

FILE INTEGRITY MONITOR

By

V. PRANEETH REDDY

UG Scholar, CSE Department, SNIST, Hyderabad, Telangana, India

(Mr. V. Prabhakar, Assistant Professor, Dr. Aruna Varanasi, Professor, & HOD
Department of Computer Science and Engineering, SNIST, Hyderabad, Telangana, India)



REENIDHI
EDUCATIONAL GROUP

REENIDHI
INSTITUTE OF
SCIENCE AND
TECHNOLOGY



ABSTRACT:

A File Integrity Monitor is a process or internal control that checks the validity of software files in an operating system or application. This is achieved by comparing the current state of the file with a known, secure baseline. Typically, a cryptographic checksum of the original file baseline is calculated and compared to the current file state to ensure that the file has not been altered improperly. Other file attributes can also be used to monitor the integrity of the file. FIM is a valuable tool for organizations to identify any unauthorized changes made to important files, which can prevent data theft or loss, and avoid the costs associated

with lost productivity, revenue, and reputation, as well as legal and compliance penalties. Additionally, compliance penalties resulting from failing to detect improper changes to critical files on their systems can be significant for organizations. These penalties can range from fines to legal action and can significantly damage a company's reputation. Implementing a file integrity monitor (FIM) can help organizations avoid such penalties by proactively monitoring changes to files and alerting system administrators when unauthorized changes are detected. With FIM in place, organizations can ensure compliance with regulations and standards, prevent data

breaches and cyber-attacks, and avoid the costly consequences of non-compliance.

INTRODUCTION

File Integrity Monitoring (FIM) has become an increasingly important aspect of cybersecurity in recent years. In today's digital landscape, cybercriminals use a wide range of sophisticated methods to bypass perimeter security defences and gain unauthorized access to an organization's network. Once inside, they can manipulate and change critical files in order to steal sensitive data, cause system failures, or execute other malicious actions. To combat these threats, many organizations have turned to FIM tools to continuously monitor their file systems for any unauthorized changes or modifications. These tools use a variety of techniques, including cryptographic hash functions, digital signatures, and other methods to detect file alterations and ensure the integrity of critical files. One of the key benefits of FIM is its ability to prevent data loss and minimize the risk of data breaches. By continuously monitoring file systems for any unauthorized changes, FIM solutions can help organizations quickly detect and respond to potential threats, reducing the risk of sensitive data being compromised.

In addition to its security benefits, FIM can also help organizations comply with industry and regulatory standards that require strict data security and integrity measures. For example, the Health Insurance Portability and Accountability Act (HIPAA), the Payment Card Industry Data Security Standard (PCI DSS), and the Sarbanes-Oxley Act (SOX) all have specific requirements related to data security and integrity, which can be met using FIM tools.

File Integrity Monitor software is an internal process that performs the act of validating the integrity of operating system and application software files using a verification method between the current file state and a known, good baseline. Organizations can use FIM to prevent data loss due to accidental or intentional deletion or modification of critical files. FIM can alert about such activities. FIM is a critical tool for organizations looking to maintain the integrity of their file systems and protect against the growing threat of cyberattacks. By continuously monitoring file systems for unauthorized changes and ensuring the integrity of critical files, FIM solutions can help organizations prevent data loss, minimize the risk of data breaches, and

comply with industry and regulatory standards.

Existing System

In the past, many organizations relied on manual methods of monitoring file integrity. This could involve periodic manual checks of files and directories to ensure that they had not been tampered with. However, this approach is time-consuming, labor-intensive, and can be prone to human error. Logs were used before FIM. Logs record various events in a system, such as successful or failed logins, access to files, changes made to the system configuration, and more. These logs can then be analyzed to detect security incidents, troubleshoot problems, and generate reports for compliance and auditing purposes.

