Lab 03 – Implementing Buffer Overflows

Name: Olivia Flores
Course: IS-3033-ON2
Date: 02/06/2024

INTRODUCTION

In Lab 03, we will be going over implementing Buffer Overflows and I will be taking you along with me on this new journey of learning how this vulnerability works under the hood. Some of the main objectives here will be to learn the basic concepts behind "smashing the stack," running C code with no input validation, running the C code with an out-of-bounds string, and running the GDB debugger with PEDA. These terms are new to me as I go along, but I am welcoming all the information as it will help me understand how buffer overflows and their exploits work and later learn how this can be prevented.

BREAKPOINT 1

As we begin this lab, I wanted to explore a little bit more about buffer overflows and the article, "Smashing the Stack" by AlephOne helped me do that in a way. It began by helping me understand why it is called "smash the stack," what **stack buffers** are and how they work along with other processes in a computer. First, I want to note that the size of a stack is already dynamically adjusted by the kernel at run time, and the examples used in this article display a stack that is grown down which could be common in computers. As I attempt to understand how stacks work, I have come to see that the local variables could be referenced by giving their offsets from Stack Pointer (SP), but when words are PUSHed or POPped off the stack, the offsets change. Those can be tracked but only with further steps and instructions, more work basically. That is when the Frame Pointer (FP) comes into play with helping reference local variables and parameters because the distances do not change from the FP. There is a code called the procedure prolog that helped me picture how the SP works along with the FP and how space is reserved for these local variables. This leads me into understanding that all variables have allocated space either at load time or run time, but here will go over how dynamic buffers overflow, which are those variables allocated at run time. To overflow is to flow, or fill over the top, brims, or bounds [1]. This overflowing happens when the stack has been corrupted by writing past the end of an array declared auto in a routine.

To help me understand further and visualize how this happens, I asked **copilot**, the AI tool, to explain the "smashing the stack" concept and this is how the AI tool responded:

"Smashing the stack is a term commonly associated with **buffer overflow vulnerabilities** in computer programs. Let me break it down for you.
In many C programming implementations, it's possible to corrupt the execution stack by writing past the end of an array declared as **"auto"** within a routine. When code does this, it's said to **"smash the stack"**. The consequences can be severe: the routine might return to a **random address**, leading to insidious **data-dependent bugs**. Essentially, this vulnerability occurs when a function call writes beyond the bounds of an array, potentially causing memory access errors and misuse of memory addresses[1][2]. Imagine a stack of bananas (bear with me!). Each banana represents a memory location. If we accidentally stack too many bananas, they spill over, causing chaos. Similarly, in programming,

exceeding buffer boundaries can lead to unintended consequences. So, next time you see a stack of bananas, think of stack smashing – it's all about keeping things in check! 🍌🔨💥 "[2]
Now I can visualize a stack of bananas representing variables in a stack.

BREAKPOINT 2

In step 2 of this lab, we will be using a snippet of C code that can be vulnerable to exploitation. We can see from the article "**Buffer Overflow - Part 1. Linux Stack Smashing**" that it is easy to identify what part of this program is insecure. Functions like **read(), gets(), strcpy()** do not check the length of the input strings relative to the size of the destination buffer - exactly the condition we are looking to exploit [3].

Here is the original code:

```
/* A simple demonstration of a stack-based buffer overflow */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int overflow(char *input) {
char buf[256];
strcpy(buf, input);
return 1;
}
int main(int argc, char *argv[]) {
overflow(argv[1]);
printf("meow =^..^=\n");
return 1;
}
```

Now let's see how a simple piece of C code will run correctly if we provide acceptable input validation. As I learned, when I run this program, it requires manual input of the characters so I entered the characters woofwoof, using the command **./vuln woofwoof**. This is not a lot of characters to cause a segmentation fault, but I can see where there still needs to be a correction in this snippet of C code.

