

CP400S

Lab 11 Report

Mohammad Faham Khan

IDs: 120976210

Introduction

In this lab, we were given code that was vulnerable to buffer-overflow attacks. The purpose of this lab was to find out how we can secure our code from buffer overflow attacks along with any other exploits.

Within this lab report, there are 2 parts, for Part 1 I've given step by step solutions for each buffer overflow exploit and how I fixed it. For Part 2 although we were asked to only point out the other exploits and potential fixes, I've gone ahead and taken the liberty to try and fix these exploits.

The code for the buffer-overflow fix and the security fix have been attached along with this lab report.

Table of Contents

Part 1 - Fixing Buffer Overflow Vulnerabilities:

- 1.1 Vulnerability 1 - Formatting of print
- 1.2 Vulnerability 2 - gets to fgets
- 1.3 Vulnerability 3 - sprintf to snprintf
- 1.4 Testing and Extra Testing

Part 3 - Finding Other Exploits:

- 2.1 Vulnerability 1 - copywrite variable code injection
- 2.2 Vulnerability 2 - malloc overloading system
- 2.3 Vulnerability 3 - User input code injection

Part 3 - Fixing Other Exploits:

- 3.1 Vulnerability 1 - Replacing system call with printf
- 3.2 Vulnerability 2 - Removing need for malloc
- 3.3 Vulnerability 3 - Replacing system call with C file functions

Conclusion

Part 1 - Fixing Buffer Overflow Vulnerabilities

Here I will fix the buffer overflow issues this program has. There were multiple issues, and some fixes also brought up other issues, which I've gone ahead and fixed as well.

Vulnerability 1:

Old Code:

```
if(argc>=2) {  
    strncpy(buf, argv[1], 100);  
    printf(buf);  
  
}
```

New Code:

```
if(argc>=2) {  
    if(strlen(argv[1]) < 100){  
        strncpy(buf, argv[1], 100);  
        printf("%s",buf);  
    }  
  
}
```

Issue 1: The issue here was that the buffer could be smaller than whatever the user may have inputted. Say that the user inputted a string that was larger than 100. Although the strncpy would be a safe copy, I felt that adding a layer of protection never hurts.

Fix 1: To add another layer of protection to this I added a checker to see if the string length is less than 100. If the length is lower than 100 only then do I print it.

Issue 2: In the same piece of code the printf(buf) is open to exploits as well. It is always unsafe to not format the printf function. (Say the user enters %s%s%s, this would cause the program to crash)

Fix 2: I add formatting %s, so that only strings would be printed.

Vulnerability 2:

Old Code (*this occurs multiple times with different variables*):

```
gets(firstname);
```

New Code:

```
fgets(firstname, sizeof(firstname), stdin);
if(firstname[sizeof(firstname)-2] != 10){
    if(strlen(firstname) >= (sizeof(firstname)-1)){
        while ((getchar()) != '\n');
    }
    else{
        firstname[strlen(firstname)-1]=0;
    }
}
else{
    firstname[strlen(firstname)-1]=0;
}
```

Issue 1: The issue here was that “gets” allows for users to enter input larger than the size of the “firstname” buffer. Because of this a buffer overflow can occur.

Fix 1: I use the secure version of gets, fgets. This allows us to only take input as large as the buffer or smaller.

Issue 2: By using fgets we have another issue where the extra user input would leak over to the 2nd input.

Fix 2: I added “while ((getchar()) != '\n');” so that we clear out the user input.

Issue 3: But now since we added “while ((getchar()) != '\n');” another issue occurs where normal user input which is less than the total buffer size would cause an error.

Fix 3: To fix this we added “if(strlen(firstname) >= (sizeof(firstname)-1))”, this checks to make sure if the user entered input more than the max buffer size.

Issue 4: Now because of our previous fix, we have another issue. If the user inputs exactly (buffer size - 2), our program would cause an error.

