3

Graded Assignment: Path Traversal and Buffer Overflow

Hiren Shah

3/29/2021

# Executive Summary

This vulnerability assignment requires insecure interactions between components that can make software applications vulnerable to an attack. The two unique vulnerabilities demonstrated in this assignment are the CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'), and the CWE-120:

I successfully fixed the vulnerable example for CWE-22 and for CWE-120 s

# Example 1 – [CWE-22: ‘Path Traversal’]

## Overview

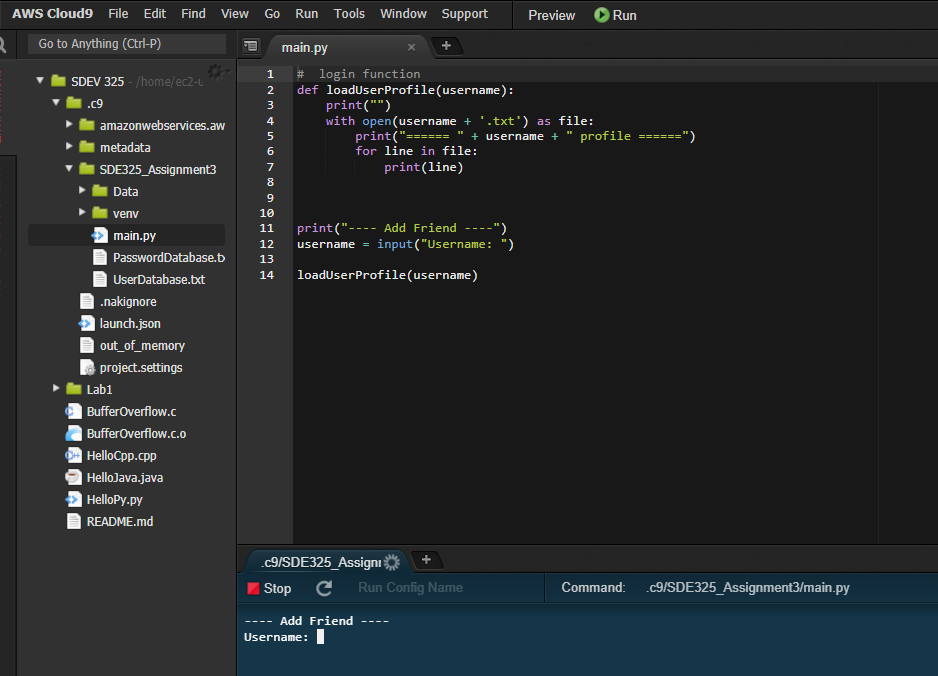
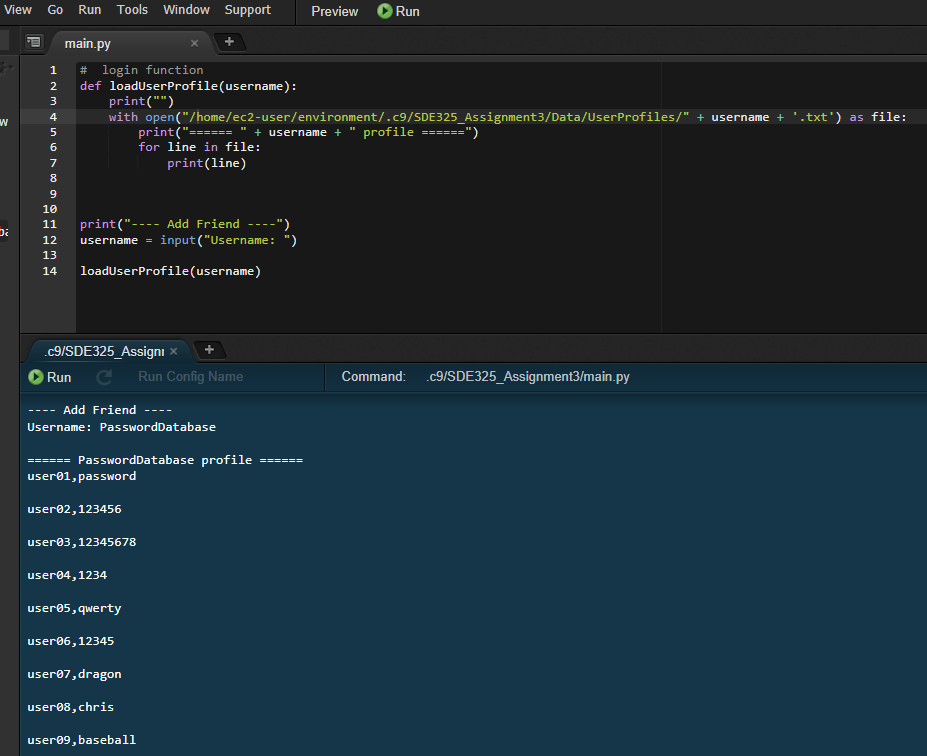
Our application is an example of a possible social media (ex. Facebook) in which you can add friends by typing in their ‘username’, which then loads their profile from an external file that is provided by the user. The application is written in Python. (Note: the whole code shows the vulnerability)

