**Buffer Overflow**

Buffer overflow is a kind of attack in binary exploitation, it occurs when a program receives data that surpasses its expected length, causing the overwriting of the entire buffer memory area on the stack. This form of attack can frequently result in the crashing of an application; thus, it is also regarded as a Denial of Service. The primary goal in buffer overflow attacks is to change the program's control flow for shell access. Overflowing stack buffers to modify return addresses or saved frame pointers is a typical strategy. Although modern defence technologies have reduced the direct exploitability of this method, it still underpins denial-of-service attacks and more advanced memory exploits, such as those exploiting the Windows structured exception handler mechanism. Moreover, complex forms of stack-based buffer overflow attacks involve corrupting code pointers, including function pointers, longjmp buffers, and global offset table entries. Hack The Box (2021) Chen, G. et al. (2013).

A diagram of a stack

Description automatically generated

Figure 1 Goedegebure, C., (2020)

This is what memory in RAM looks like when a program is run (figure 1). At the top of the memory is the kernel which is the core part of an operating system, it manages system resources and communicates between hardware and software components. Bellow the kernel is the stack where the buffer overflow will essentially happen. Bellow that is the heap this is a region of memory used for dynamic memory allocation where large objects are allocated. Bellow that is data where the global and static variables initialized by the programmer are held. Finally at the bottom is text and it contains the actual compiled code and executable instructions, for this reason this part of the memory is read only as these can not be altered with on run time.

To simplify it buffer overflow occurs when more data is entered into an array than its capacity cause data to be written outside the array, to spill out into other parts of the stack the buffer is in.

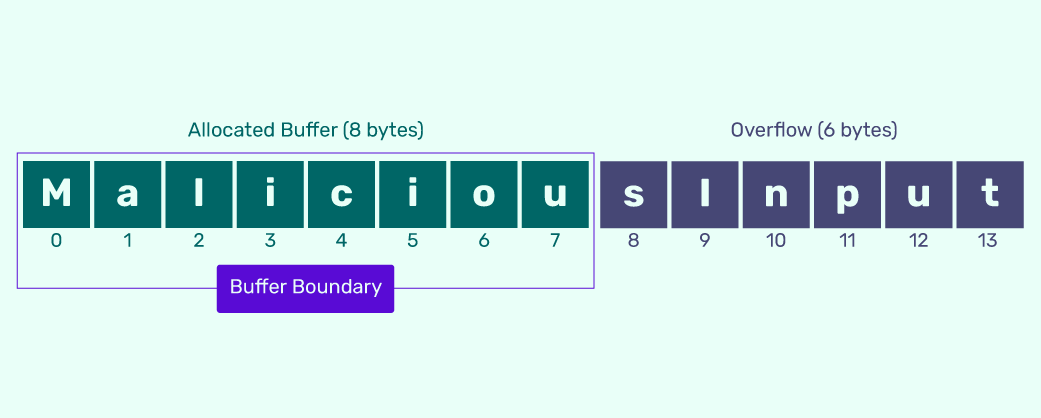
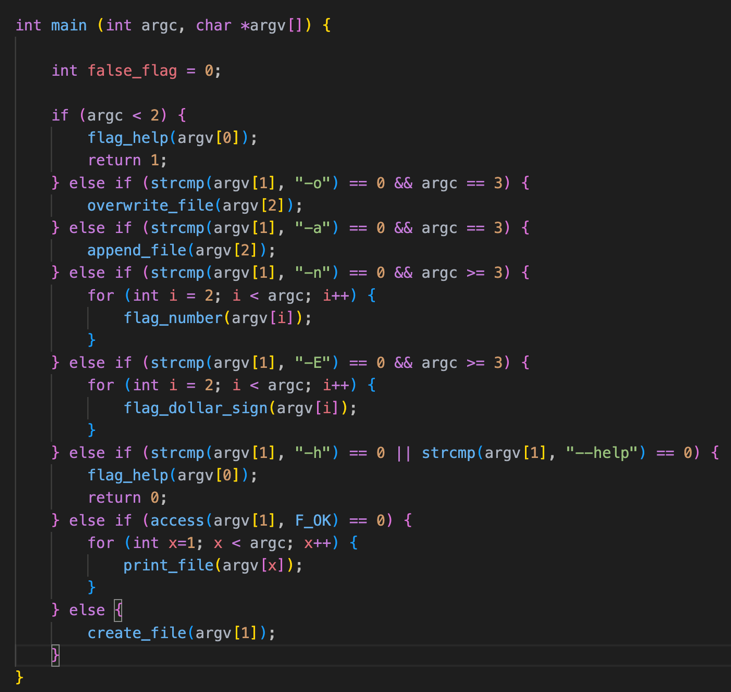


