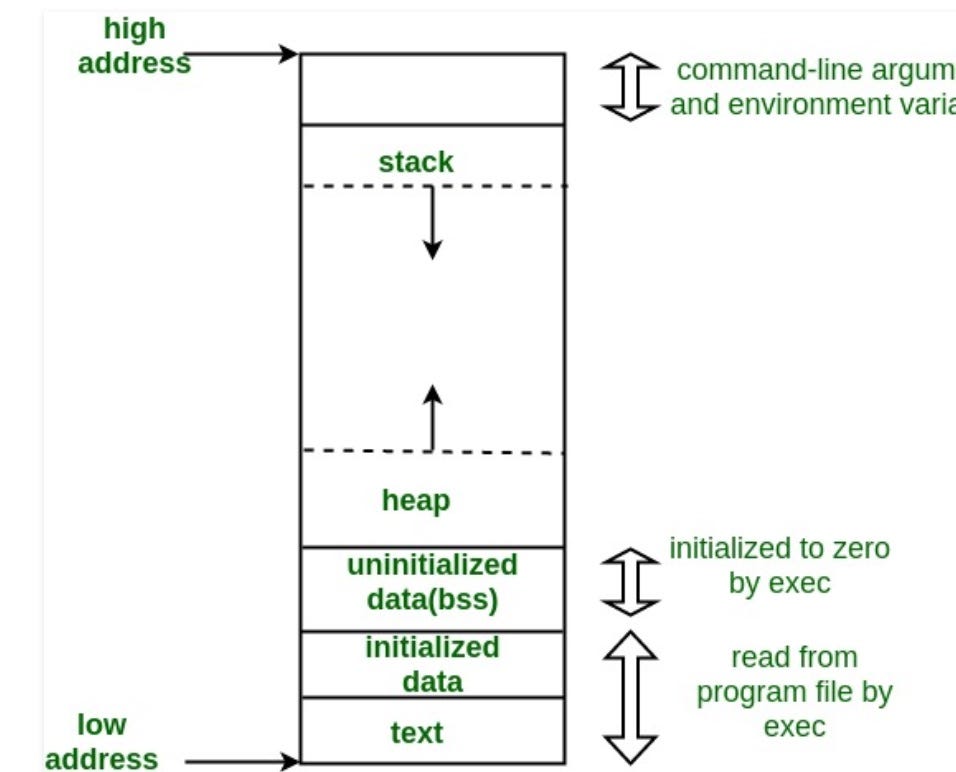
A buffer overflow is a software vulnerability that occurs when a program writes more data to a buffer than it can hold, potentially overwriting adjacent memory and causing unexpected behavior. This error can be exploited by attackers to gain unauthorized access to systems or manipulate program execution. Buffers are memory regions used to temporarily store data while it’s being transferred, and when the data exceeds the buffer’s allocated size, it can overflow into other memory areas, which can corrupt data or expose sensitive information. There are two main types of buffer overflow attacks: stack-based and heap-based. Stack-based overflows are more common and occur in the stack memory, which is used during function execution. Heap-based overflows are more difficult to exploit and affect memory allocated for dynamic operations during runtime.

Buffer overflows are a significant concern in software security, particularly in languages like C and C++, where memory manipulation is allowed, and there are no built-in protections against buffer overflow errors. If programmers fail to control buffer sizes or use vulnerable functions, such as gets, which does not limit input size, security vulnerabilities arise. This can allow attackers to execute arbitrary actions, such as deleting files, stealing data, or using the system for further attacks. As a result, buffer overflow vulnerabilities can lead to serious security issues, often compromising the integrity of a system.

Let’s learn a bit about how memory runs and few important registers before jumping into the exploitation of a vulnerability present in my custom cat command to achieve stack-based buffer overflow. Below is memory structure



Source: [https://cdn-images-1.medium.com/v2/resize:fit:1200/1\*8b9-Z3FV6X9SP9We8gSC3Q.jpeg](https://cdn-images-1.medium.com/v2/resize:fit:1200/1*8b9-Z3FV6X9SP9We8gSC3Q.jpeg)

**Stack:** This is the place where all the function parameters, return addresses and the local variables of the function are stored. It’s a LIFO structure. It grows downward in memory(from higher address space to lower address space) as new function calls are made. We will examine the stack in more detail later.

