**Comp434 – Project 2**

**Format String Vulnerability**

Ertan Can Güner

60610

**Pledge of Honor**



“I hereby declare that I have completed this individually, without support from anyone else. I hereby accept that only the below listed sources are approved to be used:

1. Course textbook,
2. All material that is made available to me via Blackboard for this course,
3. Notes taken by me during lectures.



I have not used, accessed or taken any unpermitted information from any other source. Hence, all effort belongs to me.”

60610, Ertan Can Güner, 14.04.2023



1. **Task 1**

Text

Description automatically generatedOur task is to provide a format string such that it crashes the server running on 10.9.0.5:9090. Simple way to crash a server with format strings is to provide any amount ‘*%s’.* In our lab, it was sufficient to provide a single ‘*%s’.* The results can be seen from Figure 1.

Figure 1: Crashing the Server.

1. **Task 2**

For task 2, we must somehow display the memory contents of the server running the program. We can achieve this by providing *‘%x’* in the format string.

* 1. **Task 2.A**

Based on my experiments, we must provide 63 *‘%x’* modifiers to reach the input buffers contents, which resides in main program, outside of the vulnerable function. Entering one more *‘%x’* to the payload we can reach and print the start of our payload, which can be seen in Figure 2.

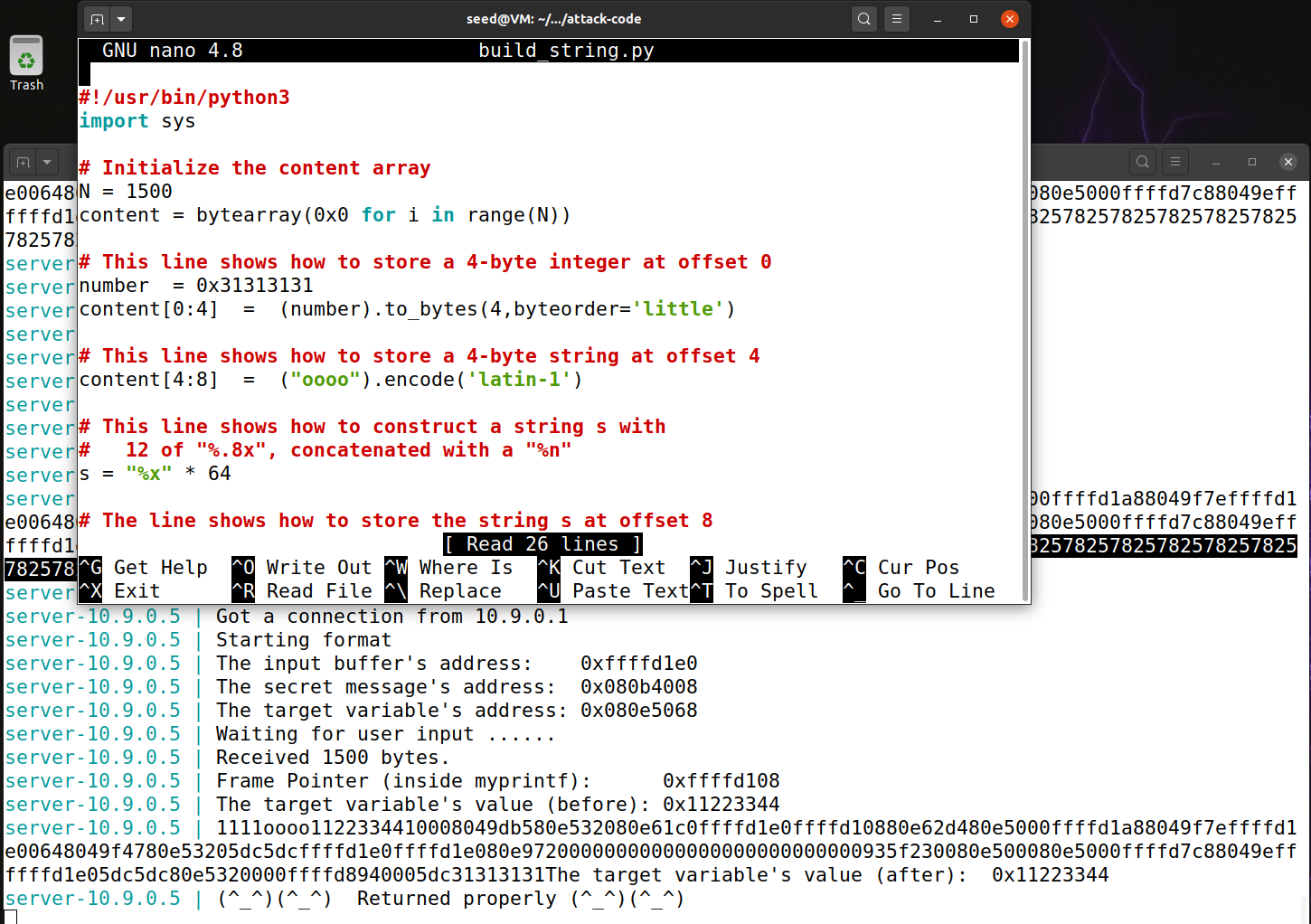


Figure 2: Reaching the Input Buffer

* 1. **Task 2.B**

We need to access a secret message from the heap. We can achieve this by putting the address of the secret message at the start of our payload and overload the va\_pointer to it and print out the string. The payload construction can be seen in Figure 3.

