HashLibrarian

4050-581-44

Michael Sweeney

Table of Contents

1. Executive Summary

HashLibrarian is a Unix/Linux based utility written in PERL and meant for assistance in a forensic investigation. It is intended to be run from the investigator's machine, with the disk image to interrogate mounted, preferably as read-only. However, HashLibrarian has been tested on non-write protected drives, and when used in this configuration, it makes no changes to the drive. Please note that HashLibrarian does no hashing, encrypting, or decrypting by itself. These functions are always piped out to GnuPG, `md5sum`, `sha256sum` and `sha1sum`. HashLibrarian does not make copies of keys, record keys or passwords, or attempt to save in locations other than where is specified in the command line arguments.

The ultimate goal of HashLibrarian is to create a database of the MD5, SHA256, and SHA1 hash values for each file that it is specified to find. These databases can be encrypted and protected, and can be loaded later for comparisons against other drives. This feature is particularly useful in the event that a series of forbidden files or contraband needs to be scanned for. A database can be quickly created by hand, but it is suggested that these databases be built in the following manner:

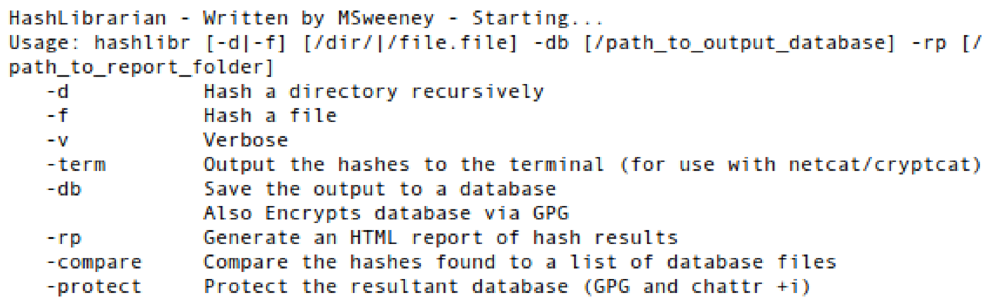
```
Hashlibrarian -d /mnt/case1 -db /home/investigator/case1/hash.db -protect
```

This command scans the specified database recursively, and creates a database at the path following `-db`. Note the `-protect` switch at the end of this command. Put simply, this protects the database from changes, and helps detect a corrupted database later on. More about database protection is explained in section 1.5.

HashLibrarian can be downloaded from OpenSourceForensics.org at this location: http://www2.opensourceforensics.org/node/191.

1.1 Options and Usage

HashLibrarian is a console tool, and is meant to be run as such. Figure 1.1.1 shows a list of command options that is presented when the user calls HashLibrarian with no arguments.



```
HashLibrarian - Written by MSweeney - Starting...
Usage: hashlibr [-d|-f] [/dir/|/file.file] -db [/path_to_output_database] -rp [/
path_to_report_folder]
   -d          Hash a directory recursively
   -f          Hash a file
   -v          Verbose
   -term       Output the hashes to the terminal (for use with netcat/cryptcat)
   -db         Save the output to a database
               Also Encrypts database via GPG
   -rp         Generate an HTML report of hash results
   -compare    Compare the hashes found to a list of database files
   -protect    Protect the resultant database (GPG and chattr +i)
```

Figure 1.1.1. The arguments provided in HashLibrarian.

The first switch mentioned is `-d`, which used as `-d /path/to/folder`. This scans a folder recursively, hashing all of the contents. This command produces no command line output when run with only this option set. It is a way to test HashLibrarian's impact on a system. To verify that this utility does not impact the system, create a few hashes of unimportant data with another program (or `md5sum` or `sha1sum`), run HashLibrarian, and then run the other program again.

The second switch is `-f`, which is used as `-f /path/to/specific/file`. This hashes a specific file and outputs the results of the MD5 and SHA1 operations to the console by default.

