ITE-379 Lab Report 1 – Buffer Overflows

Zachary Handel

9/27/2022

# Description

During this lab, we are tasked with performing a buffer overflow. To start, we are instructed to use the SEED labs VM (which has previously been installed for the purposes of the class). This lab will go through the steps of a buffer overflow as well as describe all observations that are made.

# Procedure

Step 1:

Step 1 is to download the necessary lab materials to perform the buffer overflow. These steps have been completed.

Step 2:

Step 2 is to download the vulnerable C program provided by the instructor (labeled vulnerable.c) into the VM.

Graphical user interface, text, application

Description automatically generated

Step 3:

In order to turn of ASLR protections (*describe*), we use” echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space”

A screenshot of a computer

Description automatically generated with medium confidence

Step 4:

Now we are to build the code using the command: gcc -g -fno-stack-protector -z

execstack -o vuln vulnerable.c

Text

Description automatically generated

During the compilation process, a warning showed up explaining that the ‘gets’ function is ‘dangerous and should not be used.’ This can be as foreshadowing for the following process of a buffer overflow.

Step 5:

Now we must analyze the source code file vulnerable.c. The functions that are considered dangerous are the ‘gets’ function as stated in the previous step. The gets function is dangerous because, according to Common Weakness Enumeration, it does not perform bounds checking on the size of its input. This means that an attacker can easily send arbitrarily-sized input to the gets() function and overflow the destination buffer.

Step 6:

Now we are to run the program with inputting a low number of A’s.

Text

Description automatically generated

As we can see, it seems as if the program ran as intended. It runs the first printf() function. Then, it runs the self-made function break\_me() and this is where we input our desired number of A’s.

Step 7:

Now, we are going to try to cause a segmentation fault. With this, I am just holding down the ‘A’ button until I feel as if there are enough to cause the segmentation fault.

Text

Description automatically generated

Here, we can see that we successfully caused a segmentation fault. This was caused by us inputting over the limit of characters within the buffer ( x>60).

Step 8:

We are now going to open the program in gdb using “gdb vuln”.

Step 9:

Now, we are going to set breakpoints around te dangerous function (which is gets()).

Text

Description automatically generated

We set breakpoints on like 5 and line 6 of vulnerable.c. This is because we want to examine the code before it hits the dangerous function of ‘gets()’ and after it hits it.

Step 10:

Inside gdb, we are now going to run some text input. This time, we are going to run a small amount of input in order to see the boundaries/layout of what is in memory.

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

Step 11:

In order to determine where the buffer we want to overflow starts in memory, we use the command ‘x /128bx buffer’. We will also do the command “info frame” to gain more information on address areas of the instructional pointer.

A screenshot of a computer

Description automatically generated with medium confidence

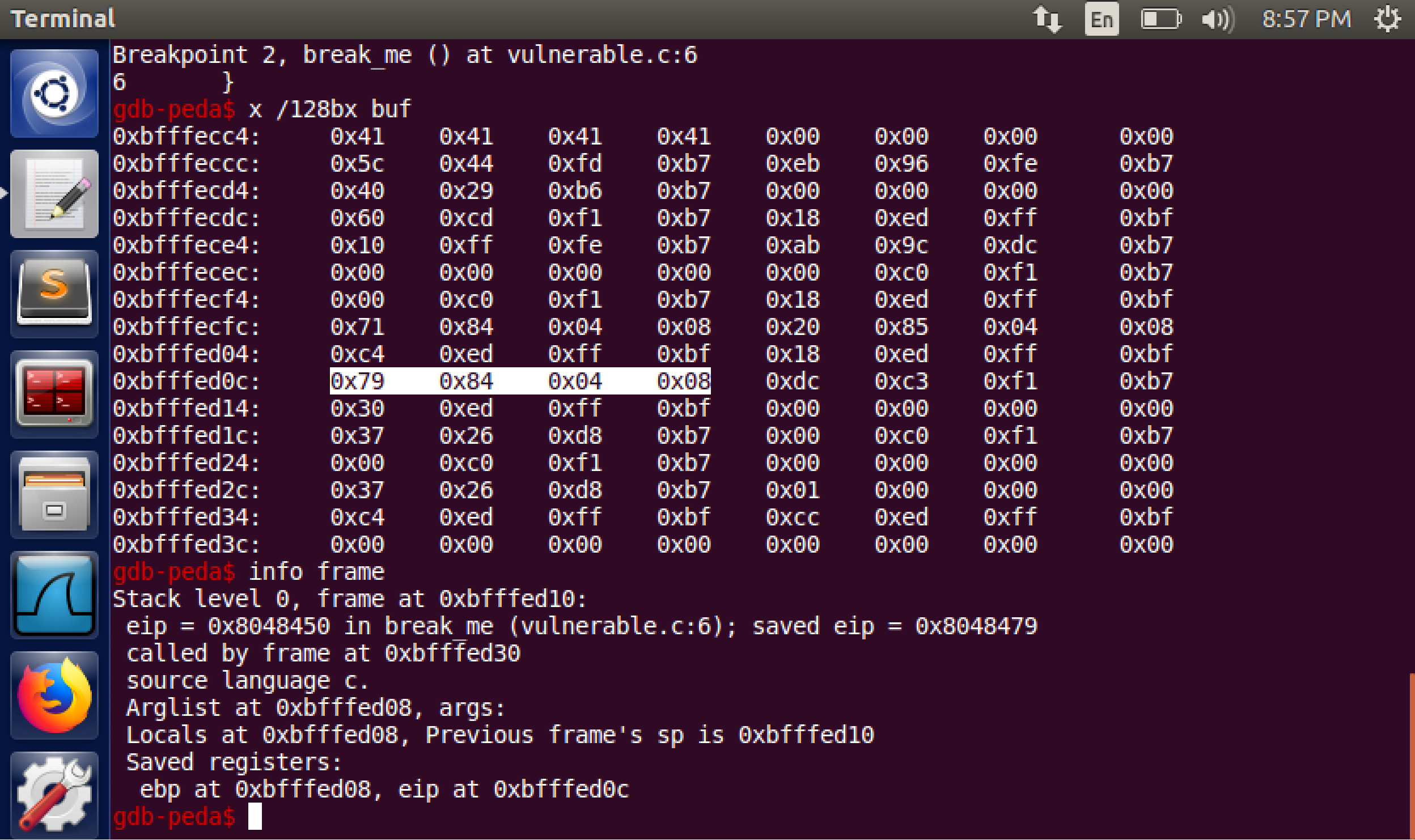
From this image, we can see that the starting address of the buffer (highlighted) is 0xbfffecc4.