Before running this command, I copied the snippet of code from the article on cocomelonc and opened a file with the command **nano vuln.c** and was automatically in the nano file, I pasted the code into this file and exited, going back to the command line prompt.
Next, I used the GCC compiler to "build" the object code, already built into this Linux distribution. Lastly, I was able to execute the code using the command **./vuln woofwoof** as shown below in Figure 1.

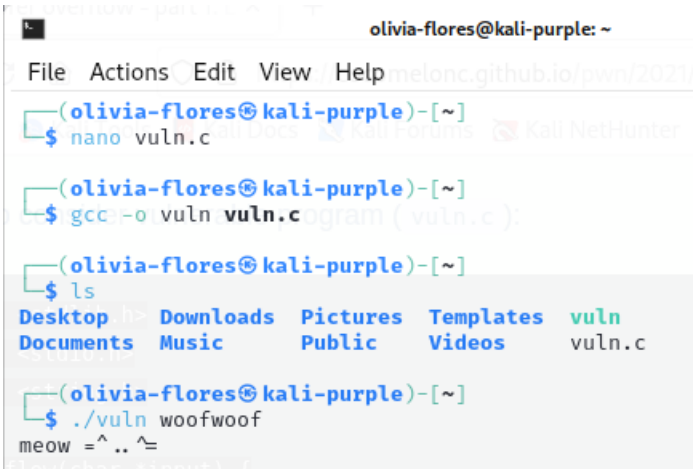


Figure 1: Acceptable input validation.

To understand the snippet of code a bit better, I asked ChatGPT to explain it more thoroughly. ChatGPT was able to explain 3 things included in this code and why this poses a security vulnerability and should not be used. First, it explained the standard header files, then the function definition known as overflow (point of interest/vulnerability), and then the main function which calls the overflow function, and prints the string inside printf(). That is the simple way of putting it, but below I have provided the explanation provided to me in more detail:

1. **Header Files:**
    - **#include <stdlib.h>**: Includes the standard library header file, providing functions involving memory allocation and other general utilities.
    - **#include <stdio.h>**: Includes the standard input/output library, used for input and output operations.
    - **#include <string.h>**: Includes the string manipulation functions.
2. **Function overflow:**
    - Takes a single argument, a pointer to a character array (**char *input**).
    - Declares a local character array **buf** of size 256.
    - Uses **strcpy** to copy the content of the input string (**input**) into the local array **buf**. This is the point of interest and potential vulnerability as it may lead to a buffer overflow.
3. **Function main:**
    - Takes command-line arguments (**argc** and **argv**).
    - Calls the **overflow** function with the first command-line argument (**argv[1]**).
    - Prints "meow =^..^=" using **printf**.
    - Returns 1.

Buffer Overflow: The vulnerability lies in the **strcpy** function in the **overflow** function. If the input string is longer than 255 characters, it will overflow the **buf** array, potentially overwriting the return address or other important data on the stack. This can lead to unexpected behavior, crashes, or even security vulnerabilities.

Note: In practice, using **strcpy** without proper bounds checking is considered unsafe. Modern programming practices recommend using safer alternatives like **strncpy** with explicit size limits to prevent buffer overflows [4].

In addition to this, I asked ChatGPT if they could add comments to this C code to make it more readable and easier to walk through the process of what is happening. I also made note of the vulnerability displayed in the original piece of code and made changes to the function overflow, where **`strcpy(buf,`**

`input);` will now be displayed as **strncpy(buf, input);** to limit the copy length and prevent what would be known as a buffer overflow [4].