**Heap:** All the dynamically allocated memory resides here. Whenever we use malloc to get memory dynamically, it is allocated from the heap. The heap grows upwards in memory(from lower to higher memory addresses) as more and more memory is required.

**Command line arguments and environment variables:** The arguments passed to a program before running and the environment variables are stored in this section.

**Uninitialized data(Bss Segment):** All the uninitialized data is stored here. This consists of all global and static variables which are not initialized by the programmer. The kernel initializes them to arithmetic 0 by default.

**Initialized data(Data Segment):** All the initialized data is stored here. This consists of all global and static variables which are initialized by the programmer.

**Text:** This is the section where the executable code is stored. The loader loads instructions from here and executes them. It is often read only.

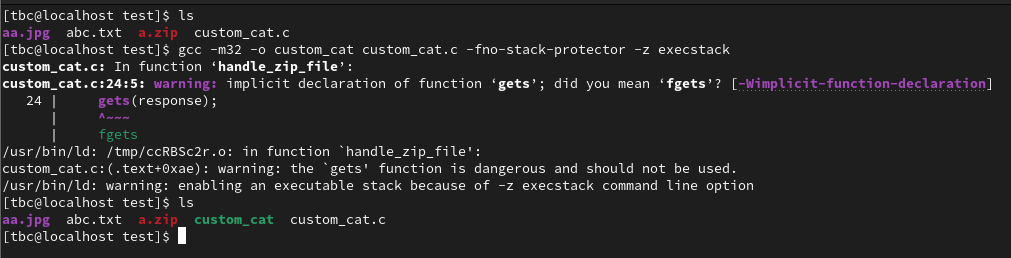
**%eip/rip**: The **Instruction pointer register**. It stores the address of the next instruction to be executed. After every instruction execution it’s value is incremented depending upon the size of an instrution.

**%esp/rsp**: The **Stack pointer register**. It stores the address of the top of the stack. This is the address of the last element on the stack. The stack grows downward in memory(from higher address values to lower address values). So the %rsp points to the value in stack at the lowest memory address.

**%ebp/rbp**: The **Base pointer register**. The %rbp register usually set to %rsp at the start of the function. This is done to keep tab of function parameters and local variables. Local variables are accessed by subtracting offsets from %rbp and function parameters are accessed by adding offsets to it as you shall see in the next section.

Now, we have some knowledge of important registers and how memory works let’s start to dive into the exploitation of the vulnerability introduced in the code:

Firstly, after introducing vulnerability I’ve compiled the code in 32 bit system, -fno-stack-protector to remove the stack protection against buffer overflow and used -z execstack to make the stack executable. Also we need to disable ASLR (echo 0 | sudo dd of=/proc/sys/kernel/randomize\_va\_space).

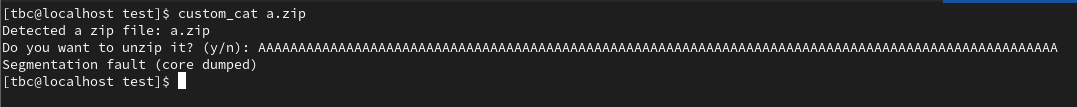


The gcc is not happy rightfully and have thrown some warning because the function gets is dangerous. We can ignore the warning and proceed as we are doing it in controlled environment and to learn (educational purpose). Now, let’s inspect the files with built-in file command of Linux and checksec a popular tool, the file looks interesting.

