cell is sorrounded by 6 mines
  C7,           // The cell is sorrounded by 7 mines
  C8,           // The cell is sorrounded by 8 mines
  UNKN_FREE,    // The cell is not displayed and it is not a mine
  UNKN_MINE,    // The cell is not displayed and it is a mine
  FLAG_FREE,    // The cell is flagged and it is not a mine
  FLAG_MINE,    // The cell is flagged and it s a mine
  MINE          // The cell is a mine
};
```

Chiarimenti Elaborato 7

```
void random_board(int board[][GAME_COLS], unsigned int rows, unsigned int cols, unsigned int i, unsigned int j, unsigned int num_mines)
{
        unsigned int a, b, c=0;

        srand(time(NULL));

        for(a = 0; a < rows; a++)
                ...
                        board[a][b] = UNKN_FREE;

        while(c != num_mines) {
                a = rand() % rows;
                ...
                if(board[a][b] == UNKN_FREE && (ABS(a,i) > 1 || ...)) {
                        board[a][b] = UNKN_MINE;
                        c++;
                }
        }
}
```

- `random_board()`: posiziona in modo casuale `num_mines` nella matrice, facendo attenzione a non posizionare nessuna mina né nella posizione `i,j` né attorno alla posizione `i,j` . Al termine della chiamata:
  - ogni posizione della matrice deve essere o `UNKN_FREE` oppure `UNKN_MINE`,
  - ci devono essere esattamente `num_mines` posizioni con valore `UNKN_MINE`,
  - nessuna posizione attorno ad `i,j` (inclusa) deve essere `UNKN_MINE`.

Chiarimenti Elaborato 7

- `flag_board()`: aggiunge un flag (bandiera) nella posizione `i,j`:
  - se la posizione `i,j` è già stata mostrata non fa nulla e ritorna 0,
  - se la posizione `i,j` è già "flaggata", rimuove il flag (`FLAG_FREE` diventa `UNKN_FREE` e `FLAG_MINE` diventa `UNKN_MINE`) e ritorna -1,
  - se la posizione `i,j` non è "flaggata" aggiunge un flag (`UNKN_FREE` diventa `FLAG_FREE` e `UNKN_MINE` diventa `FREE_MINE`) e ritorna 1.

```
int flag_board(int board[][GAME_COLS], unsigned int rows, unsigned int cols, unsigned int i, unsigned int
j)
{
        int c = 0;
        switch(board[i][j]) {
                case UNKN_FREE:
                        board[i][j] = FLAG_FREE; c = 1;  break;
                case UNKN_MINE:
                        …
                case FLAG_MINE:
                        board[i][j] = UNKN_MINE; c = -1; break;
                case FLAG_FREE:
                        …
                default:                    c =  0; break;
        }
        return c;
}
```

Chiarimenti Elaborato 7

- `display_board()`: mostra il contenuto nella posizione `i,j`
  - se la posizione `i,j` è già stata mostrata oppure è "flaggata" non fa nulla e ritorna 0,
  - se la posizione `i,j` non è già stata mostrata e contiene una mina, posiziona la costante `MINE` in `i,j` e ritorna -1,
  - se la posizione `i,j` non è già stata mostrata e non contiene una mina, posiziona in `i,j` la costante enumerativa che indica il numero di mine che circondano la posizione `i,j` (i.e. `C0, ..., C8`). Se la posizione `i,j` non ha mine attorno (i.e. diventa `C0`) allora tutte le posizioni non scoperte attorno ad `i,j` sono rivelate. Lo stesso algoritmo viene ripetuto fintanto che sono scoperte posizioni marcate con `C0`. La funzione ritorna il numero complessivo di posizioni rivelate.

```
int display_board(int board[][GAME_COLS], unsigned int rows, unsigned int cols, unsigned int i, unsigned int j)
{
        if(board[i][j] == UNKN_MINE) {
                board[i][j] = MINE;
                return -1;
        } else {
                return …;
        }
}
```

Chiarimenti Elaborato 7

- **`expand_board()`**: mostra il contenuto delle posizioni ancora non svelate attorno ad `i,j`. Il comportamento è esattamente lo stesso della **`display_board()`**, valore di ritorno compreso, ma applicato a tutte le celle non svelate attorno ad `i,j`. Casi in cui la funzione non fa nulla e ritorna 0:
  - se la posizione `i,j` non è già stata mostrata,
  - se la posizione `i,j` non contiene attorno un numero di celle flaggate pari al numero mostrato in `i,j`.

```
int expand_board(int board[][GAME_COLS], unsigned int rows, unsigned int cols, unsigned int i, unsigned int j)
{
        int a, b, ndis, tndis=0;
        SET_LIMIT

        if (board[i][j] > C0 && board[i][j] < C8) {
                if (count_flagged(…) == board[i][j]) {
                        for (a = i0; …)
                                for (b = j0; …)
                                        if (…)
                                                if (IS_UNKNOW(…)) {
                                                        …
                                                }
                        return tndis;
                }
        }
        return 0;
}
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Chiarimenti Elaborato 7

```
static int count(int board[][GAME_COLS], unsigned int rows, unsigned int cols, int i, int j) {
        int a, b, c = 0;
        SET_LIMIT

        for (…)
                for (…)
                        if (a != i || b != j)
                                c += IS_MINE(…);

        return c;
}
```

Chiarimenti Elaborato 7

```
static int count_flagged(int board[][GAME_COLS], unsigned int rows, unsigned int cols, int i, int j) {
        int a, b, c = 0;
        SET_LIMIT

        for (…)
                for (…)
                        if (a != i || b != j)
                                c += IS_FLAG(…);

        return c;
}
```

Chiarimenti Elaborato 7

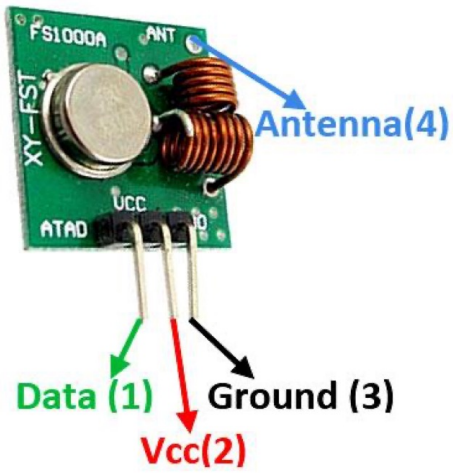
```
static int setup_unknown(int board[][GAME_COLS], unsigned int rows, unsigned int cols, int i, int j) {
        int s = 0;
        if(i >= 0 && i < rows && j >=0 && j < cols && board[i][j] == UNKN_FREE) {
                int c = count(board,rows,cols,i,j);

                switch(c) {
                        case 0: board[i][j] = C0;
                                s += setup_unknown(board,rows,cols,i-1,j);
                                …
                                break;
                        case 1: board[i][j] = C1; break;
                        …
                }
                return s+1;
        }
        return s;
}
```

Embedded C – TX/RX

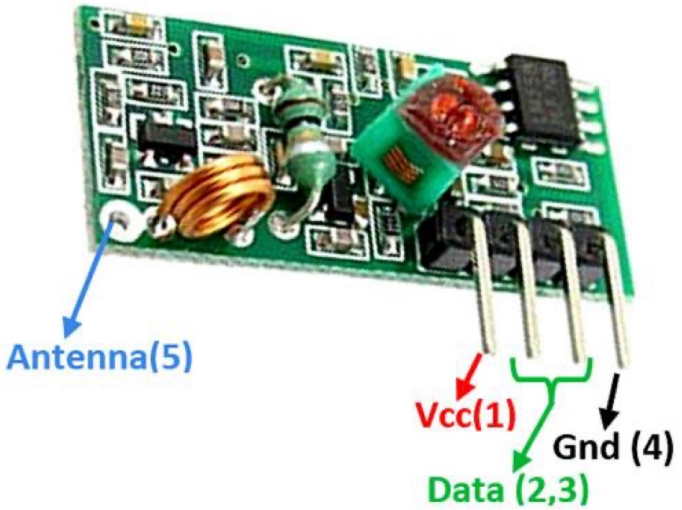
ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