Adjusted code:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Function that demonstrates a stack-based buffer overflow
int overflow(char *input) {
   // Local character array (buffer) of size 256
   char buf[256];
   // Copy the input string into the local buffer using strncpy
   strncpy(buf, input);
   // Return 1 (not particularly meaningful in this context)
   return 1;
}

int main(int argc, char *argv[]) {
   // Call the overflow function with the first command-line argument
   overflow(argv[1]);

   // Print "meow =^..^="
   printf("meow =^..^=\n");

   // Return 1 (not particularly meaningful in this context)
   return 1;
}
```

BREAKPOINT 3

As we move into step 3 of this lab, I will introduce us to a website by Matt Godbolt, called **Compiler Explorer**. This is said to be a very useful site that allows us to see code translated from all different languages, but we are going to use it to translate C code into assembly and binary for this particular lab because it will allow us to see how these languages are read by humans compared to a computer's hardware. As I am running through this translator, I was able to stop and read about assembly language and what it is. If you are like me and need a quick run through, I am here to do just that. An assembly language is a type of low-level programming language that is intended to communicate directly with a computer's hardware. Unlike machine language, which consists of binary and hexadecimal characters, assembly languages are designed to be readable by humans. It is obvious that humans would not be able to construct modern software programs by explicitly programming ones and zeros, so that is why we must all rely on various layers of abstraction that can allow themselves to articulate their commands in a format more intuitive to humans [6].