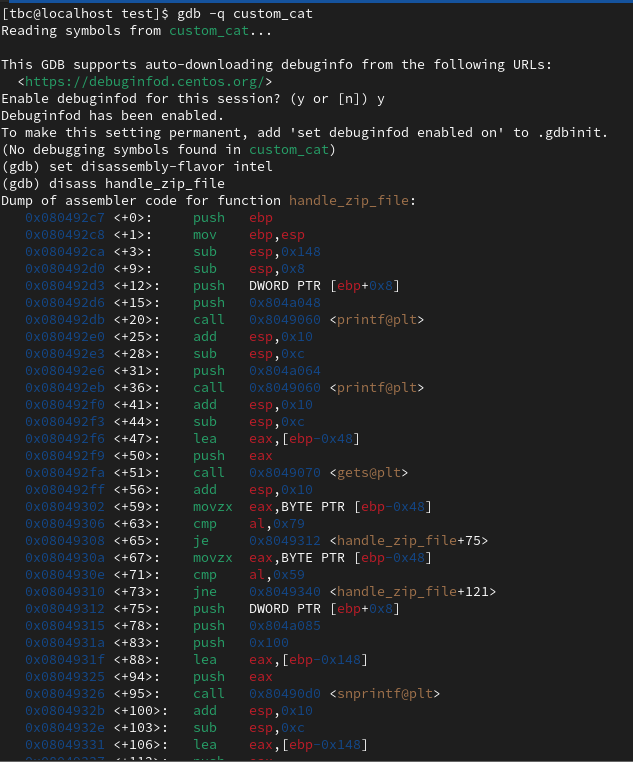
A black screen with white dots

Description automatically generated

To identify a buffer overflow vulnerability, I typically use two main approaches. The first is source code or assembly analysis (if accessible) to find known vulnerable functions like gets, scanf, strcpy, strcat, or sprintf. When I spot these, I check if the input is under my control, if the destination can safely store the result, and if size restrictions are in place to prevent overflow. For example, in this case, the use of gets—which doesn’t limit input size—makes it inherently vulnerable. The second approach is blind testing, where I target areas accepting user input by injecting long strings to see if the application crashes or throws an error. Although this method is basic and might require specific input formats to trigger vulnerabilities, it can uncover issues and be enhanced with automated fuzzers, though that’s what we’ll avoid because we are trying to learn everything. Here, I’ve input a string of 100 characters (A\*100) and have successfully crashed the program.



The reason is because the return address has been overwritten with my long "AAAAA...", (e.g. 100 × "A") and therefore, at the end of the epilog, when executing the instruction ret, the CPU tries to jump at the address stored in what used to be the return address but is now 0x41414141 (AAAA in ASCII). 0x41414141 is not a valid address, hence the crash.

Let’s validate this assumption with gdb (gdb –q custom\_cat):  


A screenshot of a computer program

Description automatically generated

When examining different sections of memory (stack, heap, .text, etc.), I noticed that there is nothing loaded around the address 0x41414141. Since this address is empty, the application crashes when it tries to access it.

To leverage a buffer overflow and control the execution flow, it’s crucial to determine the exact offset in the payload where the return address gets overwritten. One effective way to find this is by sending a unique pattern, such as "AAABBBCCCDDDEEEFFFGGGHHH..." (each letter repeated three times), using a debugger like GDB.

A black background with white text

Description automatically generated

The return address has been overwritten with 0x61615a5a, which corresponds to "aaZZ" in ASCII. However, since GDB displays data in little-endian format, the value should be read backward as "ZZaa". Referring to the generated pattern, "ZZaa" is located at an offset of 76.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | A | A | B | B | B | C | C | C | D | D | D | E | E | E | F | F | F | G | G | G | H | H | H | I | I | I | J | J | J | K | K | K | L | L | L | M | M | M |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 |
| N | N | N | O | O | O | P | P | P | Q | Q | Q | R | R | R | S | S | S | T | T | T | U | U | U | V | V | V | W | W | W | X | X | X | Y | Y | Y | Z | Z | Z |
| 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |
| a | a | a | b | b | b | c | c | c | d | d | d | e | e | e | f | f | f | g | g | g | h | h | h | i | i | i | j | j | j | k | k | k | l | l | l | m | m | m |
| 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 |

To verify that, let’s send a string of 76 "A" followed by 4 "B":

A screenshot of a computer

Description automatically generated

Manually building a pattern and counting the offset is tedious and impractical, especially when dealing with large buffers, like arrays of a thousand bytes. It becomes even more challenging when the alphabet is exhausted, and counting offsets manually is inefficient. Automating this process is the obvious solution, and fortunately, scripts already exist for this purpose. The most well-known are pattern\_create.rb and pattern\_offset.rb from Rapid7, included in the Metasploit framework—a popular tool for application security assessments. However, since these scripts require the entire Metasploit framework, a simpler alternative is to use the online version developed by zerosum0x0, which eliminates the need for installation.

A computer screen with white text

Description automatically generated

The execution flow has been redirected to the address 0x63413563, indicating that the return address has been overwritten with this value. To determine where this occurred, we need to identify the offset in our pattern where the values c (0x63), 5 (0x35), A (0x41), and c (0x63) appeared (reversed because of little-endian format).

A screenshot of a computer

Description automatically generated

So, the offset is 76 proved from manual and automated method. So now, we just need to replace the four B's (i.e. add address/group of instructions after 76 ‘A’) with the address of the function (or group of instructions) we want to execute.