Step 12:

As stated above, the ‘info frame’ command will allow you to see the current instructional pointer (labeled EIP) address which is 0xbfffed0c

Step 13:

In order to determine how many bytes we need to read from the buffer to create a buffer overflow, we need to keep in mind what kind of system we are running on. We are running on a 32bit system, we are going to see that 4 bytes make up the address. We also must keep in mind that the address read as “little Indian” (read backwards). Henceforth, the system will read the address backwards up to 4 bytes.



The highlighted portion of the picture is the 4 bytes that our EIP is consisted of. The EIP value is 0x8048479 (we exclude the leading 0 for logical purposes).

The math to find out how many bytes of information we need to store to execute the buffer overflow is as follows:

* Start with where the EIP is located at: bfffed0c
* Then, subtract the start of the buffer: bfffecc4
* This in decimal is 72 bytes of space we need to fill

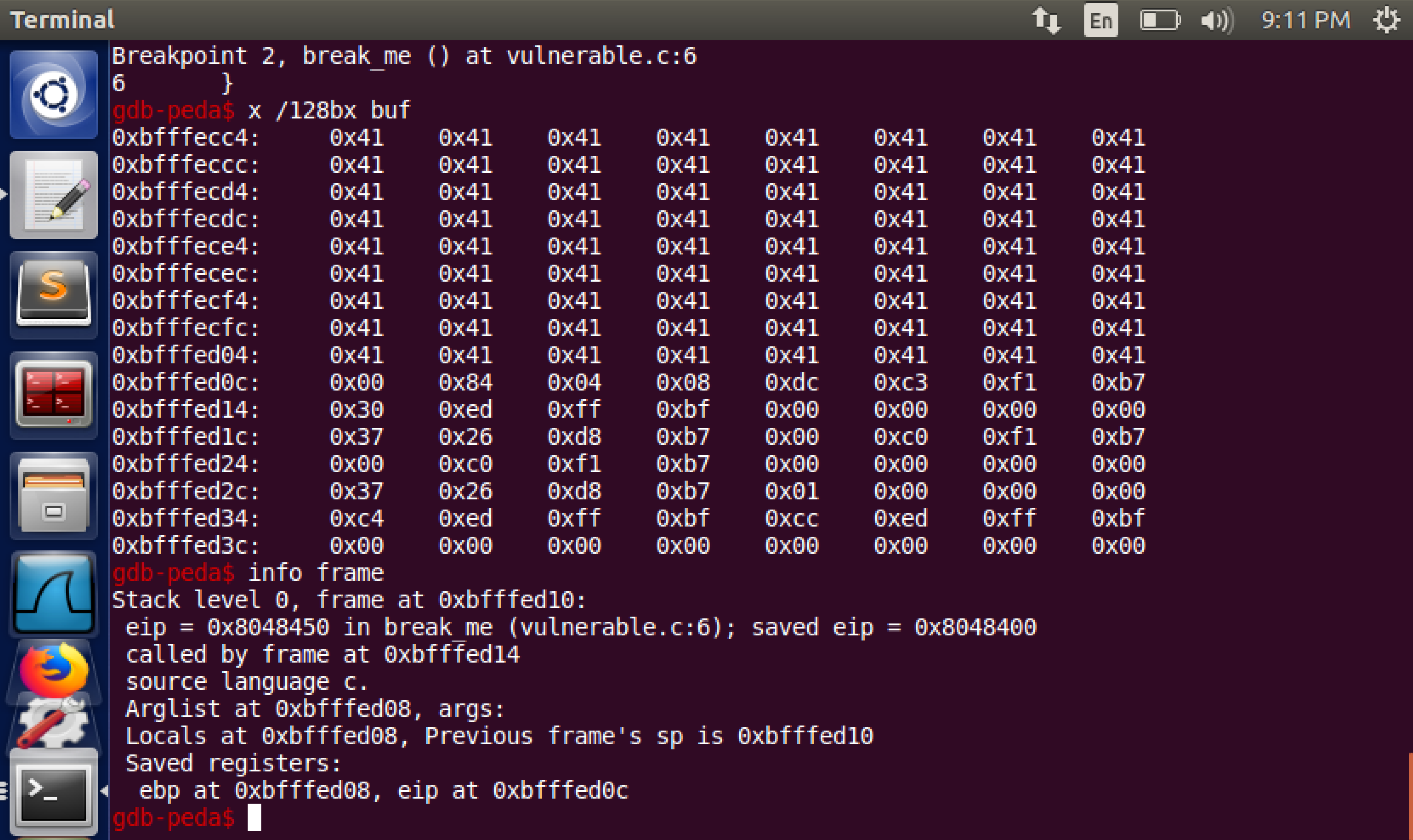
Step 14:

