RedShL Intrusion Detection System

Course: 3809ICT Applied Network Security | Date: 07/05/2019

Group:

* Taylor Radke (s5094922)
* Lucas Geurtjens (s5132841)
* Sebastian Perry (s5132483)

Table of Contents

[Introduction 3](#_Toc8115527)

[Program Functionality 3](#_Toc8115528)

[RedShL 3](#_Toc8115529)

[Flags 3](#_Toc8115530)

[State 3](#_Toc8115531)

[Verification 3](#_Toc8115532)

[Program Modules 4](#_Toc8115533)

[Test case Results 4](#_Toc8115534)

[Summary 5](#_Toc8115535)

# Introduction

The program was designed to detect changes to a given set of files and directories within a Unix operating system. It evaluated this by creating a verification file that could be used as a base case comparison to potential changes. In practical use, the program could eventually be used to detect potential intrusions and file tampering within the system, heightening integrity.

# Program Functionality

## RedShL

RedShL acted as the main of the program and tied the flags, state and verification modules together. Firstly, it parses user input into the flags function to discover which options are being called and if they are valid. From this, the program decides which functions to call, based on which flags were used. Each valid call will always result in the creation of a verification state and applying a verification check to the directory.

## Flags

Flags allowed the tracking of input option flags. Here, here program checked each argument of the input user parameters against the valid candidate option cases. When a valid case is met, its specific variables are stored. Furthermore, error checking allowed the detection of incorrect option parameters and warned the user when they were input.

## State

State allowed the creation of the verification file. Here, for each file/directory, the program stored the following in each line as a string:

* Inode (Basic info about a file/directory)
* File name
* Absolute Path
* File type
* Owner ID
* Group ID
* Access Privileges
* Time Last Modified
* Last Time Accessed

*Exclusively for files and not directories:*

* SHA1
* Word count

With this, the program now has a record of the initial state of the files that will become tracked and can use it during the verification process.

## Verification

After the user has (optionally) made changes to the files, the verification module provides the comparisons with the verification file and the potentially changed files. This functions by reading in the files from the potentially changed files and comparing them against the known verification file state.

# Program Modules

|  |  |
| --- | --- |
| Options | Description |
| -c name | Create a verification file called “name”. This assumes a verification file has not yet been created. |
| -o name | Display results of the comparison check on screen and saved to an output file. This assumes a verification file has already been called (however, does not yet contain the state mapping). |
| -t directory | Choose a directory to track (skipping user input) . This is used to specifically tell the program which directory will be tracked without needing the user input. |
| -v file | Select a pre-existing verification file to use against the current state. This is used when the user already has an existing verification file, hence, does not need to create a new state mapping. |
| --help or -h | Display a help message explaining how to use the program. This lists all the program modules the user has access to. |
| --display-results | Displays the results of verification on screen. This will avoid having to create an output file. |

Table 1: Command Line Options

# Test case Results

|  |  |  |
| --- | --- | --- |
| Command | Input | Output |
| sh RedShL.sh | No directory input | Error: is not a directory |
| sh RedShL.sh --help | None | Displays help |
| sh RedShL.sh -c ok.txt | test\_dir | Created verification file then tested the information |
| sh RedShL.sh -v | None | Error: -v: option requires an argument |
| sh RedShL.sh -v ok.txt -t test\_dir | None because -t allows for the directory to be input as an argument | Verification finished with 0 failing |
| sh RedShL.sh -c test.mp3 | test\_dir | This worked creating the saving the verification as an mp3. This shows it is robust to some unusual file types. |
| sh RedShL.sh -v test.mp3 | test\_dir | It successfully verified all files. |
| Sh RedShL.sh -v test.mp3 | test\_dir  (Various changes were made in test\_dir) | It recognised all changed instance within test\_dir. |
| sh RedShL.sh -v test.mp3 -o output.txt | test\_dir | It recognised all changed instance within test\_dir and output the results into a text file |

Table 2: Test Case Results

It was observed that the program functioned successfully with the given variety of input test cases seen in Table 2. Firstly, error handing succeeded in finding errors where required options were missing. Similarly, the program was also able to handle unusual file types (.mp3) and still function correctly. As well, the program was able to verify when changes were made and when they weren’t made and output them to a file accordingly. Overall, there were no observed major errors and the program appeared to behave as expected.

# Summary

Overall, it was observed that the program functioned successfully. It was able to predict different types of file changes and did not incur any false positive or false negative errors. With further investigation and improvement, this program could eventually become capable of system wide intrusion detection checks similar to anti-virus.