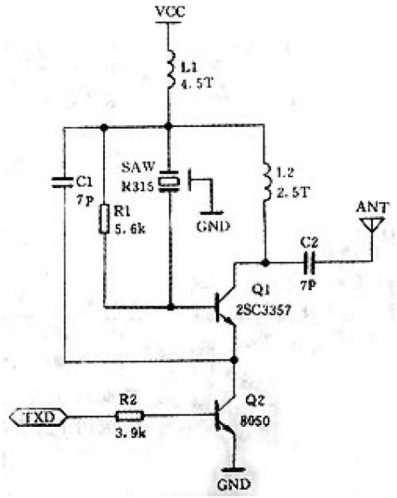
Embedded C – TX/RX

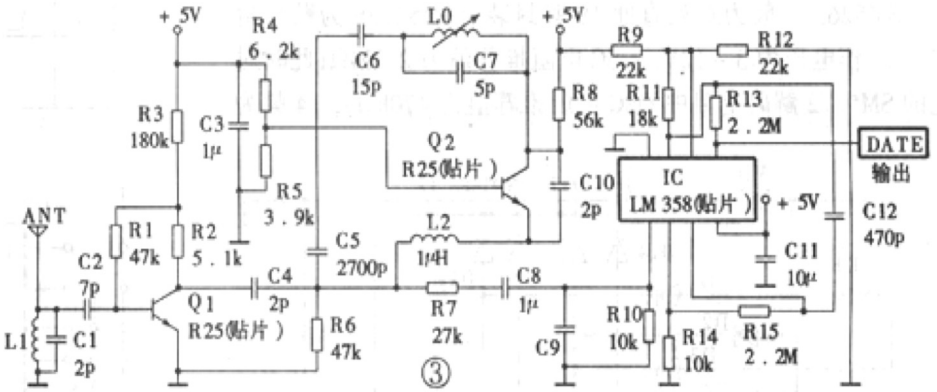
TX: MX-FS-03V



RX: MX-05V







Embedded C – TX/RX

Trasmettitore MX-FS-03V

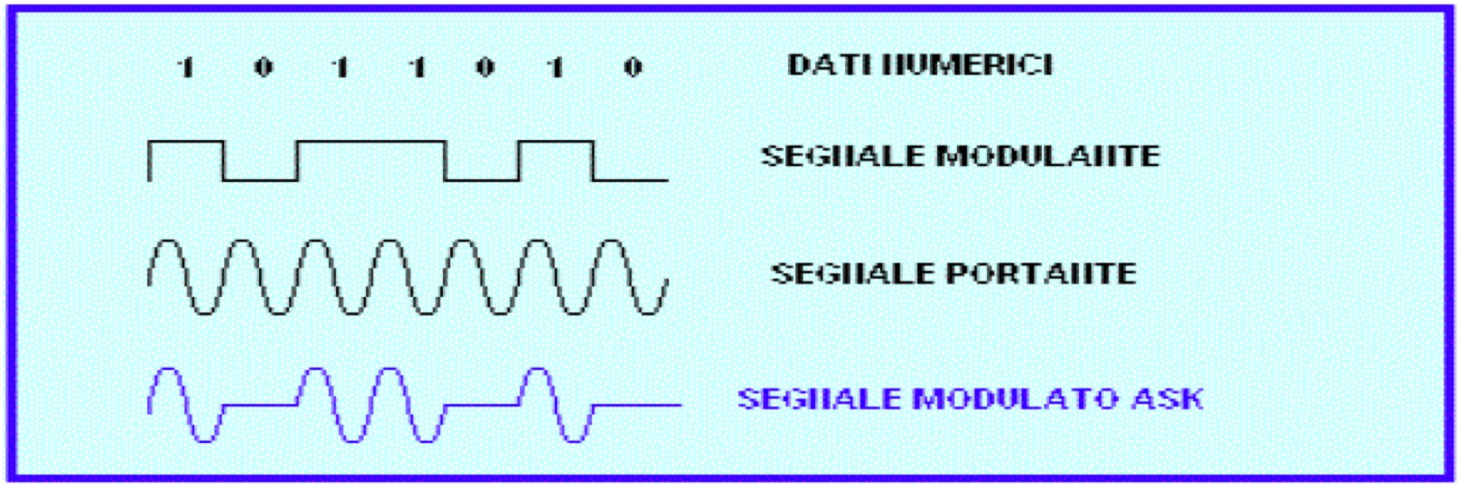
- Alimentazione : da 3.5V a 12V, la potenza varia a seconda dell'alimentazione.
- Potenza : 10mW circa.
- Distanza raggiunta dal segnale : da 20 a 200 metri in campo aperto (20m a 3.5V e 200m a 12V)
- Dimensioni : 19mm x 19mm
- Velocità : 4kbit/s

Ricevitore MX-05V

- Alimentazione : 5v
- Corrente richiesta : 4mA
- Dimensioni : 30mm x 14mm x 7mm

Amplitude Shift Keying Modulation (ASK)

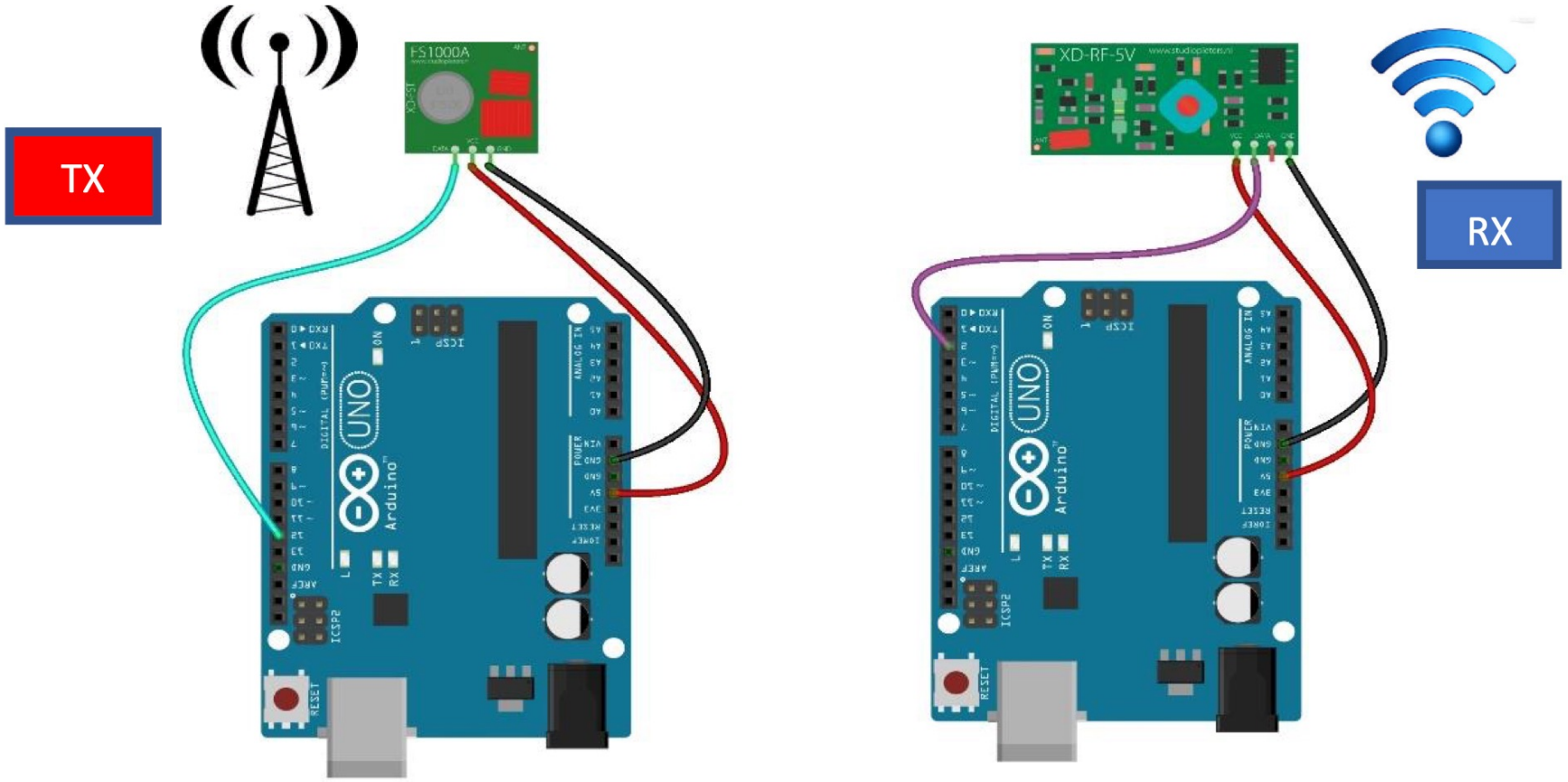
Nella modulazione **ASK** l'ampiezza della portante sinusoidale viene fatta variare in correlazione al segnale digitale modulante. Nel caso più semplice e più comune (detto anche On Off Keying) in corrispondenza dello zero logico il segnale modulato ha ampiezza zero o prossima allo zero, mentre in corrispondenza dell'uno logico ha ampiezza pari a quella della portante non modulata.