Now that we have covered information about this resource, lets copy the **vuln.c** code above into the lefthand pane at Compiler Explorer. The righthand pane offers numerous options to display the corresponding assembly code. Assembly instructions vary, depending on the computer architecture that

is in play. For this exercise, I selected the X86-64 architecture, the C language, and the C compiler "x86-64 gcc 11.2", and all of the options under Output except for linking [5].

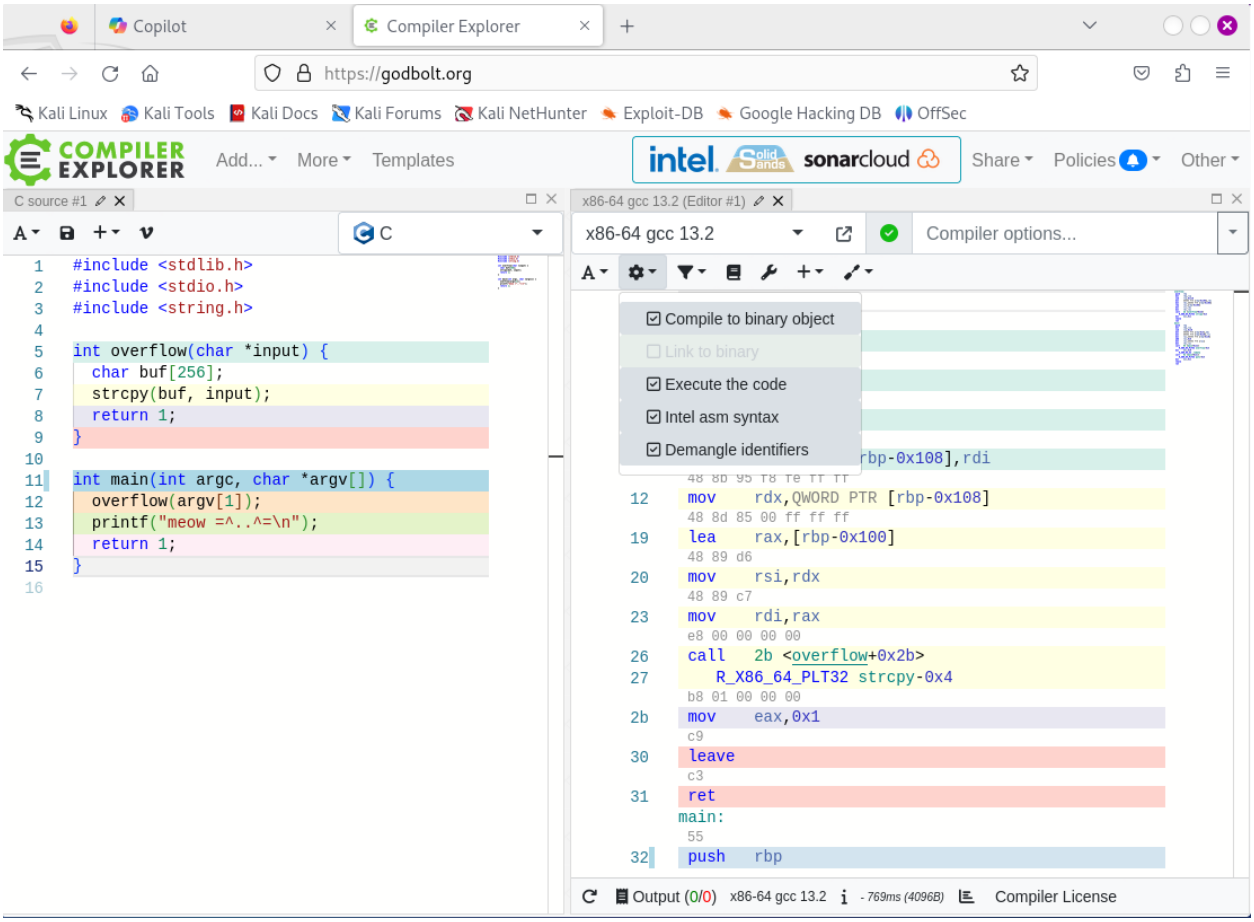


Figure 2: Details of Architecture and language being translated.

Next, I clicked on the **+ "Add new"** tab in the lefthand pane and selected **Execution Only** to pull up the execution pane. In the righthand page, an Executor pane should appear. Then, I clicked on the **>_ "Execution arguments"** option to pull up a text box to input user text that will be an argument to the program call at the command line [5]. I did also note that the Compiler Explorer prohibits the running of vulnerable code, so I went ahead and input an in-bounds, or acceptable input, as a command line argument: **woofwoof** shown in Figure 3.

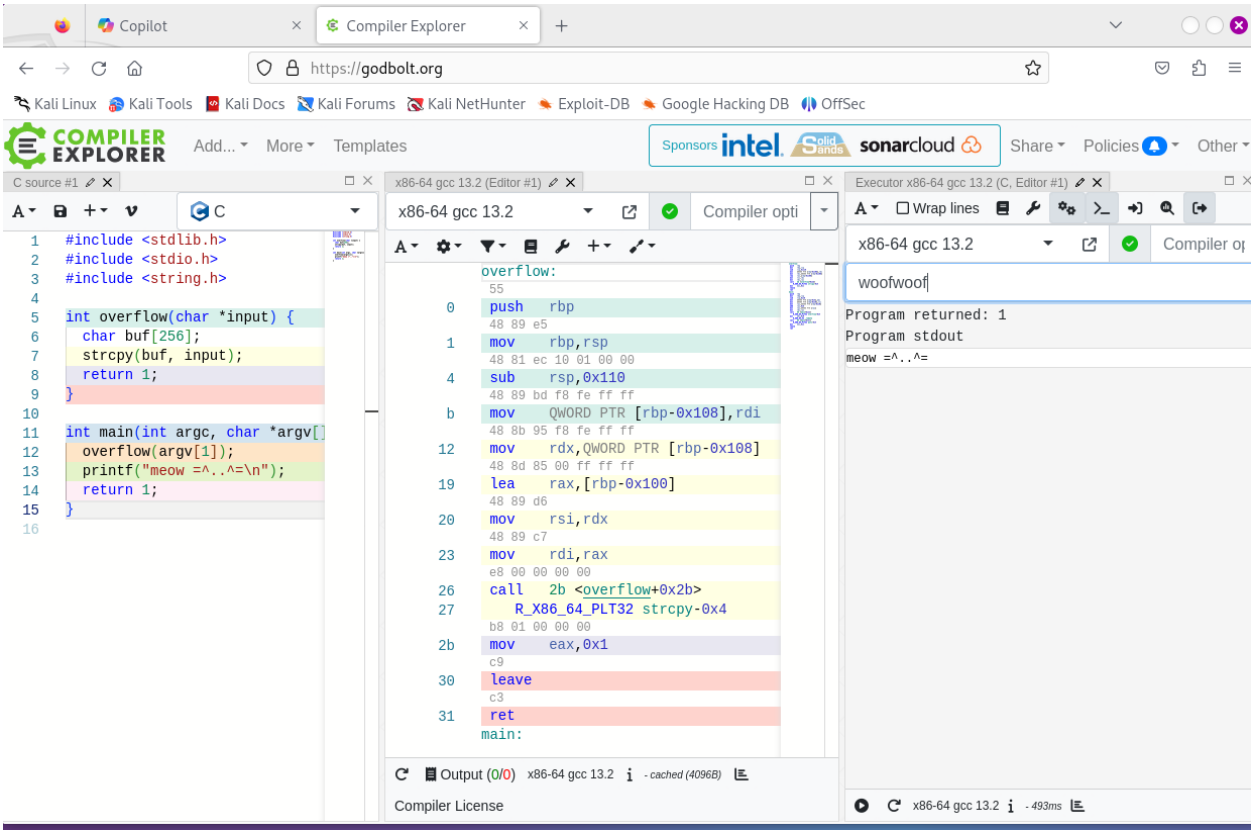


Figure 3: Compiler Explorer running acceptable input.

I want to elaborate a little more on this line of C code: **strcpy(buf, input);** which is used to copy the contents of the 'input' string into the character array or buffer 'buf'. It assumes that 'buf' has enough space to fit the contents of 'input'. If 'buf' is not large enough, it may lead to buffer overflow issues as described in Breakpoint 2 of this lab [4]. This line of C code translated into assembly language is displayed below.
Line of C code Translated into assembly language (opcodes & values):

Opcodes: **Values:**

