Jon Franck

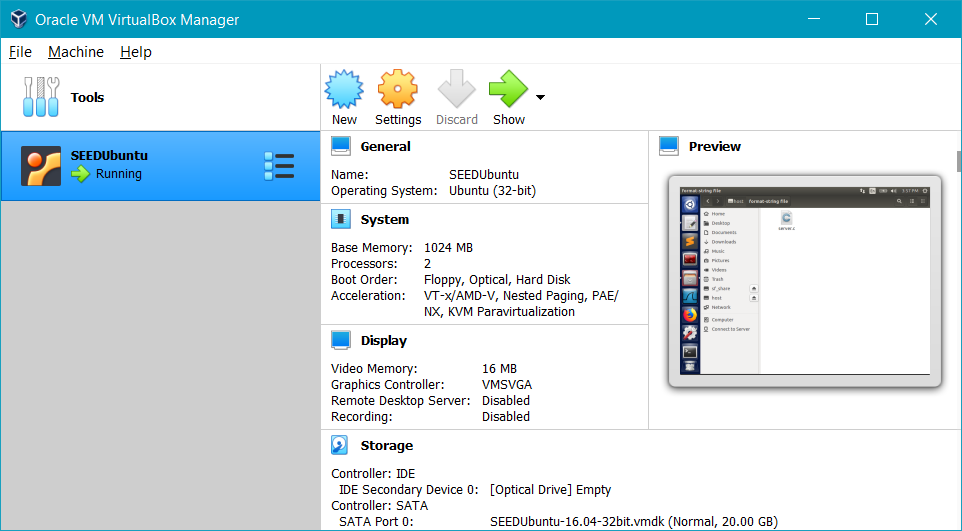
2/5/2021

CSC 4413

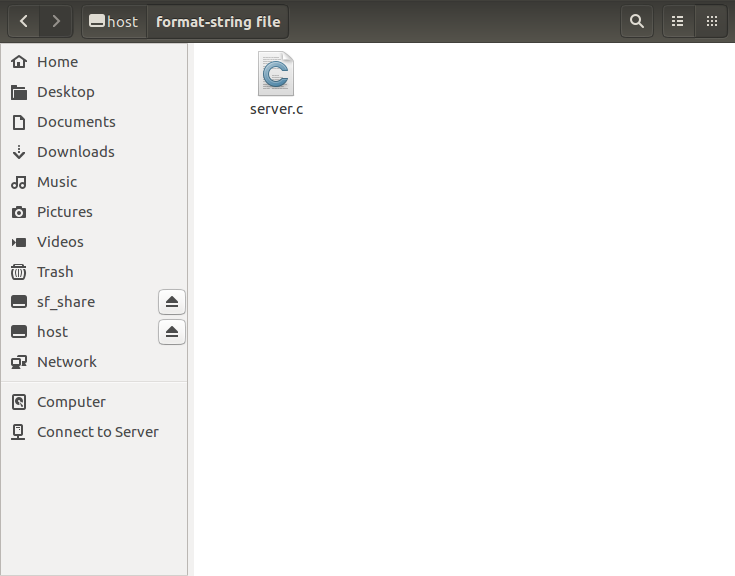
# Format String Lab

**Objective:** Learn how to exploit the format string vulnerability to achieve the following damage: (1) crash the program, (2) read the internal memory of the program, (3) modify the internal memory of the program, and most severely, (4) inject and execute malicious code using the victim program’s privilege.

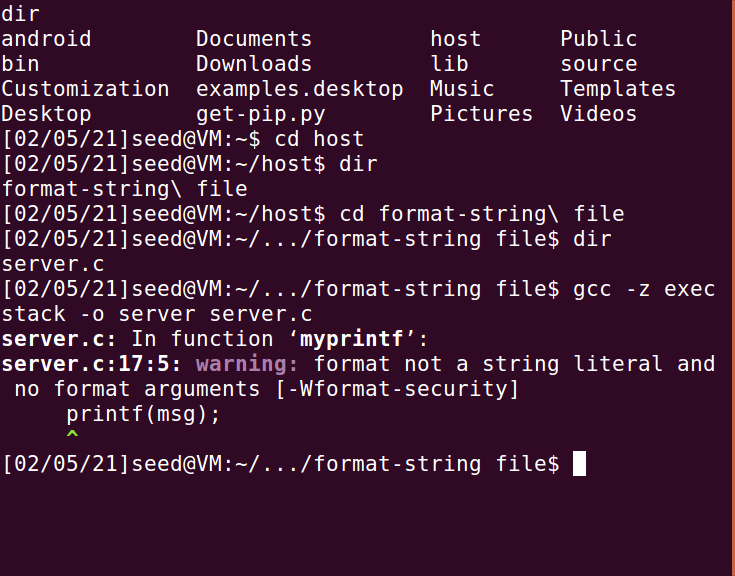
## Step 1: Configure the Virtual Image



SEEDUbuntu downloaded from <https://seedsecuritylabs.org/lab_env.html>



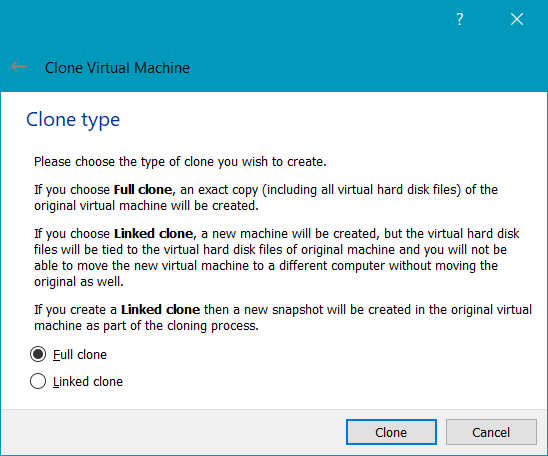
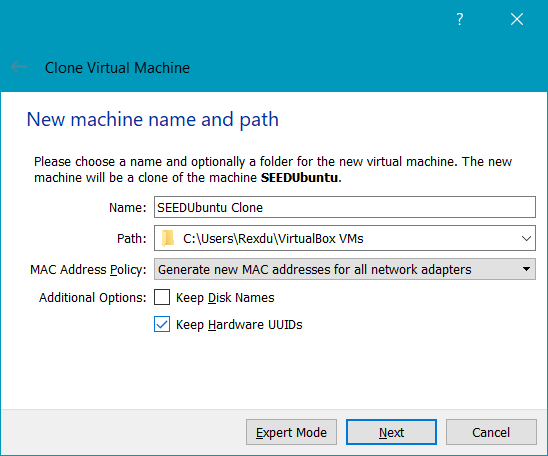
Folder Sharing configured to allow access of files between the host and the VirtualBox operating system.