In assenza di portante il segnale di disturbo presente in uscita del ricevitore, va ignorato altrimenti potrà essere causa di letture sbagliate e/o di instabilità del software che lo gestisce. Una possibile soluzione consiste nel cominciare a leggere il messaggio solamente dopo la ricezione di un token di alcuni byte scelti dal programmatore, questo permette di ignorare ogni pacchetto che non cominci con una sequenza ben definita, sequenza che se abbastanza complessa, permette di eliminare del tutto il difetto del disturbo in assenza di portante.

Embedded C – TX/RX

PARTE 3: ESEMPIO DI TRASMISSIONE DATI TRA DUE ARDUINO



Embedded C – TX

Questo è il codice da caricare sull'Arduino che funge da Trasmittente (TX)

```
#include <RH_ASK.h>
#include <SPI.h> // Not actually used but needed to compile

RH_ASK driver;

void setup()
{
    Serial.begin(9600);     // Debugging only
    if (!driver.init())
         Serial.println("init failed");
}

void loop()
{    char cstr[0],*msg1;
     char *msg = "Z";
for (int i=0; i<100; i++){
    msg1=itoa(i,cstr,10);
    driver.send((uint8_t *)msg1, strlen(msg1));
    driver.waitPacketSent();
    delay(200);
  }
  delay(15000);
}
```

itoa() — Convert int into a string

Format

```
#define _OPEN_SYS_ITOA_EXT
#include <stdlib.h>

char * itoa(int n, char * buffer, int radix);
```

General description

The itoa() function coverts the integer n into a character string. The string is placed in the buffer passed, which must be large enough to hold the output. The radix values can be OCTAL, DECIMAL, or HEX. When the radix is DECIMAL, itoa() produces the same result as the following statement:

```
(void) sprintf(buffer, "%d", n);
```

with buffer the returned character string. When the radix is OCTAL, itoa() formats integer n into an unsigned octal constant. When the radix is HEX, itoa() formats integer n into an unsigned hexadecimal constant. The hexadecimal value will include lower case abcdef, as necessary.

Returned value

String pointer (same as buffer) will be returned. When passed a non-valid radix argument, function will return NULL and set errno to EINVAL.

Embedded C – RX

Questo è lo sketch da caricare su Arduino che funge da Ricevente (RX):

```
#include <RH_ASK.h>
#include <SPI.h>
RH_ASK rf_driver;
int count=0;
int Packet_Loss =0;
int Packet_Loss_perc = 0;
int A[5];
int i;

void setup() {
  // put your setup code here, to run once:
rf_driver.init();

Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
uint8_t buf[2];
uint8_t buflen = sizeof(buf);
while(count<5){
 if (rf_driver.recv(buf, &buflen)) // Non-blocking
    {
     Serial.print("count:");
     Serial.println(count);
     A[count]=(int)atoi((char*)buf);
     Serial.println(A[count]);
     count++;
     }
   } //end of while
  for(i=1;i<5;i++){

     Packet_Loss = Packet_Loss+(A[i]-1-A[i-1]);

   }

Packet_Loss_perc=Packet_Loss/(A[4]-A[0]);

if ( Packet_Loss <0) {

 Serial.print("ERROR");

}

else {

 Serial.print("Perdita dei pacchetti:");

 //Serial.println(Packet_Loss);

 Serial.println(Packet_Loss_perc);

 }

 count=0;

 Packet_Loss =0;
}
```

Buffer Overflow

Cosa può succedere se si provoca un Buffer Overflow?

Un qualunque errore, come l'assenza di specifici controlli, può, in determinate condizioni, indurre il programma ad eseguire operazioni non previste e non gradite.

L'Hacker se riesce ad individuare queste problematiche scopre una vulnerabilità del programma che può sfruttare a proprio vantaggio, come ad esempio **diventare root del sistema**.

Per tale motivo stanno nascendo nuovi linguaggi che permettono di ottenere livelli di sicurezza molto più avanzati sul fronte della gestione della memoria, tra questi **GO di Google** e **RUST della Mozilla Foundation**.

Cosa è un BUFFER?

E’ uno spazio di memoria fisica allocato nella memoria RAM dove vengono memorizzati temporaneamente dei dati durante l’esecuzione di un programma (runtime).

In alcuni casi un buffer può essere una memoria di appoggio per il trasferimento di dati da un dispositivo ad un altro con lo scopo di migliorare le prestazioni. Si pensi per esempio allo streaming video in cui il player scarica una quota parte del video in una memoria di appoggio (buffer) eseguendone lo streaming da esso.

Durante l’esecuzione da un lato c’è il lettore che preleva i dati dal buffer, dall’altro il collegamento assicura che nuovi dati continuino ad afferire al buffer.

In questo modo cali di velocità della connessione o micro interruzioni del servizio non influiranno sulle prestazioni del flusso video che apparirà sempre fluido.

Cosa è un BUFFER?

Nel nostro caso concentriamo la nostra attenzione sui buffer allocati da un programma in esecuzione in una zona di memoria RAM pensata per contenere dati con dimensione prefissata; per esempio un array di interi.

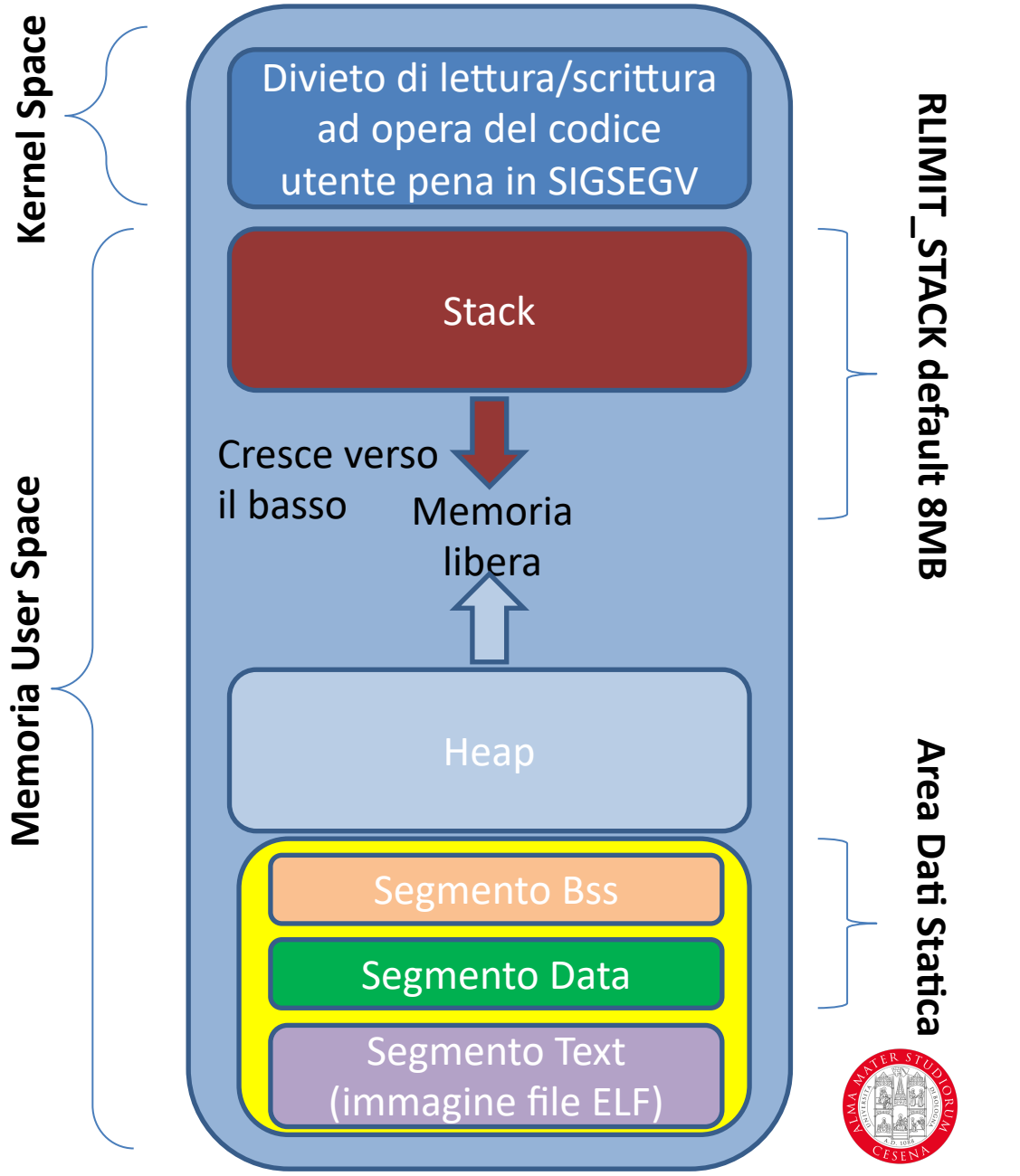
Quando al suddetto buffer si inviano più dati di quelli che può contenere si ha il cosiddetto **BUFFER OVERFLOW**, un «traboccamento» dei dati dal buffer che può portare a condizioni di vulnerabilità se l'overflow non viene in qualche modo controllato/gestito.