Proposed System

FIM systems have emerged as a solution to the limitations of traditional security measures and manual monitoring. By continuously monitoring file integrity in real-time, FIM systems can quickly detect any unauthorized changes to files and directories, and alert security personnel to act. This proactive approach helps organizations to quickly respond to potential security breaches and minimize

the impact of cyberattacks. FIM, on the other hand, can monitor all files and directories on a system, including those that are not logged. Additionally, FIM can detect changes in real-time and trigger alerts or actions, which can help organizations respond quickly to potential security incidents

SYSTEM

ARCHITECTURE

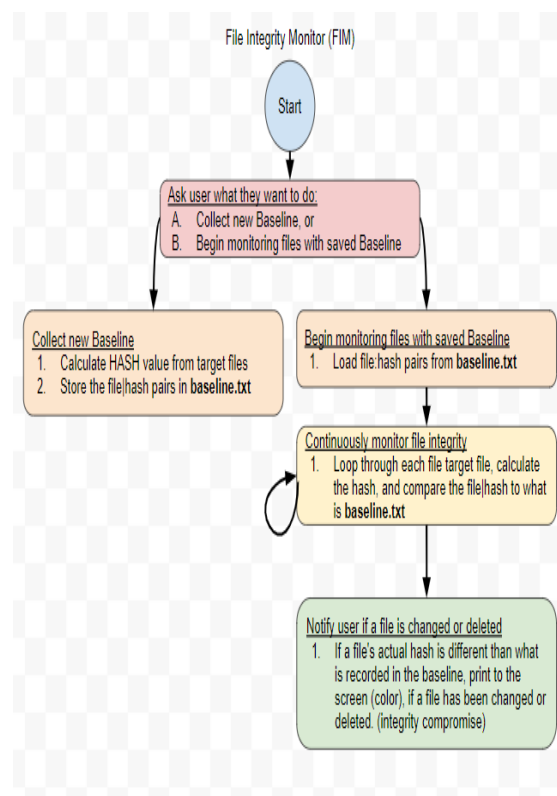


Fig 1: Flow of the code

The above diagram is about the flow of the code. To build this software we need a

PowerShell ISE to run the script at first the end-user gets the option to choose either to collect a new baseline or to begin monitoring files with saved baseline. When creating new baseline, the path of the files and their respective hashes are appended to the document baseline.txt and to avoid the confusion with the new baseline and previous baselines, every time a user creates a new baseline the previous one gets cleared.

Once the baseline is created the live monitoring starts by comparing the hash values that are created previously which are stored in baseline.txt. The code loops through each file target file and calculates the hash values. If a file's actual hash is different than what is recorded in the baseline then an alert will be send to the user by specifying the file path that has got changed along with the information whether the file contents are changed or file is created or destroyed.

