CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Atharv Singh Patlan
190200
CS253A Assignment 4

1. What is Path traversal vulnerability?

Reading and writing to files can be an important path of any program or web server. In order to access these files, a file path is usually taken as input from the user, and the desired action is performed. However if this process is being done on a controlled server (that is the user is not the owner or the administrator), then the owner would want that the user can not access files on the server, or look at files other than those owned by the user.

A malicious user makes use of special elements of Bash path syntax: '/' representing the directory name separator (or a shortcut to move to the root of the system) and '..' denoting parent directory, in order to move out of a restricted directory, and access other files in the system. Thus, an attacker makes use of commands consisting of several "../" to move out of their restricted directories, or simply access files using absolute paths (like "/home/user2/doc.txt") which the user should not be able to access.

Other methods to exploit this vulnerability are by using ASCII codes of these symbols if a filter has been applied to somehow identify such patterns. Other symbols, which are used in Windows file systems, can also be similarly exploited, such as '..\' or the '\' symbol

2. Attack Model

My model for exploiting this vulnerability is to implement a simple text reader in python that takes as input the file a user wants to read, or a folder of which a user wants to know the contents. The vulnerable code can be exploited in this sample, as even though the program asks for a relative path to file, the user can simply move to the parent directory by using ../ or simply provide an absolute path of any file they want. This allows the attacker to view the contents of any folder in the entire system and also read any file they want on the system. Using ASCII codes, however, will not work for the vulnerable code as well because python takes the file input as strings and can not accept nonstring inputs, which mitigates this vulnerability.

3. How to run the codes.

The submission has the following directory structure:

```
.
├── notallowed
│   ├── diary.txt
│   └── password.txt
├── report.pdf
└── user
    ├── alice.txt
    ├── safe.out
    ├── safe.py
    ├── script.sh
    ├── test.in
    ├── vuln.out
    └── vuln.py
```

Here, ‘user’ is the directory that the user should be able to access, hence the codes vuln.py and safe.py are executed in the user folder. The user should not be able to access the ‘notallowed’ directory or any other directory outside ‘user’. So run **cd user** before executing.

NOTE: The steps written below can be automatically executed with the inputs in the file ‘**test.in’** using **./script.sh** from the user directory in the submission. The outputs of **script.sh** on **vuln.py** are stored in **vuln.out**, and the outputs from **safe.py** are stored in **safe.out**.

1. Run vuln.py, and follow the prompts:

   ```
   Enter 1 to list files in a directory
   Enter 2 to view files
   Anything else to quit:
   Enter Input:
   1

   Enter path to file relative to your user directory:
   ../notallowed
   Contents of ../notallowed folder are:

   diary.txt
   Password.txt
   ```

   Clearly, this shouldn’t be happening in a safe file.

2. Run safe.py and follow the prompts:

   ```
   Enter path to file relative to your user directory:
   ../notallowed
   Cleaned path by regex: ./notallowed
   Folder ./notallowed not found
   ```

The safe file was able to prevent access to the folder outside the user's restricted folder.

4. Comparing the vulnerable and safe scripts

The difference in the working of the vulnerable and safe scripts is that the vulnerable script directly accesses any file requested by the user, blindly trusting the user input thus making it vulnerable, allowing the user to access any file on the system.

On the other hand, the safe script first processes the input file path, and cleans the input of any tries to access any directory outside the user's restricted directory.

We first add the symbol "./" to the beginning of any path, which converts any input into a path relative to the 'user' directory (an input like "/home/user2' gets converted to ".//home/user2" which is the same as "./home/user2 which does not exist")

A very common way in order to find attacks on relative paths is to use regex matching to detect the presence of any "../" commands in the path. However, attackers are aware of this and hence this regex test can be fooled by this input: "....//". The regex will detect one "../" and remove it, resulting in the final path to be "../".

Instead, I use the below pseudocode in order to mitigate this issue:

```
1. for i from 0 to len(filename) - 3 do
2.    if filename[i:i+3] == "../" then
3.       filename = filename[:i] + filename[i+3:] //removes ../
4.       if i > 1 ? i -= 3 : else i = -1 //i stays in range
```

In this way, the input path is validated properly, and a cleaned path is returned in order to get the secure file outputs.

5. Preventing Path Traversal attacks

Path Traversal attacks can be prevented by:

1. Finding ways to perform a task without taking any user file path inputs (although this might be difficult)
2. Ensure that any user input is preceded by the path to the user's directory (or './' like I did) to prevent any absolute path attacks.
3. Make use of any programming libraries to normalize any path input, get their absolute paths, such as python's **os.path.normpath()** and **os.path.abspath()**, and then check if the path requested is actually within the allowed folder or not.
4. Directly reject the user input if any attempts to provide an absolute path or move to parent directory are observed.