**Python File Integrity Monitor Project**

**Intro**

This project is a python3 script that is able to hash files into SHA256 format and monitor files hashed in the SHA256 format for changes. I chose to use SHA256 as it creates longer and more complex hashes than other hashing formats like MD5 and SHA-1, thus storing the data in the file more securely than those other hashes.

The script is able to monitor the integrity of files by comparing the current hash of the file with the has that has been stored by the script before. This ensures integrity of the contents of the files hashed and stored in the directory as the hash of the file is based on the contents of the file, and so would be different if the contents of the file were altered. This is useful for if certain files need to be protected from changes. For example, if a file contained sensitive data, or data critical to a company, a file integrity monitor would detect if a threat actor has tampered with the data in the file and alert the user of the monitor.

1. Plan p.1
2. Code p. 3
3. Conclusion p. 9

**Plan**

The plan for this project is as follows:

1. The script must allow the user to choose between hashing a file or monitoring the files in the folder.
2. If the user chose to hash a file, it hashes the file then writes the hash key (name of the file) and the hash to a csv file.
3. If the user chooses to monitor the files, it will continually compare the saved hashes in the file with the current hashes of the file, so as to detect if the contents of the file have changed.

Before coding this, I’ve broken down how I plan on coding this further.

To hash the file:

1. The script must first create a hash of the file.
2. It will then write the key and hash to a csv file, *hashes.csv*.
3. The script must first check to ensure that the file has not already been hashed and stored in the csv file. If it has, it will alert the user.

To monitor files:

1. The file must first read the keys and corresponding hashes from the *hashes.csv* file.
2. It will then find the keys and re-hash them, to create a hash that can be compared to the saved file hashes.
3. It will then compare the hashes. If the hashes match, then the integrity of the file is kept. If the hashes do not match, it must alert the user that the file contents have been altered. The alert should contain information on which file has been altered and when it was altered.
4. It must also be able to alert the user to any succeeding changes or changes to any file. It cannot break and must continue monitoring.

**Code**

First, I started by creating a function that would create a hash of the contents of a file. To do this I had to import ‘hashlib’, a python library that provides modules to create hashes of files.

A computer screen shot of text

Description automatically generated

This is the whole function. It returns a SHA256 hash of the file passed into the function.

Next, I wrote a function to write the hash to the file. It first takes the file name the user inputted and passes that into the ‘hash\_file’ function I just created.

A black background with colorful text

Description automatically generated

Then, it reads the ‘*hashes.csv’* file to see if the file has already been hashed.

A screen shot of a computer code

Description automatically generated

I stored the csv file as a dictionary, to make reading and writing to and from it with the csv function easier. Consequently, I used the ‘csv.DictReader’ module to read the contents of the csv file as a python dictionary. I then read the file row by row using a ‘for’ loop, and added the dictionaries read from the file to a new list ‘*h*’.

I imported csv at the top of the python file and created a *FIELDNAMES* constant. I then added the fieldnames “key” and “hash” and put the fieldnames at the top of the csv file.

A black background with text and symbols

Description automatically generated

A computer screen with numbers and letters

Description automatically generated

The csv file now contains the fieldnames on the first line, along with the key, followed by its hash.

If the file has been hashed, it tells the user and does not write the hash to the ‘*hashes.csv’* file.

A black screen with colorful text

Description automatically generated

The ‘for’ reads each dictionary from the ‘*h*’ list and gets the value stored under the dictionaries key “key”, comparing it the user’s input which has been passed into the function as ‘*file\_name*’.

If it hasn’t already created a hash of that file, it writes the key (file name) and hash to the ‘*hashes.csv’* file.

A computer code on a black background

Description automatically generated

This code opens the ‘*hashes.csv’* file and stores the contents as a variable named *hashes*. The “a” means to open it and append to it. It then creates a writer using the ‘csv.DictWriter’ module, using the *hashes* variable and *FIELDNAMES* constant. Then it uses the writer to write a new dictionary to the file where the “key” is the file name and the “hash” is the hash we just created.

Now that the plan for hashing the file has been completed, I moved onto creating a function to monitor the files that have been hashed.

Using the plan as guidance, the first thing the function must do is read the keys and hashes stored in the *hashes.csv* file.

