Encoding vs Hashing vs Encryption

Encoding, hashing, and encryption are three distinct concepts used in computer science and cryptography, each serving different purposes and with different properties.

Let's review each of them:

1. Encoding:

Encoding is the process of converting data from one format to another format that is suitable for transmission, storage, or processing.

- The objective of encoding is to ensure that the data is correctly interpreted by the receiving system.
- **Encoding does not provide any form of security**; it is merely a representation transformation.
    1. https://simple.wikipedia.org/wiki/Encoding_method
    2. https://en.wikipedia.org/wiki/Code
    3. https://en.wikipedia.org/wiki/Character_encoding

Common examples of encoding include:

- ASCII (American Standard Code for Information Interchange): Encoding text characters into a numerical representation.
- Base64: Encoding binary data into a format that can be safely transmitted over text-based protocols, such as email or URLs.
- UTF-8: A variable-length character encoding system that supports a wide range of characters for different languages.

Key points:

- **Encoding does not provide any form of security**.
- It's primarily used for ensuring **compatibility** and **readability** of data across different systems or protocols.

2. Hashing:

Hashing is a **one-way function** that takes an input (or 'message') and returns a fixed-size string of bytes, typically a hexadecimal or binary representation. The output of a hash function is known as a hash value or hash code. Hashing is commonly used for data integrity verification and indexing.

Properties of hash functions:

- Deterministic: For the same input, a hash function always produces the same output.
- Fast computation: Hash functions are designed to be computationally efficient.
- Avalanche effect: A small change in the input should result in a significantly different hash output.
- Pre-image resistance: It **should be computationally infeasible** to reverse the hash function and obtain the original input.
- Collision resistance: It should be difficult to find two different inputs that produce the same hash output. **The larger the bits, the more difficult it is to find inputs that produce the same output.**

Common hashing algorithms include:

- MD5 (Message Digest Algorithm 5)
- SHA-1 (Secure Hash Algorithm 1)
- SHA-256 (Secure Hash Algorithm 256-bit)

Key points:

- Hashing is irreversible; you cannot obtain the original input from the hash output.
- It is commonly used for data integrity verification, password storage, and cryptographic protocols.

The output sizes of MD5, SHA-1, and SHA-256 are typically represented in hexadecimal notation. Each hexadecimal digit represents 4 bits of the output. So, to determine the number of characters required to represent the output of each hash function, we divide the output size (in bits) by 4 (since each hexadecimal digit represents 4 bits).

- **MD5 (Message Digest Algorithm 5)**:
    - Output size: 128 bits
    - Number of hexadecimal characters: 128 bits / 4 = 32 characters
- **SHA-1 (Secure Hash Algorithm 1)**:
    - Output size: 160 bits
    - Number of hexadecimal characters: 160 bits / 4 = 40 characters
- **SHA-256 (Secure Hash Algorithm 256-bit)**: (subset of SHA-2)
    - Output size: 256 bits
    - Number of hexadecimal characters: 256 bits / 4 = 64 characters

---

Future Proofing:

https://en.wikipedia.org/wiki/SHA-3

SHA-3, or Secure Hash Algorithm 3, is a cryptographic hash function and a member of the SHA-3 family of hash functions. It was designed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, and selected as the winner of the NIST hash function competition in 2012. SHA-3 was developed as an alternative to the SHA-2 family of hash functions, providing improved security and performance.

Here are some key points about SHA-3:

1. **Algorithm**: SHA-3 operates on input data to produce a fixed-size hash value, typically represented as a string of hexadecimal characters. It uses the Keccak permutation-based sponge construction, which differs from the Merkle-Damgård construction used by SHA-1 and SHA-2. The sponge construction provides resistance against various cryptographic attacks, including collision attacks.
2. **Security**: SHA-3 offers a high level of security against collision attacks, pre-image attacks, and second pre-image attacks. Its security is based on the properties of the underlying sponge construction and the cryptographic properties of the Keccak permutation.
3. **Output Size**: SHA-3 supports several output sizes, including 224, 256, 384, and 512 bits. The choice of output size depends on the specific application's requirements for security and efficiency.
4. **Usage**: SHA-3 is used in various cryptographic applications, including digital signatures, message authentication codes (MACs), key derivation functions (KDFs), and blockchain technologies. It's also integrated into cryptographic protocols such as TLS (Transport Layer Security) and SSH (Secure Shell).