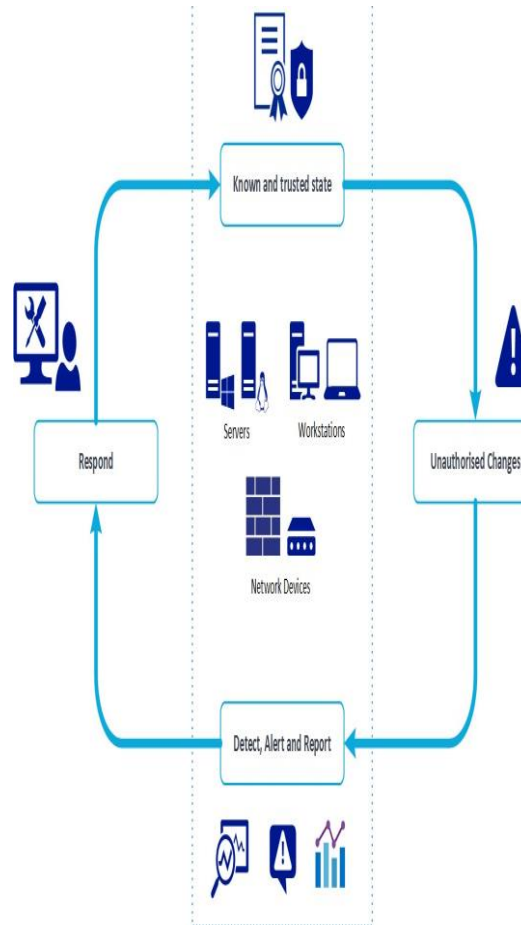


Fig 2: FIM architecture overview

ENCRYPTION ALGORITHMS

PowerShell provides various internal encryption algorithms that can be used for securing data. To view these encryption algorithms in PowerShell, we can use the "Get-CipherSupportedProvider" command. This command lists the available cryptographic service providers (CSPs) and the encryption algorithms that they support. The number of encryption algorithms provided depends on the CSPs installed on the system.

Some of the commonly used encryption algorithms in PowerShell include AES (Advanced Encryption Standard), DES (Data Encryption Standard), TripleDES (Triple Data Encryption Standard), RC2 (Rivest Cipher 2), and RC4 (Rivest Cipher 4).

AES is a symmetric encryption algorithm used for securing sensitive information. It uses a block cipher with a key size of 128, 192, or 256 bits. DES and TripleDES are symmetric block ciphers that use a key size of 56 bits. They are less secure than AES and are being phased out in favour of more secure algorithms.

RC2 is a symmetric key block cipher that uses variable length keys, with a key length of up to 2048 bits. It was widely used in the past, but it has been deprecated due to security concerns. RC4 is also a symmetric key stream cipher that is commonly used in encryption protocols such as SSL and WEP. However, it has also been deprecated due to security concerns.

PowerShell provides SHA512 as one of the encryption algorithms that can be used to generate a hash value for a file. In the code I constructed, the Get-FileHash cmdlet is

used to calculate the hash value for each file using the SHA512 algorithm. The resulting hash value is then compared to the hash value stored in the baseline.txt file to detect any changes to the file.

SHA512 is a hash function that generates a fixed-size output (512 bits) from any input data of arbitrary size. It is a member of the SHA-2 family of hash functions, which also includes SHA-256, SHA-384, and SHA-224. SHA-512 is more secure than SHA-256 because it has a larger output size, making it more resistant to collisions and brute-force attacks.

POWERSHELL

PowerShell ISE is a tool for automation of repetitive tasks. This can include tasks such as file backups, system checks, and software installations. PowerShell also provides the ability to create and manage Windows services, configure network settings, and manage user accounts. It provides a File System Watcher class in PowerShell to monitor the target folder for any new files. This class provides various events that you can subscribe to, such as Created, Changed, Deleted, etc. You can write a PowerShell script that runs in the background and subscribes to the Created

event of the File System Watcher class. Whenever a new file is created in the target folder, the script will be triggered, and you can start processing the new file.

The PowerShell script which I constructed helps users to monitor changes to files in a target folder. The script does this by calculating a hash of each file in the target folder and comparing it to a saved "baseline" hash value for each file. If a file's hash value has changed, the script will notify the user that the file has been modified.

The script is designed to be easy to use and understand, with clear prompts for the user to select either to collect a new baseline hash or begin monitoring files with a saved baseline hash. The script also makes use of PowerShell functions to make the code more modular and reusable.

The script makes use of several built-in PowerShell commands and functions to perform its tasks. For example, the `Get-ChildItem` cmdlet is used to retrieve all files in the target folder and its subfolders. The `Get-FileHash` cmdlet is used to calculate the hash value of each file, using the SHA512 algorithm.

In addition to these built-in cmdlets, the script defines its own functions to perform

certain tasks. The `Calculate-File-Hash` function calculates the hash value of a file, while the `Erase-Baseline-If-Already-Exists` function deletes any previously saved baseline hash file, so that a new one can be created.

The script also makes use of various PowerShell constructs, such as `foreach` loops, conditional statements (`if/elseif`), and dictionaries, to perform its tasks. The script uses a dictionary to store the baseline hash values for each file, so that they can be compared to the current hash values when monitoring the files for changes.