A computer code with text

Description automatically generated with medium confidence

This code is similar to the code in the *write\_hash* function that reads from the *hashes.csv* file to check if the key is already hashed and saved in the file.

Unlike that code, we won’t be checking to see if its already saved but we need the keys and hashes saved to the *hashes.csv* file to check if the hash has changed.

Next, I used a while loop to start monitoring the files. The first thing the function needs to do when monitoring the files is create a new hash for the files. This way, it can compare what the current hash is to the hash we have saved. If the hash is the same, the contents of the file have not changed, but if it is different, then the contents have changed. A screen shot of a computer program

Description automatically generated

The above code first tells the user the date and time and that the script has begun monitoring the files. It then starts a while loop to continually check each file.

Then, I created an empty list called *current\_hash*. This is to store the newly made the hashes of the files.

The script then iterates through the stored keys and creates hashes of the files using the *hash\_file* function I made earlier. It then adds a dictionary with the key and newly made hash to the *current\_hash* list.

Finally, the script has to compare the hash it has saved with the newly created hash to check if they match.

A computer screen with text

Description automatically generated

This code iterates through both the list of saved hashes and the newly made hashes. It then finds the keys that match in each list and asks if the hashes match. If they do, it is the end of the *while* loop, and so the code loops and creates and compares new hashes again. This allows for the script to constantly check the file hash and update in real time.

If the hashes don’t match, the script outputs a message to the user, logging the time and file that was altered.

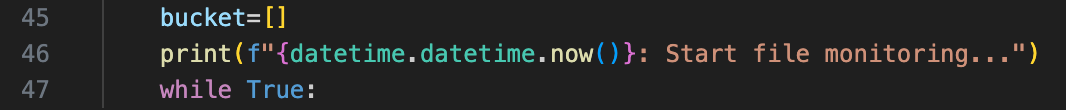
As we can see from this output, the script correctly identifies if the file hashes do not match in real time, outputting a log message for the user to see the contents of the file have been altered.

A screen shot of a computer screen

Description automatically generated

However, as this is in a while loop, and therefore constantly checking for changes, it outputs the message every time the loop checks the file hashes. Ideally, this would not be the case and the message would only be displayed once. This means if the same file is altered again at a later time or date, a new log would be displayed and it would be clear for the user to see the file has been altered more than once.

To remedy this, I created an empty list called ‘bucket’ outside of the while loop. In the ‘bucket’ list, which I would add the changed hashes to. It is important this is outside of the while loop, so that it does not reset to a blank list each time the loop runs.



Then, in the while loop, when it would check to see if the hashes were changed it would also check to see if the hash was in the ‘bucket’ list. If it isn’t already in the ‘bucket’ list, then the script is allowed to output the log message. At the same time, it adds the new hash to the bucket, so as to stop it repeating the log messages.

A computer screen shot of text

Description automatically generated

This made it so the script would not print the output for a file that’s been changed once, more than once. This is the output if I keep changing different files:

A screen shot of a computer

Description automatically generated

As you can see by the difference in time (as noted by the seconds part of the time), the script correctly only outputs a log message when the file has been altered. This makes reading the logs clear and simple, giving the user only the information necessary to know exactly when the files were changed.

Finally, I created the main function that runs the program. This function asks for the user’s choice of hashing a file or monitoring the files and runs the correct function we had made before.

A screenshot of a computer program

Description automatically generated

I embedded the user input into a while loop in the *main()* function, so that the user can hash a file, or multiple files, and immediately start monitoring the files.

The final code is as follows:

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Conclusion**

The script does everything I planned for it to be able to do and works correctly, without errors. It’s able to hash files and write the hashes to a csv file, and it’s able to monitor those files for changes in their hashes, thus ensuring the integrity of the files.

One problem I overcame whilst coding the script was forcing the script to only output the log message once. This is because it was inside a *while* loop, so the script outputted the message whenever the loop ran. I solved this by storing the new hashes in an empty list, so the loop would not run the *if* statement if the hash had already been found. This means the script would only print one log message per time the file was changed, making it easy to see how many times and when the file was changed.

One limitation of the script is that it can only monitor files in the same directory as the script. If I were to remake the project, I would ask the user which directory they want to monitor, as the user can currently hash files from other directories by inputting the full path to the file.