5. **Performance**: While SHA-3 offers strong security guarantees, its performance characteristics may vary depending on the chosen output size and implementation. Generally, SHA-3 is designed to be efficient and suitable for use in both software and hardware implementations.
6. **Standardization**: SHA-3 was standardized by the National Institute of Standards and Technology (NIST) in FIPS PUB 202, published in August 2015. It has since become widely adopted in various security-critical applications and protocols.

Overall, SHA-3 represents a significant advancement in cryptographic hash function design, providing improved security properties and performance compared to its predecessors, particularly in the face of emerging cryptanalytic techniques.

3. Encryption:

Encryption is the process of converting plaintext (original data) into ciphertext (encrypted data) using an algorithm and a secret key. The primary purpose of encryption is to ensure confidentiality and privacy of data, making it unreadable to unauthorized users. Encryption schemes typically involve two main operations: encryption (converting plaintext to ciphertext) and decryption (converting ciphertext back to plaintext) typically via a cipher.

Types of encryption:

- Symmetric encryption: The same key is used for both encryption and decryption. Examples include AES (Advanced Encryption Standard) and DES (Data Encryption Standard).
- Asymmetric encryption (public-key encryption): Two different keys are used, one for encryption and one for decryption. Examples include RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography).

Key points:

- Encryption provides confidentiality by making data unreadable without the decryption key.
- It's commonly used to protect sensitive information during transmission or storage, such as credit card details, login credentials, and private communications.

In summary, encoding is about representing data in a different format, hashing is about generating a fixed-size unique representation of data, and encryption is about transforming data into an unreadable form for confidentiality. Each serves its own purpose in the realm of computer science and cryptography.

Linux Commands Review:

echo

The `echo` command in Linux is a simple utility used to display text or variables on the standard output (usually the terminal). It's commonly used in shell scripts, command-line environments, and system administration tasks.

Displaying Text: The most common use of `echo` is to display text on the terminal.

echo "Hello, world!" would result in the terminal printing Hello,world!

cat

The `cat` command in Linux is short for "concatenate." It's primarily used for displaying the contents of files, concatenating files, creating new files, and combining the content of files.

Displaying File Contents: The most common usage of `cat` is to display the contents of one or more files to the terminal.

cat filename   This would output the contents of the file to the screen.

vim

The `vim` command is a powerful text editor commonly used in Unix-like operating systems. Here's a brief overview of its functionality:

1. **Editing Text**: Vim allows users to create, edit, and modify text files. It provides a wide range of features for navigating, searching, and manipulating text efficiently.
2. **Modes**: Vim operates in different modes:
   - **Normal mode**: This is the default mode for navigating and executing commands.
   - **Insert mode**: In this mode, users can directly input and edit text.
   - **Visual mode**: Used for selecting and manipulating blocks of text visually

Vim is a versatile and efficient text editor suitable for both casual editing tasks and complex programming work. While it has a steep learning curve initially, mastering Vim can greatly improve productivity and efficiency in text editing workflows.

md5sum

The `md5sum` command in Linux is used to calculate and display the MD5 checksum of one or more files. The MD5 checksum is a 128-bit cryptographic hash function that produces a fixed-size hash value (usually represented as a 32-character hexadecimal number) from the input data. This hash value is unique to the input data, and even a small change in the input data will result in a significantly different hash value.

The basic syntax of the `md5sum` command is:

```
md5sum [options] [file(s)]
```

For example, to calculate the MD5 checksum of a file named `example.txt`, you would use:

```
md5sum example.txt
```

This would produce output similar to:

```
1a79a4d60de6718e8e5b326e338ae533  example.txt
```

The first part of the output (`1a79a4d60de6718e8e5b326e338ae533`) represents the MD5 checksum of the `example.txt` file, while the second part (`example.txt`) is the name of the file.

You can use the `md5sum` command to verify the integrity of files. For example, after transferring a file over the internet, you can calculate its MD5 checksum on both the sending and receiving ends. If the MD5 checksums match, it indicates that the file has been successfully transferred without any alterations. If they don't match, it suggests that the file may have been corrupted during transmission.