The third option is `-v`, which represents a verbose option. This outputs additional information about what HashLibrarian is doing at any given time, along with some hashes and hash information. It is recommended that this option be run with HashLibrarian the first few times an investigator uses it. This should be done so that the method of operation becomes obvious.

The fourth option is `-term`, which is very different from `-v`. This option takes the hash information that is usually piped to a database and adds it to the screen instead. Some additional messages are displayed, but not the same kind of messages are exposed as those displayed when running in verbose mode. To output hash information from HashLibrarian to `netcat`, the following command can be used:

```
Hashlibrarian -d /mnt/case1 -term | nc x.x.x.x 8888
```

This will send a list of the hash information to a destination machine, provided it has a copy of netcat set up to be a listener.

The `-rp` switch is used to generate an HTML report of the hash results, with any discrepancies during a database comparison flagged. This HTML report uses CSS for styling and JavaScript for some small functionality (`<div>` showing and hiding).

To utilize HashLibrarian's ability to compare hash databases, use the `-compare` option, followed by as many paths to databases as desired. Note that more databases (and larger ones) will result in a longer operation time. When using protected files, pass the path of the original, unencrypted database. HashLibrarian will do the rest. It will not read from the original database when performing operations, it will decrypt the protected database and use that instead.

When a protection operation is initiated, HashLibrarian automatically takes advantage of the `chattr` utility to `+i` the output files. This location, write, and rename locks these files. Running HashLibrarian with the `-nochattr` option eliminates this behavior, allowing for easier movement of the database files.

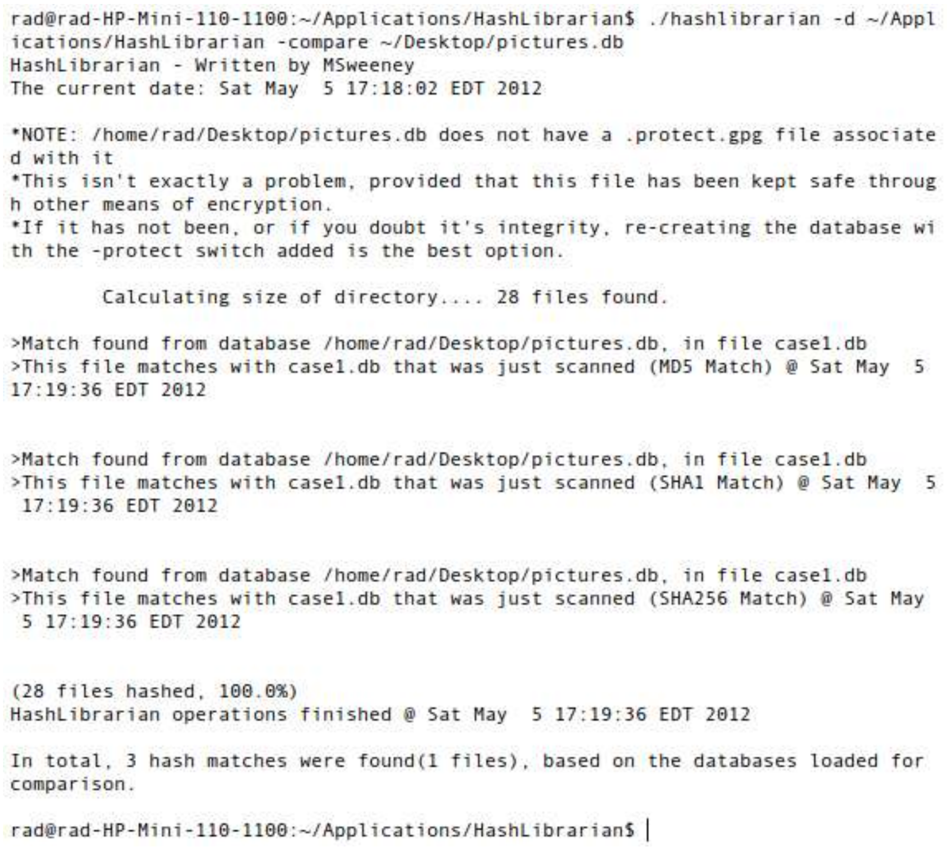
The switch `-protect` is explained in section 1.4 of this document.