Come è fatta la MEMORIA?

Per chiarire meglio come si presenta un buffer overflow, partiamo dalla descrizione della memoria.

La memoria allocata dal sistema operativo per un processo (sinonimo di programma nella nostra accezione) può essere suddivisa in tre zone:

- Un'area **TEXT**
- Un'area dati **statica**
- Un'area dati **dinamica**



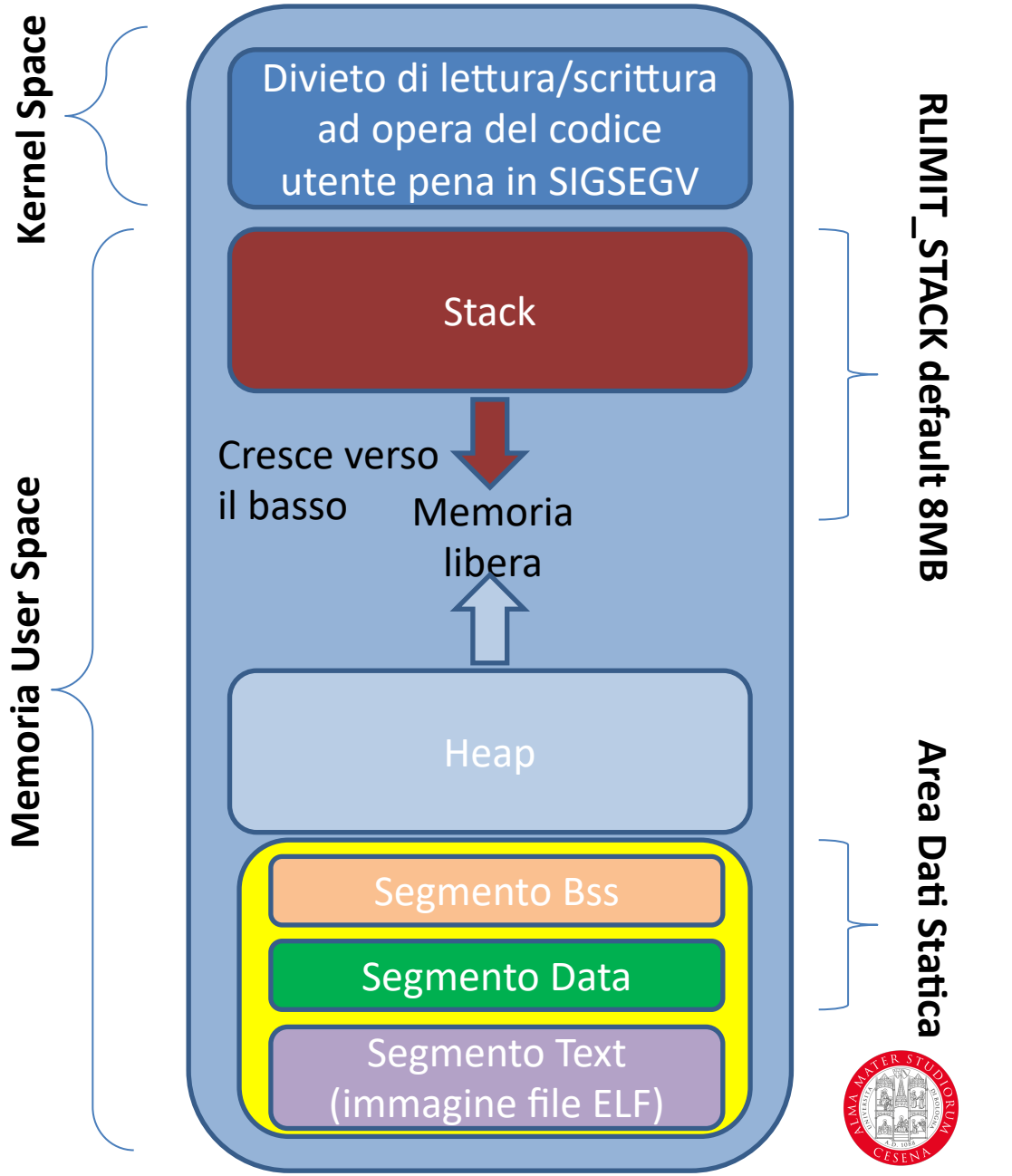
Area TEXT

L'area **TEXT** contiene le istruzioni Assembly del programma in esecuzione.

E' una zona di memoria a cui si può accedere in **SOLA lettura**, infatti memorizzando le istruzioni del programma non si può permettere a chiunque di cambiare casualmente durante l'esecuzione.

Qualsiasi tentativo di scrittura in questo segmento porta irrimediabilmente a un segnale **SIGSEGV** (SIGnal SEGmentation Violation), riferimento di memoria non valido con conseguente interruzione del programma ad opera del sistema operativo.

L'area TEXT, a causa della sua modalità in sola lettura, non potrà essere vulnerabile ad attacchi di tipo buffer overflow.



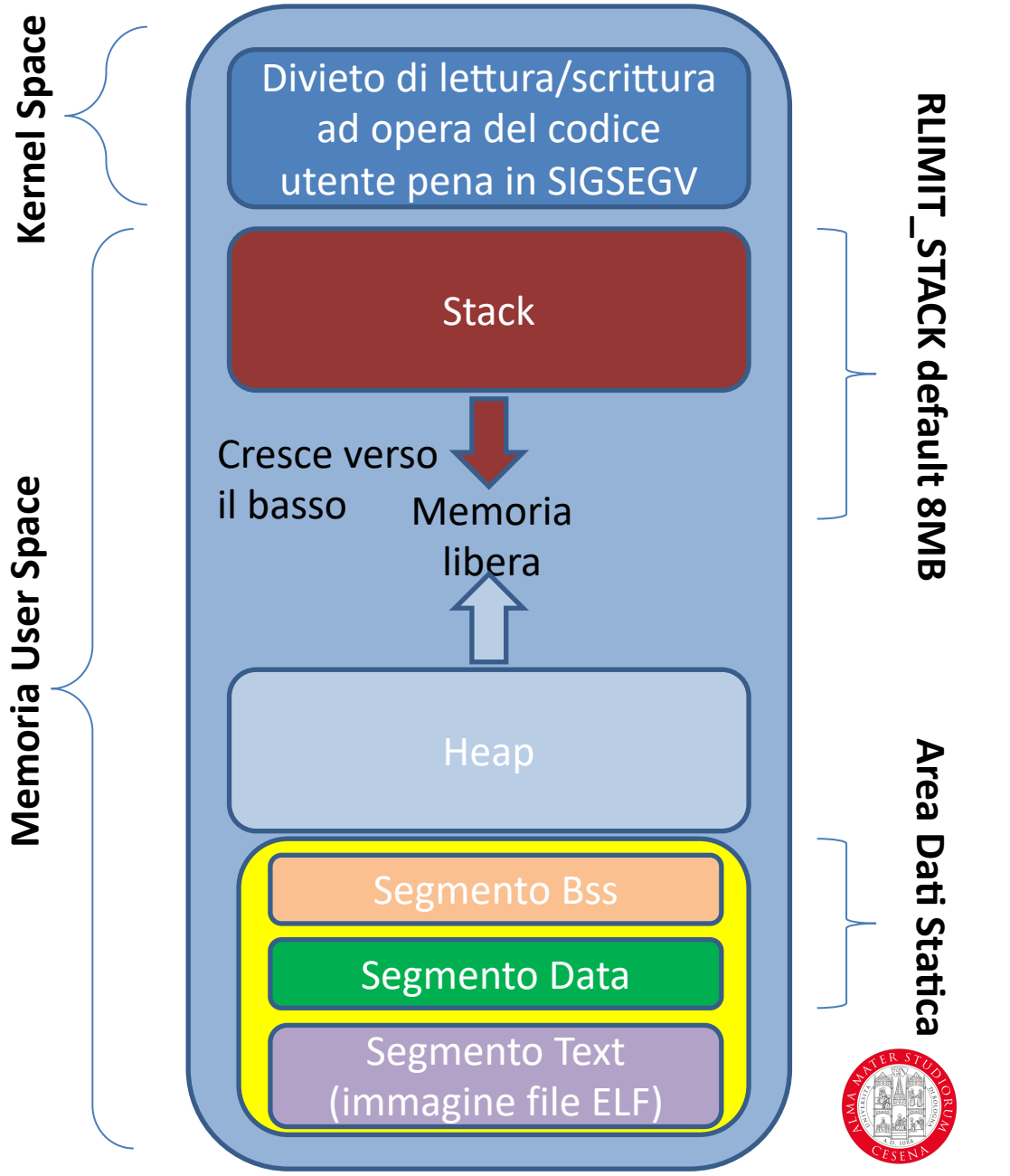
**per approfondire i segnali e il loro significato dare il comando «`man signal`»*