Figure 2 Indusface, (2024)

As you can see in figure 2 the array can hold up to 8 bytes, but the user/hacker/attacker entered 14 bytes into it where the last 6 bytes overflown outside the array/buffer. You can overflow an array by one byte like the number 1 or by 50 bytes or by however much bytes you want, this is important to know because if you overflow the right buffer your buffer overflow attack you could go into the return address of the stack your buffer is in, considering that the buffers we are talking about are local variables.

A screenshot of a cell phone

Description automatically generated

Figure 3 Goedegebure, C., (2020) Figure 4 Goedegebure, C., (2020)



Figure 5

This is what a stack looks like, as you can see at the top the name-parameter, so when the cat command is called the first thing that is pushed on the stack is the compiled run name which would look something like figure 5, where the name I initiated the program with was cat, so later when I call the function I would type “./cat”. After the name parameter the return address is pushed on to the stack because we call the function create\_file() for argv[1] in the main as seen in figure 4 on the last line. And therefore, the computer would want to know where to return after the function is completed and that’s why it pushes the return address of argv[1] on the stack, the return address is where we want our buffer overflow to reach to overwrite it, this is explained more in depth later in the article. And then the base pointer is pushed which is used to refer to parameters and local variables. And finally, the buffer is pushed as the content[10] array in figure 6, the buffer in figure 3 looks huge visually because the picture I found was of a buffer of a hundred bits, but the buffer I am working with is 10 bits; it is like that for sake of easier explanation.

A screen shot of a computer program

Description automatically generated

Figure 6

To further elucidate it is important to note what is a stack in memory management. Stack is a section of memory that keeps track of function calls and local variables. When a function is called, the return address is pushed onto the stack. When the function finishes, the computer pops this address off the stack to know where to resume in the code. The return address represents the location in code the computer is going to return to after it is done executing the function that was called, typically a user can not interact with or see the return address.

A screenshot of a test

Description automatically generatedA diagram of a base point

Description automatically generated

Figure 7 Goedegebure, C., (2020) Figure 8 Goedegebure, C., (2020)

This type of attack is considered dangerous because this overflow corrupts other important data in the stack, by which it can overwrite it as visualized in figure 7, an example of the important data is the return address of the stack. Normally it would result in a crash because the overwritten data would not point to a valid program instruction as shown in figure 8, but if the attacker is to type an actual return address they would be able to execute another function in the program, this type of user-controlled buffer overflow can happen when a program tries to take user input like the cat command I created; when you call the normal cat command you would create a file and then input data into the file if this data was larger than the buffer then a successful buffer overflow attack happened.

A diagram of numbers and a return

Description automatically generated

Figure 9 Yoo, A., (2021)

You might ask how is overwriting the return stack beneficial? To give a simple example to this answer consider an ATM (automated teller machine) and say there is a function in the ATM that dispenses money. If there was a buffer overflow in the ATM when it prompts to type in the pin, if the attacker typed in just the right return address of a function during the overflow they would overwrite the return address of the get pin function with the return address of the dispense money function as shown in figure 9 if the return address for the dispense money function was 4892 then the function would be executed, so when the ATM is done getting the pin the computer would look at the stack and pop the return address, since it would be the last thing pushed on the stack after the buffer (and the base pointer), and would start executing the dispense money function Yoo, A., (2021).