1.2 Performing Forensic Tasks with HashLibrarian

When a piece of logical media has been captured, it should be mounted as read only. This advisory is not given as protection against HashLibrarian; this program will only save where specified in the passed arguments. It is simply good forensic practice. Suppose that this disk image is mounted at `/mnt/case1`. Once mounted, the device can then be explored by HashLibrarian. It is suggested that a protected database is made of the entire disk at first. This is accomplished by running `hashlibrarian -d /mnt/case1 -db /home/investigator/case1 -protect`.

This command recursively hashes the disk image at `/mnt/case1`, creates a database of the SHA1, SHA256, and MD5 hashes of every readable file, and then creates an encrypted backup and a protection file with the hashes of both the encrypted an unencrypted files (respectively). See section 1.4 of this document for more information about how HashLibrarian protects its databases. Note that protecting databases is not required, but is good practice.

Now that a protected database has been made that contains the original hashes, the disk image can now be scanned through a list of flagged hashes. A database of flagged hashes can be created by recursively hashing the folder that contains the flagged files, protecting it, and then running the following command against the disk image that needs to be searched: `hashlibrarian –d /mnt/case1 –compare /home/investigator/flagged/flagged.db`. After this command is executed, HashLibrarian will return results live about the status of the scan. When a match is found, a line will be added to the terminal, noting which hashes match. It is easy to note whether or not this is an accidental hash collision (however unlikely), because HashLibrarian will state which hashes match. Figure 1.2.1 depicts a match, as discovered by a similar command.



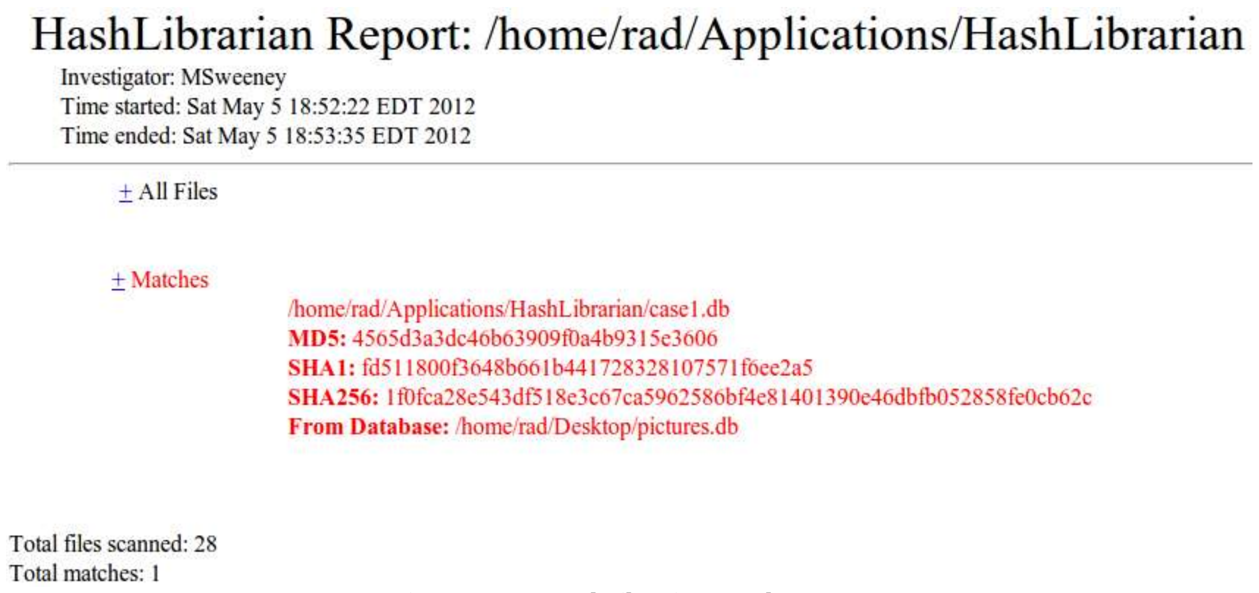
```
rad@rad-HP-Mini-110-1100:~/Applications/HashLibrarian$ ./hashlibrarian -d ~/Appl
ications/HashLibrarian -compare ~/Desktop/pictures.db
HashLibrarian - Written by MSweeney
The current date: Sat May  5 17:18:02 EDT 2012

*NOTE: /home/rad/Desktop/pictures.db does not have a .protect.gpg file associate
d with it
*This isn't exactly a problem, provided that this file has been kept safe throug
h other means of encryption.
*If it has not been, or if you doubt it's integrity, re-creating the database wi
th the -protect switch added is the best option.

        Calculating size of directory.... 28 files found.

>Match found from database /home/rad/Desktop/pictures.db, in file case1.db
>This file matches with case1.db that was just scanned (MD5 Match) @ Sat May  5
17:19:36 EDT 2012


>Match found from database /home/rad/Desktop/pictures.db, in file case1.db
>This file matches with case1.db that was just scanned (SHA1 Match) @ Sat May  5
 17:19:36 EDT 2012


>Match found from database /home/rad/Desktop/pictures.db, in file case1.db
>This file matches with case1.db that was just scanned (SHA256 Match) @ Sat May
 5 17:19:36 EDT 2012


(28 files hashed, 100.0%)
HashLibrarian operations finished @ Sat May  5 17:19:36 EDT 2012

In total, 3 hash matches were found(1 files), based on the databases loaded for
comparison.

rad@rad-HP-Mini-110-1100:~/Applications/HashLibrarian$ |
```

