Describe the stack in the address space of a process in the VM (in generalities).

The stack in a process's address space in a virtual memory environment is a data structure that stores information related to function calls, local variables, and control flow. It is used for maintaining the control flow of a program. It is a Last In First Out (LIFO) data structure and it grows downwards. Each function call creates a call frame on the stack.

Addresses where in memory the stack would be located (specifically).

Which direction, relative to overall memory, does a stack consume memory when it grows? What limits the size of the stack?

High	stack (grows downwards)
	heap (grows upwards)
	data
Low	text

Text: Holds executable code
Data: Holds Global Variable
Heap: memory – allocated during runtime
Stack: Return Address, Function parameters, local variables

Stack is located at the higher end of the memory however the precise location of the stack within memory varies depending on the operating system and its configuration. It grows in the opposite direction of the memory consuming memory from higher address towards the lower address. The size of stack can be limited by hardware constraints such as available physical memory and software constraints such as system configuration defined by an OS or a compiler. Stack grows downwards in memory (i.e., towards the heap).

Explain how program control flow is implemented using the stack.

As a program runs, it's important to keep track of a few things when one function calls another. First, we need to save the return address, which acts as a kind of marker for the compiler to know where to continue once the called function finishes its job. Additionally, any data passed between functions as parameters should be stored in memory because it might be needed after the called function completes its work. All this information is organized within a stack frame. So, when all the necessary addressing is done, the stack becomes empty, and the stack pointer moves on to the next part or function in the program..

How does the stack structure get affected when a buffer of size 'non-binary' is allocated by a function (i.e., a buffer size which causes misalignment within the stack)? [Also known as 'non-binary']

When a buffer of size 'non-binary' is allocated by a function, it leads to a misalignment in the stack. The misalignment can cause overwriting unintended data on the stack. Misalignment occurs when data on the stack is not evenly divisible by the word size, causing inefficient memory access and potential security vulnerabilities.

Create a diagram that includes the following:

i) **What does the stack structure look like when data is pushed onto the stack and popped off the stack?**
ii) **Show what register values are placed onto and used with the stack.**
iii) **Where would arguments be placed on the stack?**
iv) **Where are local user variables placed on the stack?**

In the figure below, we can see the structure of stack when data is pushed into it.

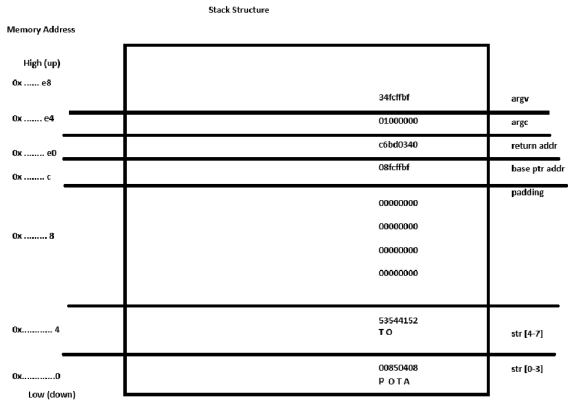


Figure 1Computer Security Principles and Practice: Stallings, Brown

Stack Structure: In a stack structure, when you push data onto the stack, it is added to memory starting from a higher memory address and moving down. So, the data is read in a higher memory address during pushing. For example, if you push the word "potato" onto the stack, "t" and "o" (str[4-7]) will be added first, and then "p" and "a" (str[0-3]) will be added as you move down in memory.
When you pop data from the stack, the data that was added first (at the lower memory address) is removed first. So, in the figure, the data at str[0-3] would be the first to be removed when you perform a pop operation.

Register When register values are saved to the stack, the actual values themselves are not directly accessible. Instead, what is placed on the stack are the memory addresses or references to these registers. These addresses are then utilized to retrieve and work with the register values when needed.

Arguments: Arguments are typically located at the top of the stack frame, which is closer to the current stack pointer or base pointer for the function

Local user variable: Local user variables on the stack are positioned below the argument counters and any potential padding, depending on the system's conventions. These local variables are located at lower memory addresses relative to the argument addresses and control structures

Write a testing program (in C) that contains a stack buffer overflow vulnerability.

The easiest way to exploit a stack buffer overflow vulnerability is by declaring a fixed size char but feeding it with more data than it can handle.

The code can be:

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

int main(int argc, char *argv[]) {
    char buffer[4];
    strcpy(buffer, argv[1]);
    printf("buffer: %s\n", buffer);
    return 0;
}
```

Here, the char buffer in main is currently of length 4. Here, if the program is compiled as ./hello we can do:
./hello abc → the code would run without any issue.
./hello abcde → The code would go into a segmentation fault as the program flow would get altered

If we were to modify the input in such a way that it returns to a certain area in memory, we would be able to run a malicious payload.

b) Show what the stack layout looks like and explain how to exploit it. (Include a diagram)

i) Include the following items:

(1) The order of parameters (if applicable), return address, saved registers (if applicable), and local variable(s).

(2) The sizes in bytes.

(3) The overflow direction in the stack.

(4) Size of the overflowing buffer to reach and overwrite the return address.

(5) Overflow data that is meaningful for an exploit (this can be general).

Order of Parameter: Mentioned in the figure.
Return Address: The address to which control will return after the function main completes. This is the system since it is the main method in our program.
Saved Registers: Base pointer EBP is saved in the stack.
Local Variables: buffer is the local variable.

high	Stack
	Argc Return address (4 bytes in 32-bit system) Base pointer Str[0-3]: buffer size 4 bytes; here goes the input data the user inputs.
	Free memory
	Heap
	Data
low	text

Size in byte: mentioned in the figure.

Overflow direction in stack: data is written from the buffer and overflows towards higher memory addresses. In this case, it's from buffer towards the return address.

Size of overflowing buffer: The buffer size is 4 bytes, so to overwrite the return address, we need to provide more than 4 bytes of input data to argv[1]

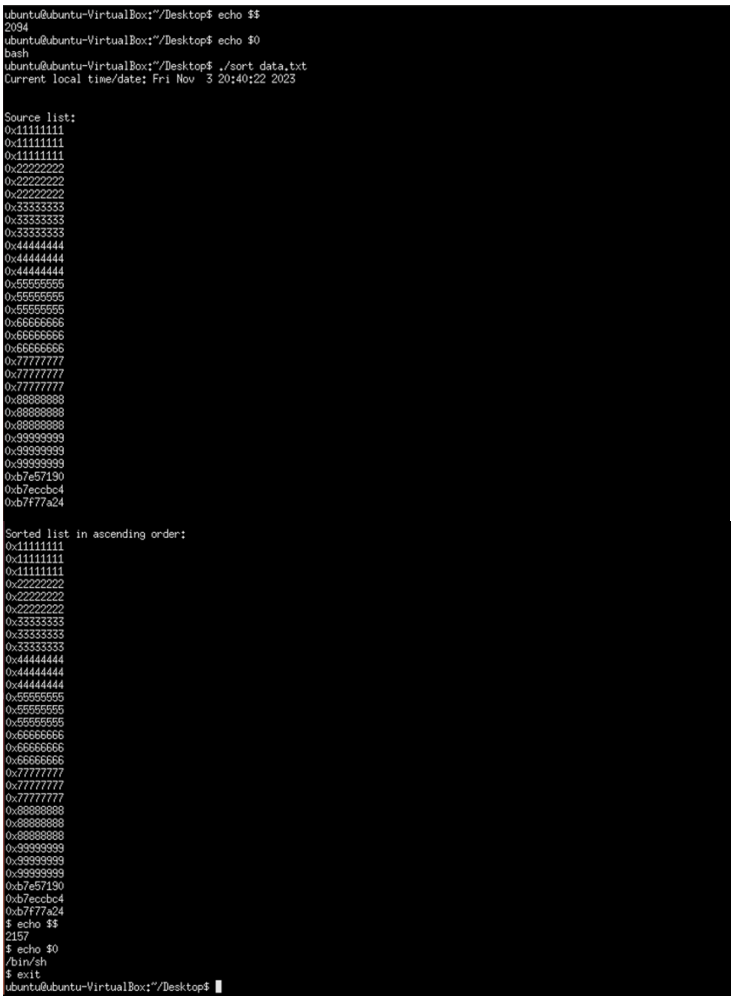
Overflow Data for exploitation: For exploitation, we need to craft an input that contains a payload we want to run.

If our code is compiled as ./exploit:

./exploit <any size 4 string> + <value to modify return address> + <value for payload>

Process of Finding the exploits

Screenshot:



Laptop Computer Specifications: Intel Core i7-7500U @ 2.70 GHz, Windows 10 Pro, 64bit processor and 64 bit Operating System

The first and foremost point of figuring out the exploit was to understand how a buffer overflow exploit work. A buffer overflow exploit takes advantage of a program's vulnerability when it writes data beyond a designated storage area. By intentionally overwriting critical information, like return addresses, attackers can seize control of the program's operation, potentially leading to unauthorized access or system compromise.

The first step was to figure out was whether we could exploit the program or not. Looking at the code, there were no security measures in process to ensure that data.txt is of appropriate size and it uses fgets which is easily exploitable. My goal was to then find the size of data.txt for which the code would result to a segmentation fault.

I then proceeded to fill data.txt with some values until I received a segmentation error. What I noticed was when doing the sorting, each data separated by a new line was treated as a new entry. But one entry could have 8 digits.

i.e. entering

a
a
a

was same as doing

aaaaaaaa
aaaaaaaa
aaaaaaaa

but if I enter more than 8 in one line:

aaaaaaaaaa

the program would treat it as:

aaaaaaaa
aa

This provided me some clue on how the data is structured. Further, what I also noticed was at the 28th line, if I enter anything I get a segmentation fault.

This means that my program flow was altered, and I was accessing a part of memory that I was not supposed to. i.e. I was changing the return address.

From here, I got a general clue of where the return address was located on the stack.

I then proceeded to find the following addresses:

- Address of system
- Address of _exit
- Address of /bin/sh

When I started, I started in gdb, I set a breakpoint inside in main using 'break main' to find the address of system using 'p system'. I did similar thing for exit but used p exit instead of p _exit. So, I had to redo it to find the address of p _exit. I used the find function to find the address of /bin/sh.

My addresses were:

- System = 0xb7e57190
- _exit = 0xb7ccbc4
- /bin/sh = 0xb7f77a24

I then set them in the 28 29 and 30th line of the document without the 0x which resulted in the code opening a /bin/sh and exiting cleanly.

As you can see in the screenshot, before the execution, the pid is at 2094 but after it changes to 2157. This shows that the buffer overflow exploit was successfully carried out. Further upon exiting, the code exits cleanly and without any segmentation fault.