```
48 8D 95 F8      mov     rdx, QWORD PTR [rbp-264]
48 8D 85 00      lea     rax, [rbp-256]
48 89 D6         mov     rsi, rdx
48 89 C7         mov     rdi, rax
E8 00 00 00 00   call    strcpy
```

Furthermore, on the architecture (X86-64) selected for this piece of code, I asked ChatGPT to help me figure out what each line of this assembly language is doing.

The **first line** shown is moving the 8-byte value stored at the address [rbp-264] into the 'rdx' register likely representing the source string. The **second line** 'lea rax, [rbp-256]' computes the effective address of [rbp-256] and stores it in the 'rax' register, likely the destination buffer. The **third line** 'mov rsi, rdx' moves the value in the 'rdx' register into the 'rsi' register. The **fourth line** 'mov rdi, rax' moves the value in the 'rax' register into the 'rdi' register. Lastly, the **fifth line**, call 'strcpy' calls the 'strcpy' function and this would then copy the content from the source address (rsi) to the destination address (rdi) [4].

BREAKPOINT 4

Next, I am going back to my Kali machine and running the code with an in-bounds string of choice and an out-of-bounds string thereafter. Before doing that though, I must make sure to update GCC to be able to run 32-bit programs. The simpler memory layout of 32-bit (rather than 64-bit) programs make it easier to analyze the effects of buffer overflows [7]. In this part of the lab, I did run into an issue where I did not make sure my network settings were set up correctly, making the process longer than it should have taken, but once that got resolved, I moved on to entering the commands displayed below.

First I entered the **sudo apt-get update** command.

```
┌──(olivia-flores㉿kali-purple)-[~]
└─$ sudo apt-get update
Hit:1 https://artifacts.elastic.co/packages/8.x/apt stable InRelease
Get:2 http://mirrors.jevincanders.net/kali kali-rolling InRelease [41.5 kB]
Get:3 http://mirrors.jevincanders.net/kali kali-rolling/main amd64 Packages [
19.6 MB]
Get:4 http://mirrors.jevincanders.net/kali kali-rolling/main amd64 Contents (
deb) [46.5 MB]
Fetched 66.2 MB in 2min 31s (439 kB/s)
Reading package lists ... Done
```

Figure 4: Getting updates for 32-bit memory layout.

Next, I entered the **sudo apt-get install gcc-multilib** which did not take very long for me once connectivity was no longer the issue like the first time I attempted to install it.