Figure 1.2.1. HashLibrarian returning a match from a database.

Note that any line beginning with a caret (>) is a signal of a match. Also note that each hash returned its own individual result. Instead of adding each, individual hash search hit together, HashLibrarian splits up the results so a more detailed view can be seen. With three separate hashing algorithms cross-examining the file, the search result is absolutely authentic. Note that one file match was found, but three hash matches were returned. More specifically, the one file was returned because a hash match occurred for SHA1, MD5, and SHA256. It is noteworthy that if the number of hash matches is ever not a multiple of three, then a hash collision occurred at some point, and the files should be more closely examined.

1.3 Reports

Finally, once results are found, they can be automatically exported to a HTML report by appending `-rp /path/to/directory` to the end of the command that is used to generate the matches. This report contains a list of when the hashing operations started and ended, the name of the investigator, a list of all files hashed (and their hashes), and all matches (along with the matched file's name and the database the hit occurred in). Figure 1.3.1 shows a screenshot of the report generated by the above investigation.



HashLibrarian Report: /home/rad/Applications/HashLibrarian

Investigator: MSweeney
Time started: Sat May 5 18:52:22 EDT 2012
Time ended: Sat May 5 18:53:35 EDT 2012

± All Files

± Matches

/home/rad/Applications/HashLibrarian/case1.db
MD5: 4565d3a3dc46b63909f0a4b9315e3606
SHA1: fd511800f3648b661b441728328107571f6ee2a5
SHA256: 1f0fca28e543df518e3c67ca5962586bf4e81401390e46dbfb052858fe0cb62c
From Database: /home/rad/Desktop/pictures.db

Total files scanned: 28
Total matches: 1

Figure 1.3.1. A HashLibrarian sample report.

This report is a combination of Cascading Style Sheets (CSS), JavaScript, and HTML. The plus symbols can be clicked to expand a larger view of the subcategory. The name of the investigator is asked for in the command line as the report is being generated, and the times are generated by the system, not HashLibrarian. Of note is the "From Database" line under the match, which specifies which database contained the hash data that matched the file that was just scanned.

It is important to know that when a report is generated in a new directory, HashLibrarian will copy `report.css` and `report.js` out of its own `report/` directory, and put them in the new directory. These will not be overwritten after every execution, and the original documents that are copied can be modified, allowing for customization in reports (the addition of logos, for example).

