Reverse Engineering Executive Document for a simple Calculator in C (revised October 2018)

Jay Chow, Kevin Hamilton, and Yuqing Wang, MSSI graduate students, JHUISI

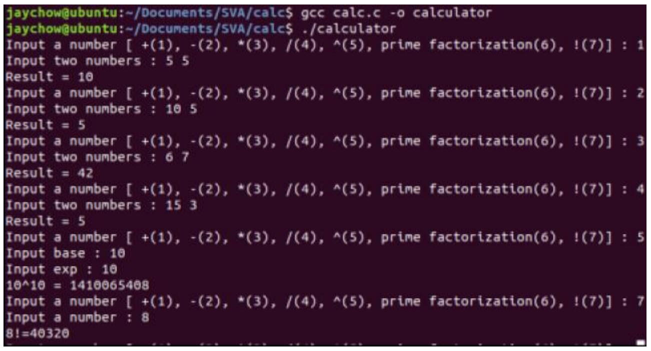
Abstract - The goal of this document is for us to develop an information leak exploit, arbitrary address read exploit and arbitrary address write exploit and to completely understand the format string vulnerability in a open-sourced, simple Calculator written in C that does operations such as addition, subtraction, multiplication, division, power and factorial.

1 Project Goals

Our goal as a group was to attain a better understanding of the format string vulnerability and how corresponding exploits, such as information leak exploit, arbitrary read exploit and arbitrary write exploit, work. We will insert a format string vulnerability into a Linux command line application and construct corresponding exploits that will launch the shellcode. We have built the this command line application in a certain way such as disabling address space layout randomization to randomize the starting addresses for the stack and heap, for example, in virtual memory. This would make it more difficult for attackers to exploit vulnerabilities. The linux command line application that we selected satisfies these properties:
- Compatible with x86-32 on Ubuntu 18.04
- Can be readily compiled using gcc 4.8
- Can be debugged using gdb8.1 and Eclipse CDT(C/C++ Development Tooling)
- Coded in C and open source project
- Accepts command line inputs from the terminal

1.1 Overview

The simple, open-sourced calculator performs a subset of mathematical operations such as addition, subtraction and factorial operations.



```
jaychow@ubuntu:~/Documents/SVA/calc$ gcc calc.c -o calculator
jaychow@ubuntu:~/Documents/SVA/calc$ ./calculator
Input a number [ +(1), -(2), *(3), /(4), ^(5), prime factorization(6), !(7)] : 1
Input two numbers : 5 5
Result = 10
Input a number [ +(1), -(2), *(3), /(4), ^(5), prime factorization(6), !(7)] : 2
Input two numbers : 10 5
Result = 5
Input a number [ +(1), -(2), *(3), /(4), ^(5), prime factorization(6), !(7)] : 3
Input two numbers : 6 7
Result = 42
Input a number [ +(1), -(2), *(3), /(4), ^(5), prime factorization(6), !(7)] : 4
Input two numbers : 15 3
Result = 5
Input a number [ +(1), -(2), *(3), /(4), ^(5), prime factorization(6), !(7)] : 5
Input base : 10
Input exp : 10
10^10 = 1410065408
Input a number [ +(1), -(2), *(3), /(4), ^(5), prime factorization(6), !(7)] : 7
Input a number : 8
8!=40320
```

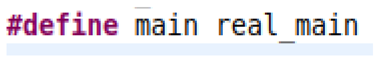
Fig. 1. Performance of simple open source calculator

As shown previously, the calculator takes inputs from command lines for each operation. For example, if the user wants to perform an add operation, the user has to first input "1" on terminal to indicate that an add operation is selected. Then the user will be prompted to enter 2 numbers for addition. Subsequently, upon entering 2 numbers separated by a space character on command line, the calculator program will perform the addition and return the result to the user.

Additionally, if a user wants to perform a factorial, a user can first enter "8", then input a number that to perform the factorial on. As such, if a user then enters "8" on the command line, 8! is performed and the result of 40320 is returned to the user. Although this calculator is not advanced or a complicated software, it gave us a better understanding of developing a format string exploit.

2 Project execution

For the information leak exploit, we altered the program such that when an arbitrary user can provide a char[] buffer that is used by the program as a format specifier for a printf in our calculator program. Next, we constructed an exploit using C that passes in a tainted format specifier that interacts with the stack. Then, we built the vulnerable program that contains a #define that renames main() as real_main(). Our exploit contains main() and our vulnerable program took in malicious arguments by calling real_main(argc, malicious_argv).