Compile the server.c program. (Make sure to use -z exec to ensure that the stack is executable)

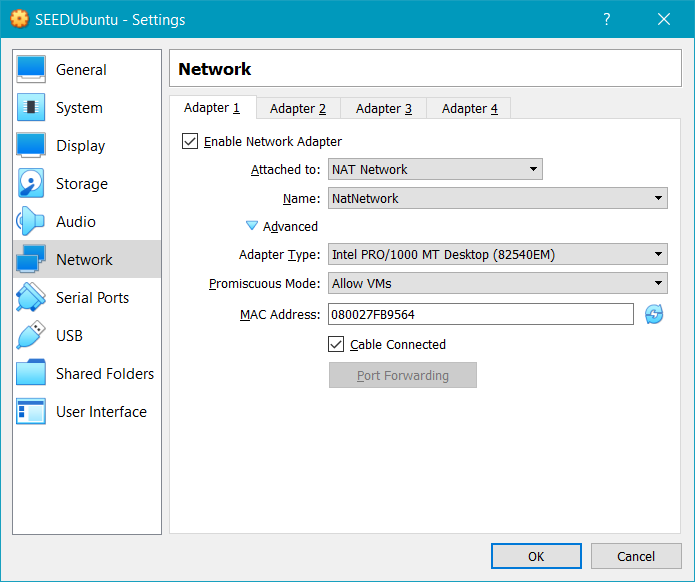
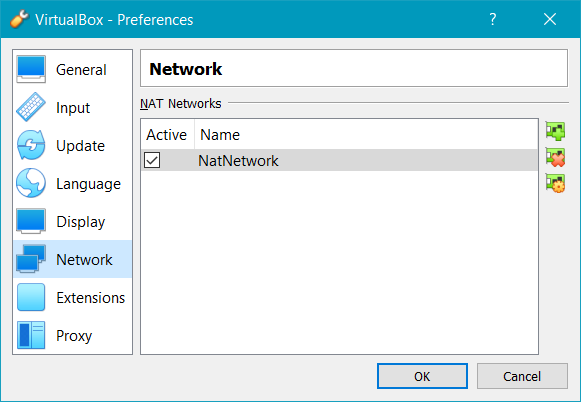
Note: The warning message is a countermeasure built into GCC against format string vulnerabilities; we can ignore it for the time being.

### Create a Clone

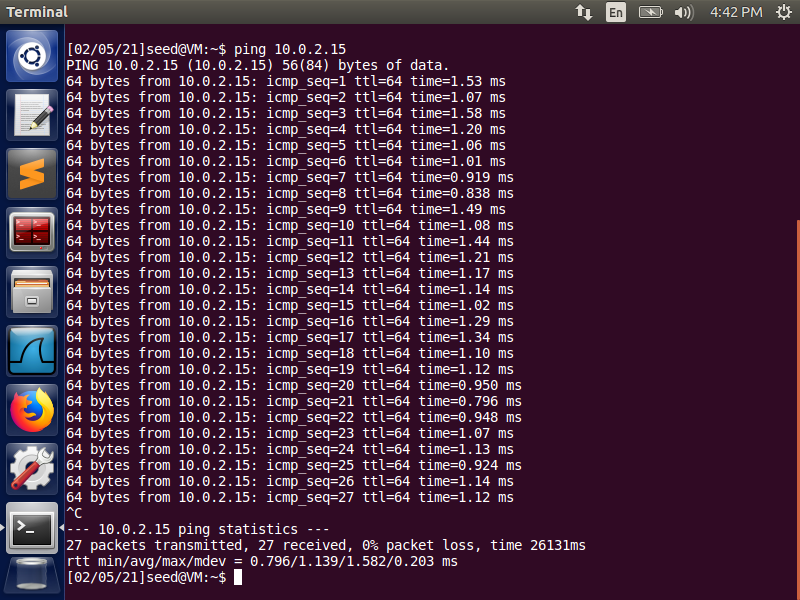
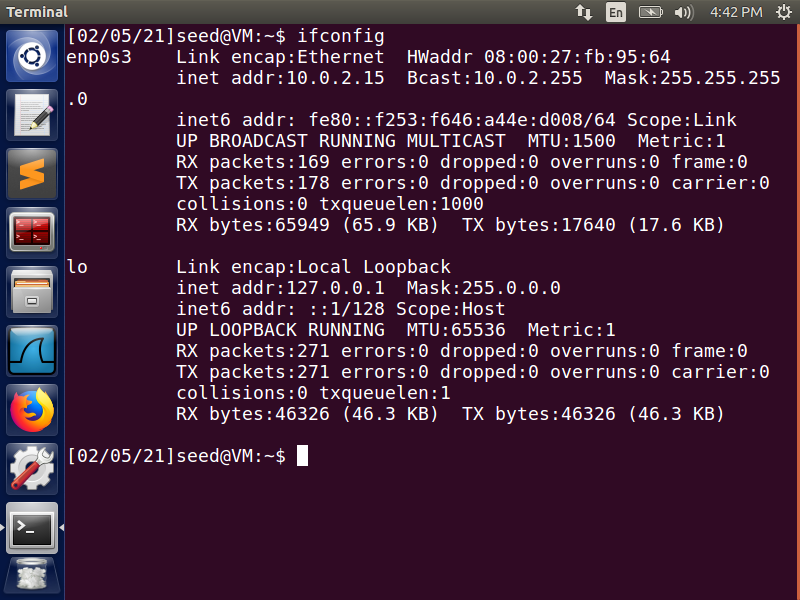


Clone the virtual machine in order to be able to run two at the same time.

