Modificare il flusso dei files eseguibili in ambiente UNIX

Disclaimer: questo tutorial è a scopo illustrativo. L'autore NON è responsabile di eventuali usi non legali delle informazioni qui contenute.

USO A VOSTRO RISCHIO E PERICOLO!

Immaginiamo di avere un programma che, per funzionare, ci richiede una password (o un numero seriale), che non abbiamo in quel momento o che abbiamo smarrito. Senza andare a sbattere la testa in lamerate di vario genere, possiamo cambiare il flusso del programma eseguibile in modo abbastanza pulito. Tutto ciò che ci serve è:

- una buona conoscenza dell'Assembly e del C
- un debugger
- un editor esadecimale

Questo tutorial dimostra come fare una cosa simile su un sistema UNIX. Come debugger userò GDB (il debugger standard della GNU, il più completo che si possa trovare in giro, installato su gran parte delle distro Linux e *BSD) e come editor esadecimale KHexEdit, l'editor esadecimale di KDE, ma ovviamente qualsiasi debugger o editor hex va bene (mi sono trovato abbastanza bene anche con BVI, un editor di files binari che usa una sintassi molto simile a VI, e con EMACS in modalità esadecimale. Inoltre, con i concetti qui esposti è possibile anche compiere azioni simili su Windows).

Partiamo da un programma semplicissimo scritto in C:

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

main() {
      char pwd[] = "prova";
      char str[30];

      printf ("Inserire la password: ");
      scanf ("%s",str);

      if (!strcmp(str,pwd))
            printf ("Password corretta\n");
      else
```

```
            printf ("Password errata\n");
}
```

chiamiamolo vuln.c e compiliamolo, chiamando il programma di output vuln:

```
gcc -o vuln vuln.c
```

E' ovvio che, finchè non inserisco la stringa "prova", il messaggio che mi compare sarà sempre "password errata". Supponiamo di non avere questa password, e nemmeno il file sorgente (questo esempio è molto semplice, ma, poste queste condizioni, potremmo anche avere a che fare con un file di setup che ci chiede un codice seriale, file di setup di cui, ovviamente, non abbiamo i sorgenti). In questo tutorial vedremo come bypassare semplicemente la verifica della condizione, NON come risalire alla password originale.

Diamo l'eseguibile vuln in pasto a gdb:

```
blacklight@blacklight:~/prog/c++$ 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)
```

Inseriamo ora un break point nella funzione main e poi eseguiamo il programma:

```
(gdb) break main
Breakpoint 1 at 0x804841a
(gdb) run
Starting program: /home/blacklight/prog/c++/vuln

Breakpoint 1, 0x0804841a in main ()
(gdb)
```

In questo modo possiamo mantenere il programma in esecuzione e accedere alle sue informazioni in “tempo reale”.

Ora:

```
(gdb) p main
$1 = {<text variable, no debug info>} 0x8048414 <main>
```

Ci permette di memorizzare la posizione della funzione main, in modo da poterci muovere all'interno del codice.

Io uso poi la direttiva:

```
(gdb) set disassembly-flavor intel
```

Questo comando ci permette di visualizzare l'output disassemblato della funzione nella sintassi Assembly dell'Intel (quella più usata), invece di vederlo nella sintassi AT&T, lo standard dei sistemi UNIX, che è un po' più “criptica” (ma è solo questione di abitudine…).

Ora disassembliamo il codice:

```
(gdb) disas main
Dump of assembler code for function main:
0x08048414 <main+0>:    push   ebp
0x08048415 <main+1>:    mov    ebp,esp
0x08048417 <main+3>:    sub    esp,0x38
0x0804841a <main+6>:    and    esp,0xfffffff0
0x0804841d <main+9>:    mov    eax,0x0
0x08048422 <main+14>:   add    eax,0xf
0x08048425 <main+17>:   add    eax,0xf
0x08048428 <main+20>:   shr    eax,0x4
0x0804842b <main+23>:   shl    eax,0x4
0x0804842e <main+26>:   sub    esp,eax
0x08048430 <main+28>:   mov    eax,ds:0x80485b4
0x08048435 <main+33>:   mov    DWORD PTR [ebp-24],eax
0x08048438 <main+36>:   mov    ax,ds:0x80485b8
0x0804843e <main+42>:   mov    WORD PTR [ebp-20],ax
0x08048442 <main+46>:   sub    esp,0xc
```

