

## Software exploitation

### Assignment 3

Mikael Romanov K1521

Maaliskuu 2018

Tekniikan ja liikenteen ala

Insinööri (AMK), tieto- ja viestintätekniikan tutkinto-ohjelma

Kyberturvallisuus



## 1 Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>2</b>
<b>2</b>	<b>Testing platform .....</b>	<b>2</b>
<b>3</b>	<b>shell_1.....</b>	<b>4</b>
<b>4</b>	<b>shell_2.....</b>	<b>8</b>
<b>5</b>	<b>Conclusion.....</b>	<b>9</b>

# 1 Introduction

This assignment's goal was to create customized executable shellcodes in assembly. The first program took shellcode as input and executed it as a function. Second program needed buffer overflow to execute buffer overflow

## 2 Testing platform

I used my Thinkpad w520 with VMware Workstation 14 (Figure 1)



Figure 1 vmware

I reserved good amount of resources for the KALI vm and they were following(Figure 2)

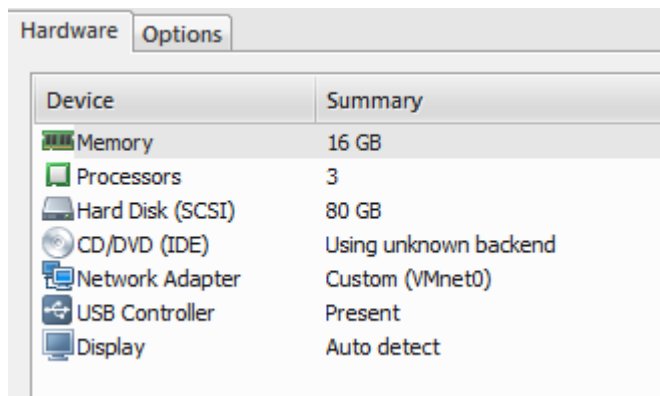


Figure 2 resources

I used Kali Linux 4.14.0-kali3-i686 (32 bit) since the amd64 had some problems with the **shellcode -f**. The gcc version was 7.2.0, the same as on the earlier assignments. (Figure 3)

```
root@kali:~/Downloads/software_exploitation/assignments/3# uname -a
Linux kali 4.14.0-kali3-686-pae #1 SMP Debian 4.14.13-1kali1 (2018-01-25) i686 GNU/Linux
root@kali:~/Downloads/software_exploitation/assignments/3# 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 3 versions

I used the make file to compile the programs. The make file was simple to us to compile the file, it only required the command **make**. The MAKEFILE uses gcc to compile the program.

After compiling the programs I turned ASLR off, because it is hard to know where the memory is allocated next. ASLR can be set off by echoing **echo 0 > /proc/sys/kernel/randomize\_va\_space** (Figure4)

```
Try to manage with less.root@kali:~/Downloads/software_exploitation-master/assignments/2/basic# echo 0 > /pr
va_spacekernel/randomize_v
```

Figure 4 aslr off

Then I used readelf to get information of the executables headers, which indicated that the program is 32bit (Figure 5)

```

root@kali:~/Downloads/software_exploitation/assignments/3# readelf -h shell_1
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF32
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  EXEC (Executable file)
  Machine:                               Intel 80386
  Version:                               0x1
  Entry point address:                   0x8048540
  Start of program headers:              52 (bytes into file)
  Start of section headers:              9068 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   52 (bytes)
  Size of program headers:               32 (bytes)
  Number of program headers:              9
  Size of section headers:               40 (bytes)
  Number of section headers:             36
  Section header string table index:     35

```

Figure 5 readelf

### 3 shell\_1

I started this assignment by modifying the hello.asm file, so that it would work with the programs. (Figure 6)

```

bits 32

global _start

section .text

_start:
    jmp message

print_hello:
    mov eax, 0x4 ;sys_write(fd, data, size)
    mov ebx, 0x1
    pop ecx
    mov edx, 13
    int 0x80