Say for example firstname, the buffer is 30, if the user inputs 28 characters, the program would not work. The reason for that is because fgets adds 1 more character, thus

making it 29, and because 29 is equal to the size-of-the-buffer -1, it would run the while loop to clear the user input. And this would cause issues.

Fix 4: I was able to find out that if the user inputs buffer-size - 2 values, then I need to just check “if (firstname[sizeof(firstname)-2] != 10)”, normally, if the user tried to overflow that “if” statement would not pass, but because they didn’t, and it still shows up as 29, the program would check and see that in fact there was no buffer overflow. Thus it would go ahead and run “`firstname[strlen(firstname)-1]=0;`”

Final Statement for Issue 2:

For this, we also could have used `fgetln()` but it was not the function I used initially and I personally wanted to work on fixing `fgets()`.

Vulnerability 3:

Old Code:

```
n=sprintf (buffer, "echo %s %s:%s:%s:%s:%s >> ./userdatabase",  
firstname,lastname,jobtitle,email,phone,company);
```

New Code:

```
n= snprintf(buffer, sizeof(buffer), "echo %s %s:%s:%s:%s:%s >>  
./userdatabase", firstname,lastname,jobtitle,email,phone,company);
```

Issue 1: This code causes a buffer overflow similar to gets, thus we need to change it.

Fix 1: I changed sprintf to snprintf so that we can define the size of the buffer and make sure no overflow occurs.


```
[03/30/2018 16:25] root@ubuntu:/home/seed/Documents/lab11# ./lab11
Welcome to ABC Company Registration
♦ 2018 Wilfrid Laurier University
Please fill out the registration form by answering questions below
First name:Hello
Last Name:I
Job title:Am
Email:Testing
Phone:This
Company:Now

Your Registration Information is:
First name:Hello
Last name:I
Job title:Am
Email:Testing
Phone:This
Company:Now

[03/30/2018 16:26] root@ubuntu:/home/seed/Documents/lab11# ./lab11
Welcome to ABC Company Registration
♦ 2018 Wilfrid Laurier University
Please fill out the registration form by answering questions below
First name:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Last Name:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Job title:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Email:a
Phone:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Company:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Your Registration Information is:
First name:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Last name:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Job title:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Email:a
Phone:aaaaaaaaaaaaaaa
Company:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

[03/30/2018 16:26] root@ubuntu:/home/seed/Documents/lab11#
```

```
[03/28/2018 20:40] root@ubuntu:/home/secd/Documents/lab11# valgrind --leak-check=yes lab11
==15615== Memcheck, a memory error detector
==15615== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==15615== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==15615== Command: lab11
==15615==
Welcome to ABC Company Registration
• 2018 Wilfrid Laurier University
Please fill out the registration form by answering questions below
First name:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Last name:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Job title:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Email:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Phone:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Company:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Your Registration Information is:
First name:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Last name:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Job title:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Email:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Phone:aaaaaaaaaaaaa
Company:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

==15615==
==15615== HEAP SUMMARY:
==15615==   in use at exit: 0 bytes in 0 blocks
==15615== total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==15615==
==15615== All heap blocks were freed -- no leaks are possible
==15615==
==15615== For counts of detected and suppressed errors, rerun with: -v
==15615== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Part 2 - Finding Other Exploits

Here we will try to exploit the code outside of just the buffer overflow. In the next section, I've gone ahead and taken the liberty to try and secure the code as well, so that there won't be any vulnerabilities. The secure code is attached along with this lab report.

Question:

In addition to buffer overflow, do you think that there are other software vulnerabilities?
_____ (Yes/No)

Answer:

Yes

There are actually a few ways to take advantage of the code here, and they mostly reside in the "system()" function, which can be exploited with code injection.

Vulnerability 1:

Code:

```
copyright = getenv("COPYRIGHT");

printf("Welcome to ABC Company Registration\n");
if (copyright!=NULL) {
    needed = snprintf(NULL, 0, "echo ♦ %s", copyright);
    cstr = malloc(needed+1);
    snprintf(cstr, needed+1, "echo ♦ %s", copyright);
    system(cstr);
    free(cstr);
} else {
    system("echo ♦ 2018 Wilfrid Laurier University");
}
```

Why this is an issue:

The way this code is executed is that it checks if the system the code is running on has an environment variable set for the term COPYRIGHT. Now what we expect is that the environment variable would be set to something like “Laurier” or “Faham Khans Company” etc. But what if someone was able to set the environment variable with malicious code?

Say that it was set to “Company ; rm userdatabase”, this would go ahead and delete the database when the code was executed. The possibility of what you can do here is endless. Say that the code runs with root access, then a user who does not have root access could be granted root access by just setting their current environment variable to whatever allows them root access.

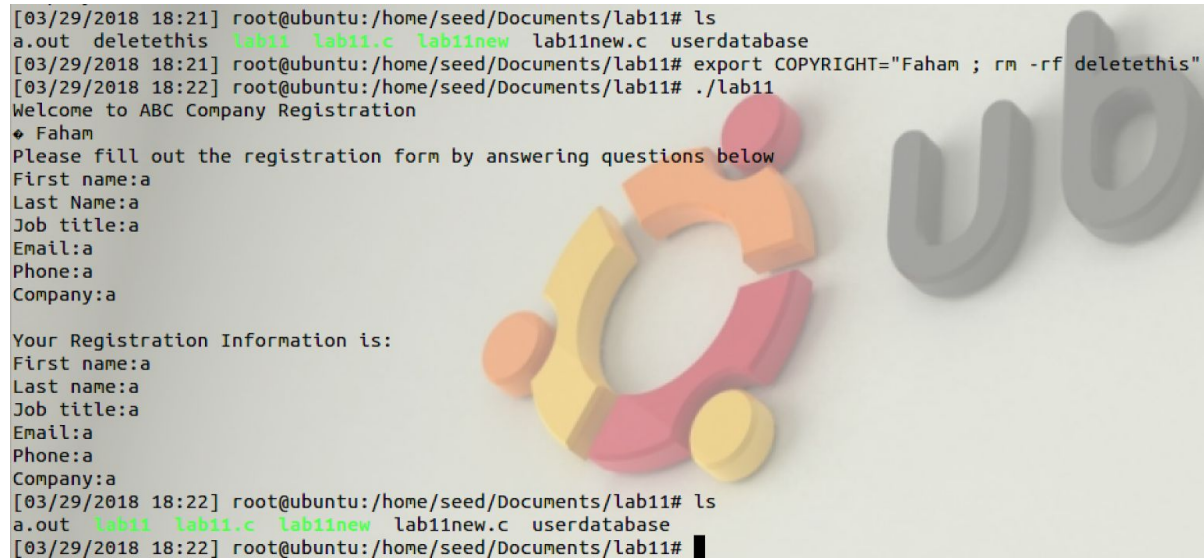
How we can solve this:

To solve this we can go ahead and get rid of the “system(cstr)” function call, and replace it with something like “printf(“%s”, cstr)” etc.

But say that for some reason it is absolutely necessary that we have the system call. In that case, we would need to sanitize the variable “copyright”.

Example with malicious input:

Here we just try to delete the file “deletethis” by injecting code into the COPYRIGHT.



```
[03/29/2018 18:21] root@ubuntu:/home/seed/Documents/lab11# ls
a.out deletethis lab11 lab11.c lab11new lab11new.c userdatabase
[03/29/2018 18:21] root@ubuntu:/home/seed/Documents/lab11# export COPYRIGHT="Faham ; rm -rf deletethis"
[03/29/2018 18:22] root@ubuntu:/home/seed/Documents/lab11# ./lab11
Welcome to ABC Company Registration
♦ Faham
Please fill out the registration form by answering questions below
First name:a
Last Name:a
Job title:a
Email:a
Phone:a
Company:a