Area Dati Statica

L'area dati statica è suddivisa in due segmenti:

- **Data**
- **Bss** (dal nome di un vecchio operatore Assembler dal nome Block Started by Symbol)

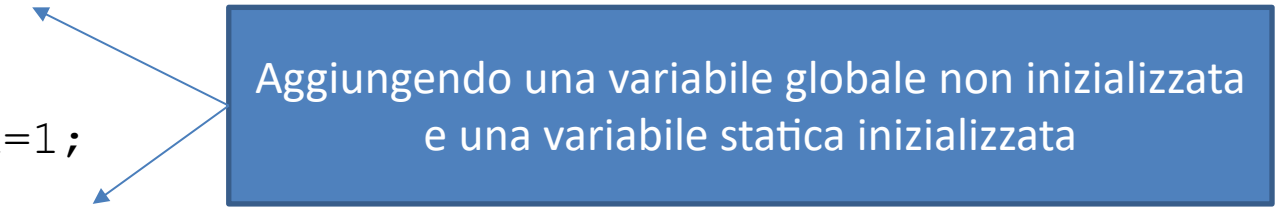
Entrambe contengono dati statici ma mentre l'area Data contiene variabili globali inizializzate, il segmento Bss ospita variabili globali non inizializzate



Area Dati Statica

Per rendersi conto di quanto appena detto è sufficiente fare un confronto con poche righe di codice:

```
#include <stdio.h>
double var_globale;
int main(void)
{
static int statica=1;
    return 0;
}
```



Dopo aver salvato le righe precedenti su due file di nome qualsiasi, esempio1.c ed esempio2.c li andiamo a compilare con

gcc –o esempio1 esempio1.c

gcc –o esempio2 esempio2.c

Gli output risulteranno eseguibili in formato **ELF (Executable and Linkable Format).**

Dando il comando **`size esempio1`** si ottiene un output del tipo:

Area Dati Statica

```
size esempio1
   text    data     bss     dec     hex filename
    800     252      16    1068     42c esempio1
```

```
#include <stdio.h>
double var_globale;
int main(void)
{
   return 0;
}
```

Dando il comando **`size esempio2`** si ottiene un output del tipo:

```
size esempio2
   text    data     bss     dec     hex filename
    800     256       8    1064     428 esempio2
```

```
#include <stdio.h>
int main(void)
{
static int statica=1;
   return 0;
}
```

Area Dati Statica

Come si può vedere le differenze si riflettono nell'ampiezza dei segmenti Bss e Data.

Infatti nel primo codice viene aggiunta una variabile globale di tipo ***double*** che in C occupa 8 byte e non essendo inizializzata viene immagazzinata nel segmento **Bss** il quale passa da 8 a 16 byte.

Nel secondo codice è stata aggiunta una variabile statica di tipo ***intero*** (4 byte in C) che viene inizializzata pertanto verrà immagazzinata in **Data** che infatti passa da 252 a 256 byte.

La durata di vita di questa parte di memoria coincide con la vita del programma e l'ampiezza è fissata poiché è nota al momento della compilazione.

Aree Vulnerabili

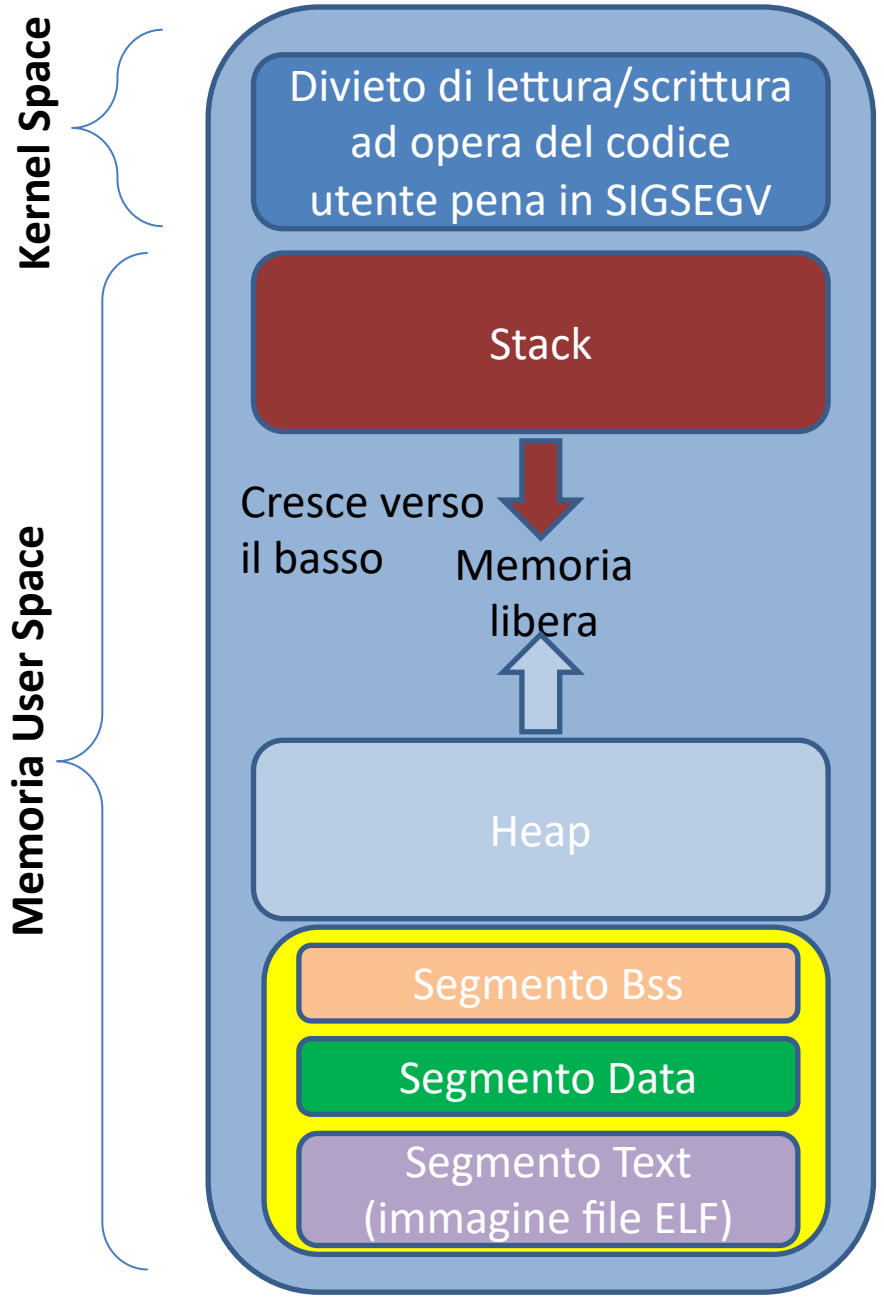
L'ultimo segmento di memoria, l'area di dati dinamica, è quella che se non adeguatamente controllata origina il problema dei buffer overflow.

E' divisa in due sezioni:

- **STACK**
- **HEAP**

Lo Stack è un'area di memoria contigua.

Ad esempio in GNU/Linux, aderendo alle specifiche System V ABI (Application Binary Interface) sulle architetture x86-64 (AMD64), lo stack cresce verso gli indirizzi più bassi.