A drawing of a computer code

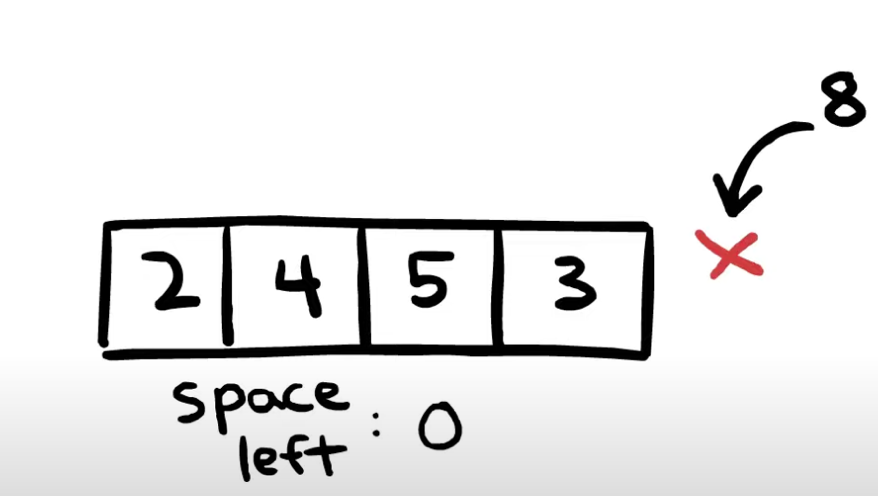
Description automatically generated

Figure 10 Yoo, A., (2021)

That is one way for a hacker to benefit from buffer overflow, but what if the hacker does not know the return address to the dispense money function, or what if there is not one even. There is still a way to benefit from buffer overflow and this way can sometimes be considered better where the hacker can write their own code in the buffer and overwrite the return address to jump back and execute the code they wrote in the buffer as shown in figure 10, this is where hackers can get creative, they can write a code that goes straight to the shell so they can have shell access in other words they would be the admin to the computer they just hacked, to go back to the example they can do literally whatever they desire with ATM, they can even delete all the functionality of the ATM and change it to be a music player. So, this way makes them no longer limited only using the code in the program, but makes their code limited to the buffer size, for example a shell code could be 25 bits and in a buffer like mine which is 10 it would make it harder for a hacker to gain shell access; but maybe there is still a possibility for them to write a shell code in 10 bits Yoo, A., (2021).

**Buffer Overflow Countermeasures**

There are a lot of ways to countermeasure/avoid a buffer overflow attack a couple of them are:

1. Runtime bounds checking.
2. Stack canary.
3. Data execution prevention.
4. Shadow stack.
5. **Runtime Bounds Checking**

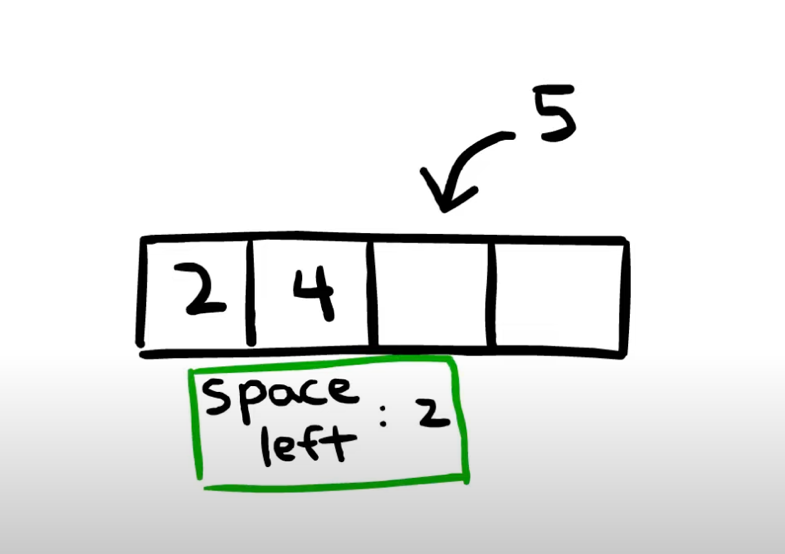


Figure 11 Yoo, A., (2021) Figure 12 Yoo, A., (2021)

