İskender Akyüz
150150150

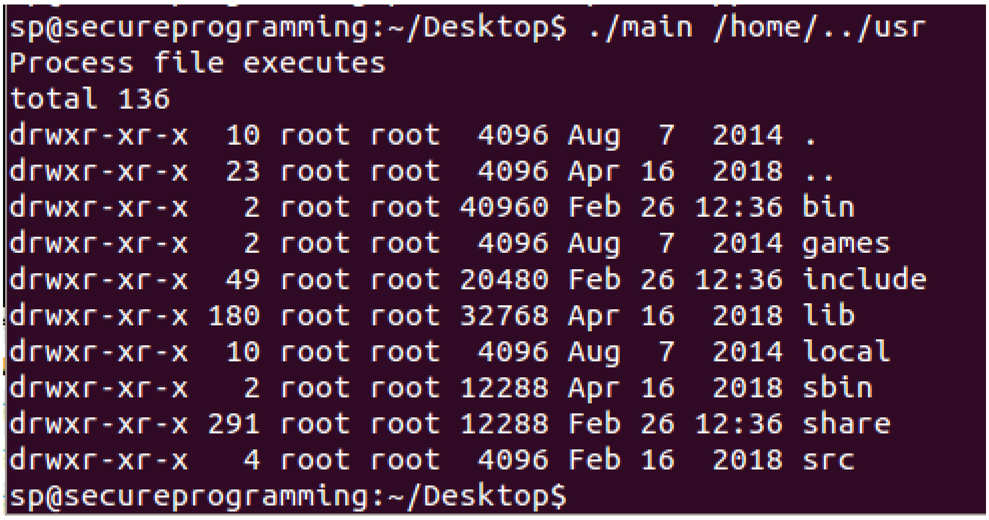
BLG460E – Homework II

Report

Part1: Directory Traversal Issues

Example 1:

./main /home/../usr => by writing this command, we can access a directory which is outside home.

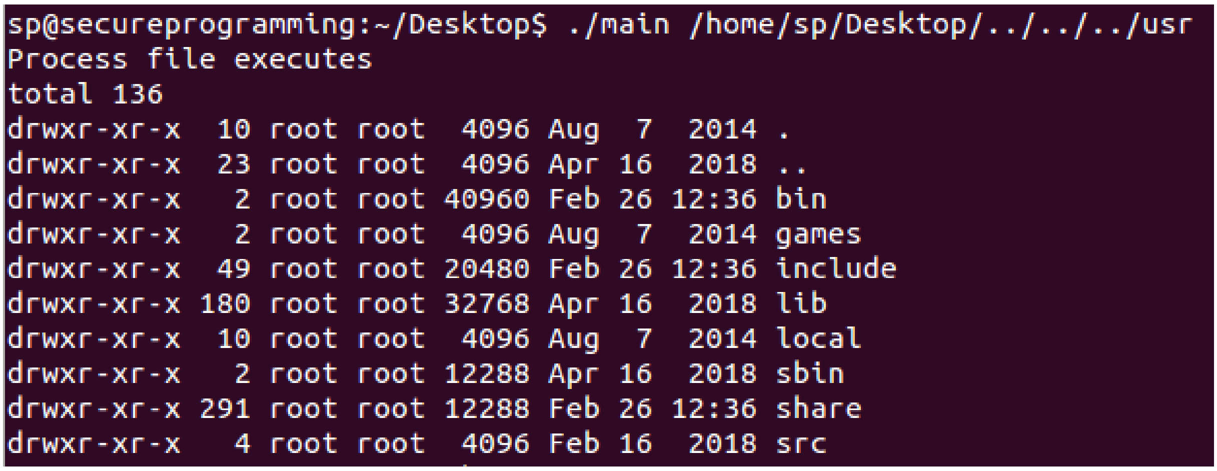


```
sp@secureprogramming:~/Desktop$ ./main /home/../usr
Process file executes
total 136
drwxr-xr-x  10 root root  4096 Aug  7  2014 .
drwxr-xr-x  23 root root  4096 Apr 16  2018 ..
drwxr-xr-x   2 root root 40960 Feb 26 12:36 bin
drwxr-xr-x   2 root root  4096 Aug  7  2014 games
drwxr-xr-x  49 root root 20480 Feb 26 12:36 include
drwxr-xr-x 180 root root 32768 Apr 16  2018 lib
drwxr-xr-x  10 root root  4096 Aug  7  2014 local
drwxr-xr-x   2 root root 12288 Apr 16  2018 sbin
drwxr-xr-x 291 root root 12288 Feb 26 12:36 share
drwxr-xr-x   4 root root  4096 Feb 16  2018 src
sp@secureprogramming:~/Desktop$
```

Since the program checks only the first 5 characters of the path given by the user, we can go up a directory by using two dots and access other files.