PROGRAM

FUNCTIONALITY

The script which I have created consists of several functions and logic for monitoring changes to files in a specified directory. The `Calculate-File-Hash` function calculates the SHA512 hash of a file at a specified path. This function is used to generate a hash value for each file in the monitored directory. This hash value is then used to compare against the baseline hash value to determine if a file has been modified. The `Get-FileHash` cmdlet is used within the function to generate the hash value.

The 'Erase-Baseline-If-Already-Exists' function checks if the baseline file exists and deletes it if it does. This function is called when the user selects option A to collect a new baseline. This ensures that the baseline is always up to date with the latest hash values for each file in the monitored directory.

The main part of the script begins with a prompt to the user to select an option, either to collect a new baseline or to begin monitoring files with the saved baseline. The user is then prompted to enter the path to the directory to be monitored.

If the user selects option A, the Erase-Baseline-If-Already-Exists function is called to delete the existing baseline if it exists. The Get-ChildItem cmdlet is used to recursively collect all files in the specified directory. For each file, the Calculate-File-Hash function is called to generate the SHA512 hash value. The file path and hash value are then written to the baseline file using the Out-File cmdlet with the -Append parameter. This creates a baseline file with the hash value for each file in the monitored directory.

If the user selects option B, the baseline file is loaded into a dictionary using the Get-Content cmdlet. The Test-Path cmdlet is

used to check if each file in the baseline exists in the monitored directory. If a file in the baseline has been deleted, the user is notified.

The Calculate-File-Hash function is called again for each file in the monitored directory. If the hash value for a file matches the baseline hash value, it is assumed that the file has not been modified. If the hash values do not match, it is assumed that the file has been modified and the user is notified. If a new file is created in the monitored directory, the user is also notified.

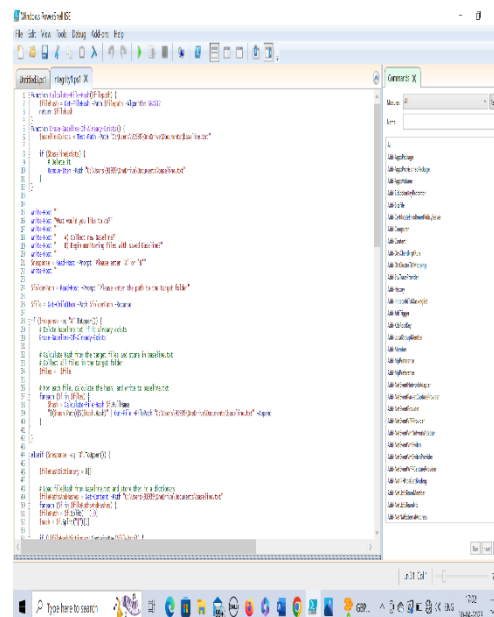


Fig 3: The FIM script

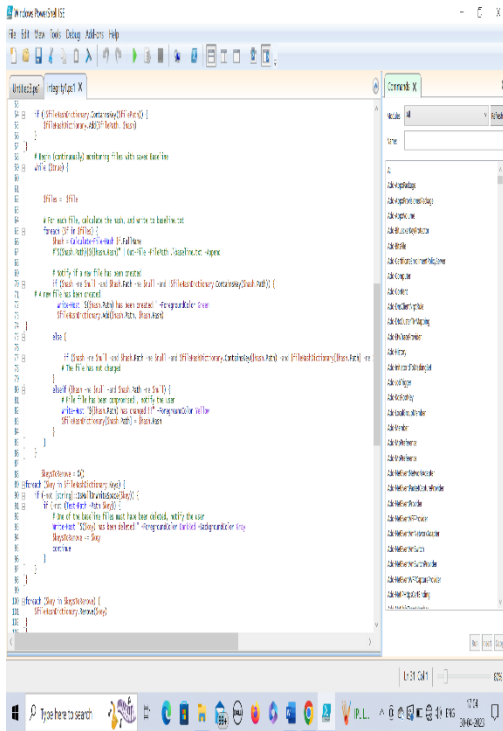


Fig 4: The FIM script Real-time monitoring

PowerShell supports FIM code through its ability to access the .NET Framework, which provides a range of classes and methods for working with files, directories, and cryptographic algorithms. In the FIM code, the System.IO namespace is used to read, write, and check the integrity of files, while the System.Security.Cryptography namespace is used to generate the SHA512 hash of the file contents. In addition, PowerShell allows for the creation of functions and modules, which can be used to encapsulate and reuse FIM code across different scripts and systems. This can