Runtime bounds checking is a very popular technique that is even used in some programming languages like python and java, the basic idea behind it is every time the user adds a number/character to an array there is code behind the scenes that will check if there space left in the array as shown in figure 11; and, if there is no room in the buffer the insertion of the number/character will not happen as shown in figure 12 and an error would be given to the user.

the main advantage to it is the actual checks happen during the execution/run time of the program rather than at compile time, this allows it to handle dynamic data structures and variable data lengths which might not be predictable before runtime. This basically makes it theoretically impossible for buffer overflow to happen. However, there is a performance cost for the extra bit of code that is running on every time a user inserts number/character into an array, that is the main reason the other programming languages like c and c++ choose to not use runtime bounds checking by default. A simple example of for changing the code in c to have run time bounds checking is changing the gets() function that reads user input to the fgets() function that also read the user input but behaves differently.

The gets() function reads a line into the buffer (content[]) as you can see in figure 6. However, it does not perform bounds checking, meaning it will continue to read characters and store them in the buffer regardless of its size. This leads to buffer overflow, as gets() doesn't know the size of the buffer and can overwrite adjacent memory if the input exceeds the buffer's capacity.

A computer screen shot of a program code

Description automatically generated

Figure 13

The fgets() function In contrast, reads a line from stdin into the buffer content[], but it also takes the buffer size as an argument “sizeof(content)” as shown in figure 14. It stops reading after n-1 characters or upon reaching a newline character, whichever comes first, and then adds a null terminator. This ensures that the buffer is not overrun, performing runtime bounds checking by respecting the buffer's size limit.