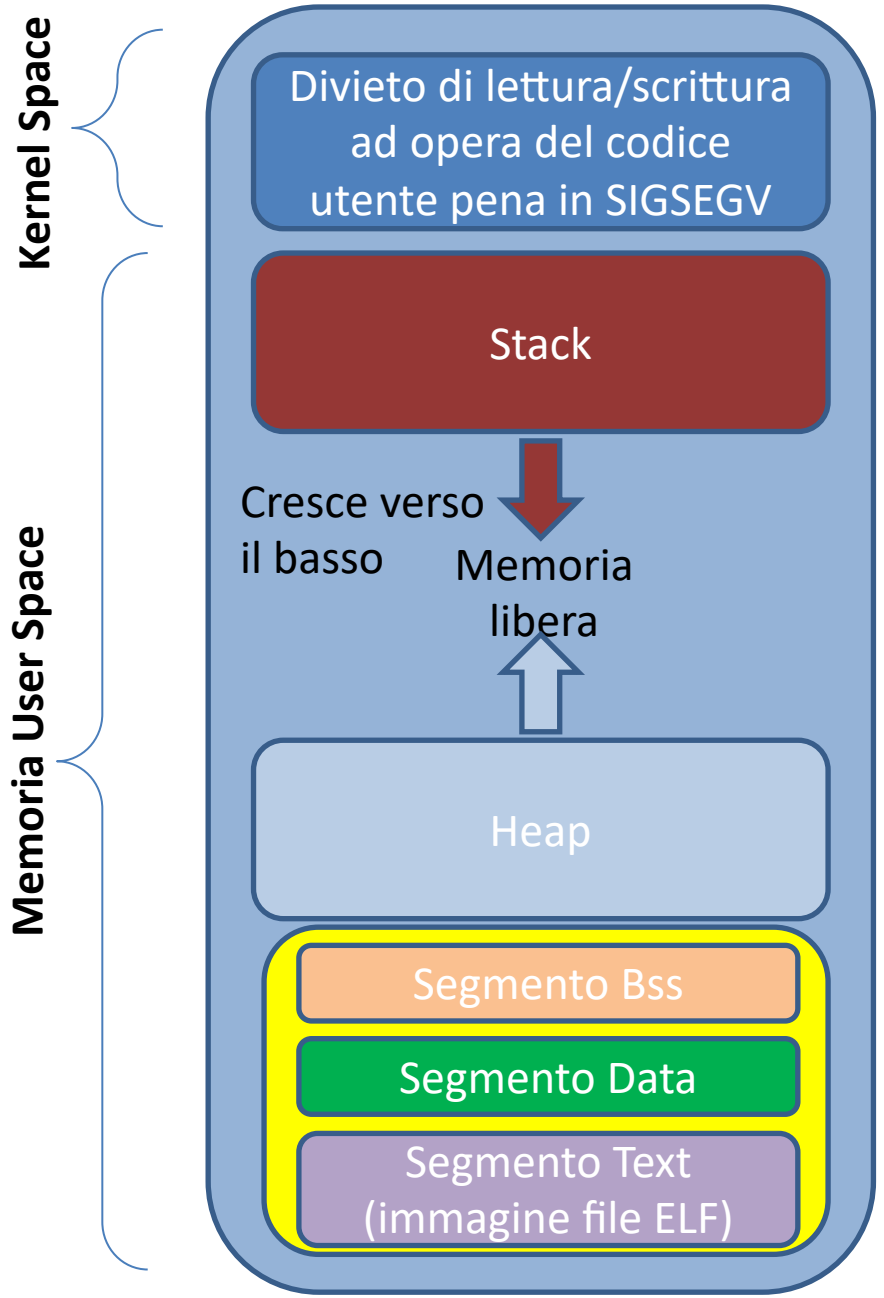


Aree Vulnerabili

Ossia gli indirizzi più piccoli identificano le locazioni più alte dello stack gestito in modalità **LIFO** (Last In First Out) con opportuni registri.

Questa porzione di memoria viene usata per allocare dinamicamente a tempo di esecuzione (runtime) elementi come le variabili locali delle funzioni, i parametri nonché l'indirizzo di ritorno a partire dal quale riprendere l'esecuzione del programma una volta terminata la funzione.

La durata di vita dello stack è pertanto temporanea.



Aree Vulnerabili

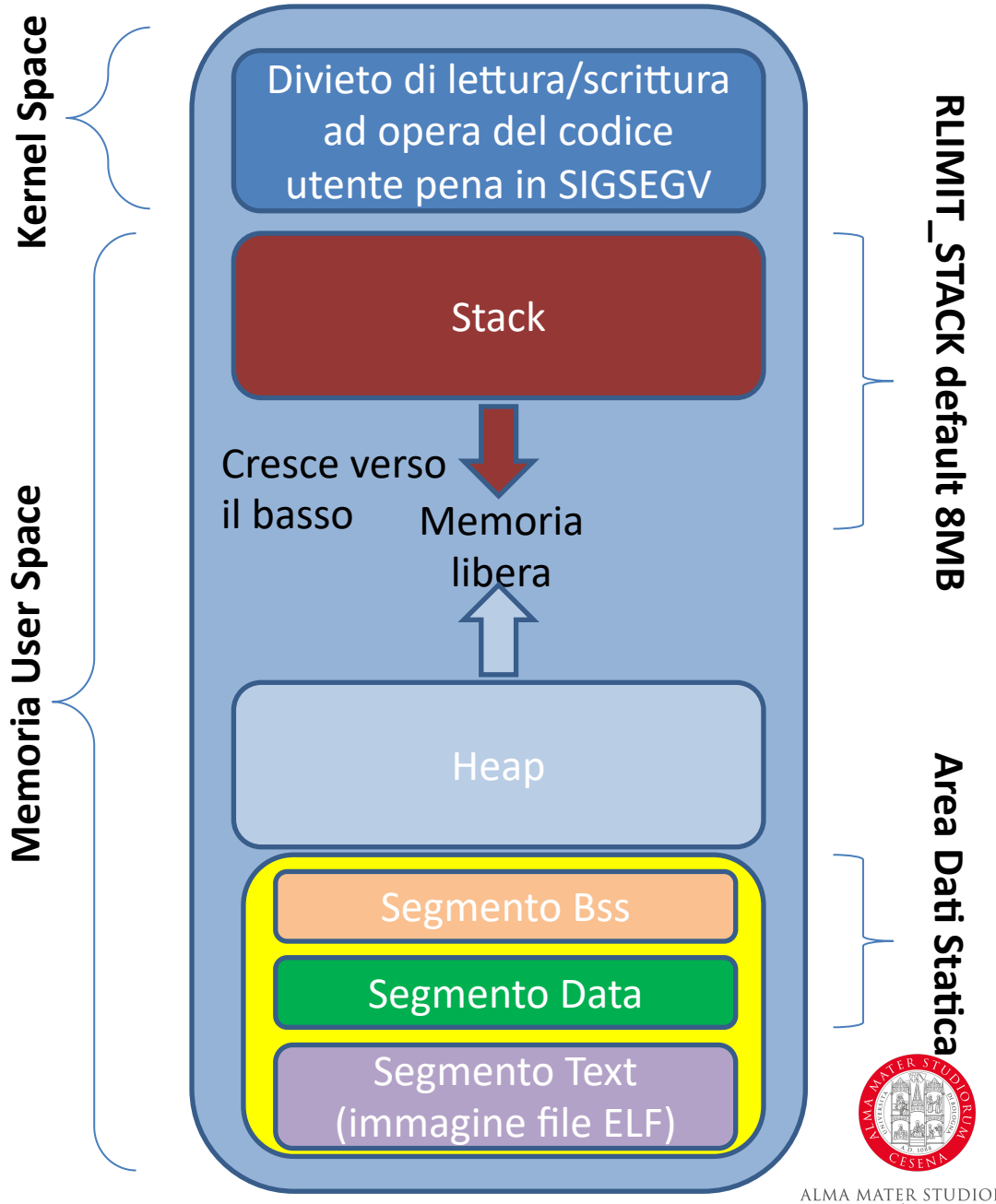
Lo HEAP è una porzione di memoria allocata dinamicamente, per esempio attraverso funzioni C come ***calloc()*** o ***malloc().***

Per approfondimenti date il comando ***man 3 malloc.***

A differenza dello stack che è un'area contigua, la memoria heap è allocata in modalità random e cresce verso indirizzi più alti.