Before building a payload we need to understand that in order to execute the function (or group of instructions) that we want we also need to execute the previous instruction that push to the stack the pointer to the function (or group of instructions). Another few issues we might face are the shellcode and address may contain non-printable characters (e.g., 0x00,0x0a), which can cause issues and the payload often cannot include the NULL character (0x00), as it typically truncates the payload, ignoring everything after it. Additionally, depending on how the program handles user input, other characters might be removed or cause unexpected behavior. These problematic characters are referred to as "bad characters." Another potential issue is that the initial input used to overflow the buffer might be too small to hold our shellcode. In such cases, we have two options: either optimize or modify our shellcode to make it smaller, or find an alternative location to place it. For example, if I can execute commands on a Linux system, I could store the shellcode in a new environment variable or pass it as a parameter when running the vulnerable application. Both environment variables and application parameters are typically placed on the stack before the main function's stack frame.

Now, I’ll find the memory address on the stack where data is stored by sending skeleton payload:

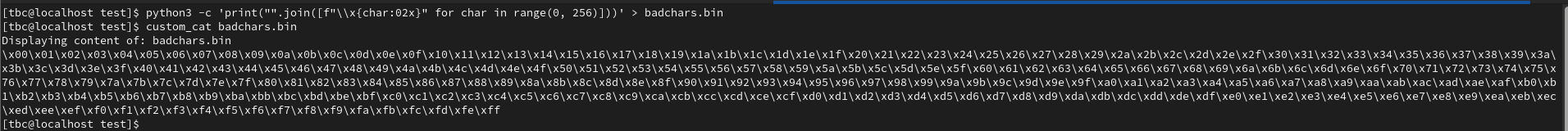
(padding (76\*’A’) + (4\*’B’) address + shellcode (51\*’C’))

A screenshot of a computer

Description automatically generated

The ret instruction from handle\_zip\_file () popped the overwritten return address and (tried to) jump to it. So, this means esp is now pointing to what was written right after the return address, i.e. our shellcode.

Now I will generate a list of all possible characters and test for bad chars:



The payload of badchars terminated at the third hexadecimal values (Highlighted in sky blue) it have printed \x09\ so, the problem is \x0a\.

A black screen with white dots

Description automatically generated

Let’s remove \x0a\ the problem and re-run it and this time instead of 50 I’ll print 70 hexadecimal values:  
A black screen with white dots

Description automatically generated

Everything ran through and terminated with \xff\ our last character. Hence, I found only /x0a/ to be bad chars.

Now let’s craft payload avoiding \x0a\ payload will look something like this:

python2.7 -c offset\*’A’(padding) + 'address' + ‘shellcode’

python2.7 -c "print (offset – (address bytes + shellcode) \*'A'(padding) + ‘shellcode’ + 'address' {another technique for crafting payload when the size of offset is big enough to hold shellcode}

Let’s generate a shellcode, for this purpose we can use msfvenom by rapid7 included in metasploit-framework or websites like shellstrom.org or we can write it ourself. For now, I’ll use code from shellcode for /bin/cat/etc/passwd from shellstrom.org (souce: <https://shell-storm.org/shellcode/files/shellcode-571.html> ) and since it meets my requirements to avoid my bad characters. One thing to consider is where we execute the payload from is it gdb or terminal itself because if we execute it from terminal we’ll have to add nop sleds as running program from terminal will shift the stack in memory because there are several aspect of the program that might slightly alter the actual location of the shellcode. (nop sled is a sequence of **nop** (No Operation) instructions used in exploits, particularly in buffer overflow attacks, to help an attacker redirect the execution flow of a program to a shellcode.) It can be useful if we run payload from terminal in that case payload would be:

python2.7 -c "print offset\*'A'(padding) + 'address' + number\_of\_nop\_instruction\*‘\x90\’ +‘shellcode’ | command

python2.7 -c "print (offset – (address bytes + shellcode) \*'A'(padding) + number\_of\_nop\_instruction\*‘\x90\’ +‘shellcode’ + 'address' | command

{Note: Number\_of\_nop\_instructions depends on shift in memory stack}

Injecting payload from the gdb :