1. **Stack canary**

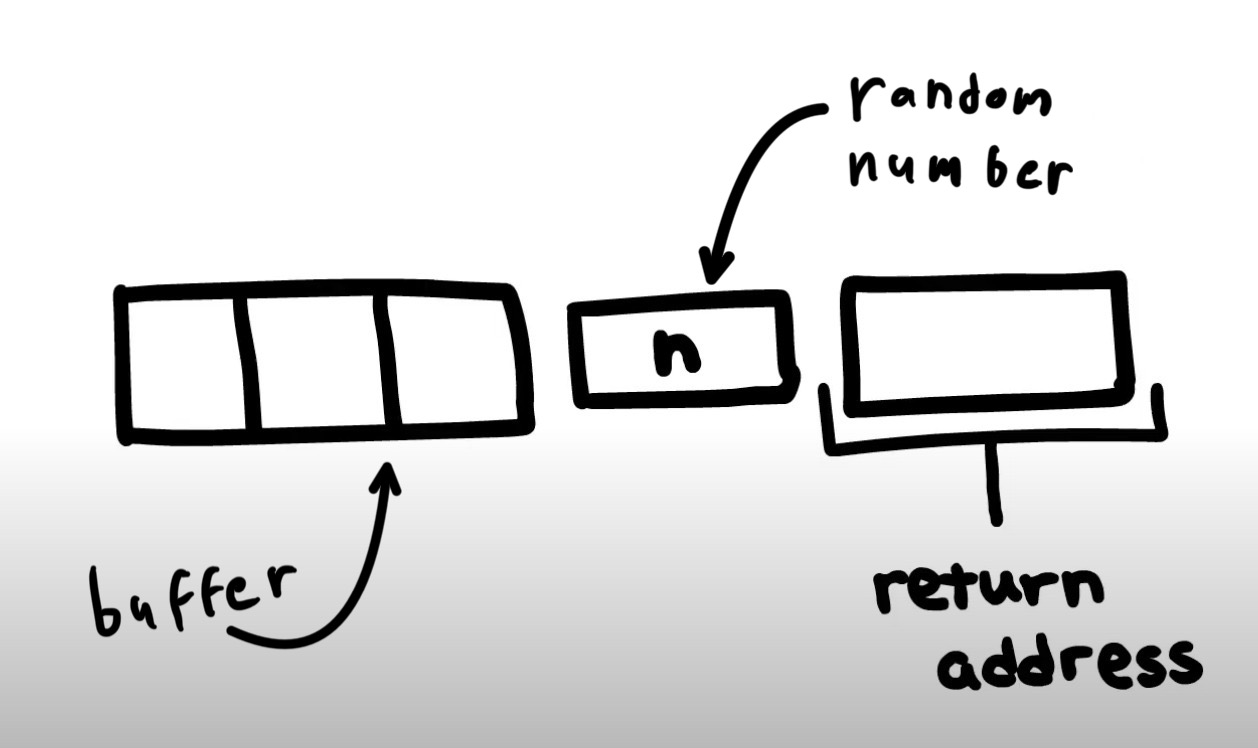


Figure 14 Yoo, A., (2021)

Stack canary is a very straight forward and efficient technique as the way it is works is, when a function is called, a small randomly generated value, known as the "canary", is placed in the stack memory just before the return address and after the buffer as shown in figure 15. When the function is ready to return, before the computer jumps to the return address the canary is checked to make sure it is still the same value “n”, if it is not the same value the program will crash, and no more code will be executed. This is a very good defence against buffer overflow because to get to the return address the buffer overflow attack must overwrite the canary; and, since the attacker does not know the canary value, the computer will be alerted to the attack and would respond with crashing the program execution before jumping to the return address.

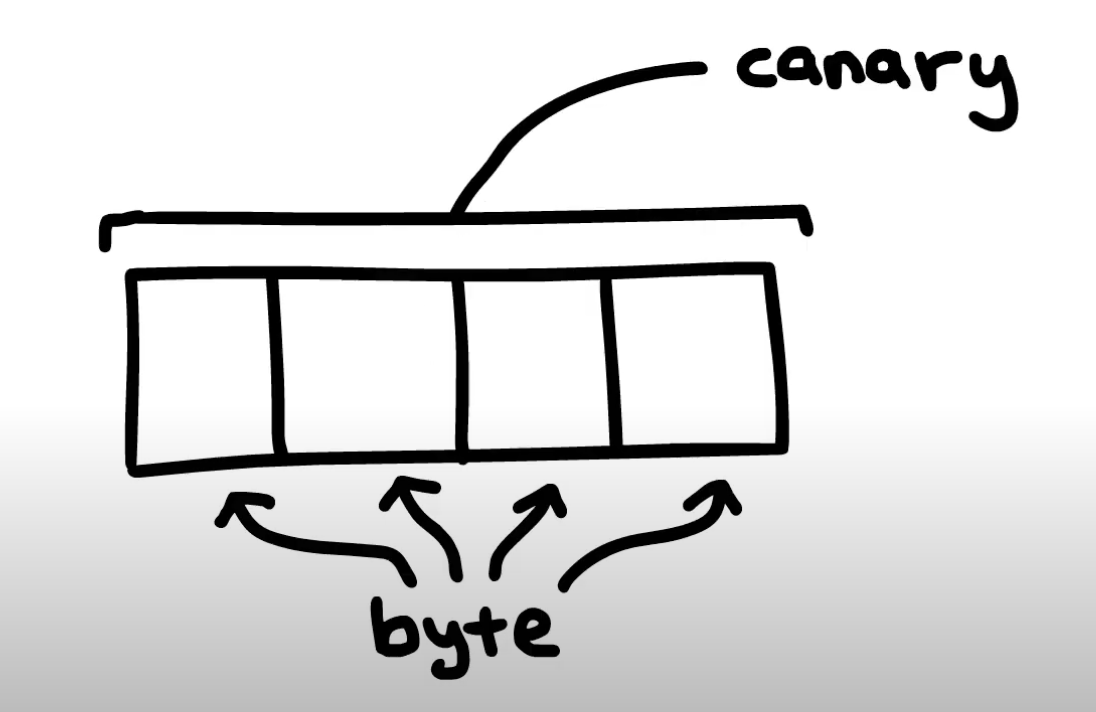


Figure 15 Yoo, A., (2021)