make it easier to manage and maintain file integrity monitoring tasks across an organization. Overall, PowerShell's support for the .NET Framework and its ability to create reusable functions and modules make it a powerful tool for performing FIM tasks on Windows-based systems.



Fig 5: Baseline.txt file

In the PowerShell script provided, baseline.txt is a file that is used to store the baseline or initial file state of the target directory. This file is stored in the user's OneDrive Documents folder at the following location:
C:\Users\91939\OneDrive\Documents

The baseline.txt file is used in the second option of the PowerShell script, which is to continuously monitor files in the target directory for any changes. The script loads the file paths and their hashes from baseline.txt and stores them in a dictionary. Then, the script calculates the hashes of the files in the target directory, compares them to the hashes in the dictionary, and notifies the user if a file has been created, deleted, or modified.

By using baseline.txt, the script can keep track of the initial state of the target directory and monitor any changes made to it. Without baseline.txt, the script would not have a reference point to compare the current state of the target directory to, making it difficult to determine if any changes have been made.

Therefore, baseline.txt serves as a reference point or baseline for the script to compare the current state of the target directory to, allowing it to detect any changes made to the files in the directory.

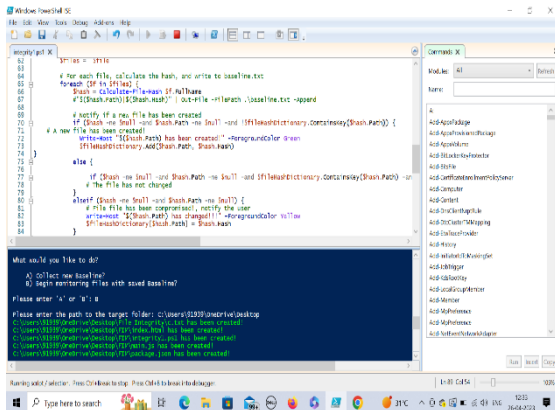
RESULTS

The provided PowerShell script is designed to collect a baseline of file hashes for a target folder and its subfolders, then monitor those files for any changes or new

additions. The script first prompts the user to choose between collecting a new baseline or monitoring files using the existing baseline.

If the user chooses to collect a new baseline, the script calculates the hash of each file in the target folder and writes it to a text file named "baseline.txt" located in the user's "Documents" folder. If the user chooses to monitor files using the existing baseline, the script reads the file paths and their associated hashes from "baseline.txt" into a dictionary. The script then continuously monitors the target folder and its subfolders for changes or new additions, comparing the file hashes to those in the dictionary to detect any modifications or new files.

Overall, this script is a useful tool for monitoring changes in a folder and detecting any unauthorized modifications to files. By comparing the current hash values with the previously saved values, it can quickly detect any changes and notify the user in real-time.