### Network Configuration

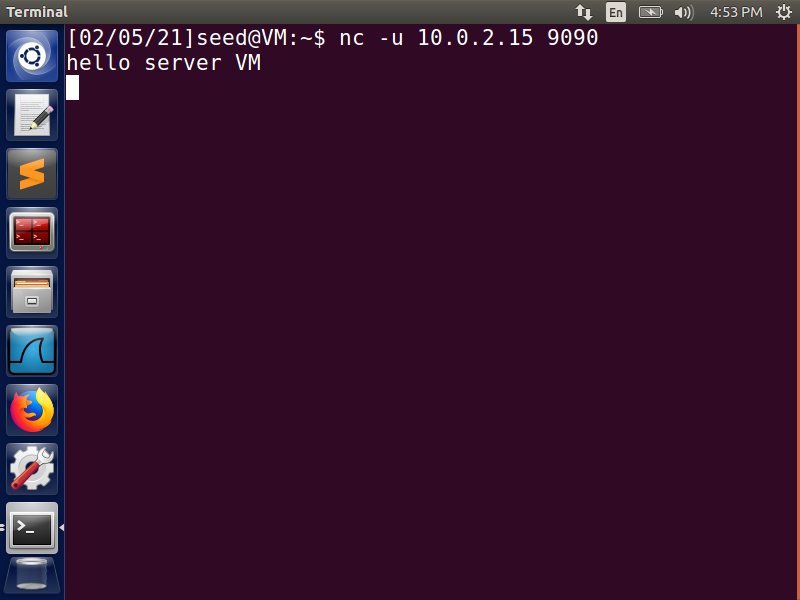
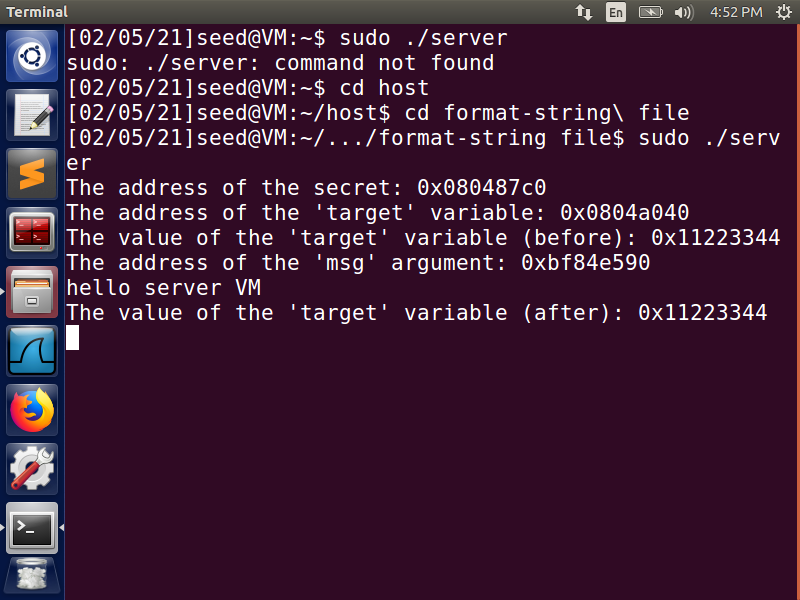


Create a new NatNetwork Adapter and enable it for both VM (make sure all other network adapters are disabled). Set the Promiscuous Mode to Allow VMs.



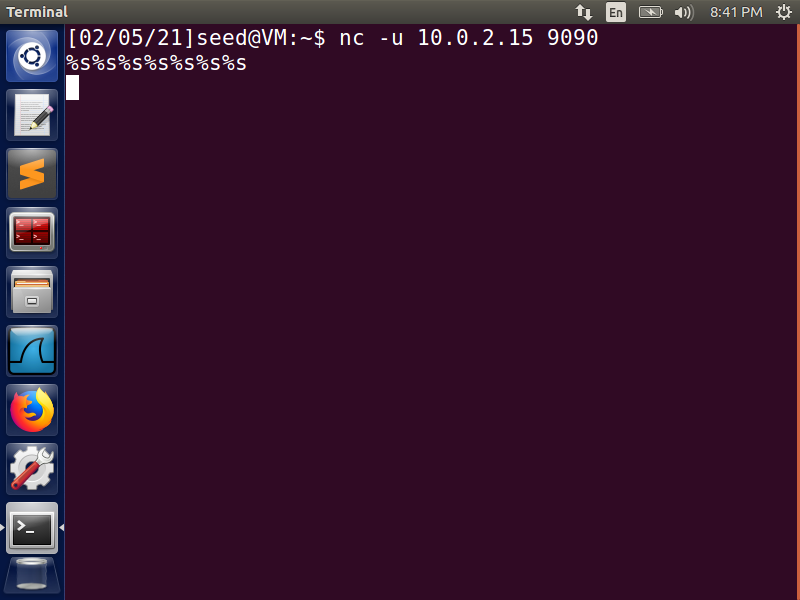
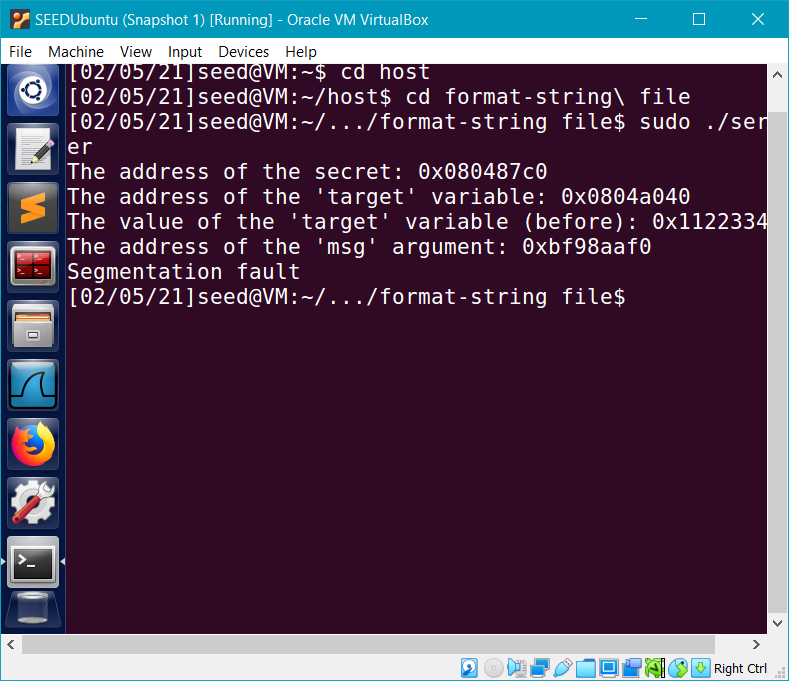
Check the VM’s IP addresses and make sure they can ping each other.