Example 2:

./main /home/sp/Desktop/../../../usr => Even if the user go deep inside the home directory, he/she can easily access the outside of it.



```
sp@secureprogramming:~/Desktop$ ./main /home/sp/Desktop/../../../usr
Process file executes
total 136
drwxr-xr-x  10 root root  4096 Aug  7  2014 .
drwxr-xr-x  23 root root  4096 Apr 16  2018 ..
drwxr-xr-x   2 root root 40960 Feb 26 12:36 bin
drwxr-xr-x   2 root root  4096 Aug  7  2014 games
drwxr-xr-x  49 root root 20480 Feb 26 12:36 include
drwxr-xr-x 180 root root 32768 Apr 16  2018 lib
drwxr-xr-x  10 root root  4096 Aug  7  2014 local
drwxr-xr-x   2 root root 12288 Apr 16  2018 sbin
drwxr-xr-x 291 root root 12288 Feb 26 12:36 share
drwxr-xr-x   4 root root  4096 Feb 16  2018 src
```

It is a file path issue. In order to solve this issue, we need deal with .. operator. Also, if there is .. operator in the path, that does not mean that the user is trying to reach outside of home. That is why an efficient solution is needed. I wrote an extra function to check if the path is valid or not.

```
int pathChecker(const char* safepath, const char *path, size_t spl) {
    size_t i = 0, pathlength = strlen(path);
    int validPath = 0, outsideSafe = 0;
    int distanceFromSafe = 0;
    while(i < pathlength) {
        // if we see a "/home" then we are in the safe zone
        if(*(path+i) == '/' && strncmp(path+i, safepath, spl) == 0) {
            // we are not in the outside of safepath, so it is set to false
            outsideSafe = 0;
            distanceFromSafe = 0;
            i += (spl -1); // skipping safepath
            // checking if we are going up from the safe directory
            if(i+3 < pathlength && strncmp(path+i+1, "/..", 3) == 0) {
                outsideSafe = 1;
                distanceFromSafe--;
                i += 3; // skipping "/.."
            }
        }
        else if(*(path+i) == '/' && i+1 < pathlength) {
            i++;
            if(!outsideSafe) {
                if(*(path+i) == '.' && i + 1 < pathlength && *(path+i+1) ==
'.') {
                    distanceFromSafe--;
                    i++;
                }
                else
                    distanceFromSafe++;
            }
            else {
                if(*(path+i) == '.' && i + 1 < pathlength && *(path+i+1) ==
'.') {
                    distanceFromSafe++;
                    i++;
                }
                else
                    distanceFromSafe--;
            }
        }
        i++;
    }
    if(distanceFromSafe >= 0
        validPath = 1;
    else
        validPath = 0;
    return validPath;
}
```

How this function works is that it calculates the distance from home folder after each / in the given path and if the distance is greater than or equal to 0, that means the given path is somewhere in the safe path, so we are good to go. In detail, after detecting a slash, we are looking the next characters.

If the next 2 characters are dots and we are already in the safe path, that means we are getting closer to the safe directory. That’s why we decrease the distance.

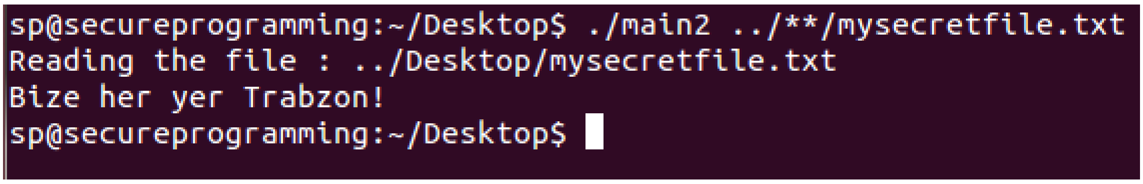
If the next 2 characters are dots and we are not in the safe path, that means we are getting closer to the safe directory. That’s why we increase the distance.

Part2: Directory File Name Handling

Example 1:

In the Linux environment, we can easily access the secret file due to file path issues.