Your Registration Information is:
First name:a
Last name:a
Job title:a
Email:a
Phone:a
Company:a
[03/29/2018 18:22] root@ubuntu:/home/seed/Documents/lab11# ls
a.out lab11 lab11.c lab11new lab11new.c userdatabase
[03/29/2018 18:22] root@ubuntu:/home/seed/Documents/lab11#
```

A lot more can be done than just removing the “deletethis” file. If the right permissions were set on the file, we could even see the shadow file etc.

Vulnerability 2:

Code:

```
cstr = malloc(needed+1);
```

Why this is an issue:

This is a potential issue because what if the user entered a value into COPYRIGHT which is larger than the size the computer can handle? This may cause the computer itself to overload or crash.

How we can solve this:

We can limit the size of the cstr value so that it wouldn't be too much.

Example:

An example here would be too extensive, but the idea would be to create a variable so large that it crashes the program.

Vulnerability 3:

Code:

```
i = system(buffer);
```

Why it is an issue:

Just like before the system call here can cause issues because we can execute code that we add after the “;” symbol. This code injection could allow the attacker to cause serious harm to the system or allow them to access data they should be unable to access.

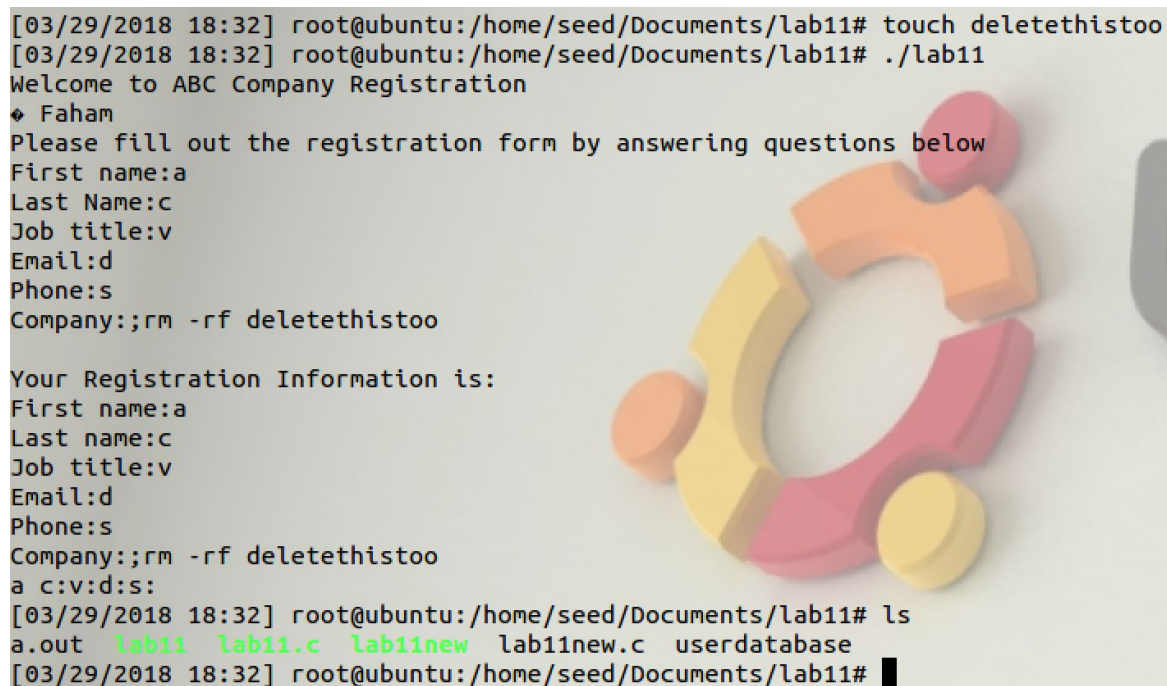
How we can solve this:

We can solve this in 2 ways.

First we can sanitize the entire input, and only then do we pass it into the system() function call.

Or we can replace system with function specifically made for writing to files.

Example with malicious input:



```
[03/29/2018 18:32] root@ubuntu:/home/seed/Documents/lab11# touch deletethistoo
[03/29/2018 18:32] root@ubuntu:/home/seed/Documents/lab11# ./lab11
Welcome to ABC Company Registration
♦ Faham
Please fill out the registration form by answering questions below
First name:a
Last Name:c
Job title:v
Email:d
Phone:s
Company:;rm -rf deletethistoo

Your Registration Information is:
First name:a
Last name:c
Job title:v
Email:d
Phone:s
Company:;rm -rf deletethistoo
a c:v:d:s:
[03/29/2018 18:32] root@ubuntu:/home/seed/Documents/lab11# ls
a.out lab11 lab11.c lab11new lab11new.c userdatabase
[03/29/2018 18:32] root@ubuntu:/home/seed/Documents/lab11#
```

Same as the previous example, we created a file called deletethistoo, and using code injection we were able to delete that file.

Part 3 - Fixing Other Exploits:

I've gone ahead and tried to make sure that this code is as secure as possible, and to do so I had to make a few changes to it, and at the same time maintain the functionality.

This section will go over the fixes for the 3 issues presented in "Part 2".

Vulnerability 1 Fix:

Old Code:

```
if (copyright!=NULL) {  
    needed = snprintf(NULL, 0, "echo 💎 %s", copyright);  
    cstr = malloc(needed+1);  
    snprintf(cstr, needed+1, "echo 💎 %s", copyright);  
    system(cstr);  
    free(cstr);  
}
```

New Code:

```
if (copyright!=NULL) {  
    printf("💎 %s\n", getenv("COPYRIGHT"));  
}
```

What I did:

Note: Although I made a lot of changes here, I will discuss the fix for only “Issue 1” from the previous part, please go to the “Issue 2 Fix” part to see a full explanation for the rest of the changes.

I removed the system call in general and replaced it with printf. The general idea of the system call was to echo out the COPYRIGHT info that the user set, and for that, we can just use a formatted printf to achieve the same goal.

Example:

```
[03/30/2018 15:47] root@ubuntu:/home/seed/Documents/lab11# export COPYRIGHT="Faham; rm -rf deletethis"
[03/30/2018 15:48] root@ubuntu:/home/seed/Documents/lab11# ls
a.out deletethis lab11 lab112 lab112.c lab11.c userdatabase
[03/30/2018 15:48] root@ubuntu:/home/seed/Documents/lab11# gcc -o lab112 lab112.c
[03/30/2018 15:48] root@ubuntu:/home/seed/Documents/lab11# ./lab112
Welcome to ABC Company Registration
* Faham; rm -rf deletethis
Please fill out the registration form by answering questions below
First name:test
Last Name:test
Job title:test
Email:test
Phone:test
Company:test

Your Registration Information is:
First name:test
Last name:test
Job title:test
Email:test
Phone:test
Company:test
[03/30/2018 15:48] root@ubuntu:/home/seed/Documents/lab11# ls
a.out deletethis lab11 lab112 lab112.c lab11.c userdatabase
[03/30/2018 15:48] root@ubuntu:/home/seed/Documents/lab11#
```

As you can see here, even though we tried to inject some code into the COPYRIGHT variable, we couldn't execute it.

Vulnerability 2 Fix:

Old Code:

```
if (copyright!=NULL) {  
    needed = snprintf(NULL, 0, "echo 💎 %s", copyright);  
    cstr = malloc(needed+1);  
    snprintf(cstr, needed+1, "echo 💎 %s", copyright);  
    system(cstr);  
    free(cstr);  
}
```

New Code:

```
if (copyright!=NULL) {  
    printf("💎 %s\n", getenv("COPYRIGHT"));  
}
```

What I did:

There was no need for us to use malloc, cstr, needed etc. Our purpose is to print out the value of COPYRIGHT if it exists, and if not we print out something else. I went ahead and removed the unneeded elements so that the attacker would have even fewer chances to take advantage of our program.

Example:

For this example, please look at the previous example since they are part of the same fix.

Vulnerability 3 Fix:

Old Code:

```
n= snprintf(buffer, sizeof(buffer), "echo %s %s:%s:%s:%s:%s >>
./userdatabase",
    firstname,lastname,jobtitle,email,phone,company);
i = system(buffer);
```

New Code:

```
n= snprintf(buffer, sizeof(buffer), "%s %s:%s:%s:%s:%s",
    firstname,lastname,jobtitle,email,phone,company);

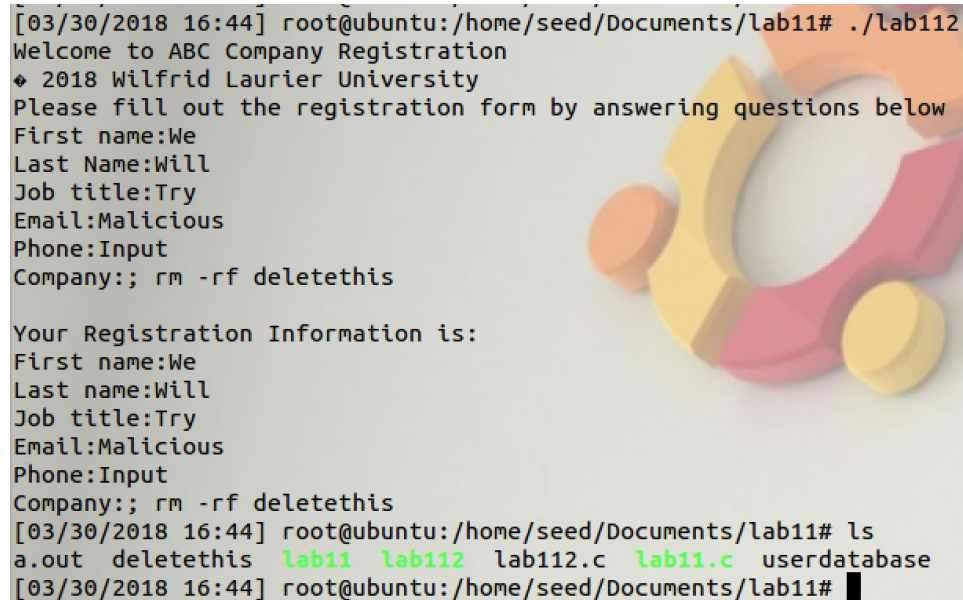
FILE *fp = fopen("userdatabase", "ab+");
if (fp == NULL)
{
    printf("Error opening file!\n");
    exit(1);
}
fprintf(fp, "%s\n", buffer);
```

What I did:

The purpose of this part was to write to the file called userdatabase. But because the system() function allows injections it would be dangerous to use it, as it allows users to inject malicious code.

To fix this we replaced it with the actual c function to read and write to files. By doing this we remove the possibility for users to inject code maliciously.

Example



```
[03/30/2018 16:44] root@ubuntu:/home/seed/Documents/lab11# ./lab112
Welcome to ABC Company Registration
♦ 2018 Wilfrid Laurier University
Please fill out the registration form by answering questions below
First name:We
Last Name:Will
Job title:Try
Email:Malicious
Phone:Input
Company:;; rm -rf deletethis

Your Registration Information is:
First name:We
Last name:Will
Job title:Try
Email:Malicious
Phone:Input
Company:;; rm -rf deletethis
[03/30/2018 16:44] root@ubuntu:/home/seed/Documents/lab11# ls
a.out deletethis lab11 lab112 lab112.c lab11.c userdatabase
[03/30/2018 16:44] root@ubuntu:/home/seed/Documents/lab11#
```

Here you can see that even with code injection, we were unable to execute the malicious code.

Part 3 Final Thoughts:

Another way to solve the issues and keep the system() functions would be to sanitize the user input. But there is always a risk involved in that so I decided to change up the code itself.

Conclusion:

In this lab report, we went over quite a few details, especially buffer overflows and other software vulnerabilities.

It was exciting to see that simple pieces of code could easily be used to cause security exploits. But at the same, time it was assuring to see that these exploits could be avoided if one takes the time to analyze their code and fix it.