W05 Lab: Homographs

Logan Holland, Noah Cook, Ryan Arveseth, Scott Malin, Tyler DeFreitas

1. Description

The purpose of this project was to learn how to build an encoding system to detect when two homographs exist. The reason this is important is because when building an application there are possible files we would want to keep out of reach from those without admin access. We can achieve this goal by ensuring that all file paths that lead to the protected files are checked and blocked. If we do not ensure this, then private information may be leaked.

2. Relevant Output

This code allows the person running the program to make 3 choices: run the program, run the tests, or exit. When two file paths which are homographs are entered, the expected output is a message saying, “The file paths are homographs”. When two non-homographs are entered, the following message should be shown, “The file paths are not homographs.”

3. Code Fragments

The first step in determining if the two file paths are homographs is to split the file paths into its parts. The function below splits a single file path into each of the various steps it takes and stores it into a vector. This allows us to evaluate the file path more in-depth.

```
26  /* Intention of this function is to take a path(in the form of a string) and separate it into its
27  component commands.*/
28  std::vector<std::string> SplitPath(std::string str) {
29
30      // path_elements holds the commands contained in the path
31      std::vector<std::string> path_elements = std::vector<std::string>();
32
33      int i = 0;
34
35      for (int ii = 0; ii < str.length(); ii++) {
36          if (str[ii] == SLASH || ii == str.length() - 1) {
37              path_elements.push_back(str.substr(i, ii - i + 1));
38              i = ii + 1;
39          }
40      }
41      return path_elements;
42  }
```

Next, the vector is passed into the SanitizePath function. SanitizePath analyzes each of the various steps that were previously broken down and then determines what the final file path would look like.

The first step taken by SanitizePath is to get the current working directory. Next, if the first command is a single slash or reference to a root drive (‘C:/’), the path is taken back to the root drive.

```
50        /* In the interest of usability, all backslashes are to be replaced by forward slashes*/
51        std::string clean_string = ReplaceAllSlashes(std::filesystem::current_path().string());
52        clean_string.push_back(SLASH);
53
54        // If the first character of the command is a slash, it starts the path from root.
55        if (path_elements[0][0] == SLASH) {
56            clean_string.erase(3, clean_string.size() - 1);
57            path_elements.erase(path_elements.begin());
58        }
```

Next, a loop runs through the vector of commands previously provided. Two dots followed by a slash (‘../’) returns to the parent directory. A single dot followed by a slash (‘./’) refers to the current directory. Other commands refer to the folder names they would like to reference. After the loop is completed, the result is a string which is the canonicalized file path.

```
74        // This loop is designed to iterate through the various commands
75        for (int i = 0; i < path_elements.size(); i++) {
76
77            // Two dots and then a slash refers to going back to the parent directory.
78            if (path_elements[i] == DOT_DOT_SLASH) {
79                // This removes everything after the second-to-last slash
80                clean_string.erase(clean_string.find_last_of(SLASH), clean_string.size() - 1);
81                clean_string.erase(clean_string.find_last_of(SLASH) + 1, clean_string.size() - 1);
82            }
83            // A single dot and then a slash refers to the current directory. Do nothing.
84            else if (path_elements[i] == DOT_SLASH) {
85
86            }
87            // Otherwise, add the current command to the path.
88            else {
89                clean_string.append(path_elements[i]);
90            }
91        }
92        return clean_string;
93
94    }
95
```

Finally, after the two file paths are put through the canonicalization process, the strings are compared to see if they are equivalent. If they are equivalent, they are homographs. Otherwise, they are not homographs.

4. Test Cases

There are nine total tests that are conducted when running the test cases: five sets which are homographs, and four sets which are not. The tests use asserts on the PathsAreHomographs() function. This function replaces uses of "\" with "/", then Sanitizes the paths and finally compares them to each other.

Homograph Test Strings:

First Path	Second Path	Expected Result	Outcome
testing.txt	./././../testing.txt	**Homographs**	**Passed**
/test/path/here/testing.txt	/./test/path/here/testing.txt	**Homographs**	**Passed**
test/path/here/testing.txt	test/../test/.././test/path/here/testing.txt	**Homographs**	**Passed**
C:/test.txt	/test.txt	**Homographs**	**Passed**
C:/test.txt	c:\./test.txt	**Homographs**	**Passed**

Non-Homograph Test Strings:

First Path	Second Path	Expected Result	Outcome
testing.txt	./../../../testing.txt	**Not Homographs**	**Passed**
/test/path/here/testing.txt	/test/../path/here/testing.txt	**Not Homographs**	**Passed**
test/path/here/testing.txt	test/../test/./../test/path/here/testing.txt	**Not Homographs**	**Passed**
test.txt	C:/test.txt	**Not Homographs**	**Passed**

5. Justification

We used the filesystem module from C++ to check both file paths that are inputted and compare them against each other. After multiple versions of the lab and independent research

we were able to create a file that will check the two file paths and decide if they are homographs. Our file was able achieve one hundred percent accuracy on our test cases. We are confident that our file will hold up to scrutiny and be able to be deployed on any system.