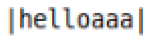


```
#define main real_main
```

Fig. 2. Line in calc.c to connect to our wrapper.c

We ensured that our program was built as a x86-32 application with an executable stack with the defense mechanism for insertion of stack canaries and virtual address randomization turned off.

For the arbitrary read exploit, we created this exploit such that a user could provide an address for an arbitrary read to a %s conversion operator. We take the provided address that we want to read from as an input to the exploit and programmatically construct the conversion specifier(format specifier) such that that it aligns with the stack so the final %08x operator makes the stack point to the specified address for the arbitrary read.



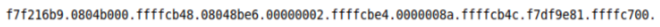
```
|helloaaa|
```

Fig. 3. String from address given

For the arbitrary write exploit, this was more challenging for us, as a group. This exploit sequence required us to programmatically construct the tainted format specifier. The beginning of the format specifier is a good location to contain the shellcode. We aligned the tack such that the sequence of four %n conversions in the format specifier use the appropriate address to precisely overwrite the vulnerable function's return pointer(address) with the 4-bytes containing the address of the shellcode. We provided these two as arguments. In this attack,we altered the tainted format specifier to contain shellcode such that it will overwrite the contents at the vulnerable function's return pointer(address) with the address of the shellcode using a sequence of four arbitrary writes via the format specifier %n conversions.

3 Accomplishments

It took us some time to fully understand format string vulnerabilities. Once were able to learn more about how to utilize the strengths of eclipse we levered them to help us understand the organize of the stack. With our simple calculator program, we were successfully able to conduct the exploits on our simple calculator program in C.



```
f7f216b9.0804b000.ffffcb48.08048be6.00000002.ffffcbe4.0000008a.ffffcb4c.f7df9e81.ffffc700.
```

Fig 4. Data leaked from the stack

4 Lessons learned

We fully understood how format string vulnerabilities could be exploited. First we learned that a format function takes a variable number of arguments and when the function acts on the format strings, it accesses the parameters given to the function. We learned that if a user could pass in a format string to a format function, like printf(), a format string vulnerability exists. As a result, the behavior of the format function is changed and a user could gain control over a target application.

From the reading, we learned that:

- The format string controls the behavior of the format function
- Format string vulnerability makes bugs exploitable as a user can supply malicious input
- We can crash the program , for example, crashing a daemon that dumps core to see useful data within the coredump or have a service not responding during DNS spoofing by printf("%s%s%s%s…").
- %n in printf is used to write the number of bytes already printed, which in turn can be saved into a variable
- we can perform a partial dump of the stack by printf("%08x.%08x.%08x\n")

Viewing memory at any location was a tougher concept to us to grasp. We learned that we have to get the format function to show the contents of memory from an address we supply as a group. We can use %s to display memory from a stack supplied address. Now, we need to ensure the address on the stack is in the right place. To do this, the we use the format string on the stack. The format function string lies on the stack itself. So if we can get this pointer to point to that format string where we have provided the address we can force printf to print the memory at that address with the '%s' format specifier. To increase the internal stack pointer of the format function towards the top of the stack to point into the format string, we used '%08x' parameters to get the pointer to point to the format string itself.For example,printf("\xe0\xcb\xff\xff_%08x.%08x.%08x.%08x.%08x|%s|") dumps memory from 0xffffcbe0 till a NULL byte is seen terminating the string.

To overwrite an arbitrary memory, we can find instructions that modify the $EIP, a register in the CPU, and take control over how these instructions modify the $EIP. Format string vulnerabilities are different from buffer overflow vulnerabilities as in buffer overflows, we overwrite the return address on the stack to point to NO-OP slide still on the stack. However, for format string attacks, we can extend the length of user input by abusing format string parameters. For example, if a printf or sprintf does not check the length, this vulnerability could be exploited to break out of the memory allocated for local buffer and rewrite the return address. To counter this, we should use secure functions such as sprintf to ensure that the outer boundaries of the buffer is not accessible and that stretching format parameters cannot be inserted.

5 References

[1] scut/team teso, "Exploiting Format String Vulnerabilities", September 1, 2001, version 1.2.

[2] StackOverflow, https://stackoverflow.com/questions/5672996/format-string-vulnerability-printf?fbclid=IwAR33YlcBh51kVmSyVvWHrxvuT9yLxENbqfRH-Ue2quUgpLRdKiQFvdeKMSw, 2011

[3] Bouchareine, https://www.win.tue.nl/~aeb/linux/hh/kalou/format.html, 2000

[4] Wikipedia, https://en.wikipedia.org/wiki/Stack_buffer_overflow