### Test the Server



Run the server on one VM and use the client VM to send a message to the server.

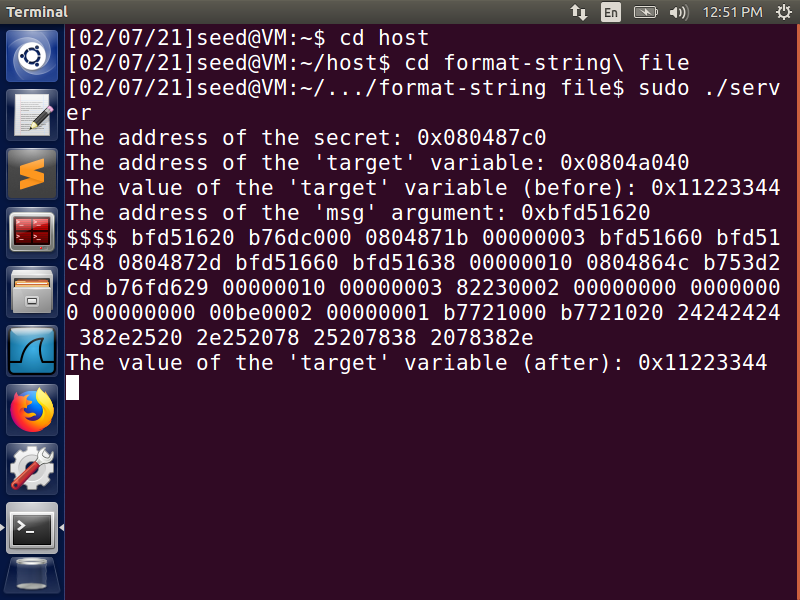
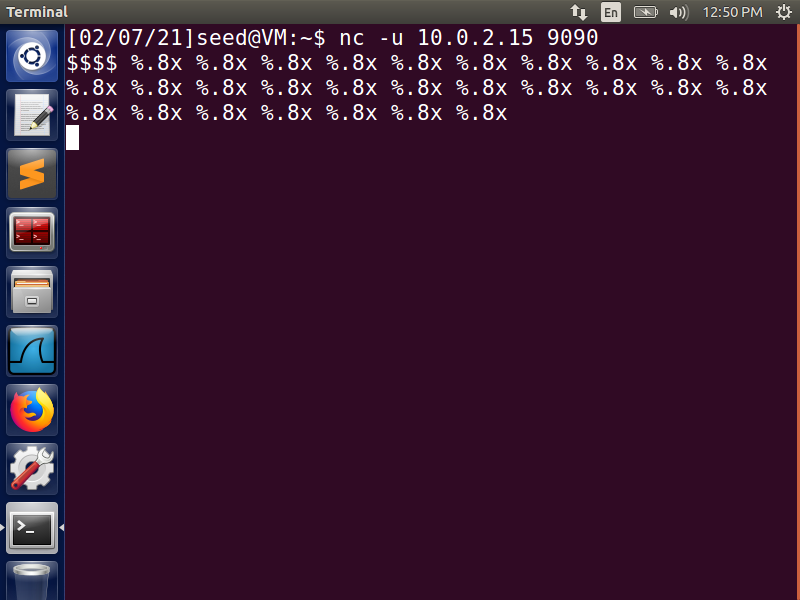
## Step 2: Crash the Program

From the client VM send a message that will cause the server program to crash when it tries to print it out using the printf() function.