LINUX REDIRECTION TOPICS

Piping (Redirecting out put of a command to another command)

In Linux, a pipe (`|`) is a powerful feature that allows you to redirect the output of one command as input to another command. This facilitates chaining multiple commands together, enabling complex operations and data processing.

Here's how a pipe is used to redirect output within Linux:

1. **Basic Syntax**: The basic syntax of a pipe is straightforward. You simply use the `|` character to connect the output of one command to the input of another command. For example:

```
command1 | command2
```

This takes the output produced by `command1` and passes it as input to `command2`.

Pipes are a fundamental concept in Unix-like operating systems, providing a concise and powerful mechanism for connecting and orchestrating command-line utilities to perform complex data processing tasks efficiently.

In Linux, redirection to a file is a powerful feature that allows you to redirect the output of a command or program to a file instead of displaying it on the terminal. It's commonly used for saving command output, logging, and creating or appending to files.

Output Redirection (`>`):

- The `>` operator is used to redirect the standard output (stdout) of a command to a file.
- If the file does not exist, it will be created. If the file already exists, its contents will be overwritten.
- Syntax: `command > filename`

Example:

```
ls > filelist.txt
```

This command lists the contents of the current directory and redirects the output to a file named `filelist.txt`.

Appending Output Redirection (`>>`):

- The `>>` operator is used to append the standard output (stdout) of a command to the end of a file.
- If the file does not exist, it will be created. If the file already exists, the output will be appended to the end of the file.
- Syntax: `command >> filename`

Example:

```
echo "Additional text" >> filelist.txt
```

This command appends the text "Additional text" to the file `filelist.txt`.

EXERCISE

Task 1

Echo

```
echo "You guys are AWESOME!" | base64
```

Collect the output. (No, I'm not telling you, you have to prove it to yourself that you got it right)

Task 2

```
echo "<output from previous command>" | base64 -d
```

The -d in base64 decodes instead of encodes

The output should be exactly what you put in the encode in Task 1

Task 3

```
echo This is evil naughty naughty malware > malware.txt
```

This redirects the output of the echo to a file. No error or feedback means that it completed successfully.

Validate that it worked by:

```
cat malware.txt
```

The output should be exactly what you put in the echo but its saved to the file.

Now lets encode the output of the file and then save the encoded output to a different file.

```
cat malware.txt | base64 > notmalwarenoreally.txt
```

The encoded output should be exactly what you put in the echo but its saved to the file.

Validate that it worked by:

```
cat notmalwarenoreally.txt
```

It should be encoded and NOT human readable.

Now let's validate, by reversing the output of the encoded message.

```
cat notmalwarenoreally.txt | base64 -d
```

You should get   This is evil naughty naughty malware

This proves that the encoding and decoding was successful.

Task 4 – HASHING and validating:

In linux one can hash a file by using the md5sum command.

Let's hash the original file that we are pretending is malware.

```
md5sum malware.txt
```

The output should be as follows:

```
de54f727fe1eb6d1d0ca9411db64024a  malware.txt
```

Now let's hash the file, which STILL contains the malware in a encoded format.

```
md5sum notmalwarenoreally.txt
```

The output should be as follows:

```
6606d6223afdfe6c969d31197e9400e1  notmalwarenoreally.txt
```

Please note the above 2 hashes are different, this means that if I was using signature based antivirus, it WOULD NOT CATCH the encoded and obfuscated malware.

Takeaway

Encoding is a technique commonly used by malware authors to evade signature-based detection by antivirus software and other security mechanisms. Here's how encoding aids in this evasion:

1. **Obfuscation of Malicious Payload**: Encoding involves transforming the malicious payload of the malware into a different format, often using techniques like Base64 encoding, XOR encoding, or custom encryption algorithms. This transformation obscures the original code, making it difficult for signature-based detection mechanisms to recognize known patterns or signatures of malware.

2. **Bypassing Signature-based Detection**: Signature-based detection relies on predefined signatures or patterns of known malware to identify and block threats. By encoding the malware, attackers can create variations of the same malware that appear different to antivirus signatures. This allows them to bypass detection by traditional signature-based antivirus software, as the encoded variants do not match the signatures in the antivirus database.

3. **Dynamic Decoding**: Malware may include routines to dynamically decode or decrypt the encoded payload at runtime. This means that the malware is encoded in its storage or transmission state but decoded and executed in memory, making it more challenging for static analysis techniques to detect the malicious behavior.