Figure 1. Showing Startup of application when running.

## Analysis of the Vulnerability

The vulnerable portion of our code is related to how the users profile’s text file is opened without ensuring it is actually a profile’s text file. This would allow a malicious user to type a path instead of a username. This would allow them to see any file that they knew the path to. (ex. ../path/password.txt)

Figure 2. Showing malicious code being executed by user entering sensitive passworddatabase file.

## Mitigation

The code uses Input Validation strategy to mitigate the vulnerability. First the application gets the full path name to the user profile’s file that the user is requesting. Then, the application compares the length of the full path to an expected path. After which the application compares each folder in the expected path to the actual path.

**# get expected path and actual path**

**expectedPath = "/home/ec2-user/environment/.c9/SDE325\_Assignment3/Data/UserProfiles"**

**actualPath = "/home/ec2-user/environment/.c9/SDE325\_Assignment3/Data/UserProfiles/" + username + '.txt'**

**actualPath = os.path.realpath(actualPath)**

**# check if actual path is within the expected directory**

**# if validatePath returned false**

**# we know the path the user gave is not correct**

**if validatePath(expectedPath, actualPath) == False:**

**print("Nice try punk. Ya aint getting nofin!")**

**return**

**# check if the path the user entered goes to the appropriate directory**

**# return true if the path the user gave us is valid (ie in the userProfiles folder)**

**# return false if the path is not valid**

**def validatePath(expectedPath, autoPath):**

**#print("expected:" + expectedPath)**

**#print("Actual:" + autoPath)**

**expectedFolders = expectedPath.split('/')**

**actualFolders = autoPath.split('/')**

**actualFolders.pop() # remove the userfile from the path (ex. user01.txt)**

**#print(expectedFolders)**

**#print(actualFolders)**

**# both paths should be the same length now**

**if len(actualFolders) != len(expectedFolders):**

**return False**

**# make sure each folder matches**

**for i in range(0,len(expectedFolders)):**

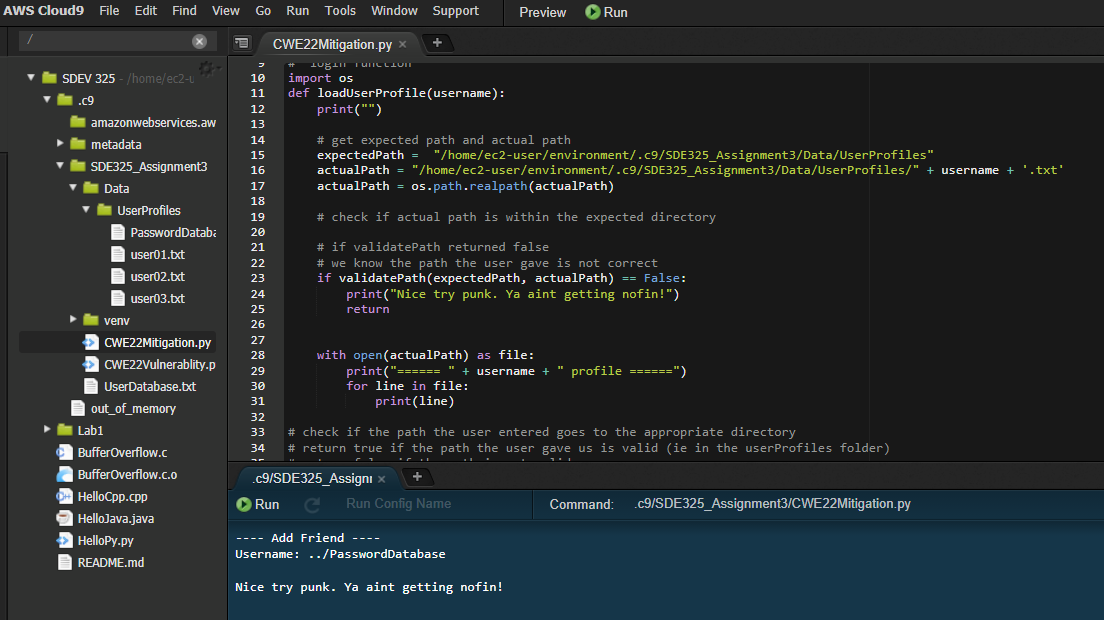
**# return false if a single folder in path does'nt match**

**if expectedFolders[i] != actualFolders[i]:**

**return False**

**# if we made it here, the path must be valid**

**return True**

Figure 3. Showing attacker using malicious code to no avail.

# Example 2 – [CWE-120: Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)]

## Overview

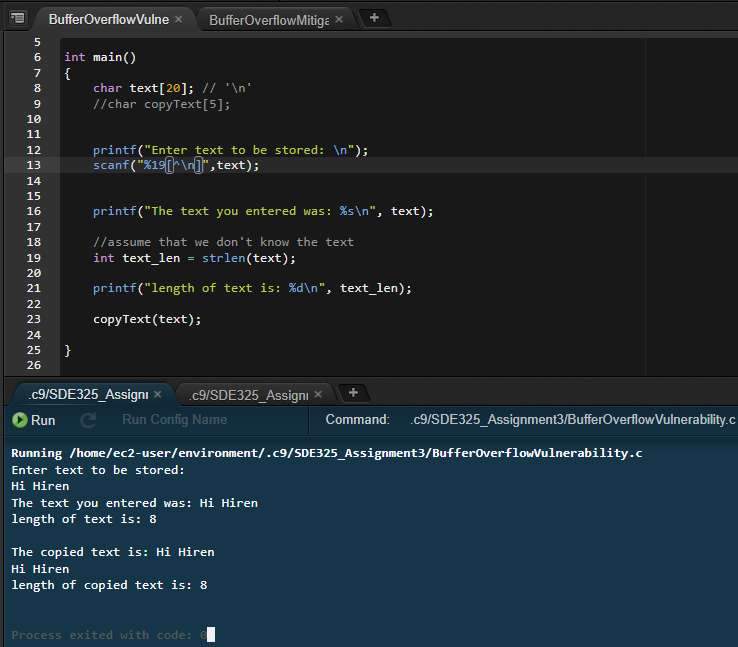
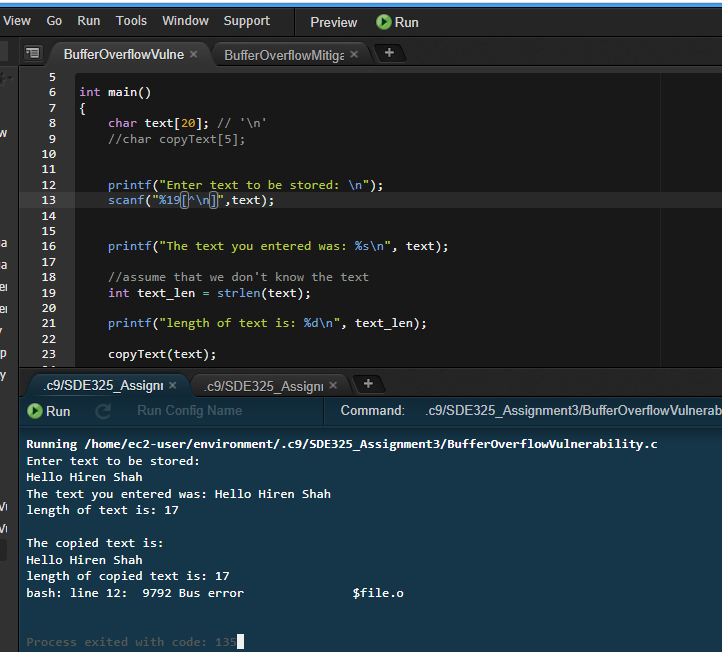
This application asks the user some input, then it copies it into another buffer, next it displays the copied buffer. The program does not check the size of the input, so in the vulnerable source file, this is shown when you get the segmentation fault. The code is mitigated by first gettgin the length of the orginal buffer, and then using that length in the malloc to dynamically allocate space for the buffer. The language I used in this application is C.