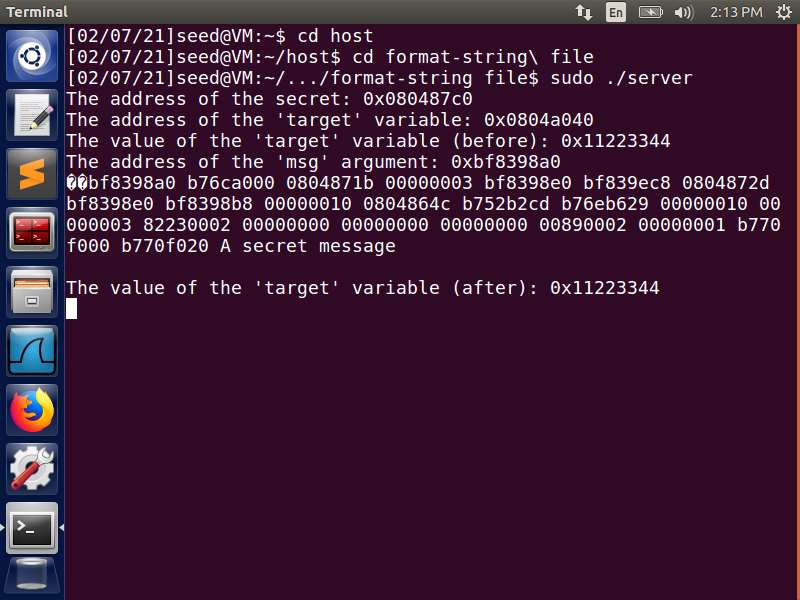
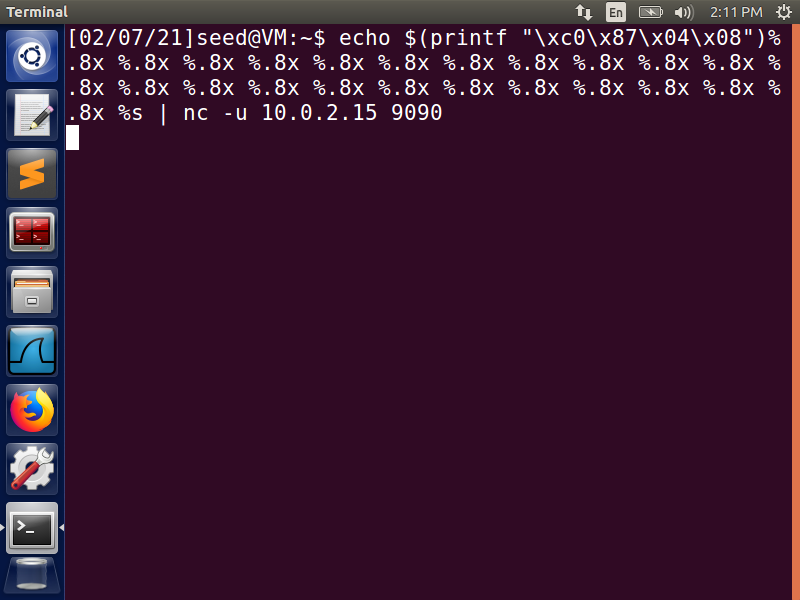
## Step 3: Print Out the Server’s Memory

**Task 3.A: Stack Data.** The goal is to print out the data on the stack (any data is fine). How many  
format specifiers do you need to provide so you can get the server program to print out the first four bytes of your input?



Since $ has a value of 24, we can expect the value of our input ($$$$) to be 24242424. We can use the %x format with a .8 precision value to format the server output as 4 hexadecimal values representing 4 bytes of data. By using multiple “%.8x” format specifiers, we can see how far back in the server’s memory the input data is stored. In this case, 24242424 is the 24th %x.

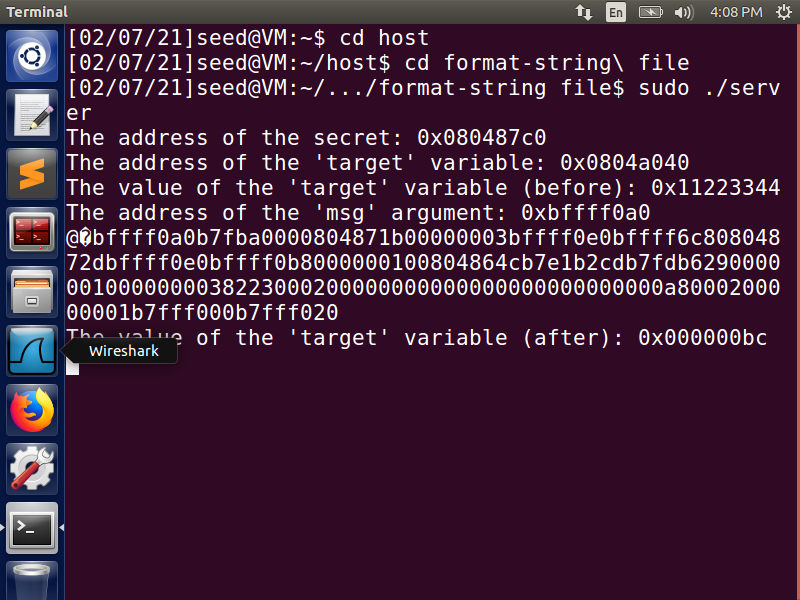
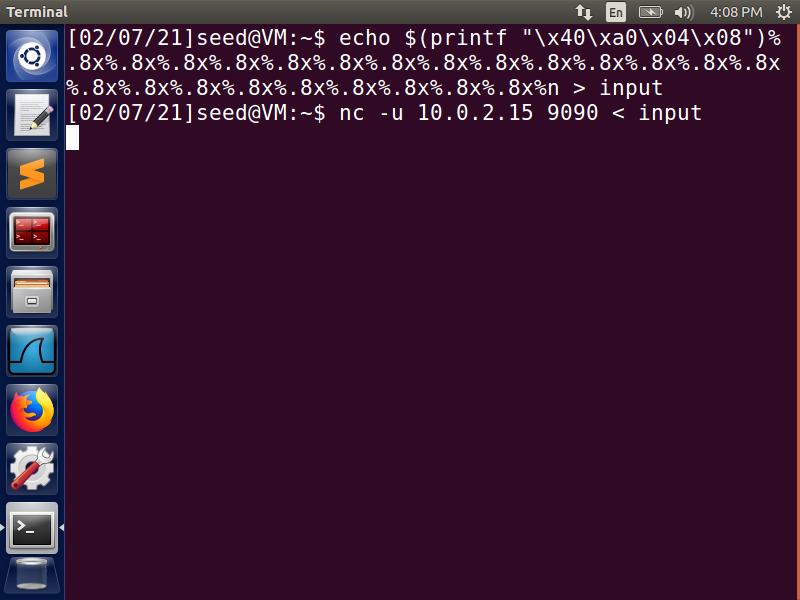
**Task 3.B: Heap Data** There is a secret message stored in the heap area, and you know its address;  
your job is to print out the content of the secret message.



