Author: Rajat Sethi

Class: CPSC-6200

Date: September 22, 2021

**Assignment 1 Report – FormatString**

**Task 1:**

By sending “%s” to the server, the program crashes because there is no 2nd string argument. As proof, there was no message with smiley faces stating that everything returned properly.



Text, letter

Description automatically generated

**Task 2.A:**

64 “%x” were needed to return the first four bytes of input. The first four “AAAA” are represents the first four bytes of input (0x41414141 in hexadecimal).

A picture containing chart

Description automatically generated



Text

Description automatically generated with medium confidence

**Task 2.B:**

Knowing that the first input parameter is 64th in the stack, I created the following script to pull out the secret message. The script creates a malicious file that contains:

* The address of the secret message
* 63 consecutive “%x” to go through the stack
* A final “%s” at the 64th position

This string will pull out the address and print what is located there.

A screenshot of a computer

Description automatically generated with medium confidence

Sending the “badfile” contents to the server yielded the secret message in the output.

Text, letter

Description automatically generated

**Task 3.A:**

By padding an arbitrary number of characters with garbage, I can change the value of the variable at the specified address to however many characters come before the %n. The following screenshot shows the python script used to make ‘badfile.’ It is very similar to Task 2’s, except it pads the string with “A” and replaces (%x 🡪 %c) and (%s 🡪 %n).

A screenshot of a computer

Description automatically generated with medium confidence

A picture containing text

Description automatically generated

**Task 3.B:**

Using a similar script that padded the input until it hit 0x5000 bytes, I received the following result. In this case, I did not brute force the padding with 0x5000 “A” characters.

A screenshot of a computer

Description automatically generated with medium confidence

Graphical user interface, application

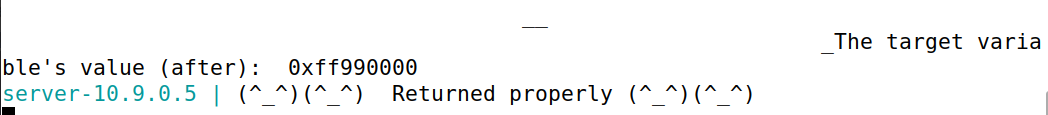
Description automatically generated

**Task 3.C:**

Using the %hn format string, I was able to change the bytes individually. First, I changed the first two bytes at 0x080e506A by padding 0xFF99 characters. Second, I changed the last two bytes at 0x080e5068 by padding 101 more characters to overflow the bytes from 0xFF99 to 0x0.

A screenshot of a computer

Description automatically generated with medium confidence



**Task 4.1:**

1. The address marked at 2, the return address, is 0x4 more bytes than the frame pointer at 0xFFD5B83C
2. The address marked at 3, the input buffer, is at 0xFFD5B910
3. As in the previous tasks, 64 “%x” specifiers are needed to reach the first input.

Text

Description automatically generated

**Task 4.2:**

Breaking into system to acquire a reverse shell using format strings is a several step process.

* Find the location of the return address.
* Find the location of the buffer.
* Create a payload that overwrites the return address and points to the middle of the buffer, which NOP sleds into the malicious shellcode.
* Open a netcat listener on a separate terminal.
* Send the malicious payload to the victim.

This screenshot shows the several addresses relevant to the program. The frame pointer is 4 bytes under the return address, which is accounted for in the payload.

Text

Description automatically generated

Using the method in Task 3, I overwrote the return address two bytes at a time. The first two bytes I placed an 0xFFFF, and then I overflowed to hit the middle of the buffer at 0xD280 (so it would run into a NOP sled instead of the garbage at the beginning).

I also put the shellcode with a standard reverse shell at the end of the input, which would be exploited using netcat.

In the end, the payload looked something like:

“Format String Return Address Overwrite – NOP Sled – Shellcode”

Graphical user interface, text, application

Description automatically generated

This screenshot represents the root shell obtained through this exploit, which was listening on a separate terminal.

Text

Description automatically generated