```
┌──(olivia-flores㉿kali-purple)-[~]
└─$ sudo apt-get install gcc-multilib
Reading package lists ... Done
Building dependency tree ... Done
Reading state information ... Done
The following additional packages will be installed:
```

Figure 5: Installing gcc-multilib.

Next, to disable the Address Space Layout Randomization (ASLR), I ran the command displayed in Figure 6 below. Configuring ASLR will be able to randomize memory addresses where system components, libraries, and executable code are loaded into a process's address space, making it more difficult for an attacker to predict the exact memory locations of critical functions and data. The randomization of memory addresses is done each time a program is executed or loaded [7]. This is like any program that runs in memory, we want it to be protected, but in the following codes we will disable it for now, so we are not randomizing memory for the purpose of the lab.

Before running the commands shown in Figure 6 and 7, I want to introduce what the **-z execstack** and **-fno-stack-protector** are, these are compiler flags passed to the GCC compiler. As shown in Lab 3 instructions:

The **-z execstack** flag is used to mark the resulting executable as an "executable stack." By default, modern operating systems mark the stack as non-executable for security reasons. Using this flag tells the linker to create an executable with an executable stack and allows code to be executed from the stack.

The **-fno-stack-protector** flag disables the stack protection mechanism implemented by the compiler. When enabled, the compiler adds a "canary value" before the function's return address on the stack. If this value is modified during the function's execution, it indicates a buffer overflow, and the program can terminate or take appropriate action [7].

I began by entering the command: **echo 0 | sudo tee /proc/sys/kernel/randomize_va_space**

```
┌──(olivia-flores㉿kali-purple)-[~]
└─$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
```

Figure 6: Configuring randomize_va_space.

Next, I ran GCC with the following parameters to disable additional protection:
gcc -z execstack -fno-stack-protector -m32 -o vuln vuln.c

Shortly after, I ran the program with my in-bounds string of choice:
./vuln woofwoof

```
┌──(olivia-flores㉿kali-purple)-[~]
└─$ gcc -z execstack -fno-stack-protector -m32 -o vuln vuln.c

┌──(olivia-flores㉿kali-purple)-[~]
└─$ ./vuln woofwoof
meow =^..^=
```

Figure 7: Disabling additional protection and running in-bound string.

Next, I ran the program with an out-bounds string showing the number of A's as 400, printed from a Python call from the shell. This clearly will overflow the buffer, and in this case could overwrite memory. As you can see below in Figure 8, it caused a Segmentation Fault which essentially is a crash, lacking bounds checking because we disabled ASLR in the first place. This also happened because it accessed memory outside the allocated buffer which was listed as `char buf[256];` shown in Breakpoint 2 of this lab.

Running the out-bound string: **./vuln $(python3 -c 'print("A" * 400)')**

```
┌──(olivia-flores㉿kali-purple)-[~]
└─$ ./vuln $(python3 -c 'print("A" * 400)')
zsh: segmentation fault  ./vuln $(python3 -c 'print("A" * 400)')