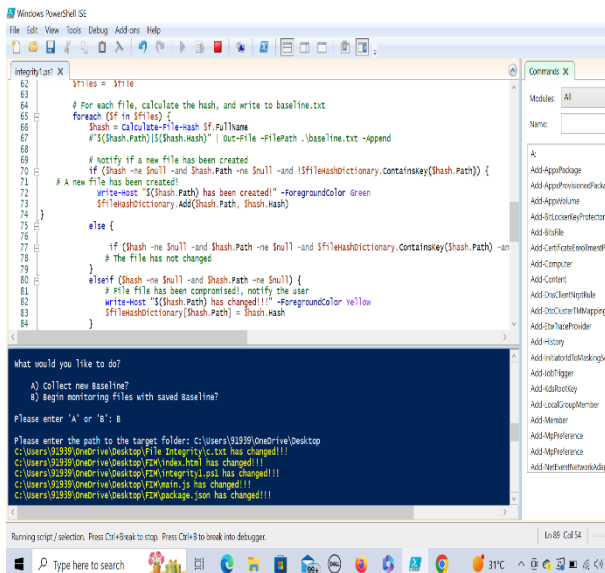


```

$files = $files
# For each file, calculate the hash, and write to baseline.txt
foreach ($f in $files) {
    $hash = Calculate-File-Hash $f.FullName
    # $($hash.Path) | Out-File -Filepath .\baseline.txt -append
    # Notify if a new file has been created
    if ($hash -ne $null -and $hash.Path -ne $null -and $FileHashDictionary.ContainsKey($hash.Path)) {
        # A new file has been created
        write-host "$($hash.Path) has been created!!" -ForegroundColor Green
        $FileHashDictionary.Add($hash.Path, $hash.Hash)
    }
    else {
        if ($hash -ne $null -and $hash.Path -ne $null -and $FileHashDictionary.ContainsKey($hash.Path)) -or
        # The file has not changed
        }
        elseif ($hash -ne $null -and $hash.Path -ne $null) {
            # File file has been compromised, notify the user
            write-host "$($hash.Path) has changed!!!" -ForegroundColor Yellow
            $FileHashDictionary[$hash.Path] = $hash.Hash
        }
    }
}

```

Fig 5: The user gets notified when a new file is created

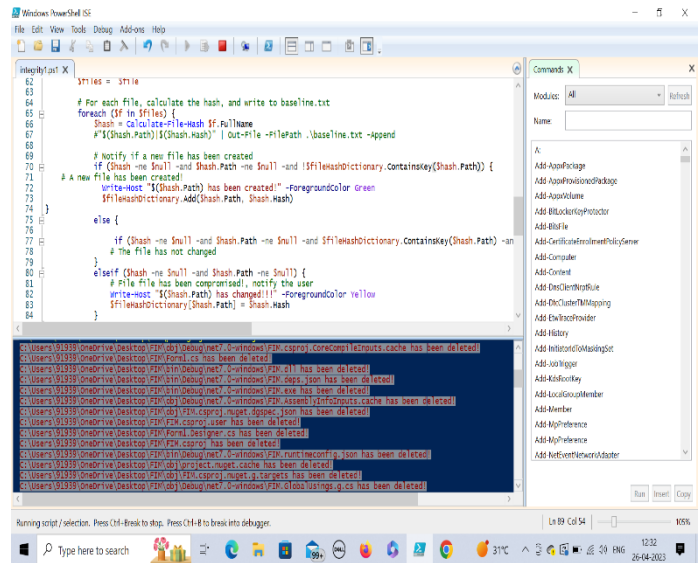


```

$files = $files
# For each file, calculate the hash, and write to baseline.txt
foreach ($f in $files) {
    $hash = Calculate-File-Hash $f.FullName
    # $($hash.Path) | Out-File -Filepath .\baseline.txt -append
    # Notify if a new file has been created
    if ($hash -ne $null -and $hash.Path -ne $null -and $FileHashDictionary.ContainsKey($hash.Path)) {
        # A new file has been created
        write-host "$($hash.Path) has been created!!" -ForegroundColor Green
        $FileHashDictionary.Add($hash.Path, $hash.Hash)
    }
    else {
        if ($hash -ne $null -and $hash.Path -ne $null -and $FileHashDictionary.ContainsKey($hash.Path)) -or
        # The file has not changed
        }
        elseif ($hash -ne $null -and $hash.Path -ne $null) {
            # File file has been compromised, notify the user
            write-host "$($hash.Path) has changed!!!" -ForegroundColor Yellow
            $FileHashDictionary[$hash.Path] = $hash.Hash
        }
    }
}