E' un'area che può essere soggetta a memory leak quando viene allocata e non viene rilasciata prima di rimuovere il riferimento ad essa. Infatti la memoria rimarrà allocata fino a che non interverrà una delle due seguenti condizioni:

- Chiamata alla funzione C ***free()***
- O la terminazione del programma



Aree Vulnerabili

Questo implica almeno per alcuni linguaggi un ciclo di vita dello heap a «discrezione del programmatore» e cosi per la sua dimensione.

In alcuni linguaggi di programmazione moderni questi aspetti vengono gestiti automaticamente pertanto è più difficile avere dei memory leak. Per esempio in Java, dove non si ha un accesso diretto alla memoria come avviene in C/C++, sarà il **Garbage Collector** (GC) a occuparsi della liberazione della memoria dello heap quando non ci saranno più riferimenti a essa.

In sostanza il GC rileva gli oggetti allocati ma non utilizzati e li elimina liberando la memoria.

A differenza dell'area TEXT, l'area dello HEAP può essere soggetta a una vulnerabilità di tipo buffer overflow nota come ***heap overflow*** o ***heap overrun*** la cui dinamica risulta essere completamente differente dal buffer overflow che interessa lo STACK.

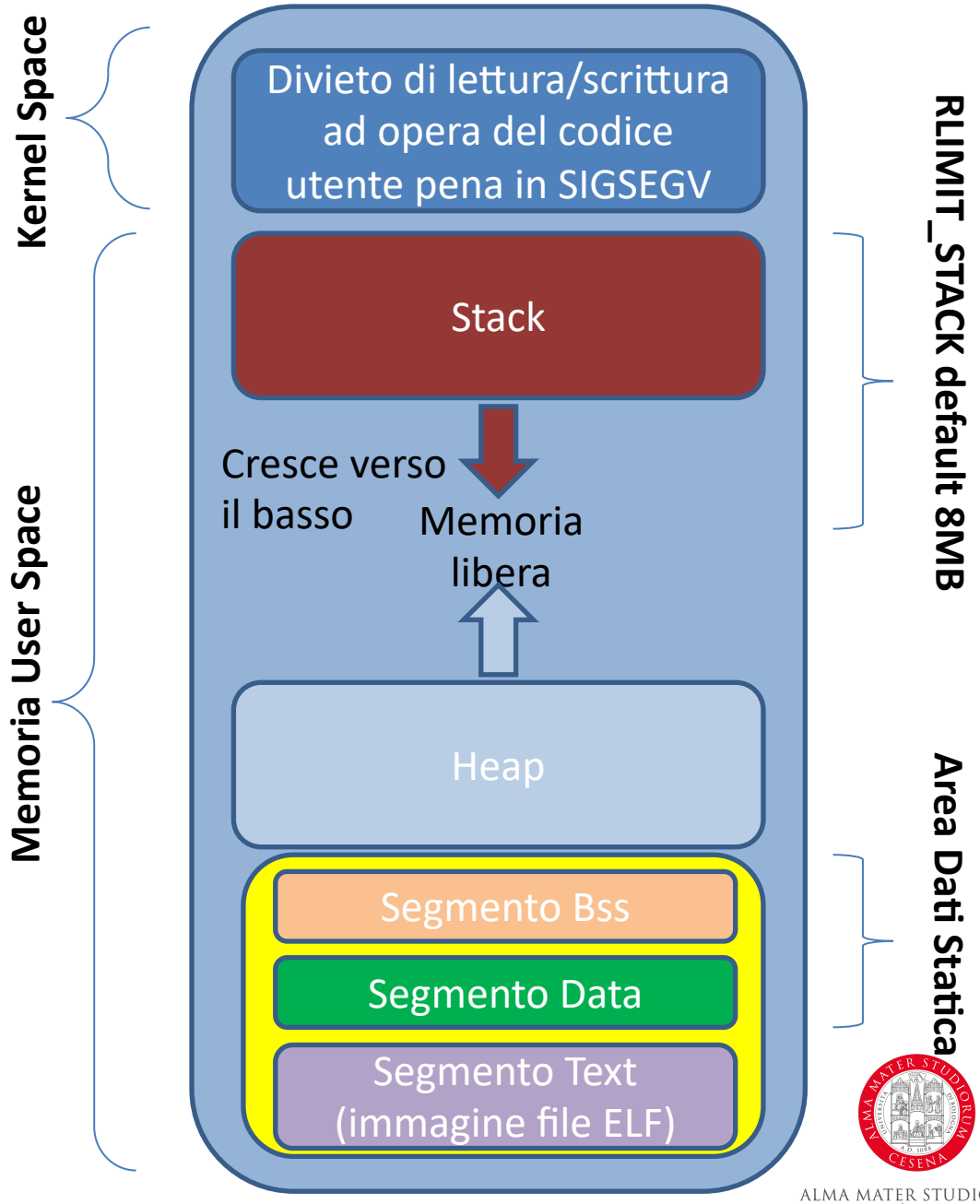
Come si crea lo STACK

Abbiamo già detto che lo stack è una struttura LIFO ossia una struttura cronologica nella quale gli elementi più recenti si trovano in cima e quelli più datati sul fondo.

Siccome lo Stack varia in continuazione durante l'esecuzione di un programma, per conoscere quale sia la cima dello stack frame corrente viene utilizzato il registro **%rsp (%esp)**(stack point register).

La posizione di ciascun elemento allocato nello stack potrebbe essere valutata come delta relativo rispetto allo stack pointer, ma il fatto che esso vari in continuazione significherebbe ogni volta aggiornare le posizioni relative con spreco di risorse e tempo per computazioni inutili.

Per tale scopo viene invece utilizzato il registro **%rbp (%ebp)** (base pointer register) il quale punta alla base dello stack frame, rendendo più facile ottenere la posizione delle variabili.



Chiamata a Funzione

Quando una funzione viene chiamata è possibile individuare tre azioni temporali consecutive:

- La Chiamata
- Il Prologo
- L'Epilogo

Nella fase di chiamata il chiamante, il ***main()***, che di per se è già una funzione allocata nello stack, esegue l'istruzione ***call*** memorizzando nello stack, quindi a indirizzi più bassi rispetto al main(), i parametri della funzione.

Nel prologo vengono aggiornati i registri di gestione riservando lo spazio alle variabili locali della funzione chiamata.

Infine nell'epilogo viene ripristinato lo stack nella condizione precedente alla chiamata della funzione.

Chiamata a Funzione

Questo effetto «yo-yo» dello stack avviene di continuo, a ogni chiamata di funzione le variabili locali vengono continuamente create e distrutte all'interno dello stack.

A causa di questo comportamento è evidente come non avremo una sola copia della variabile, come accade per esempio nell'area Bss e Data per le variabili globali/statiche, ma una loro cotinua creazione e distruzione.

Consideriamo qualche riga di programma per illustrare la dinamica.

Useremo il debugger ***gdb*** (*sudo apt-get install gdb*).

Scriviamo le seguenti righe di codice e salviamole nel file **Somma.c**

```
#include <stdio.h>
//
void somma(int a1, int a2, int a3)
{
   int risultato = 0;
   risultato = a1+ a2+ a3;
   printf("Somma: %d\n", risultato);
}
}
int main()
{
   int addendo1 = 10;
   int addendo2 = 20;
   int addendo3 = 30;
   somma(addendo1, addendo2, addendo3);


   return 0;
}
```

Chiamata a Funzione

Compiliamo il programma con il comando:

gcc –S Somma.c && gcc Somma.c –o Somma

Questo genererà il file **Somma.s** contenente il codice Assembly corrispondente in sintassi AT&T, e l'eseguibile **Somma** che se eseguito visualizzerà la somma dei tre interi.

Per comprendere cosa accade potremmo aprire il file **Somma.s** con un normale editor di testo leggendo le istruzioni e le chiamate alle funzioni.