```
0x08048445 <main+49>:   push   0x80485ba
0x0804844a <main+54>:   call   0x8048328 <printf@plt>
0x0804844f <main+59>:   add    esp,0x10
0x08048452 <main+62>:   sub    esp,0x8
0x08048455 <main+65>:   lea    eax,[ebp-56]
0x08048458 <main+68>:   push   eax
0x08048459 <main+69>:   push   0x80485d1
0x0804845e <main+74>:   call   0x8048308 <scanf@plt>
0x08048463 <main+79>:   add    esp,0x10
0x08048466 <main+82>:   lea    eax,[ebp-24]
0x08048469 <main+85>:   lea    edx,[ebp-56]
0x0804846c <main+88>:   sub    esp,0x8
0x0804846f <main+91>:   push   eax
0x08048470 <main+92>:   push   edx
0x08048471 <main+93>:   call   0x80482f8 <strcmp@plt>
0x08048476 <main+98>:   add    esp,0x10
0x08048479 <main+101>:  test   eax,eax
0x0804847b <main+103>:  jne    0x804848f <main+123>
0x0804847d <main+105>:  sub    esp,0xc
0x08048480 <main+108>:  push   0x80485d4
0x08048485 <main+113>:  call   0x8048328 <printf@plt>
0x0804848a <main+118>:  add    esp,0x10
0x0804848d <main+121>:  jmp    0x804849f <main+139>
0x0804848f <main+123>:  sub    esp,0xc
0x08048492 <main+126>:  push   0x80485e7
---Type <return> to continue, or q <return> to quit---
0x08048497 <main+131>:  call   0x8048328 <printf@plt>
0x0804849c <main+136>:  add    esp,0x10
0x0804849f <main+139>:  leave
0x080484a0 <main+140>:  ret
0x080484a1 <main+141>:  nop
0x080484a2 <main+142>:  nop
0x080484a3 <main+143>:  nop
0x080484a4 <main+144>:  nop
0x080484a5 <main+145>:  nop
0x080484a6 <main+146>:  nop
0x080484a7 <main+147>:  nop
0x080484a8 <main+148>:  nop
```

```
0x080484a9 <main+149>:  nop
0x080484aa <main+150>:  nop
0x080484ab <main+151>:  nop
0x080484ac <main+152>:  nop
0x080484ad <main+153>:  nop
0x080484ae <main+154>:  nop
0x080484af <main+155>:  nop
End of assembler dump.
(gdb)
```

Quelle che ci interessano sono queste righe:

```
0x0804846f <main+91>:   push   eax
0x08048470 <main+92>:   push   edx
0x08048471 <main+93>:   call   0x80482f8 <strcmp@plt>
0x08048476 <main+98>:   add    esp,0x10
0x08048479 <main+101>:  test   eax,eax
0x0804847b <main+103>:  jne    0x804848f <main+123>
```

Come si può intuire, nelle prime due righe il programma salva sullo stack gli indirizzi delle due stringhe da confrontare (la password autentica e la stringa inserita dall'utente), provvisoriamente salvati sui registri eax e edx, per poi chiamare, attraverso la direttiva call, la funzione strcmp della libreria C che effettua il confronto tra le due stringhe inserite. Effettua quindi un'operazione per allineare lo stack e azzera il registro eax, e poi, cosa più importante, se le due stringhe non coincidono salta ad una nuova etichettadel programma (quella che, nel nostro caso, ci darà il messaggio “password errata”) attraverso la direttiva JNE (“salta all'etichetta specificata se i dati forniti non coincidono”). Quello che vogliamo fare è proprio eliminare questo salto condizionato, in modo che le istruzioni che sarebbero eseguite se la password inserita fosse corretta vengano eseguite anche in caso contrario. Quello chefaremo in questo tutorial è sostituire, direttamente nel codice eseguibile, al codice operativo corrispondente all'istruzione jne in linguaggio macchina il codice operativo dell'istruzione NOP (No OPeration, un'istruzione che non fa niente), in modo che il codice, in Assembly, diventi così:

```
0x0804846f <main+91>:   push   eax
0x08048470 <main+92>:   push   edx
0x08048471 <main+93>:   call   0x80482f8 <strcmp@plt>
0x08048476 <main+98>:   add    esp,0x10
0x08048479 <main+101>:  test   eax,eax
```

```
0x0804847b <main+103>:  nop
; In questo modo non c'è più il salto condizionato, e ciò che c'è dopo nel codice
viene eseguito in ogni caso
```

Dobbiamo solo armarci di un buon editor esadecimale e di tanta, tanta pazienza.

L'istruzione JNE, nel nostro caso, si trova al byte 103 (main+103) del nostro programma: ci serve solo sapere qual è il suo codice operativo in esadecimale in modo da poterlo cercare all'interno del programma e sostituirlo con la nostra NOP.

Per farlo, facciamo una carrellata veloce dei 102 byte prima della nostra istruzione, traducendoli direttamente in linguaggio macchina codificato in esadecimale:

```
(gdb) x/102xb
0x8048415 <main+1>:     0x89    0xe5    0x83    0xec    0x38    0x83    0xe4    0xf0
0x804841d <main+9>:     0xb8    0x00    0x00    0x00    0x00    0x83    0xc0    0x0f
0x8048425 <main+17>:    0x83    0xc0    0x0f    0xc1    0xe8    0x04    0xc1    0xe0
0x804842d <main+25>:    0x04    0x29    0xc4    0xa1    0xb4    0x85    0x04    0x08
0x8048435 <main+33>:    0x89    0x45    0xe8    0x66    0xa1    0xb8    0x85    0x04
0x804843d <main+41>:    0x08    0x66    0x89    0x45    0xec    0x83    0xec    0x0c
0x8048445 <main+49>:    0x68    0xba    0x85    0x04    0x08    0xe8    0xd9    0xfe
0x804844d <main+57>:    0xff    0xff    0x83    0xc4    0x10    0x83    0xec    0x08
0x8048455 <main+65>:    0x8d    0x45    0xc8    0x50    0x68    0xd1    0x85    0x04
0x804845d <main+73>:    0x08    0xe8    0xa5    0xfe    0xff    0xff    0x83    0xc4
0x8048465 <main+81>:    0x10    0x8d    0x45    0xe8    0x8d    0x55    0xc8    0x83
0x804846d <main+89>:    0xec    0x08    0x50    0x52    0xe8    0x82    0xfe    0xff
0x8048475 <main+97>:    0xff    0x83    0xc4    0x10    0x85    0xc0
```

Ora osserviamo che la nostra istruzione “vittima” in linguaggio macchina è lunga 2 byte (infatti, comincia all'etichetta main+103 e all'etichetta main+105, se osserviamo il dump Assembly di gdb, c'è già un'altra istruzione), quindi vediamo qual è il suo codice esadecimale dicendo a GDB di visualizzare il codice esadecimale dei prossimi 2 byte:

```
0x804847b <main+103>:   0x75    0x12
```

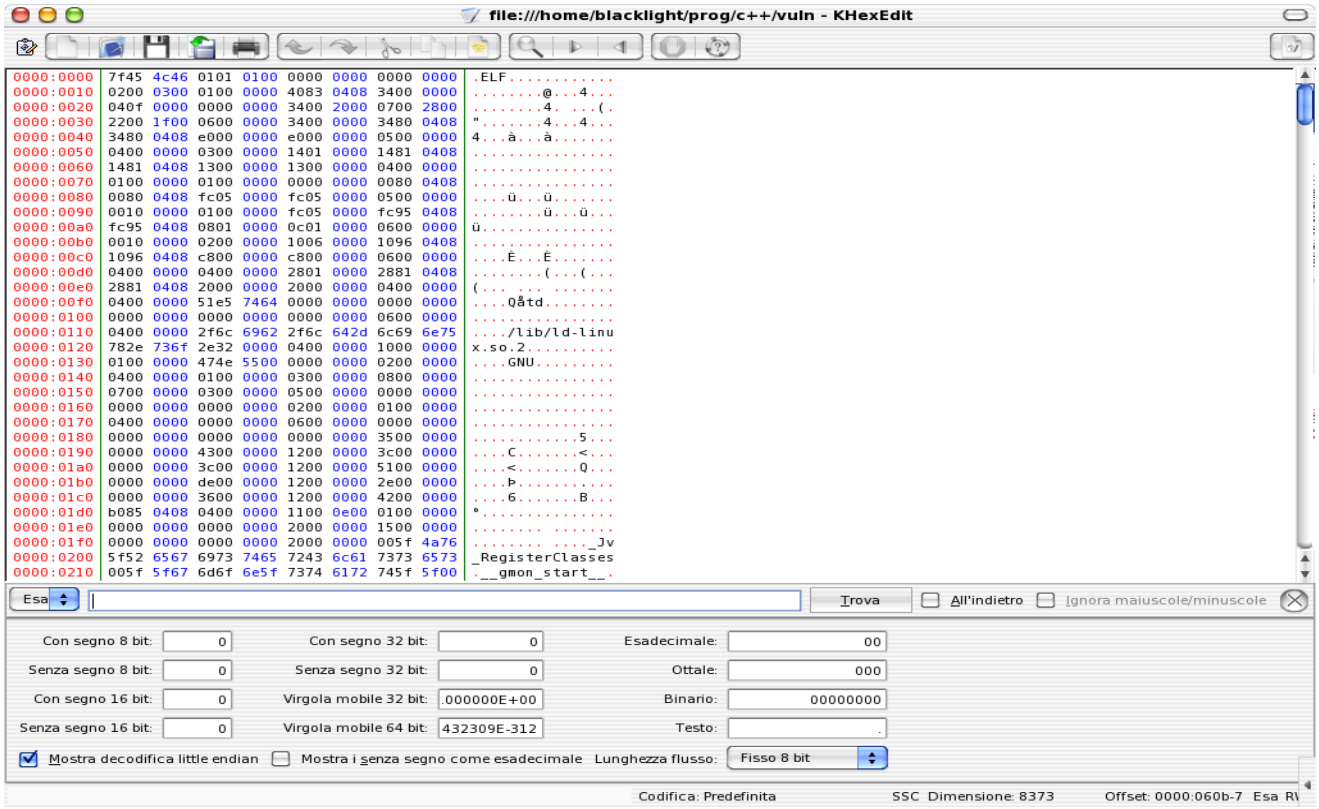
Questo è il codice operativo che corrisponde, in Assembly, all'istruzione:

```
jne    0x804848f <main+123>
```

Ora sappiamo che nel codice esadecimale del programma dobbiamo cercare la stringa 7512[1]

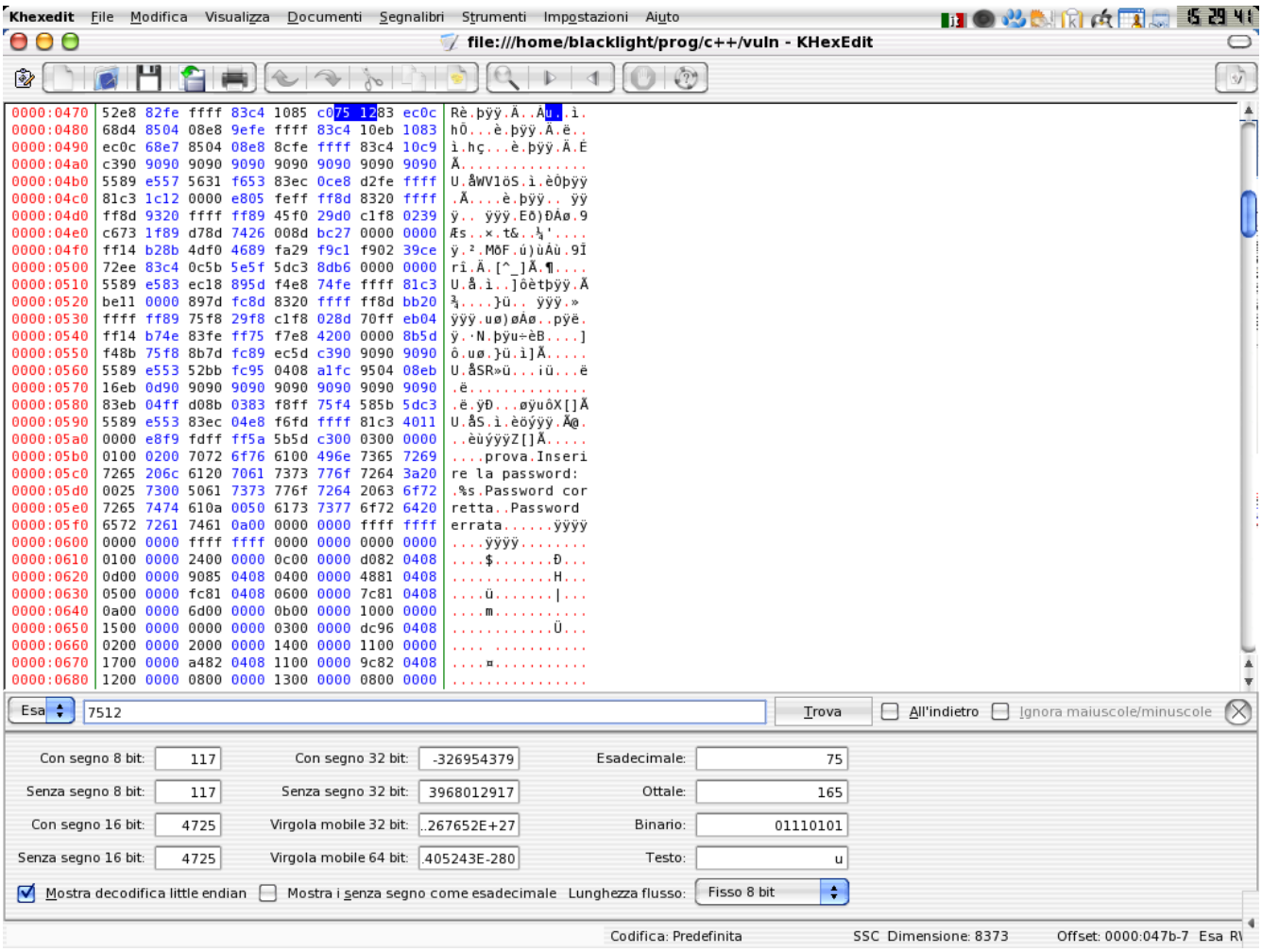
```
blacklight@blacklight:~/prog/c++$ khexedit vuln
```

Appena aperto l'editor esadecimale ci dovremmo ritrovare di fronte a una schermata del genere:

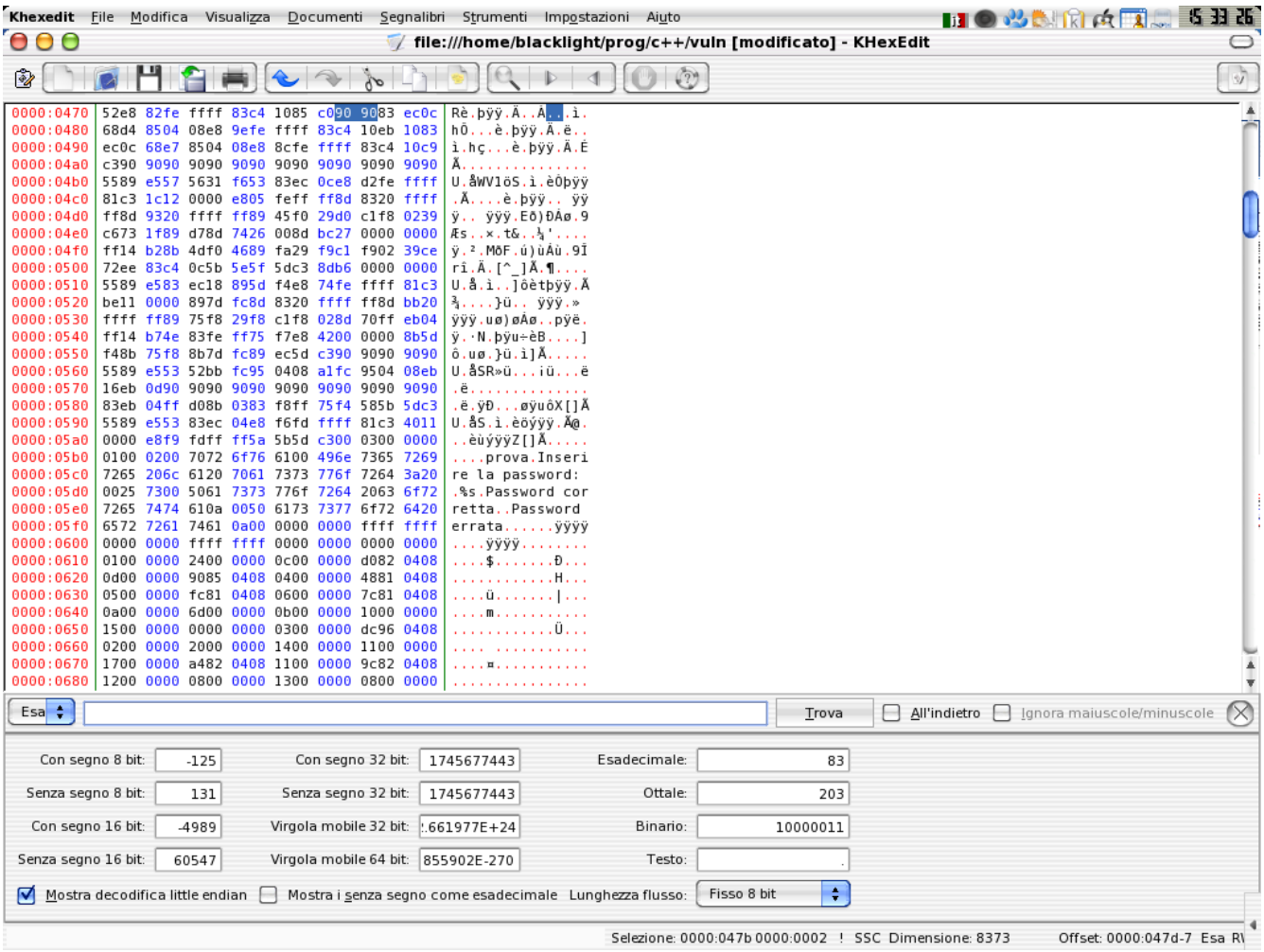


Cerchiamo ora la stringa esadecimale che ci interessa modificare. Nel mio caso sarà proprio la stringa 7512, ma se non ottenete alcun risultato vuol dire che dovreste cercare la stringa "capovolta" (1275):

1 Anche nel mondo UNIX, i sistemi operativi usano standard diversi per memorizzare le istruzioni in un file eseguibile. Sulla mia Slackware Linux lo standard è quello "diretto", ovvero i codici esadecimali delle istruzioni vengono memorizzati nell'eseguibile così come sono presenti nel programma: in questo caso dovrò quindi cercare la sequenza 75 12. Mi è capitato di testare però questo stesso esempio su una vecchia versione della Debian, e lì le sequenze di codici venivano memorizzate in ordine inverso, ovvero dovevo cercare la sequenza 12 75. Credo che abbia qualcosa a che fare con le convenzioni Little Endian e Big Endian dell'uso della memoria, ma non sono pronto a metterci la mano sul fuoco.