```

Fig 6: The user gets notified when a file content gets changed



```

$files = $files
# For each file, calculate the hash, and write to baseline.txt
foreach ($f in $files) {
    $hash = Calculate-File-Hash $f.FullName
    # $($hash.Path) | Out-File -Filepath .\baseline.txt -append
    # Notify if a new file has been created
    if ($hash -ne $null -and $hash.Path -ne $null -and $FileHashDictionary.ContainsKey($hash.Path)) {
        # A new file has been created
        write-host "$($hash.Path) has been created!!" -ForegroundColor Green
        $FileHashDictionary.Add($hash.Path, $hash.Hash)
    }
    else {
        if ($hash -ne $null -and $hash.Path -ne $null -and $FileHashDictionary.ContainsKey($hash.Path)) -or
        # The file has not changed
        }
        elseif ($hash -ne $null -and $hash.Path -ne $null) {
            # File file has been compromised, notify the user
            write-host "$($hash.Path) has changed!!!" -ForegroundColor Yellow
            $FileHashDictionary[$hash.Path] = $hash.Hash
        }
    }
}

```

Fig 7: The user gets notified when a file gets deleted

MERITS

There are many merits when it comes to this project such as providing a simple and efficient way to monitor changes in a target folder and notifying the user when changes occur. The script can be used to detect unauthorized modifications to critical files, which can be particularly useful in security-sensitive environments. Additionally, the script can be easily modified to fit specific monitoring requirements, such as adding or removing file types, changing the notification mechanism, or modifying the hashing algorithm.

Another merit of this project is that it leverages the built-in features of PowerShell, which is a powerful and versatile scripting language that is widely

used in Windows environments. The script utilizes several PowerShell cmdlets, such as Get-ChildItem, Get-FileHash, Test-Path, and Out-File, which can simplify the implementation of monitoring functionality. Additionally, the script is designed to be run continuously, which means that it can be used to monitor changes over an extended period without requiring manual intervention. Overall, this project provides a useful and customizable solution for monitoring changes in a target folder using PowerShell.

LIMITATIONS

One major limitation of this project is that it relies on the accuracy of the baseline file. If the baseline file is not accurate or has been tampered with, then the monitoring process will not work as intended. Additionally, if the baseline file is lost or deleted, the monitoring process will need to be restarted from scratch.

Another limitation is that this script only monitors the file system and does not include monitoring of network-based file shares or cloud-based storage. This means that any changes made to files in these locations will not be detected or monitored by this script. Finally, this script does not

provide any method for automatically restoring files that have been compromised, which could be a significant limitation in the event of a security breach.

SOFTWARE AND HARDWARE REQUIREMENTS

The software requirements for this project include Windows PowerShell 5.1 or later and .NET Framework 4.7 or later. These are typically included by default on modern versions of Windows, but may need to be installed separately on some older versions.

As for hardware requirements, this project can run on any computer capable of running the software requirements listed above. However, the performance may be affected by factors such as the speed of the processor and the amount of available RAM. Additionally, the amount of storage required will depend on the number and size of the files being monitored, as well as the frequency of monitoring.

CONCLUSION

The implementation of file integrity monitoring (FIM) using PowerShell demonstrated in this project provides a solid foundation for detecting unauthorized changes to critical files and directories. As cyber threats continue to evolve, FIM plays an increasingly important role in detecting and preventing security breaches. This project showcases the value of using PowerShell to automate FIM processes, which can significantly reduce the workload for security professionals. While this script provides a basic FIM solution, organizations must consider their specific security requirements to implement an effective FIM system. Best practices should be followed to ensure the integrity of the system and prevent false positives or negatives. Additionally, FIM should be part of a comprehensive security strategy that includes other security controls such as access controls, threat intelligence, and incident response. Overall, this project demonstrates the importance of FIM in ensuring the security of critical data and the value of automation in streamlining security operations.