1.4 Database Protection

When HashLibrarian is given the `-protect` switch, the program prepares itself to encrypt and protect the database file created by the current operation, but also to protect both of these files. See figure 1.4.1 for a graphical explanation of what occurs after the database is created, but before execution completes.

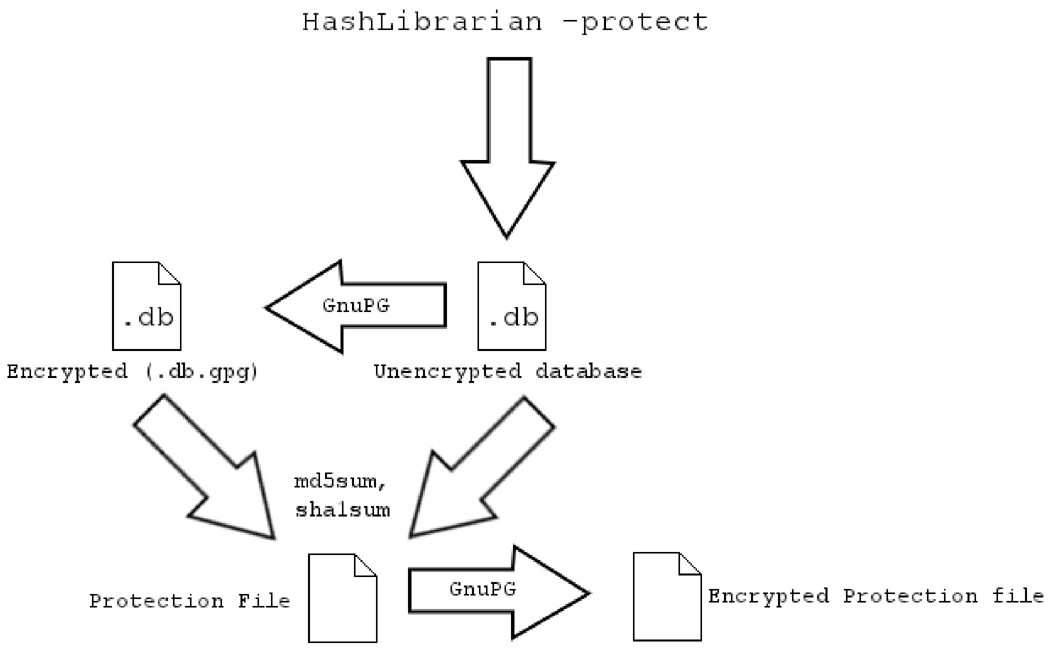


Figure 1.4.1. How HashLibrarian protects its databases.

Note that `sudo chattr +i` is run on each of the files after creation. This is a further protection measure, and can be reversed by running `sudo chattr -i` on the file to remove it. As the diagram shows above, the `-protect` option spawns three extra files. Table 1.4.1 shows these files and their purpose.

Table 1.4.1. The different files created by `-protect`

Filename	Purpose
Database**.db**	**The standard .db file that is created when `-db` is used.**
Database**.db.gpg**	**An encrypted version of the standard .db file. The only file that is read from when a protection file is found.**
Database**.db.protect**	**Contains both MD5 and SHA1 hashes from both database.db and database.db.gpg. Unencrypted.**
Database**.db.protect.gpg**	**Contains the same information as Database.db.protect, but encrypted with GPG. Also known as a protection file.**

When HashLibrarian loads a protected file, it first decrypts the `database.db.protect.gpg` file, prompting the user for their passphrase as needed. It then scans the hashes from the unencrypted protection file, and compares them to the Database.db and Database.db.gpg files located in the same directory. If there is a match, the two files are considered verified, and are loaded into memory. If there is a discrepancy, then HashLibrarian will notify the user and exit.

1.5 Current Condition Overview

HashLibrarian has had a large amount of time put into it over the four-week course of this project. Many, many bugs and glitches were uncovered, sometimes spoiling databases and making them unreadable, and sometimes PERL hashes were populated in reverse, or not at all. Slowly, these kinks were worked out, and HashLibrarian began to shape up. It eventually had to be re-written, but once it had been, everything seemed to work even better than before.