Since this technique is very straight forward there is 2 ways to bypass it the first one being a stack leak, which is when a hacker can trick the system to show the data in the stack and the main thing the hacker looks for in the data is the canary value, in that way the hacker would perform a normal buffer overflow attack and just add the canary value before the return address. The second way to bypass the canary is by guessing the value, the “easiest” guess would be on a 32-bit system since the number will be a random number between 0 and roughly 4 billion (4,294,967,295), and because this value changes every run, the chance of guessing this value would be 1 over 4 billion (4,294,967,295). But some modern service architectures, like HTTP, start with a master program and spawn worker programs for every request. When a worker program is spawned on Linux it uses fork() which is a Linux system call that clones the current program, and the thing about fork() is that it uses the same stack canary as the master program; thus, a hacker gets as many tries as he needs to figure out the canary value. There is a way to decrease the guess to be at a maximum of 1024, as a 32-bit canary value is essentially a buffer of 4 bytes, as shown in figure 16 a visualisation of that, and every byte is a random value between 0 and 255 therefore a hacker would start guessing each byte’s value until they find the canary value Yoo, A., (2021).

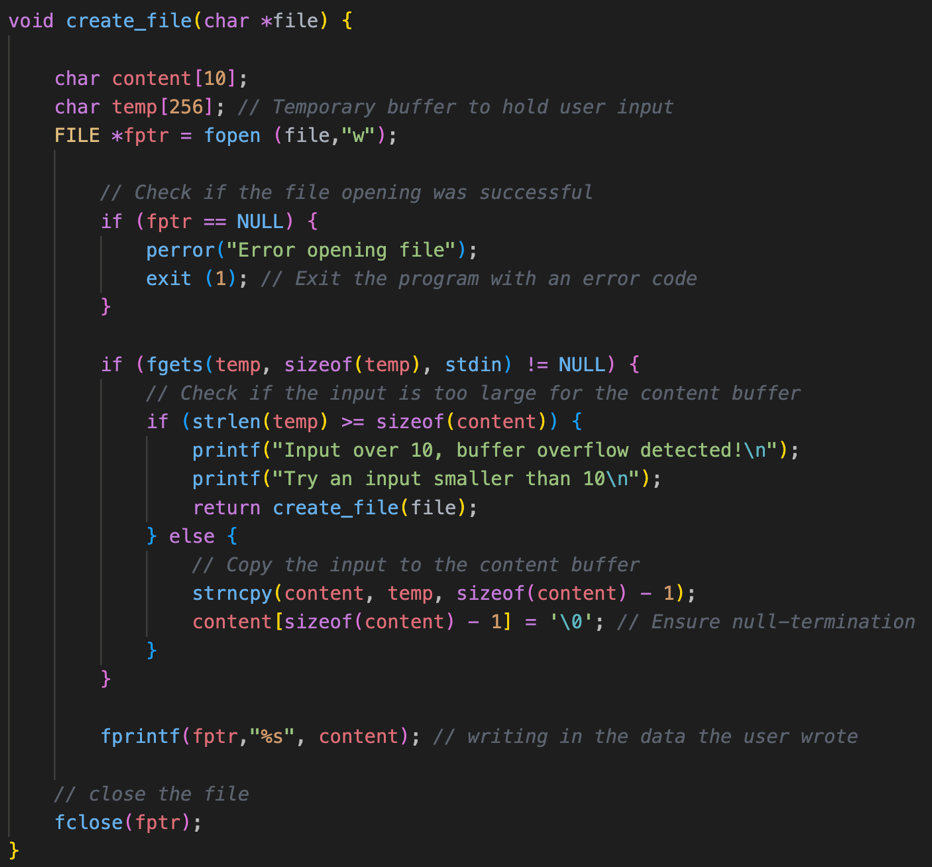
1. **Data execution prevention**

Data execution prevention or DEP is also known as executable space prevention or also NX bit. The way it works is it marks the entire stack as non-executable, so whenever the stack gets overwritten and something else is to be executed and access violation would pop up to the hacker and the program will crash. DEP also created a policy called write xor execute which prevents anything in memory to be writable and executable at the same time, which in-turn prevents buffer overflow attacks since their core is writing code in the buffer and executing it. However, there is one downside to DEP which is, it does not stop a hacker from overflowing the buffer and overwriting the return address with a pre-existing address, like dispense money which was talked about previously Yoo, A. (2021).