┌──(olivia-flores㉿kali-purple)-[~]
└─$
```

Figure 8: Output of out-bound string; Segmentation Fault.

BREAKPOINT 5

Lastly, I want to demonstrate how I investigated the segmentation fault done in breakpoint 4 of this lab. This is done using a debugger known as GNU Debugger (GDB) which is a simple command-line tool used for debugging programs written in various programming languages, allowing us to examine and manipulate the internal state of a program during its execution [7].
First, I needed to update the gdb installation on my Kali machine using the command shown below:
sudo apt install gdb

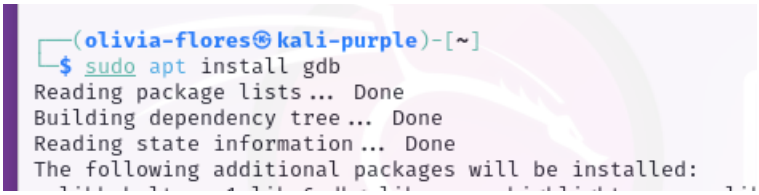


Figure 9: Updating GDB installation on Kali.

Next, I cloned and installed the PEDA package for GDB. PEDA (Python Exploit Development Assistance) is a plugin extension for GDB that makes it easier to work with the debugger. I was able to do this using the following command:

git clone https://github.com/longld/peda.git ~/peda

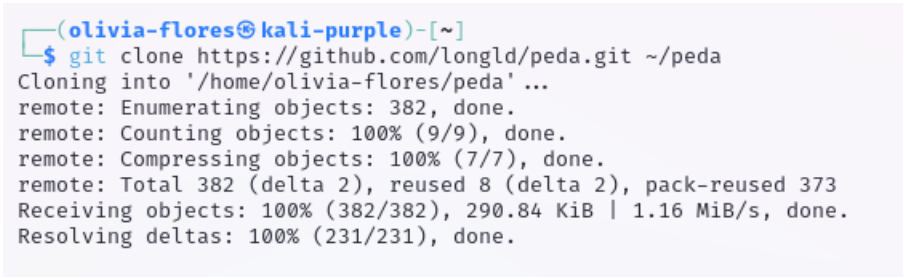


Figure 10: Installing PEDA package for GDB.

Now, I can start GDB with PEDA on this program using the following command:
gdb -q ./vuln

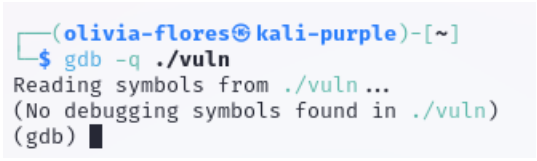


Figure 11: Starting GDB with PEDA.

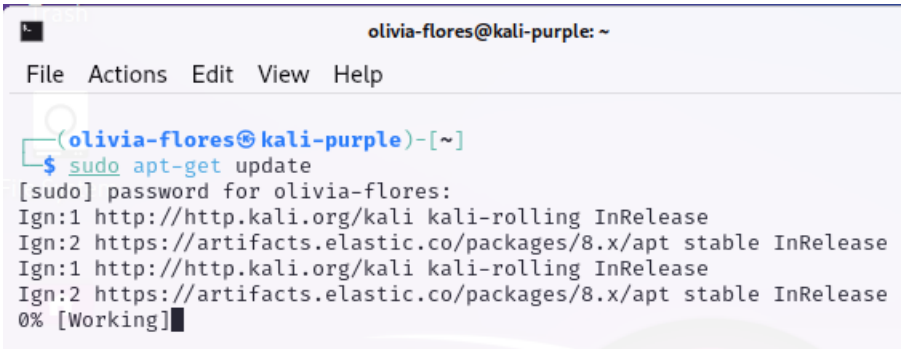
After getting into the PEDA interface, I was able to run the program with an out-of-bounds input string, shown as:
[gdb-peda$] **r $(python3 -c 'print("A" * 400)')**

After running the program with the out-of-bounds string, we can note the “41’s” corresponding to the 400 A’s entered at print(), overwriting another set of words in memory. Since we entered a large number of A’s we were going to cause the system to crash, undoubtedly. You can see from the Figure below, the Extended Instruction Pointer (EIP) loaded a set of A’s into itself and tried to go there and execute code, which caused the program to crash since there was no legitimate program to be ran, at memory address 0x41414141 [8]. I was able to reference the walkthrough by James Lyne as he was explaining how you can see this stack buffer overflow happen just by looking at the memory address ranges. As we now learned, from references presented to us and working hands on with GDB and PEDA, how to cause a stack buffer overflow to occur.