Secret message is “A secret Message”.

**Text

Description automatically generated**

Figure 3: Extracting the Secret Message.

1. **Task 3**

We managed to traverse and read the memory contents and for this task we must overwrite a specific memory location. We can achieve this by using *‘%n’* modifier in our format string. This modifier writes the number of characters that is printed out to the by printf. We can also use length modifiers to adjust the number of characters printed out so we can write any data to the memory of the program.

* 1. **Task 3.A**

First, we need to write any arbitrary number to the target variables address which can be retrieved from the servers print out. We first enter the target address to our payload and as same as before we add *‘%x’* modifier to our payload so we can reach the input buffer. Then we insert *‘%n’* at the end so when we reach the address, we can modify the contents. The payload construction can be seen in Figure 4.

Text

Description automatically generated

Figure 4: Writing to Target Address

* 1. **Task 3.B**

Now we must write 0x5000 to the target address. We can achieve this by using width modifiers such as *‘%8x’* or *‘%.8x’.* These width modifiers print out characters as specified value whether the data in the memory is smaller than it. Our payload is “0x080e5068%.8x…%.8x%.Dx%n”, there are total of 62 *‘%.8x’* modifiers in the payload, which are used to reach the input buffer. We reserve the last *‘%.Dx’* to reach our desired value. The calculation is as follows. We encode the “0x080e5068” as fourGraphical user interface, text, application

Description automatically generated bytes and each *‘%.8x’* 8 bytes. 20480 = 4 + 8\*62 + D and D turns out to be 19980. The payload can be seen in Figure 5.

Figure 5: Writing 0x5000 to Target Address

* 1. **Task 3.C**

Now we must write 0xAABBCCDD to the target address but if we use the same method in 3.B this takes too long to reach our target. So, we have to divide number into two parts 0xAABB and 0xCCDD and use ‘*%hn’* to write the first half and the second half of the number. This modifier allows it to write two bytes at a time other than *‘%n’* which writes 4 bytes of memory. Our payload is “0x080e506A@@@@0x080e5068%.8x...%.8x%.Dx%hn%.Ex%hn”. Our victim machine is 32-bit and uses little endian encoding which means LSB is stored in the higher address so we divide the address offset by 2 bytes and decide where the small part of the number goes to be efficient. We have to calculate the two values D and E. When we reach D the total number of characters has to equal to 0xAABB so we decrease the .8 modifiers (we have a total of 62 of them) and the starting bytes which are 12. And E value is basically 0xCCDD – 0xAABB because we already printed out 0xAABB number of characters. We need to divide the two address’ by ‘@@@@’ because after we write to the first memory we have another *‘%x’* modifier so it will move the va\_pointer 4 bytes, so each ‘@’ character is 1 byte in memory so we need 4 of them. Payload construction can be seen in Figure 6.

Text

Description automatically generated

Figure 6: Writing 0xAABBCCDD to Target Address

1. **Task 4**

We can read, write to memory as we please so for this task we need to inject code such that we open a revers shell to the server. We achieve this by finding the return address (which is 4 byte apart from stack frame pointer ebp) and overwriting it so it executes our code. We follow the same steps in Task 3.C and now we write some address in our input buffer so when the vulnerable method in the server finished execution and it’s time to return it returns to our payload and continues execution from there.

**Q1)** Address marked by 2 is the return address of the caller function, which is main() in our case. When myprintf() completes its instructions return address is popped to continue execution in main. Address marked by 3 is the pointer of the user input, or payload in our case.

**Q2)** Because we use static address’ by turning of va\_space randomizer the total amount of ‘%x’ we must use to access is 63, which means the offset is 63\*4 = 252 bytes. In real life, this amount is randomized in runtime to make it harder for adversaries to locate the memory locations.

* 1. **Initial Shell**

Text

Description automatically generatedWithout modifying the shellcode given to us, we find the return pointer address which is on top of the ebp register. Return address starts from 0xFFFFD73C, we determine this from the servers print out. We start by initializing our payload by filling it by no-op instruction. Then we add the shellcode at the end of our payload. At the start we follow the same steps from Task 3.C and construct our payload. The result can be seen in Figure 7.

Figure 7: Constructing Payload to Open a Shell in the Server.

* 1. **Reverse Shell**

Text

Description automatically generatedWe can inject code using buffer overflow attack, now we need to open a reverse shell to so we can infiltrate the server. We use the same steps in Task 4.1 but modify the shellcode payload to redirect its connection using tcp to our machine, which is 10.9.0.1 on port 9090 and change the stdin and stderr file descriptors so that the output and input is from our machine. Modified shell payload can be seen in Figure 8. Results can be seen in Figure 9.

Figure 8: Changing Shell Payload to Enable Reverse Shell.

Graphical user interface, text

Description automatically generated

Figure 9: Running the Reverse Shell Payload