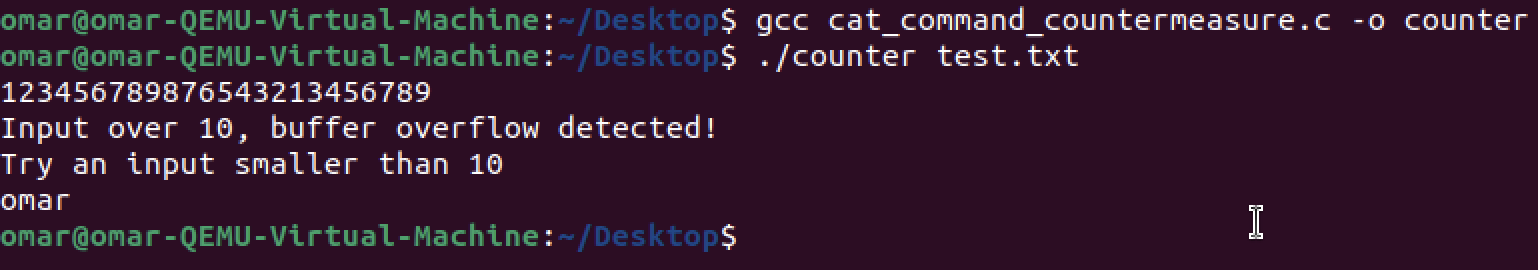
1. **Shadow Stack**

Shadow stack is a separate stack in the memory that holds copies of addresses from the original stack. The way it works is it provides a check before using the return address of the original stack to return to another function, so for a program to carry on with its functionality the return addresses should be equal otherwise if the address was overwritten in any other the way, then the address will not match and the program will crash, therefore it enforces control flow integrity Yoo, A. (2021).

**This is what I implemented:**

****

**This is the terminal:**

****

**Here are the file contents:**

**A screenshot of a computer

Description automatically generated**

**References:**

Hack The Box (2021) Cybersecurity Glossary [online]. Available from: <https://www.hackthebox.com/resources/hacking-glossary#chapter-2>

Chen, G. et al., (2013) SafeStack: Automatically Patching Stack-Based Buffer Overflow Vulnerabilities. IEEE transactions on dependable and secure computing. [Online] 10 (6), 368–379.

Indusface, (2024) What is a Buffer Overflow Attack? [online]. Available from: <https://www.indusface.com/learning/what-is-buffer-overflow-attack/>

Goedegebure, C., (2020) Buffer Overflow Attacks Explained [online]. Available from: <https://www.coengoedegebure.com/buffer-overflow-attacks-explained/>

Yoo, A., (2021) Buffer Overflow [video]. YouTube. Available from:

https://youtu.be/AD-iXWANggo?si=43JRqMYr7DBRSH11

Yoo, A., (2021) Stack Canary[video]. YouTube. Available from: <https://youtu.be/N7kGd76evsM?si=Np-7D-9Rsw5bmfAc>

Yoo, A., (2021) Data Execution Prevention [video]. YouTube. Available from: https://youtu.be/3Bu8-cdyGHQ?si=GK9S1WVVePlUNt3w

Yoo, A., (2021) Shadow Stack [video]. YouTube. Available from: https://youtu.be/ebIXo9OaNRg?si=wxg5m0juJ9N61HTl