Figure 4. Showing program running within parameters.

## Analysis of the Vulnerability

In the vulnerable example the size of the copied buffer is only 10 char. long as specified, while the original buffer has a size of 20 char. If the input from the user is more than 10 char., it will throw a segmentation fault when copying into the other buffer.

Figure 5. Showing a memory error.

**/\***

**\* Vulnerable copy**

**\*/**

**void copyText(char\* text){**

**char copyText[10];**

**strcpy(copyText, text);**

## Mitigation

In the mitigated example we first checked the size of the input using the strlen(). Once the size is for the copied buffer is known, it is dynamically created using malloc(). This code does not throw the memory error, hence the code is mitigated. This way the original buffer could be any size for it to be copied over and the copied text method will still work.

**//assume that we don't know the text**

**int text\_len = strlen(text);**

**// 10 \* 8bytes = 80bytes;**

**char \*copyText = malloc(sizeof(char) \* text\_len);**

**if(copyText == NULL){**

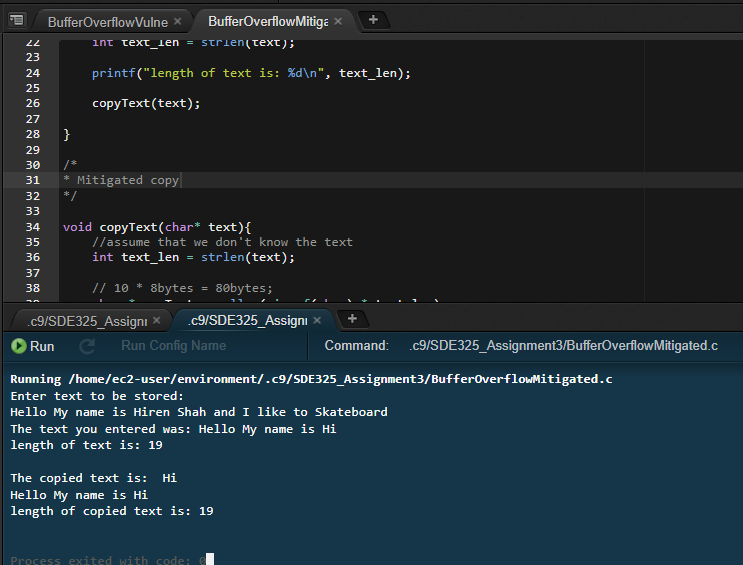
**printf("Not enough memory to create");**

**return;**

**}**

**strcpy(copyText, text);**

Figure 6. Showing input of chars > 20 and still working.



# References