We will now test this math out by filling the 72 bytes with memory. *Note: We need to keep in mind that ONLY the 4 bytes after the 72 bytes we fill will be the EIP.*

Graphical user interface

Description automatically generated with medium confidence

In the image above, we can see the data without the inputted values (*Note, the values of the bytes in memory have changed for the EIP, but the same process still applies from above).*



In the image above, we can see the data with the inputted values (72 bytes of A’s).

Now, lets overwrite the EIP with the A’s we have been inputting (76 bits).

A screenshot of a computer

Description automatically generated with medium confidence

In the image above, it has been highlighted where the EIP has been overloaded.

Step 15:

We will now exit gdb with the ‘q’ command

Step 16:

We are now going to write some shell code in order to execute an attack on our EIP.

For the sake of the project, we are going to go to shell-storm.org to find a shell to use in order to execute our attack.

*NOTE: When trying to write the shellcode2.py file to a new file, it is denying me permission. I am hoping this does not ruin the rest of the lab.*

Graphical user interface, text, application, email

Description automatically generated

Here is the shellcode used to execute the attack. I used the same code Ms. Burchum used in her tutorial video for comparison reasons. The shellcode variable is 21 bytes, and in order to fill up the rest of the data (total of 76 bytes including the EIP) we need to add on 55 bytes of extra data.

Step 17:

We will now write our shellcode2.py to a new file called shellout2

Step 18:

A screenshot of a computer

Description automatically generated

After running the shellcode, here is the data. The attack code can be seen at the start of the buffer. Then, you can see the inputted NOP sled of \x90 until we get to the EIP. Then, the EIP is filled in with the EIP address.

Step 19:

After continuing the program, I unfortunately did not get it to do what the attack was intended to do.

After some playing around, I found some errors I made along the way:

* I did not set the correct location to start the program back to execute the attack code. This made it so it would either a. run as it was normally running or b. just give me a segmentation fault.

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated with low confidenceText

Description automatically generatedGraphical user interface

Description automatically generated

The screenshots above show me doing the following:

1. I ran the program with the shellcode
2. I checked to see if I was using the proper starting address of buf
3. I ran past the breakpoint of the gets() function
4. I checked to see if everything was executed correctly
5. I finished the program to see if it worked

*Note: All has worked, but the product of this work has indeed resulted in me not having a life (all jokes)*

Step 20:

Since my exploit worked, I am going to continue with the lab by running it as a command outside the gdb. I am going to use the command ‘{ cat <shellout2> ; cat - } | vuln

Text

Description automatically generated

I tried 2 options: one being shellout2 and the other being the actual python3 file of shellcode2.py. Both showed the same result and I expect it is doing as it should.

Step 21:

Some things I learned in this exercise include the following:

* I learned more about addresses and how to manipulate code to point to certain addresses.
* I learned about how the EIP works and how we manipulate that.
* I learned about certain functions that are dangerous to coders.
* I learned more about GDB and how we can use it.  
  I learned more about how to navigate a Linux bash terminal.

Here are some frustrations I found during the process:

* Finding the correct number of bytes in order to access the EIP and input the beginning address of the buffer
* Having to rewrite the shellcode into a new file repeatedly in order to use it in gdb
* Navigating the data table was hard at first, but with some practice it became easy to understand
* Even the littlest syntax error in either code, an address, or executing the program in gdb can cause unwanted and confusing errors and that led to a rabbit hole of confusion

With that being said, I enjoyed this process. I am more comfortable using Linux now and I learned the importance of securing code input. It was exciting to see that the exploit did in fact work and it helped me have the mind set of an attacker. By this, now I know how to avoid them in my code.

Finally, I understand that this is a very small scale of a buffer overflow. I want to learn more about how they are applied in real world instances as it will give me insight to how dangerous buffer overflows can actually be.