Since we know the address of the secret, we can find where in the memory that address is and rather than formating the data with “%.8x” to show the address value, we can use “%s” to show the string value.

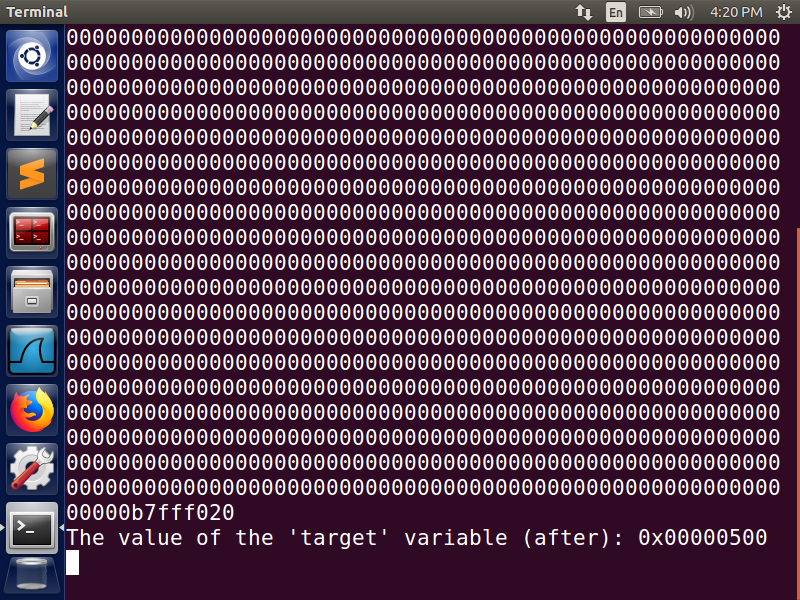
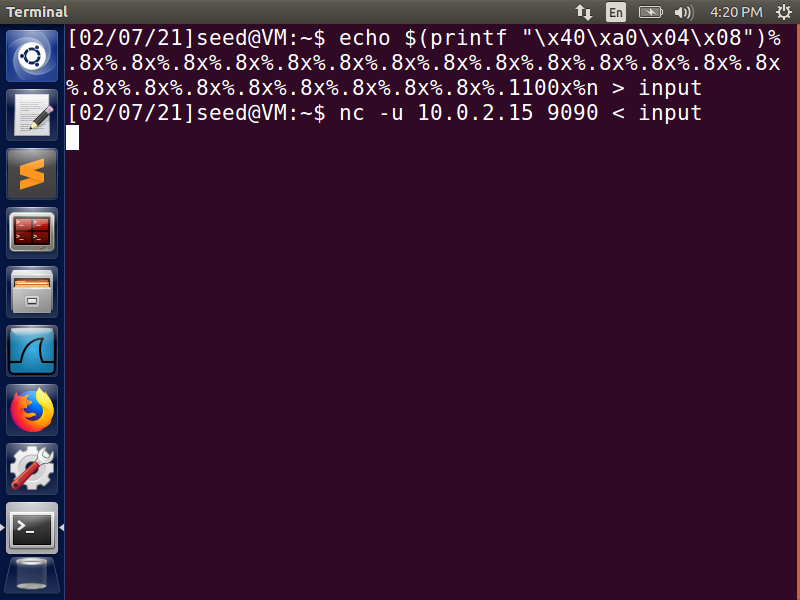
## Step 4: Change the Server’s Memory

**Task 4.A: Change the value to a different value.** In this sub-task, we need to change the content of the target variable to something else. Your task is considered as a success if you can change it to a  
different value, regardless of what value it may be.



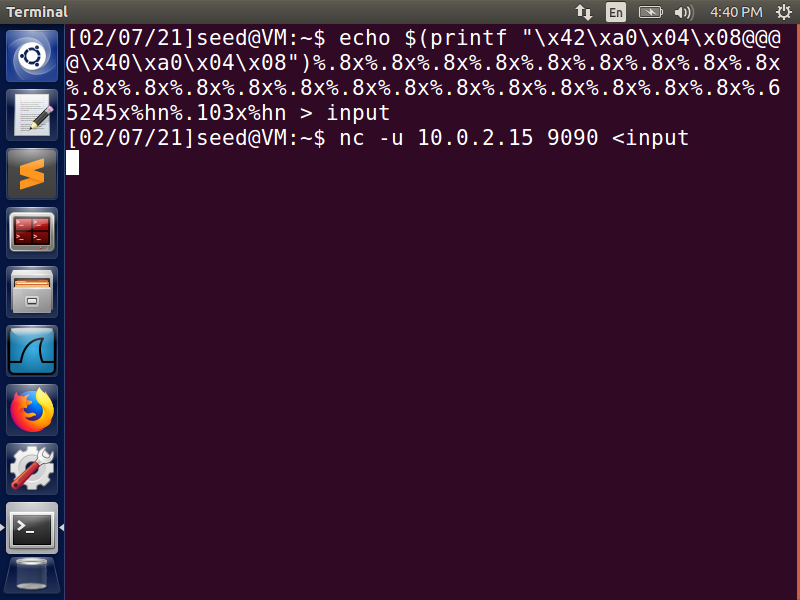
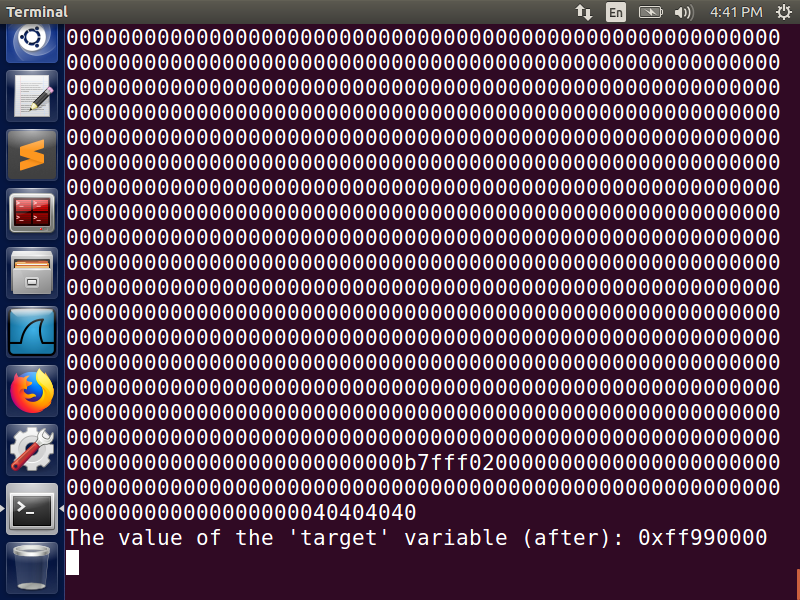
Using %n we can change the value of the ‘target’ variable to the value of the number of characters before the format specifier. In this case, there are 188 characters printed before the %n so the value is changed to BC (which has a value of 188).