Because of the work put in, HashLibrarian flawlessly reads the command line arguments, validates them, checks for garbage data, and politely informs the user of the correct way to operate the program. Internally, the program is documented reasonably well. Externally, the –v (verbose) switch allows the user to see far more detailed information about the operation of the program.

HashLibrarian should not have a problem functioning in many different ways, and should not return errors. It is of note that `use strict` and `use warnings` are both specified in the code, making PERL be far more aware of errors and issues that may occur. The reports that are generated are simple, and are generated with care.

Paths that are passed are checked for validity, as are the command line options. Also analyzed is whether or not options are missing. For example, a user shouldn't be trying to compare a database to a directory without specifying the directory. A file can't be hashed if no file is mentioned.

In terms of its protected databases, HashLibrarian is very careful. It cryptographically encrypts its databases via GPG, and verifies that the decrypted copies are valid via hashing. If a file is different after decryption, it cannot be used, and HashLibrarian will cease operations. It is notable that HashLibrarian will not take the status of the unencrypted database file into account while decrypting and using a protected database file. The protected database file will always be used.

1.6 The Security Flaw

During my writeup of HashLibrarian, I suddenly realized that there was a large security flaw in the way that HashLibrarian handles its encrypted databases. When HashLibrarian decrypts its database files, it overwrites the unencrypted database file in the directory that the protected database file is in with the unencrypted hash data, and then reads from that file.

The problem with this, I found, is that what if a malcontent removes offending hashes from this database, and then write locks the file with something like `chattr +i`? It turns out that this flaw causes HashLibrarian to detect nothing but whatever the attacker left in the database file. Doing this, I was able to make HashLibrarian not find a match of all but two files in the `/bin` directory on my Linux machine, even though I had used HashLibrarian to create a valid database seconds earlier. Aghast, I made a few changes to the code. I made HashLibrarian check to see if the file is writable before beginning the decryption operation. If it finds that the file is not writable, then it exits. This effectively eliminated the problem.

2. Identifying a Need

Before HashLibrarian was developed, I did some research on what the needs of the forensic community were. There are OpenSSL utilities installed on many Unix/Linux machines right from installation, and these utilities are able to take a wildcard as an argument and hash a directory. These utilities do not do it recursively, and they do not create any kind of record outside of command line output. EnCase provides a way to create a Hash Set and compare it to a directory or disk, but EnCase does not run on Linux or Unix. Even if it did, there is a possibility that these hashes and results would be tied up in an EnCase format, and not something generally readable through a utility such as `cat`.

There is the DigiSec Hash Utility [1], which allows for some hash comparison abilities, but it is a closed source program that only operates on Windows. There is Quick Hash, but this utility does not database its hashes, even though this feature is planned, it does not plan on encrypting these backups [2].

As such, I identified a series of shortcomings that the available applications had: Open-Source, the ability to create databases for easy scanning, the ability to protect this output, the ability to run on Linux/Unix, and the ability to create an easy to share HTML report of the results of any database comparison operation. From this information, I created HashLibrarian.

3. Development

HashLibrarian was written in PERL because of PERL's affinity for file-based operations and parsing output from various sources, along with it's handy hash data structure. This data structure works almost like a small database, taking a key and returning a value. This was immensely useful throughout the program, since a filename could be the key for a hash, and the value would be the MD5 hash, for example. However, one issue with using PERL is that the user's machine must have a PERL interpreter installed. This means that HashLibrarian cannot be executed on a powered-on machine just encountered in the field. The ultimate role of this utility is to be on an investigator's machine, with a disk image mounted of what is to be interrogated.

Many, many issues arose during the development of HashLibrarian, as can be expected from any program's development. Some issues of note were a particular mistake where global variables `@ComparisonDatabasePaths` was accidentally defined locally inside of `sub populateComparisonHashes`, which meant that any debug code written to dump the contents of `@ComparisonDatabasePaths` to the console for checking was always mysteriously blank.