A black screen with white lines

Description automatically generated

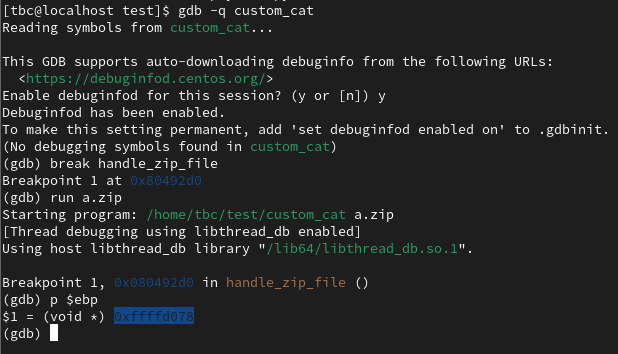
Okay, I got the desired passwd file now I want to get a live shell for that I’ll automate a little bit by writing a simple script in python. Payload script looks like this:

A computer screen shot of a program code

Description automatically generated

FIG: payload.py

I’ve imported pwntools library, which is a toolkit for building exploits and interacting with binaries. Then, made a padding of 76 bytes which is my calculated offset. Then automated the return address by simply getting the base pointer address {Got from running program as (run a.zip) after adding break point at vulnerable function handle\_zip\_file and printing out ebp address as (p $ebp)}



and adding few bytes to it I also specified 32 bit format which is handled by pwntools to little endian format. After which to make sure my shellcode lands on desired location I added few nop\_sleds and automated even the shellcode with pwntool which is then assembled into raw binary. Then I combined all parts in one file and wrote to a temporary file named payload.

Upon execution from terminal itself as:

Cat payload | custom\_cat a.zip {The process terminated and upon digging a little bit I found out since the shell has nothing to do it terminates immediately after execution. So, knowing cat reflects every command we put when we run cat without any arguments like:

A screen shot of a computer

Description automatically generated I simply added ; cat hoping it does not terminate immediately.}

Firing up the payload with ; cat like this:

(cat payload ; cat) | custom\_cat a.zip

A screen shot of a computer

Description automatically generated

Boom we get the live interactive shell. Now I’ll validate that it can be done without automating the shellcode crafting by replacing the script to craft shellcode with a shellcode (source : https://shell-storm.org/shellcode/files/shellcode-841.html ) in a different file say payload1.py and outputting file in payload1.

A black background with white text

Description automatically generated

FIG: vim payload1 ()

Upon firing up the payload let’s see what happens (Note: I adjusted bytes to be added as 125 instead of earlier 120 after viewing the stack.)

A screenshot of a computer program

Description automatically generated

Boom a shell again as expected.

* I identified the vulnerability by looking at the source code and finding the function gets(), which is unsafe and prone to buffer overflow.
* I used a long string with a specific pattern that helped identify the correct offset.
* I ran custom\_cat in gdb and sent the generated string to trigger the buffer overflow.
* After the application crashed, I looked at the value of eip (the instruction pointer) to see where it was pointing and converted that to ASCII.
* I sent the ASCII value obtained in the previous step to a second script to calculate the correct offset where the buffer overflow occurs.
* I constructed a skeleton payload : offset \* "A" + "BBBB" + estimate\_of\_shellcode\_size \* ‘C’ This payload consists of the calculated offset, filler bytes ("BBBB"), and the shellcode I want to execute.
* I once again ran custom\_cat in gdb and sent the skeleton payload to check the effect of the overflow. Aanlysed stack.
* After the application crashed, I looked at the value of esp (the stack pointer) to find the shellcode address.
* Using the address found in esp, I built our final payload : offset \* ‘A’ + ‘address’ + shellcode.
* Finally, I executed the custom\_cat application with the crafted payload to trigger the execution of our shellcode.
* I also showed much easier and faster automated way by spawning shell making use of pwntools and nop\_sleds.

How to be safe from buffer overflow:

This question may arise after seeing how dangerous this attack is, well there is always a chance that this attack can happen especially in modern time when every country is preparing their own cyber army. Moder operating system are generally quite secured compared to older operating system but this doesn’t mean they are not vulnerable like I demonstrated earlier it is in fact vulnerable if can bypass few protections even with this simplest technique of buffer overflow. To make our system secured we can always keep stack protection on and memory randomization enabled as a user. And as a developer we should implement combinations of secure coding practices, secure compiler options and enable runtime protections. Also a developer should use safe functions, validate input and perform bound checking to prevent buffer overflow. Additionally adopting modern languages like rust, go etc which offer inherent memory safety, can eliminate many buffer overflow risks entirely.