    mov eax, 0x1 ;sys_exit()
    int 0x80

message:
    call print_hello
    db "Hello, world" , 0ah

```

Figure 6 new hello.asm

I assembled it and then disassembled to see if there were null bytes in the register. And as seen below, the hello.asm had null bytes in few locations.(Figure 7) The null bytes need to be removed, because it means end of string in C.

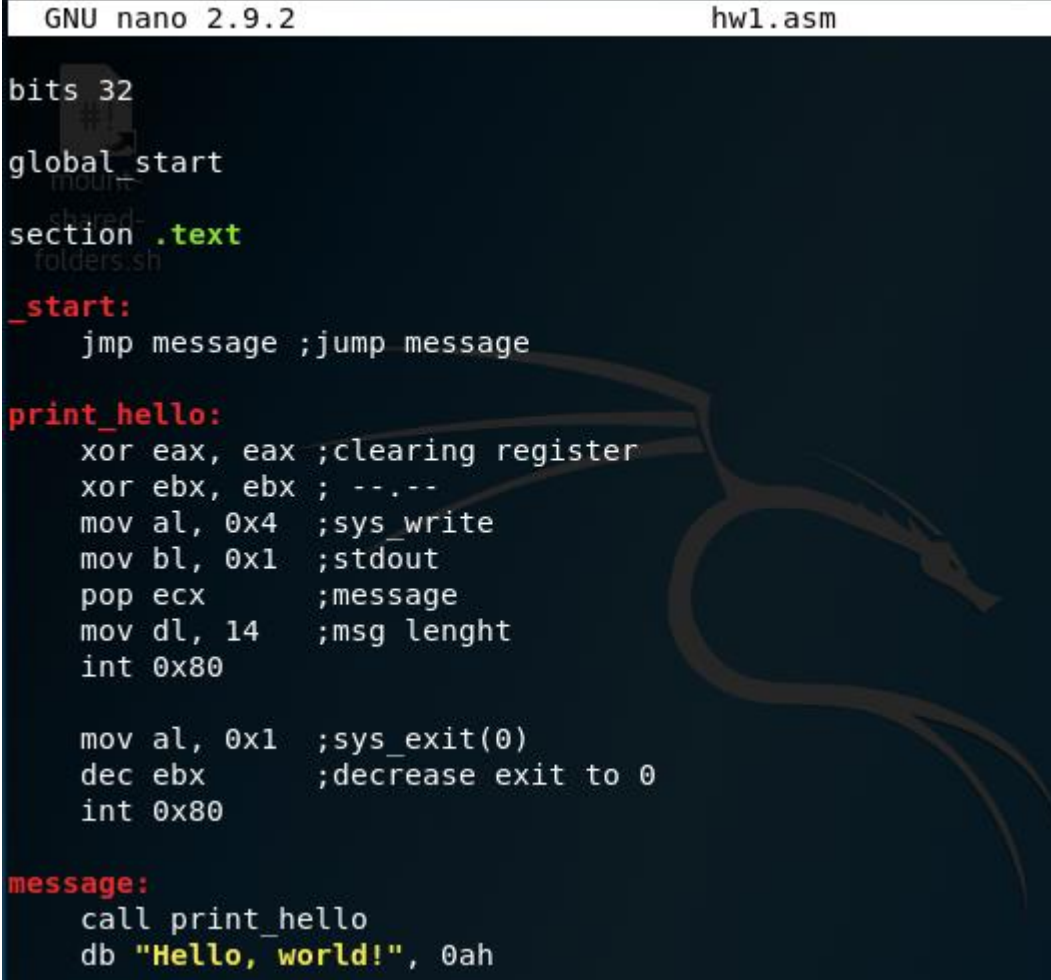
```

root@kali:~/Downloads/software_exploitation/software_exploitation-master/assignments/3# ndisasm -b32 helloworld
00000000 EB19 jmp short 0x1b
00000002 B804000000 mov eax,0x4
00000007 BB01000000 mov ebx,0x1
0000000C 59 pop ecx
0000000D BA0D000000 mov edx,0xd
00000012 CD80 int 0x80
00000014 B801000000 mov eax,0x1
00000019 CD80 int 0x80
0000001B E8E2FFFFFF call 0x2
00000020 48 dec eax
00000021 656C gs insb
00000023 6C insb
00000024 6F outsd
00000025 2C20 sub al,0x20
00000027 776F ja 0x98
00000029 726C jc 0x97
0000002B 64 fs
0000002C 0A db 0x0a

```

Figure 7 ndisasm

I then edited the hello.asm (renamed it to hw1) and removed all the null bytes by making the register 8bits long, because the values won't need extra space. First the register is cleared by Xor. Then changed the syswrite, stdout, message length and sys exit to 8 bits. And decreased ebx register to 0 for clean exit. (Figure 8)



```
GNU nano 2.9.2 hw1.asm

bits 32

global start

section .text

_start:
    jmp message ;jump message


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

    mov al, 0x1  ;sys_exit(0)
    dec ebx      ;decrease exit to 0
    int 0x80

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

Figure 8 edited hello.asm

I saved the file and made 3 different versions of it, only thing that changed was the sentence that was in use. Then I assembled them (Figure 9)



```
root@kali:~/Downloads/software_exploitation/assignments/3# nasm hw1.asm & nasm hw2.asm & nasm hw3.asm
```