Per una comprensione più facile usiamo le funzioni disassemblate usando il comando:

gdb –q ./Somma

Appena ***gdb*** restituisce il prompt digitate ***disas main*** e al nuovo prompt ***disas somma***

Chiamata a Funzione

Otteniamo:

```
(gdb) disas main
Dump of assembler code for function main:
   0x080483f6 <+0>: push   %ebp
   0x080483f7 <+1>: mov    %esp,%ebp
   0x080483f9 <+3>: and    $0xfffffff0,%esp
   0x080483fc <+6>: sub    $0x20,%esp
   0x080483ff <+9>: movl   $0xa,0x1c(%esp)
   0x08048407 <+17>: movl   $0x14,0x18(%esp)
   0x0804840f <+25>: movl   $0x1e,0x14(%esp)
   0x08048417 <+33>: mov    0x14(%esp),%eax
   0x0804841b <+37>: mov    %eax,0x8(%esp)
   0x0804841f <+41>: mov    0x18(%esp),%eax
   0x08048423 <+45>: mov    %eax,0x4(%esp)
   0x08048427 <+49>: mov    0x1c(%esp),%eax
   0x0804842b <+53>: mov    %eax,(%esp)
   0x0804842e <+56>: call   0x80483c4 <somma>
   0x08048433 <+61>: mov    $0x0,%eax
   0x08048438 <+66>: leave
   0x08048439 <+67>: ret
End of assembler dump.
```

```
(gdb) disas somma
Dump of assembler code for function somma:
   0x080483c4 <+0>: push   %ebp
   0x080483c5 <+1>: mov    %esp,%ebp
   0x080483c7 <+3>: sub    $0x28,%esp
   0x080483ca <+6>: movl   $0x0,-0xc(%ebp)
   0x080483d1 <+13>: mov    0xc(%ebp),%eax
   0x080483d4 <+16>: mov    0x8(%ebp),%edx
   0x080483d7 <+19>: lea    (%edx,%eax,1),%eax
   0x080483da <+22>: add    0x10(%ebp),%eax
   0x080483dd <+25>: mov    %eax,-0xc(%ebp)
   0x080483e0 <+28>: mov    $0x8048500,%eax
   0x080483e5 <+33>: mov    -0xc(%ebp),%edx
   0x080483e8 <+36>: mov    %edx,0x4(%esp)
   0x080483ec <+40>: mov    %eax,(%esp)
   0x080483ef <+43>: call   0x80482f4 <printf@plt>
   0x080483f4 <+48>: leave
   0x080483f5 <+49>: ret
End of assembler dump.
```

Esempio pratico

```c
#include <stdio.h>


int main(void) {
        leggistringa();
        return(0);
}
void leggistringa(void) {
        long    num = 0;
        char    buff[8];


        gets(buff);
        printf("Se mi vedi non puoi avere dubbi 8-)\n");
}
```

La funzione **gets()** può essere considerata a tutti gli effetti come la funzione più pericolosa esistente nella libreria standard del linguaggio C e difatti molti compilatori visualizzano dei bei warning quando si cerca di utilizzarla. Tale funzione legge dallo standard input (tastiera) la stringa che dopo verrà buttata nel buffer specificato, con l'unica accortezza di sostituire il carattere line-feed (l'invio a capo che abbiamo digitato per terminare l'immissione dati) con un byte NULL.

La particolarità e la pericolosità della funzione sta nel fatto che non controlla se la stringa che ha immesso l'utente è più grande del buffer dove verrà collocata.

Nello stack dovranno essere tenuti in considerazione esattamente **12 bytes** in quanto abbiamo gli

- 8 bytes di "buff" più i
- 4 bytes di "num" (un numero long in memoria infatti occupa appunto 4 bytes e comunque la logica a 32bit degli attuali processori divide tutto in 4 bytes alla volta).

Da notare che spesso se si usano dei buffer o altre variabili non inizializzate, la memoria necessaria verrà allocata solo quando verranno effettivamente utilizzate.

Esempio pratico

Una volta compilato tale codice (gcc bof.c –o bof) avremo che la funzione ***leggistringa()*** contiene il seguente codice macchina (gdb –q ./bof → disas main e disas leggistringa)

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048444 <+0>: push   %ebp
   0x08048445 <+1>: mov    %esp,%ebp
   0x08048447 <+3>: and    $0xfffffff0,%esp
   0x0804844a <+6>: call   0x8048458 <leggistringa>
   0x0804844f <+11>: mov    $0x0,%eax
   0x08048454 <+16>: mov    %ebp,%esp
   0x08048456 <+18>: pop    %ebp
   0x08048457 <+19>: ret
End of assembler dump.
(gdb) disas leggistringa
Dump of assembler code for function leggistringa:
   0x08048458 <+0>: push   %ebp
   0x08048459 <+1>: mov    %esp,%ebp
   0x0804845b <+3>: sub    $0x28,%esp
   0x0804845e <+6>: mov    %gs:0x14,%eax
   0x08048464 <+12>: mov    %eax,-0xc(%ebp)
   0x08048467 <+15>: xor    %eax,%eax
   0x08048469 <+17>: movl   $0x0,-0x18(%ebp)
   0x08048470 <+24>: lea    -0x14(%ebp),%eax
   0x08048473 <+27>: mov    %eax,(%esp)
   0x08048476 <+30>: call   0x8048344 <gets@plt>
   0x0804847b <+35>: movl   $0x8048560,(%esp)
   0x08048482 <+42>: call   0x8048374 <puts@plt>
   0x08048487 <+47>: mov    -0xc(%ebp),%eax
   0x0804848a <+50>: xor    %gs:0x14,%eax
   0x08048491 <+57>: je     0x8048498 <leggistringa+64>
   0x08048493 <+59>: call   0x8048364 <__stack_chk_fail@plt>
   0x08048498 <+64>: leave
   0x08048499 <+65>: ret
End of assembler dump.
```

L'istruzione CALL all'indirizzo `0x0804844a` farà si che l'attuale EIP (`0x0804844f` appunto) venga immagazzinato in memoria in un determinato indirizzo, cosicché esso potrà essere ripreso quando verrà invocata l'istruzione RET al termine della funzione ***leggistringa***().
La prima istruzione di ***leggistringa***() salva EBP nello stack, mentre la seconda copia su EBP il valore di ESP. Ricordiamoci che EBP puntava all'inizio del vecchio stack prima che entrassimo in ***leggistringa***(). Esso serve appunto per riappropriarci del nostro vecchio stack appena terminata la funzione.

Esempio pratico

Dopo aver avviato il nostro programma, ./bof, inseriremo la stringa "1234567" che occuperà alla perfezione il buffer di 8 bytes chiamato buff in quanto 7 numeri occuperanno i primi 7 bytes e l'ottavo sarà un NULL byte che serve a delimitare la stringa.
L'esito è:

```
./bof
1234567
Se mi vedi non puoi avere dubbi 8-)
```

Esempio pratico

Ora, invece di digitare "1234567", inseriremo proprio 20 bytes di dati, ossia:

8 per il buffer chiamato ***buff***

4 per il numero long chiamato ***num***

4 per il valore di **EBP** precedentemente salvato

4 per il valore di **EIP** precedentemente salvato

La stringa scelta è "123456781234aaaabbbb":

```
./bof
123456781234aaaabbbb
Se mi vedi non puoi avere dubbi 8-)
*** stack smashing detected ***: ./bof terminated
======= Backtrace: =========
/lib/libc.so.6(__fortify_fail+0x50)[0xb77e7890]
/lib/libc.so.6(+0xe583a)[0xb77e783a]
./bof[0x8048498]
./bof[0x804844f]
./bof[0x8049ff4]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 7604282    /home/apirodd/bof
08049000-0804a000 r--p 00000000 08:01 7604282    /home/apirodd/bof
0804a000-0804b000 rw-p 00001000 08:01 7604282    /home/apirodd/bof
09548000-09569000 rw-p 00000000 00:00 0          [heap]
b76d2000-b76ec000 r-xp 00000000 08:01 4456508    /lib/libgcc_s.so.1
b76ec000-b76ed000 r--p 00019000 08:01 4456508    /lib/libgcc_s.so.1
b76ed000-b76ee000 rw-p 0001a000 08:01 4456508    /lib/libgcc_s.so.1
b7701000-b7702000 rw-p 00000000 00:00 0
b7702000-b7859000 r-xp 00000000 08:01 4460426    /lib/libc-2.12.1.so
b7859000-b785b000 r--p 00157000 08:01 4460426    /lib/libc-2.12.1.so
b785b000-b785c000 rw-p 00159000 08:01 4460426    /lib/libc-2.12.1.so
b785c000-b785f000 rw-p 00000000 00:00 0
b7870000-b7874000 rw-p 00000000 00:00 0
b7874000-b7875000 r-xp 00000000 00:00 0          [vdso]
b7875000-b7891000 r-xp 00000000 08:01 4460410    /lib/ld-2.12.1.so
b7891000-b7892000 r--p 0001b000 08:01 4460410    /lib/ld-2.12.1.so
b7892000-b7893000 rw-p 0001c000 08:01 4460410    /lib/ld-2.12.1.so
bfc44000-bfc65000 rw-p 00000000 00:00 0          [stack]
Annullato
```