**Task 4.B: Change the value to 0x500.** In this sub-task, we need to change to the content of the  
target variable to a specific value 0x500. Your task is considered as a success only if the variable’s  
value becomes 0x500.



Since the hexadecimal 500 has a decimal value of 1280, we need to print 1280 characters before the %n. There are 180 characters before the 23rd %x so we have to change that 23rd “%.8x” to “%.1100x.” This gives us the 1,280 characters before %n (1,280 – 180 = 1,100), resulting in the hexadecimal 500.

**Task 4.C: Change the value to 0xFF990000.** This sub-task is similar to the previous one, except  
that the target value is now a large number.

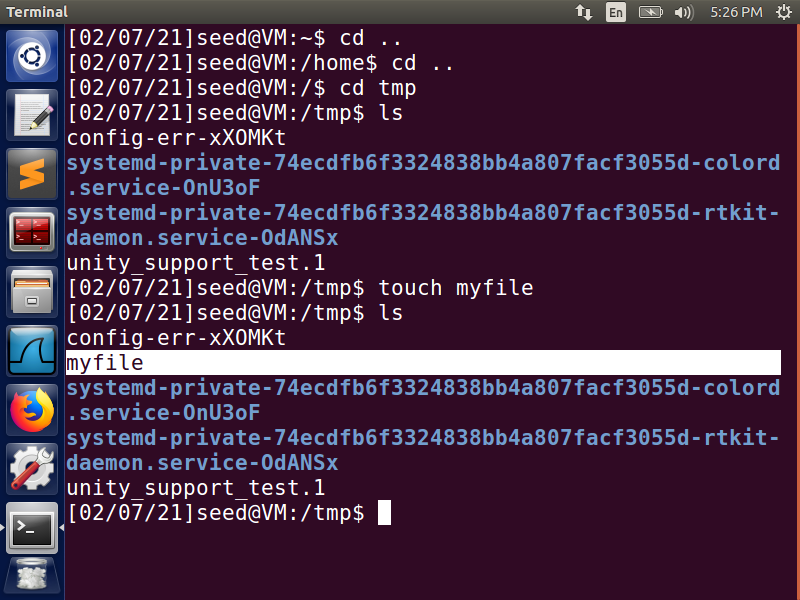
 

Because the new value is so large, we can split the target four-byte memory space into 2 two-byte memory spaces and use %hn (instead of %n) to modify them. For the first we simply use the same method to print 65,433 characters before the first %hn (FF99 hexadecimal = 65,433); therefore we use 65,433 – 188 for the 23rd %x, i.e. %.65425x.

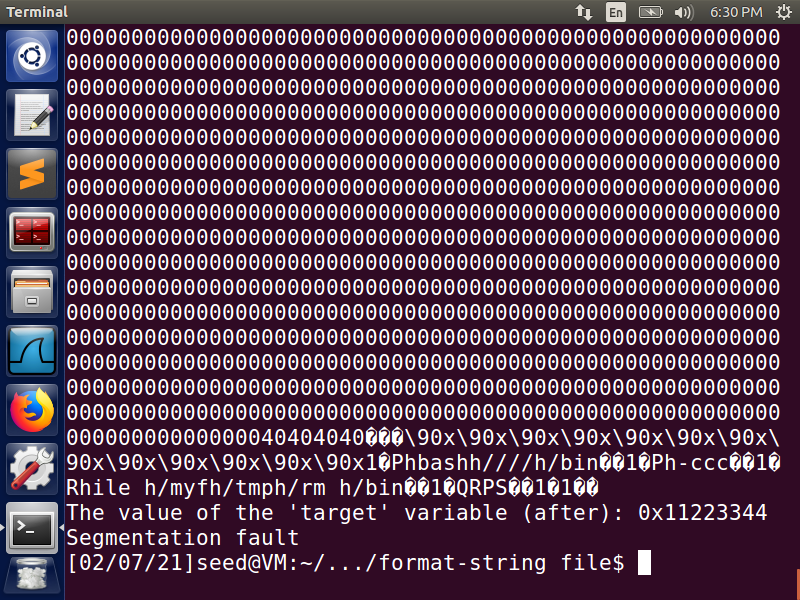
The second memory space is more complicated as you can’t directly store a value as small as 0000, so we must casue an overflow to get the value. As 2­­16 = 65,536, if we can get that many characters in the 16-bit memory space before our target, it will give us a value of 0000 for the target memory space. We already have 65,433 characters so by adding 103 more (%.103x) we overflow that 65,536 value limit to the next memory space (the target).

## Step 5: Insert Malicious Code into the Server

Now we are ready to go after the crown jewel of this attack, i.e., to inject a piece of malicious code to the server program, so we can delete a file from the server.



First, create a new file that we can use to test the malicious code.