Figure 9 assembling



To be sure about the removing of the null bytes, I disassembled the hello.asm again and this time the register didn't show any null bytes. This means that the shellcode should run. (Figure 10)

```
root@kali:~/Downloads/software_exploitation/assignments/3# ndisasm -b32 hw1
00000000 EB12      jmp short 0x14
00000002 31C0      xor eax,eax
00000004 31DB      xor ebx,ebx
00000006 B004      mov al,0x4
00000008 B301      mov bl,0x1
0000000A 59        pop ecx
0000000B B20E      mov dl,0xe
0000000D CD80      int 0x80
0000000F B001      mov al,0x1
00000011 4B        dec ebx
00000012 CD80      int 0x80
00000014 E8E9FFFF call 0x2
00000019 48        dec eax
0000001A 656C      gs insb
0000001C 6C        insb
0000001D 6F        outsd
0000001E 2C20      sub al,0x20
00000020 776F      ja 0x91
00000022 726C      jc 0x90
00000024 64210A    and [fs:edx],ecx
```

Figure 10

The I just started executing the shellcodes with cat. In the first one I used cat to pipe the command, because it wouldn't work any other way. (Figure 11)

```
root@kali:~/Downloads/software_exploitation/assignments/3# cat hw1 | ./shell_1 -f -
Hello, world!
```

Figure 11

The second one needed just the file name, I inputted the filename and as expected it runned like it supposed to. (Figure 12)

```
root@kali:~/Downloads/software_exploitation/assignments/3# ./shell_1 -f hw2
Hello, welt.
```

Figure 12 hello welt

```
root@kali:~/Downloads/software_exploitation/assignments/3# hexdump -C hw2
00000000 eb 12 31 c0 31 db b0 04 b3 01 59 b2 0d cd 80 b0 |..1.1....Y....|
00000010 01 4b cd 80 e8 e9 ff ff ff 48 65 6c 6c 6f 2c 20 |.K.....Hello, |
00000020 77 65 6c 74 2e 0a                hello.asm hw1 |welt..| hw2
00000026
```

Figure 13 hexdump welt

The last one needed some padding to be executed correctly, since -t pushes to the stack also. (Figure 14)

```
root@kali:~/Downloads/software_exploitation/assignments/3# ./shell_1 -t cat hw3
Segmentation fault
root@kali:~/Downloads/software_exploitation/assignments/3# ./shell_1 -t "$(cat hw3)"
^C
root@kali:~/Downloads/software_exploitation/assignments/3# ./shell_1 -t "a$(cat hw3)"
Segmentation fault
root@kali:~/Downloads/software_exploitation/assignments/3# ./shell_1 -t "aa$(cat hw3)"
Bonjour le monde
root@kali:~/Downloads/software_exploitation/assignments/3# ./shell_1 -t
```

Figure 14 Bonjour le monde

## 4 shell\_2

I started debugging the shell 2 and checked how many characters need to be printed until owerflow. (Figure 15)

```
(gdb) run -t $(python -c 'print "\x41"*79')
Starting program: /root/Downloads/software_exploitation/assignments/3/shell_2 -t
$(python -c 'print "\x41"*79')

Program received signal SIGSEGV, Segmentation fault.
0x08414141 in ?? ()
```

Figure 15 overflow try

The correct answer was 80 characters to fully overflow. (Figure 16)

```
(gdb) run -t $(python -c 'print "\x41"*80')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Downloads/software_exploitation/assignments/3/shell_2 -t
$(python -c 'print "\x41"*80')

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

Figure 16 owerflow success

```

(gdb) info registers
eax      0x2      2
ecx      0x50     80
edx      0xbffffec -1073745972
ebx      0x414141 1094795585
esp      0xbffff01c 0xbffff01c
ebp      0x414141 0x41414141
esi      0xb7fa1000 -1208348672
edi      0x414141 1094795585
eip      0x414141 0x41414141
eflags   0x10286 [ PF SF IF RF ]
cs       0x73     115
ss       0x7b     123
ds       0x7b     123
es       0x7b     123
fs       0x0      0
gs       0x33     51

```

## 5 Conclusion

This assignment was also hard, way more than the earlier assignments. I tried the second shell but didn't manage to finish it sadly the shellcode didn't run for some reason, so my documentation ended to the overflow. First shell was pretty simple after reading the books chapter.