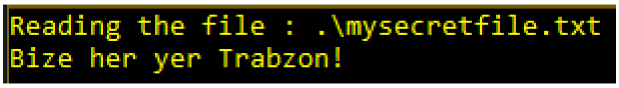
./main2 ../**/mysecretfile.txt => by writing this command, we can trick the program and access the file.



```
sp@secureprogramming:~/Desktop$ ./main2 ../**/mysecretfile.txt
Reading the file : ../Desktop/mysecretfile.txt
Bize her yer Trabzon!
sp@secureprogramming:~/Desktop$ 
```

This attack also works on Windows.

./main2 .\mysecretfile.txt



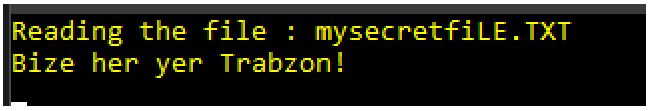
```
Reading the file : .\mysecretfile.txt
Bize her yer Trabzon!
```

Example 2:

In the Windows environment, we can access the secret file by using case sensitivity.

./main2 mysecretfiLE.TXT

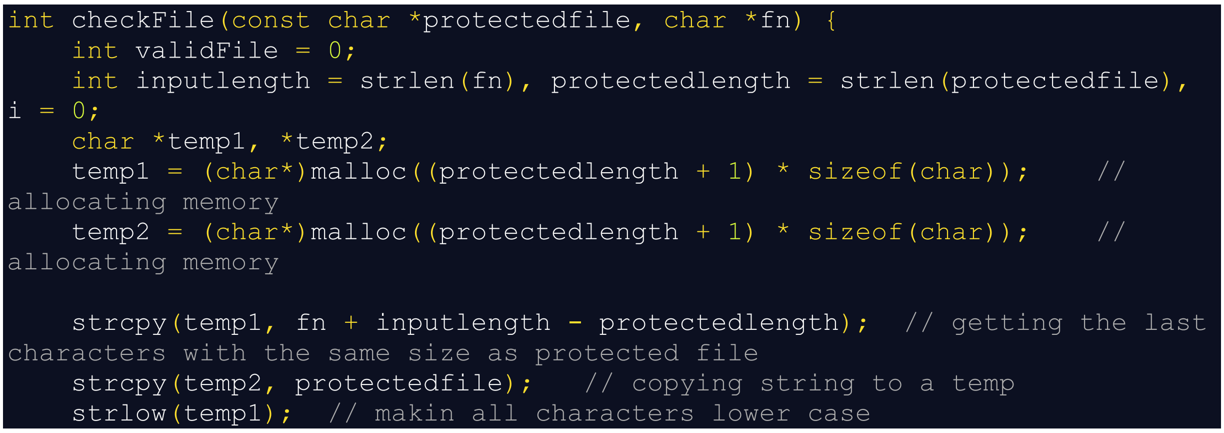
Even though some characters are capital letters, program give access to read the file due to the issues with Windows OS.



```
Reading the file : mysecretfiLE.TXT
Bize her yer Trabzon!
```

We also tried this approach in Linux environment in order to access the secret file; however, it did not work.

In order to prevent these kinds of attacks, I wrote a function that checks the given input. Since the main goal is to prevent user from accessing the protected file, any input that ends with the name of the protected file is an attempt to read the file. That’s why last characters of the given input should be checked. The other thing to consider is case-sensitivity. Since file name comparisons are case-insensitive in Windows environment, this issue should be solved.



```
int checkFile(const char *protectedfile, char *fn) {
    int validFile = 0;
    int inputlength = strlen(fn), protectedlength = strlen(protectedfile),
i = 0;
    char *temp1, *temp2;
    temp1 = (char*)malloc((protectedlength + 1) * sizeof(char));     //
allocating memory
    temp2 = (char*)malloc((protectedlength + 1) * sizeof(char));     //
allocating memory

    strcpy(temp1, fn + inputlength - protectedlength);  // getting the last
characters with the same size as protected file
    strcpy(temp2, protectedfile);   // copying string to a temp
    strlow(temp1);  // makin all characters lower case
    strlow(temp2);  // makin all characters lower case
    if (strcmp(temp1, temp2) == 0)  // comparing two strings
        validFile = 0;  // since they are the same, prevent user reaching
the protected file
    else
        validFile = 1;  // it is okay, user can read the file
    free(temp1);    // freeing memory
    free(temp2);    // freeing memory
    return validFile;
}
```

The function above checks the given input and returns 0 or 1 according to the validity of the input. In order to prevent user reading the protected file, the given input cannot have the protected file name at the end. That is why the last part of the input is compared with the protected file name. If they match, then the user is prevented from reading it. Before checking if they are the same, all the characters of both the given input and the protected file are converted to lower case characters since Windows is case-insentive.