Questo è il codice corrispondente in linguaggio macchina al salto condizionato. Quello che dobbiamo fare è sostituirlo con una NOP, l'istruzione che non fa niente. Il codice operativo della NOP in esadecimale è 0x90 (almeno su quasi tutti i sistemi UNIX), il che vuol dire che l'istruzione è lunga 1 byte. L'istruzione di salto condizionato però è lunga 2 byte (0x75 0x12), quindi ci mettiamo 2 NOP (ovvero sostituiamo, ai codici esadecimali 75 e 12, 90 e 90):



Il gioco è fatto. Salviamo il file ed eseguiamo il programma:

```
blacklight@blacklight:~/prog/c++$ ./vuln
Inserire la password: blacklight
Password corretta

blacklight@blacklight:~/prog/c++$ ./vuln
Inserire la password: prova
Password corretta

blacklight@blacklight:~/prog/c++$ ./vuln
Inserire la password: unix
Password corretta
```

Come vedete ora, qualsiasi sia la passwordche inseriamo il programma esegue sempre le istruzioni che esegue in caso di password corretta.

Ovviamente, non dovete prendere subito per oro colato i numeri, gli indirizzi e i codici operativi presenti in questo tutorial. Basta compilare questo programmino di esempio con una versione di GCC diversa dalla mia (3.4.6) e la lunghezza delle direttive in esadecimale o la posizione dell'istruzione all'interno del codice potrebbe cambiare.