ISTANBUL TECHNICAL UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

BLG 460E: Secure Programming
CRN: 21574

ASSIGNMENT 2

Instructor : Asst. Prof. Dr. Mehmet Tahir Sandıkkaya
Teaching Assistant : Res. Asst. Ayşe Sayın

150180905 : FATIMA RAHIMOVA

April 5, 2022

Contents

1 Introduction

In this assignment, canonicalization issues are discussed, given code files analyzed and after finding vulnerabilities in the codes, important changes were made in the codes in order to defeat such issues. Actually code modifications are different for different operating systems(explained in more detail in the report in below). In the first part of the assignment, the aim is try to find directory traversal vulnerability in the C code and try to modify code so that attacker cannot bypass the "naive" security check. In the second part of the assignment, the given code was analyzed and doing important changes on the code to verify its functionality against file name handling based canonicalization issues.

2 Part 1

In the first part of the assignment, given code validates a path name whether it is in the /home directory or not. But actually, not in a accurate and proper way. The given code segment is shown in below:

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

void processFile(const char *path){

   printf("Process file executes\n");
   char samplecommandtoexecute[100]={"ls -la "};
   strcat(samplecommandtoexecute,path);
   int status = system(samplecommandtoexecute);
   return;

}

void main(int argc, char *argv[]){

   const char *safepath = "/home";
   size_t spl = strlen(safepath);

   char *fn = argv[1];

   if(!strncmp(fn,safepath, spl)){
     processFile(fn);
   }
   else {
     printf("Path specified is not valid!\n");
     return;
   }

   return;
}
```

Figure 1: Given C code of first part

As you can see code takes an input from command arguments with assuming user will enter proper input path. But it might not work like that, user can enter different file paths to skip the validation performed in the code. If we enter "\home" as an input,

program works as required and all files in "\home" directory will be listed in the user's terminal. But however, if attacker enters a path like "\home\..\", attacker can access the list of all files in the upper directory of "\home". For example, if attacker enters a path like "\home\..\..\", attacker will get the whole list of all files in upper directory of upper directory of "\home". This is the vulnerability of the given code, this code accepts such inputs, however it should not. You can see output of the program, before making some necessary modification:

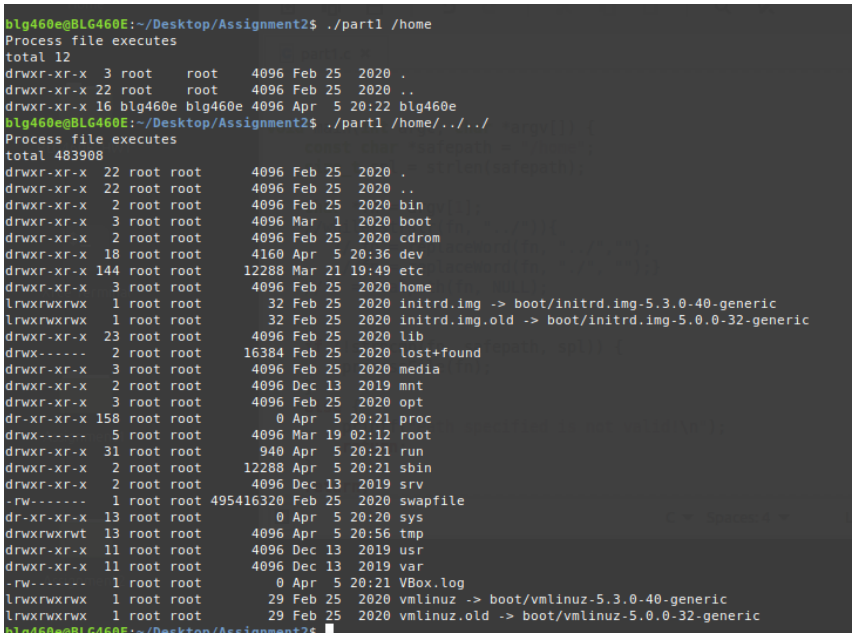

Figure 2: Output of the given C code with not proper input

As you can see in above, we get all files in "\home" directory, which is true. However, by entering the second path as input, we get all files in the 2 upper directory of "\home", which is the bad thing.

Now, we should modify the code in order to defeat this vulnerability. Generally, it can be said program should not allow file paths which includes "../" or "./", since they have special meaning.There are 2 different defeat methods depending on the operating system:

- **Linux Operating System:**

  User or attacker usually will enter relative path to the program as input, however we should convert it to the absolute path. In other words, we convert the given path name to its canonical form. For this purpose, we can use a function called realpath() in Linux operating system. After we send the entered path to realpath() function as an input, resolution of the given path name will not contain of any ".", "..". The modified code is given:

```c
void main(int argc, char *argv[]) {
    const char *safepath = "/home";
    size_t spl = strlen(safepath);

    char *fn = argv[1];
    fn = realpath(fn, NULL);

    if (!strncmp(fn, safepath, spl)) {
        processFile(fn);
    }
    else {
        printf("Path specified is not valid!\n");
        return;
    }
    return;
}

```

Figure 3: Modified code for Linux Operating System

As you can see I added realpath() function to the code, and now we can test our program, and see if it will accept "\home\..\..\" input and will list up directories of "\home" or it will not:

```
blg460e@BLG460E:~/Desktop/Assignment2$ gcc -o part1 part1.c
blg460e@BLG460E:~/Desktop/Assignment2$ ./part1 /home
Process file executes
total 12
drwxr-xr-x  3 root    root    4096 Feb 25  2020 .
drwxr-xr-x 22 root    root    4096 Feb 25  2020 ..
drwxr-xr-x 16 blg460e blg460e 4096 Apr  5 20:22 blg460e
blg460e@BLG460E:~/Desktop/Assignment2$ ./part1 /home/../../
Path specified is not valid!
blg460e@BLG460E:~/Desktop/Assignment2$ ./part1 /home/../
Path specified is not valid!
blg460e@BLG460E:~/Desktop/Assignment2$
```

Figure 4: Output of the code after making modifications

As you can see in above, after doing necessary modifications in the code, it does not list upper directories, even if attacker entered not proper file path.

- **Windows Operating System:**

  In Windows Operating system, realpath() function will not work actually. We can replace all "../" and "./" characters with "", as long as there is no "../" or "./".

```
while(strstr(fn, "../")){
    fn = replaceWord(fn, "../","");
    fn = replaceWord(fn, "./", "");}
```

Figure 5: Modified code for Windows Operating System

However, it is a fact that it is not an such excellent solution. Other way of doing this is using GetFullPathName() function for Windows operating system. However, we should standardize on either short or long(better to use) file names.

3 Part 2

In this part of the assignment, we should figure out whether the given code segment will able to defeat any canonicalization issues regarding with the file name handling. Firstly, I created naivepasswordfile.txt and naivecreditcarddetails.txt files in the same directory with part2.c code (contents of both files, my password is 1234). These 2 txt files are important, and reguler users should not access their contents. However it is possible to access their contents and read sensitive information inside them, with attacks. Similarly, depending on the operating system attacks change, for example case sentivity, 8.3 file naming convention attacks do not work for Linux operating system, however we can successfully bypass "naive" security check with these attacks on Windows operating system.

- **Linux Operating System:**

  As I mentioned before, case sentivity, 8.3 file naming convention attacks do not work for Linux Operating system. However instead of entering *naivepasswordfile.txt* or *naivecreditcarddetails.txt*, if we enter **"./naivepasswordfile.txt"**, or **"./naivecreditcarddetails.txt"**, we can read their contents succesfully as you can see in below:

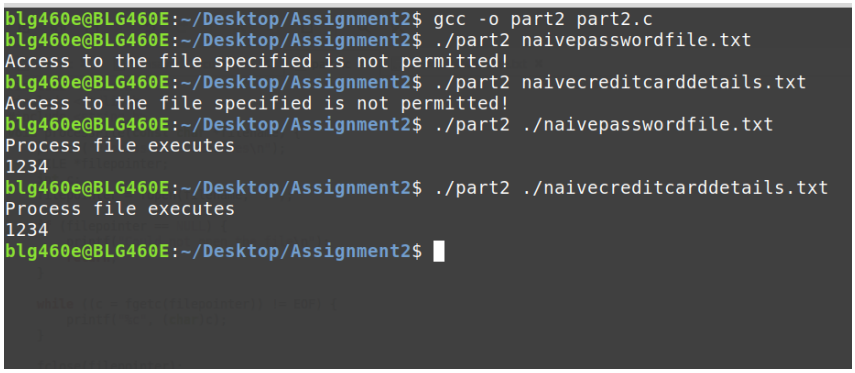

Figure 6: Reading contents of the important txt files, accessing necessary information

As you can see in above, if we enter **"./naivepasswordfile.txt"**, or **"./naivecreditcarddetails.t** we can easily read contents of the important txt files, however they include sensitive data. But we already accessed them, by bypassing "naive" security check. In order to overcome this issue I used, realpath() function again. You can see modified function in below:

```
24  id main(int argc, char *argv[]) {
25    const char *protectedfiles[] = {"naivepasswordfile.txt", "naivecreditcarddetails.txt"};
26    const char *protectedpasswordfile = realpath ("naivepasswordfile.txt", NULL);
27    const char *protectedcardfile = realpath("naivecreditcarddetails.txt", NULL);
28    char *fn = argv[1];
29
30    if (strcmp(realpath(fn, NULL), protectedpasswordfile) != 0 && strcmp(realpath(fn, NULL), protectedcardfile) != 0) {
31        processFile(fn);
32    }
33    else {
34        printf("Access to the file specified is not permitted!\n");
35        return;
36    }
37
38    return;
39
```

Figure 7: Modifying the given code for Linux

With the help of realpath() function, we can defeat against this issue, and now attacker cannot access the contents of these files in the mentioned way:

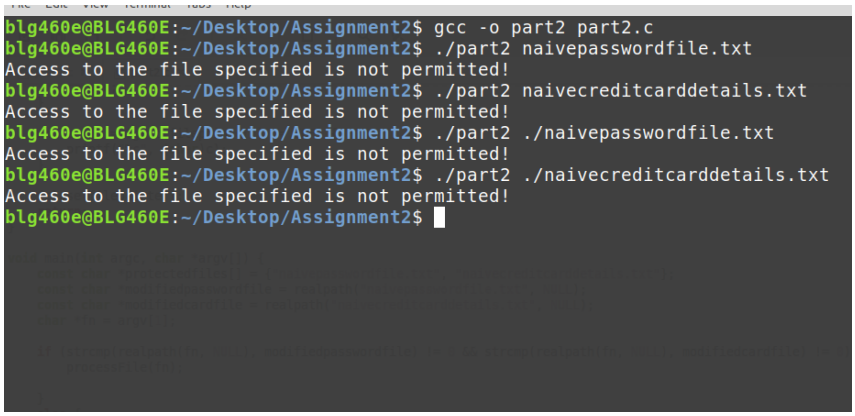

Figure 8: Attacker cannot access to the sensitive information

- **Windows Operating System:**

  As I mentioned before, there are some attacks that can be done successfully in Windows, however they do now work in Linux. One of these attacks is case sensitivity. In Windows Operating System, if we entered "NAIVEPASSWORDFILE.TXT" or "NAIVECREDITCARDDETAILS.TXT", instead of "naivepasswordfile.txt", "naivecreditcarddetails.txt", we can easily access to these txt files and their contents as well. Another attack that works in Windows is 8.3 file naming convention. In this attack, attacker enter a input file name like "naivep 1.txt", instead of "naivepasswordfile.txt", or attacker may enter "naivec 2.txt" instead of entering "naivecreditcarddetails.txt".

  In order to overcome these issues, I added some code lines to the given code segment, I added command lines to the code in order to make it more understandable:

```
29  const char *protectedcardfile = realpath("naivecreditcarddetails.txt", NULL);
30  char *fn = argv[1];
31  for(int i=0; fn[i] != '\0'; i++){
32  //defeating case sensitivity attack
33      if(fn[i]>='A' && fn[i]<='Z'){ // if the given input file name consists of uppercase letters
34          fn[i]=tolower(fn[i]); // make them lowercase(with the help of tolower function)
35      }
36      // defeating 8.3 file naming convention attack
37      if(fn[i] == '~'){ // if file includes any tilda
38          printf("Invalid input!"); // print error message
39          return; // do not accept the given invalid error
40
41      }
42  }
43
```

Figure 9: Defeat methods for Windows attack

Now, attacker cannot attack to our modified program with these attack methods.

REFERENCES

[1] BLG 460E - Secure Programming Course Slides.