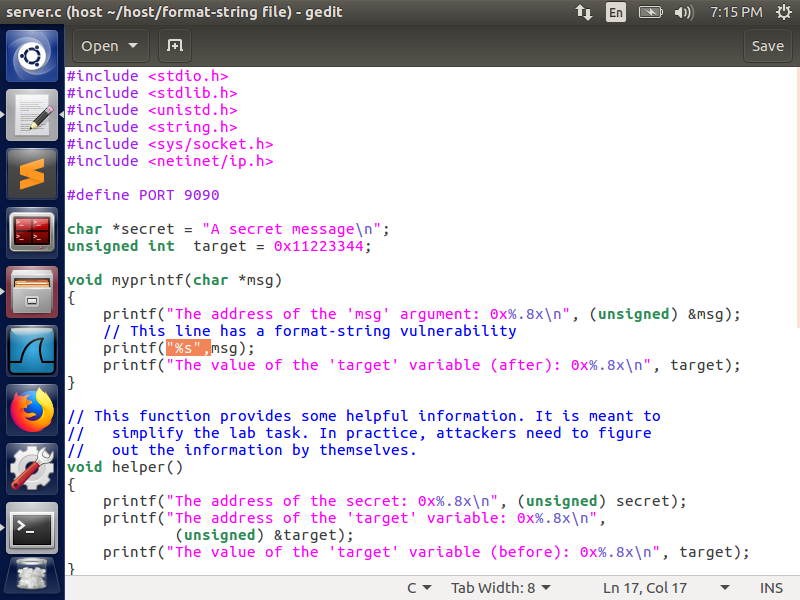
 

The goal is to modify the Return address, which is 0xBFFFF09C and is stored at the beginning of the buffer. We break this into two 16-bit (two-byte) memory addresses rather than a single 4-byte address so that the process is fast (as in the previous task). We then use the precision modifier to store address of the malicious code in the return address. The address of the malicious code is 0xBFFFF15C, therefore we use 48,963 for the first storage location ( 48,963 + 188 = 49,151; BFFF) and 12,637 for the second (49,151 + 12,637 = 61,788; F15C).

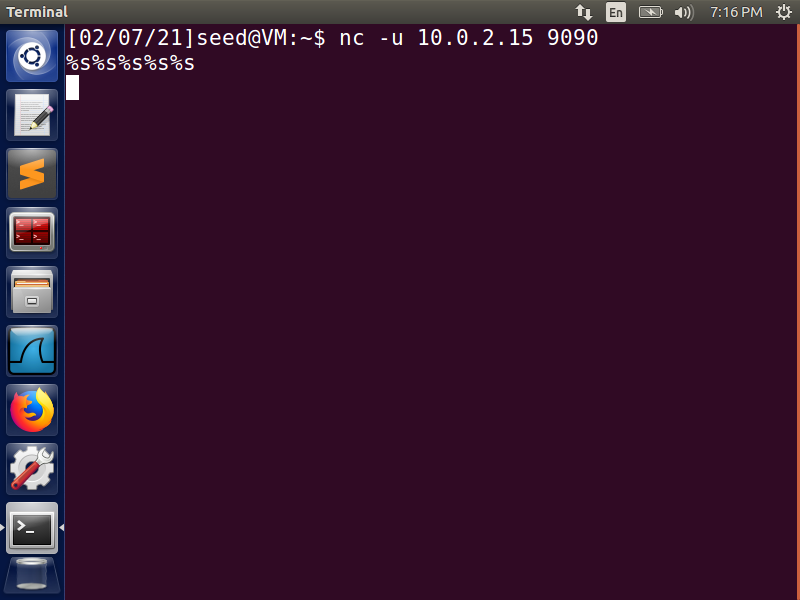
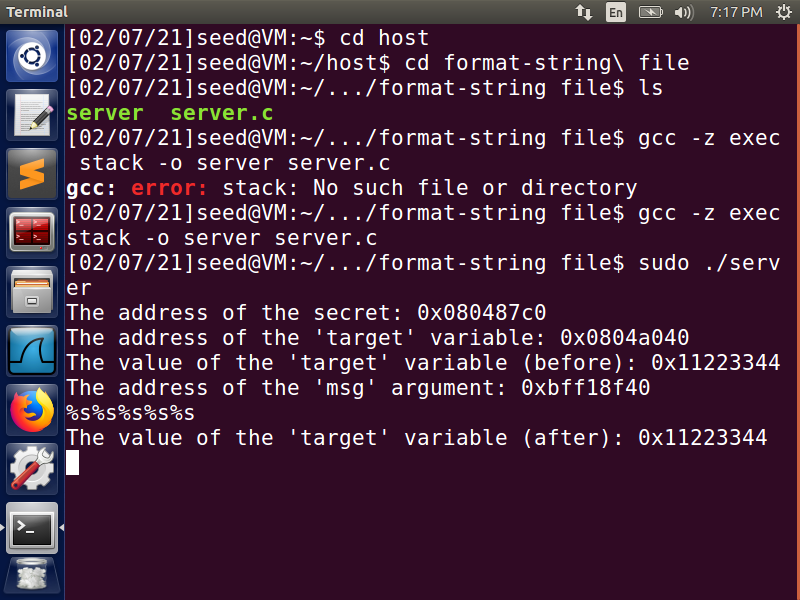
Unfortunately, my command causes a Segmantation Fault, so the malicious code is unable to execute and the file is not deleted; however I am unable to figure out why this persists.

## Step 6: Fixing the Problem

Remember the warning message generated by the gcc compiler? Please explain what it means. Please fix the vulnerability in the server program, and recompile it.



Add “%s”, before the msg argument inside the printf() call.

As seen above, there is no longer a compiler warning and the attack does not crash the server program - meaning the fix worked. Essentially the vulnerability that existed was due to a variable for user input that does not have a precise format specifier. By adding the %s specifier, we ensure that the user input is only treated as a string and cannot be executed or return memory data.