Ultimately, PERL was an excellent choice because of its hash data types. For a period during the development, I considered writing this program in C++ instead. This ultimately did not happen because of not only time constraints, but because PERL simply stood out as the better option. Speed was taken into consideration, but even in PERL, a 1.6ghz netbook with 2 Gigabytes of Random Access Memory (RAM) can run operations fairly quickly, even while a copy of Windows XP was being virtualized in the background in Oracle VirtualBox. Speed was ultimately determined not to be of high concern.

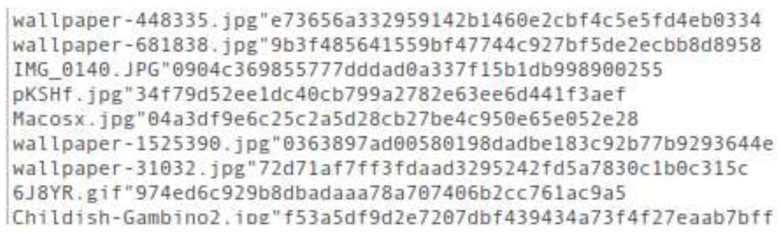
The choice of outsourcing the hashing operations to the OpenSSL utilities was an easy one, since these utilities are well-written, commonly used, and there is no need to reinvent the wheel.

4. Tips for Use

Here are some handy tricks for HashLibrarian that are not obvious features, but are still noteworthy.

1. It is possible to actually dump the contents of the hashing of *only* a single file into a database by using `hashlibrarian –f /path/to/file –db /path/to/database`.

2. Custom databases can be created by making a new file with a .db extension. Inside, add a line such as: `/path/to/file”hash`, with one entry per line. Figure 4.1 shows the contents of a HashLibrarian database. It’s best to add a SHA1, MD5, and SHA256 hash for the files to the database also.



```
wallpaper-448335.jpg"e73656a332959142b1460e2cbf4c5e5fd4eb0334
wallpaper-681838.jpg"9b3f485641559bf47744c927bf5de2ecbb8d8958
IMG_0140.JPG"0904c369855777dddad0a337f15b1db998900255
pKSHf.jpg"34f79d52ee1dc40cb799a2782e63ee6d441f3aef
Macosx.jpg"04a3df9e6c25c2a5d28cb27be4c950e65e052e28
wallpaper-1525390.jpg"0363897ad00580198dadbe183c92b77b9293644e
wallpaper-31032.jpg"72d71af7ff3fdaad3295242fd5a7830c1b0c315c
6J8YR.gif"974ed6c929b8dbadaaa78a707406b2cc761ac9a5
Childish-Gambino2.jpg"f53a5df9d2e7207dbf439434a73f4f27eaab7bff
```

Figure 4.1. The contents of a HashLibrarian database.

3. A great way to use HashLibrarian is to take all files that you’d like detected and add them to a single directory. Create a protected database of this folder in a location that is known for holding databases. Keeping all of the databases in one place makes comparisons far easier, since the path needs to be given every time.

4. HashLibrarian is not particular about what order the command line arguments are given in. It’s possible to start HashLibrarian with the `–compare` operation first, and then follow with a series of locations to scan. However, each option must be followed by a path, if it requires it.

5. Create a GPG Key just for HashLibrarian. It helps if it is named something like “Michael Sweeney (HashLibrarian)”, since if that user has their primary private key compromised, their evidence will still remain secure.

6. HashLibrarian database files are generally not movable. `Chattr +i` is used to protect these files. Since `sudo` is required to remove the protections provided by `chattr`, this adds an additional layer of security. To suppress this behavior, use the `–nochattr` switch when starting HashLibrarian.

5. References

[1] "DigiSec Hashing Utility Free Download." PlanetAlsh.com. http://www.planetalsh.com/download/digesec.php

[2] "Quick Hash." Open Source Digital Forensics. http://www2.opensourceforensics.org/node/156.