Assignment 2 – System Integrity Verifier

SYLVAIN RONCORONI, Blekinge Institute of Technology

1 Introduction

The aim of this assignment was to implement a simple System Integrity Verifier. It's a tool that can detect changes made in files, folders or subfolders of a given directory. It can be used to monitor important files, on a server for example, and to be sure that no modifications were made, so it can be used to know if a malicious program or person is making some changes or has access to the folders.

2 Design and implementation

The SIV can be run in two modes: initialization or verification. Firstly, I'll present you the initialization and then the verification.
My SIV is implemented in Java, simply because it's the language I'm most familiar with and there are a lot of standard libraries.

2.1 Initialization

The goal of this mode is to go through the files and folders of a given directory, collect information about them then store it in a verification file and finally output some statistics about the directory in a report file.

So, the directory to monitor, the verification and file and the report file must all be given to the program as arguments, and the verification and report files must be outside the monitored directory.

The information gathered from files and folders are:

- The absolute path
- The size
- The name of the owner (user)
- The name of the owner (group)
- Access rights
- Last modification date
- Computed message digest of file content

The program supports the following hashing functions to digest the content of a file: MD-2, MD-5, SHA-1, SHA-256, SHA-384 and SHA-512. They were chosen because they are implemented in native Java.

The first part of the program, in both initialization and verification mode, is to verify the parameters: check that the monitored directory exists and is a directory, check that the report and verification files are outside the directory, check the hashing function, that all the needed parameters are given etc.

Once everything is verified and correct, the program can start.

It works by iterating over each file of the monitored directory and get the information of the file. If there is a subfolder, then the same operation is executed inside the subfolder. The program stops when there is no more file or folder, or when the maximum depth level is reached. It is implemented using a recursive function, here's the pseudo-code:

ALGORITHM 1: Explore directory

```
function exploreDirectory(directory, hashingFunction, depth)
        infos = new List<>()
        for each file f in directory
                if f.isFile()
                        hashedContent = hashingFunction(f.content)
                        infos.add(f.getInformations() + hashedContent)
                else
                        if depth > 0
                                infos.add(f.getInformations())
                                infos.add(exploreDirectory(f, hashingFunction, depth − 1)
```

```
                else
                        infos.add(f.getInformations())
        return infos
```

This is a very high-level view of how the exploration of the files works, but it gives a good idea. When we have the information of the files and folders, it's time to write them into the verification file. To do that, the program iterates over each element of the list of information and write them in the file. The structure of the file is:

- The hashing function in the first line, as a string. For example: sha1

Note that the hashing function written is the one given in arguments. So, if the argument -H SHA-1 is given, the first line will be SHA-1 not sha1, but Java considers that these are the same.

- On each following line, the following information separated by a | (pipe) symbol:
  - Absolute path to the file/folder (String)
  - Size in bytes (long)
  - User owning the file/folder (String)
  - Group owning the file/folder (String)
  - Permissions stored in symbolic notation, example: rwxrwxr-- (String)
  - Date of last modification measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970) (long)
  - Digested content of the file (String), empty string if folder
  - Type: f for file and d for directory/folder (String)

I chose the pipe symbol because this is rarely used in filenames, so the parsing of the verification file will work. Each file/folder takes one line. For example, a verification file can look like that:

```
sha1
/etc/directory/file.txt|40|user|group|rw-r--r--|1544193176991|4b3cb49927ff88720a959aeecd7db6374b7d7d50|f
/etc/directory/folder|0|user|group2|rwxrwxrwx|1544108501363||d
```

Note that some non-empty folders can have a size of 0 bytes if the maximum depth was reached, since (in Java at least) you can't get the size of a folder directly, you need to add the size of each file/subfolder contained in it. So, if the maximum depth is reached, the size is never computed and is set to 0.

2.2 Verification

The goal of this mode is to go through all the files/folders in the monitored directory, collect information (like in initialization mode) and then compare with the verification file to know the changes, if any, done in the monitored directory.

To get the information about the directory, the same function as in the initialization mode is used (see above). The verification file is then read and parsed to get the information, this is done simply by getting the hashing function from the first line then each line is read, and information are separated using the | (pipe).