```
Reading symbols from ./vuln...
(No debugging symbols found in ./vuln)
gdb-peda$ r $(python3 -c 'print("A" * 400)')
Starting program: /home/olivia-flores/vuln $(python3 -c 'print("A" * 400)')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
Warning: 'set logging off', an alias for the command 'set logging enabled', is depreca
ted.
Use 'set logging enabled off'.

Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecat
ed.
Use 'set logging enabled on'.

[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x41414141 ('AAAA')
ECX: 0xffffd250 ('A' <repeats 12 times>)
EDX: 0xffffce74 ('A' <repeats 12 times>)
ESI: 0x56558eec ("@aUV\001")
EDI: 0xf7ffcba0 --> 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffce00 ('A' <repeats 128 times>)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xffffce00 ('A' <repeats 128 times>)
0004| 0xffffce04 ('A' <repeats 124 times>)
0008| 0xffffce08 ('A' <repeats 120 times>)
0012| 0xffffce0c ('A' <repeats 116 times>)
0016| 0xffffce10 ('A' <repeats 112 times>)
0020| 0xffffce14 ('A' <repeats 108 times>)
0024| 0xffffce18 ('A' <repeats 104 times>)
0028| 0xffffce1c ('A' <repeats 100 times>)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

CONCLUSION

In conclusion to Lab 03, I was able to learn how to implement Buffer Overflows and see how this vulnerability works. I got somewhat familiar with C code and learned that it has the ability to manipulate low-level details with pointers as shown throughout this entire lab. I got familiar with the term "smashing the stack" and had fun reading more about it as we were practicing hands on- running C code with no input validation, running the C code with an out-of-bounds string, and lastly, running the GDB debugger with PEDA. I also enjoyed getting familiar with all these different tools and resources that can come in handy when dealing with a fast-growing world of cybersecurity. I am constantly learning that something so small such as a simple coding error goes a long way, which leads us here to learn about buffer overflows and how to prevent them using lines of code such as the one we went over in Breakpoint 2, **strncpy**.



REFERENCES

[1] Smashing the Stack for Fun and Profit by Aleph One. https://insecure.org/stf/smashstack.html. [Accessed: February 08, 2024.]

[2] "Please explain the "smashing the stack" concept." Microsoft Copilot: Your Everyday AI Companion, https://ceto.westus2.binguxlivesite.net/. [Accessed: February 08, 2024.]

[3] "Buffer Overflow - Part 1. Linux Stack Smashing." *Cocomelonc*, 19 Oct. 2021, https://cocomelonc.github.io/pwn/2021/10/19/buffer-overflow-1.html. [Accessed: February 09, 2024.]

[4] "Can you explain this snippet of C code more thoroughly and comments to understand it better" *ChatGPT, GPT-3.5 January 2022 version, OpenAI*, https://chat.openai.com/chat. [Accessed: February 09, 2024.]

[5] Godbolt, Matt. *Compiler Explorer*. https://godbolt.org/. [Accessed: February 09, 2024]

[6] "Assembly Language." *Investopedia*, https://www.investopedia.com/terms/a/assembly-language.asp. [Accessed: February 09, 2024.]

[7] Mitra, Department of Information Systems and Cyber Security. [Online] "Lab 03 –Implementing Buffer Overflows" UTSA, 2023 [Accessed: February 07, 2024.]

[8] How They Hack: Buffer Overflow & GDB Analysis – James Lyne. www.youtube.com, https://www.youtube.com/watch?v=V9lMxx3iFWU. [Accessed: February 10, 2024.]

COLLABORATION

In this lab, I made use of the lab instructions provided to us, generative AI tool interactions as I was trying to understand specific lines of C code presented to us and being able to translate to assembly language. Also used these tools to understand where in the C code stood a vulnerability causing the buffer overflow. I was able to grasp a lot of information on this type of vulnerability from the articles and video referenced above in the references section.