**Comparing Files with Hashes**

When hashing large numbers of files in PowerShell, the output can be exported to a comma-delimited (CSV) text file for easy archival and comparison at a later time, for example (this is one command that wraps to multiples lines):

*dir -Path C:\Data -File -Recurse |  
Get-FileHash -Algorithm MD5 |  
Export-Csv -Path C:\Temp\Output.csv*

And with MD5DEEP or SHA256DEEP, the output can be redirected to a TXT file:

*md5deep64.exe -r C:\Data > Output.txt*

But how can we compare two files filled with hundreds of thousands of hashes to detect file system changes? PowerShell does include another cmdlet, Compare-Object, which can be used to compare arrays of any type, but unfortunately it is awkward to use and *slow*. Is there something better which has been optimized just for file hashes?

You can download a script, [Compare-FileHashesList.ps1](http://cyber-defense.sans.org/blog/downloads), to quickly compare two files of hash data to show what files are new, which files have gone missing, and which files have changed since the first hash dump was created. The script is in the public domain and can be downloaded [here](http://cyber-defense.sans.org/blog/downloads) in the SEC505 zip file.

(SEC505 is the SANS six-day course on "[Securing Windows and PowerShell Automation](http://www.sans.org/sec505)". The script is from one of the hands-on labs in the course. There are many other PowerShell scripts in the SEC505 zip file too. This blog has [articles](http://cyber-defense.sans.org/blog/category/powershell) on how to use several of them.)

**Using the Compare-FileHashesList.ps1 Script**

To use the script, first save two hash dump files of the same folder(s) using a command similar to the ones shown above with either Get-FileHash or one of the \*DEEP tools. Choose any hashing algorithm you wish. In real life, you might make a hash dump every night or every weekend in order to track file system changes. To just test the script, though, you can make one hash dump (before.csv), make a copy of it (after.csv), then edit that copy in Notepad (after.csv) in order to have some changes to detect, e.g., delete a few lines, change some file names, and change some of the hexadecimal hash values. This will be good enough to get started.

Once you have your two hash dumps, compare them with the script like this (make sure to save the file with a ".csv" file name extension when using Export-Csv or with a ".txt" extension when using MD5DEEP):

*.\Compare-FileHashesList.ps1 -ReferenceFile .\before.csv -DifferenceFile .\after.csv*

The output will look similar to the following:

*Status Path  
——- ——****Changed****C:\Data\file1.exe****New****C:\Data\doc44.doc****Missing****C:\Data\file3.ps1*

If the status of a file is "Changed", then it's hash is different in the two dump files. If the status is "New", then the file does not exist in the first file (before.csv), but does exist in the second file (after.csv). If it is "Missing", then the file existed in the first (before.csv), but no longer exists in the second (after.csv). (The files do not have to be named "before" and "after" of course, these are just examples.)

By default, files whose hashes have not changed are not output by the script, but they can be shown too: just add the -IncludeUnchanged switch when you run the script. Just like when we originally hashed the target files, so the output of the script can also be saved to a third CSV file using the Export-CSV cmdlet.

If you only need to see a summary of the changes, use the -SummaryOnly switch, like this:

*.\Compare-FileHashesList.ps1 -ReferenceFile .\before.csv -DifferenceFile .\after.csv -SummaryOnly*

The summary does not display each new, missing or changed file separately. The output looks like this:

*StartTime : 8/17/2015 9:18:48 PM  
FinishTime : 8/17/2015 9:18:50 PM  
RunTimeInSeconds : 1.915  
TotalDifferences : 12  
New : 3  
Missing : 4  
Changed : 5*

If you use the -Verbose switch, then the same summary details are displayed, but these details are not actually a part of the output stream of file objects, so they won't mess up your code when you pipe the output of this script into other scripts or export the output to another CSV file.

To see the embedded help in the script, run:

*Get-Help -Full .\Compare-FileHashesList.ps1*

**Performance and Choice of Hashing Algorithm**

MD5 is an obsolete hashing algorithm, but it is much faster than SHA-256. For most file integrity-checking purposes, MD5 is still fine, but you'll have to decide what is more important: performance or reducing the probability of a hash collision. (On my test machine, which has a solid state drive, using MD5 with the Get-FileHash cmdlet will hash data at an average rate of 408MB/sec, while SHA-256 only does 103MB/sec.)

The performance of the Compare-FileHashesList.ps1 script is not affected by the choice of hashing algorithm. Performance is mainly dependent on the number of lines in the two files. (On my test machine, which has an Intel Core i7-4790 at 3.60GHz, two large CSV files can be compared at 77,000 entries/second with PowerShell 5.0 on Windows 10.) Also, using the -IncludeUnchanged switch imposes a performance penalty too, so avoid this switch when you don't need this output.

You can choose any tool for producing the hashes. You can use Get-FileHash, MD5DEEP, or any other tool that outputs in the same format as MD5DEEP. If you have a choice, save your hashes to a file using UTF-8 encoding to reduce the size of the file.

Try to use PowerShell 3.0 or later to benefit from in-memory compilation of some of the code in the script with the [DLR](http://dotnet.dzone.com/articles/powershell-3-finally-dlr). PowerShell 5.0 on Windows 10 gives the best performance so far, probably because of under-the-hood DLR improvements.

Doing the comparison of these hash lists in PowerShell is convenient because the output of the script is an array of objects (not raw text that requires parsing), scripts can be easily edited and customized (unlike binaries), and PowerShell scripts are easier to use with PowerShell remoting when you want to hash files on large numbers of systems over the network.