Then, I end up with two lists: one containing the information from exploring the directory, and one with the information from the verification file.

To get the files which changed, I iterate over the first list and for each file, I get the corresponding file in the second list and compare each information one by one, here's the pseudo-code:

ALGORITHM 2: Changes detection

```
foreach file f1 in list1
        foreach file f2 in list2
                if f1.absolutePath == f2.absolutePath
                        String warning = "";
                        if f1.size != f2.size
                                warning += "size"
                        if f1.digestedContent != f2.digestedContent
                                warning += "content";
                        ...
```

```
                        if warning != “”
                                writeInReportFile(warning)
                        list2.remove(f2)
                        f1andf2compared = true
                        break
        if f1andf2comprared == true
                list1.remove(f1)
                continue
```

A string containing each change made to a file/folder is created and written in the report file (in theory, in the code another string is created using all the warnings and the file is only written once).
When two files are the same (same absolute path), they are compared and then removed from their respective list, this allows the program to be a bit faster (it's useless to iterate over files already compared) and gives us two new lists at the end.

The list which previously contained the actual information from files of the monitored directory now contains only the new files/folders since they were no match in the verification file, and the list which previously contained the information from the verification file now contains only the deleted files. Using these two lists, I can write in the report files the new and deleted files/folders.

2.3 Classes

The program is separated in four classes:

- Main is the entry-class, the one that contains the main() method. It only contains one method and has no attributes.
- FileMaster is doing all the operations on the files (exploring directory, verify if file exist, read from files, write report and verification files). All the methods are static since this class has no attributes, no need to instantiate an object.
- FileInfo is a representation of the information gathered on one file/folder. A FileInfo object is printed on each line of the verification file, and so a List of FileInfo is created when reading this file. When exploring the directory, a List of FileInfo is also created.
- ReportInfo is the representation of the information stored in the report file. It contains some statistics, path to important files (monitored directory and verification file) and the List of FileInfo.

3 Usage

First, you need to have Java installed on your machine. You can use either the Java Development Kit (JDK) version 8 from Oracle, or OpenJDK version 10 or above, no external libraries are required.

Then, extract the src folder of the SIV in a directory, for example /home/user.
You now need to compile the Java code, this is done by using the following command in the terminal, from the directory you chose before:

javac src/siv/.java*

You can now run the program with the following command:

java -cp src/ siv.Main <arguments>

You need to specify the correct arguments to run the program (see examples below).
The -h argument shows a help page, use the following command:

java -cp src/ siv.Main -h

To run the initialization mode, you need to use the -i argument, and specify the directory to monitor with -D, the verification file with -V, the report file with -R and the hashing function with -H for example:

java -cp src/ siv.Main -i -D important_directory -V verificationFile -R report.txt -H sha1

This will run the initialization mode, gather information about important_directory, create the verification database in verificationFile and write a report in report.txt. The directory to monitor needs to exist, but if the verification and report files do not exist, they will be created. However, if they already exist, you will be asked if you want to overwrite them.

To run the verification mode, you need to use the -v argument and specify the monitored directory with -D, the verification file with -V and the report file with -R, for example:

java -cp src/ siv.Main -v -D important_directory -V verificationFile -R report2.txt

This will run the verification mode and write in report2.txt if any changes were detected by comparing important_directory and verificationFile.

Note that the paths to the directory to monitor and the files can be given in either absolute or relative path.

4 Limitations

This SIV is quite simple so it still has some limitations. One of them is that we can't know if someone has read a file, someone can have access to your files and only read them, make no modifications at all.

Another limitation is that it's possible to open the verification file and modify it, thus making changes made on files undetectable if done correctly. To prevent that, the entire information about one file should be hashed, not only the content, but then it's not possible to know exactly which attribute of a file has changed.

The maximum depth of the program (when exploring the directory in initialization mode) can also be a limit. If the directory is very deep, then not all the files and subfolders are scanned, so the security is not optimal and the size for some folder cannot be computed (see explanation in the part about initialization). But this can be changed easily, simply change the value of MAX_DEPTH in Main.java and put a higher number if needed, then compile and it will work.

Since the pipe symbol is used a separator in the verification file, the program will fail in verification mode if a file or a folder in the monitored directory has this symbol in its name. Other characters are supported.