4. **Polymorphic and Metamorphic Techniques**: Encoding is often combined with polymorphic and metamorphic techniques, where the malware continuously modifies its code structure and appearance to generate new variants that evade signature-based detection. These variations may include changes in encoding algorithms, encryption keys, or code obfuscation techniques, further complicating detection efforts.

5. **Delay Detection**: Even if security solutions eventually recognize and analyze the encoded malware, the initial encoding can delay detection, giving the malware a window of opportunity to execute its malicious activities before being detected and blocked.

Encoding helps malware evade signature-based detection by disguising its true nature, making it more challenging for security solutions to identify and mitigate threats effectively. To combat this evasion technique, security vendors employ advanced detection methods such as heuristic analysis, behavior-based detection, sandboxing, and machine learning algorithms to detect and block encoded malware variants.

Relevance to PowerShell

Base64 encoding is commonly used in obfuscating PowerShell malicious code to evade detection by security software and to make the code less readable to human analysts.

1. **Transforming the Payload**: Malicious PowerShell scripts contain commands or payloads that perform nefarious activities, such as downloading and executing malware, stealing sensitive information, or compromising the system. Before embedding the payload in the script, attackers encode it using Base64 encoding.

2. **Base64 Encoding**: Base64 encoding converts binary data into a text-based format consisting of printable ASCII characters. Each group of three bytes (24 bits) from the binary data is transformed into a group of four Base64 characters. The encoded data is then embedded directly into the PowerShell script as a string literal.

3. **Obfuscation**: By encoding the payload using Base64, attackers obscure the true nature of the malicious code. This obfuscation technique makes it difficult for security software and analysts to recognize the malicious commands or payloads within the script.

4. **Decoding at Runtime**: When the PowerShell script is executed, it includes a routine to decode the Base64-encoded payload dynamically. This decoding process reverses the Base64 encoding and retrieves the original binary data, which contains the malicious commands or payload.

5. **Execution of Malicious Activities**: Once the payload is decoded, the PowerShell script can execute the malicious commands or activities embedded within it. These activities may include downloading and executing additional malware, establishing persistence on the compromised system, or exfiltrating sensitive data.

6. **Evading Detection**: Base64 encoding helps in evading signature-based detection by security software. Since the encoded payload appears as a string of seemingly innocuous characters, it does not match known signatures of malicious code in antivirus databases. Additionally, the dynamic decoding process at runtime further complicates detection efforts.

Base64 encoding is a popular obfuscation technique used in PowerShell malicious code to hide the true intent of the script and evade detection by security software. While it adds a layer of complexity for defenders, security solutions can employ advanced detection techniques such as heuristic analysis, behavior monitoring, and anomaly detection to identify and mitigate the threat posed by obfuscated PowerShell scripts.

PowerShell has been increasingly utilized by attackers in various cyberattacks due to its powerful scripting capabilities and widespread presence in Windows environments. Here are a few notable instances and trends related to PowerShell usage in cyber breaches:

1. **Notable Incidents**:
   - PowerShell has been implicated in numerous high-profile cyber breaches and incidents over the years. For example, it was used in the NotPetya ransomware attack in 2017, where attackers leveraged PowerShell scripts to spread the malware laterally across networks.
   - PowerShell has also been utilized in various other types of cyberattacks, including credential theft, lateral movement, data exfiltration, and executing malicious payloads.
2. **Trend of PowerShell Usage**:
   - Security researchers and organizations have observed a rising trend in the use of PowerShell by cybercriminals and threat actors. This trend is driven by several factors, including PowerShell's built-in capabilities for system administration and automation, its ability to execute commands and scripts without writing to disk, and its widespread presence in Windows environments.
   - Threat actors often leverage PowerShell to evade detection by security solutions, as it allows them to execute malicious activities directly in memory, bypassing traditional antivirus and endpoint detection mechanisms.
3. **Statistics and Reports**:
   - Various cybersecurity reports and studies may provide insights into the prevalence of PowerShell usage in cyber breaches. These reports may analyze trends, common attack techniques, and the percentage of incidents involving PowerShell-based attacks.
   - Organizations and cybersecurity firms may also release advisories or whitepapers detailing specific incidents or campaigns involving PowerShell-based attacks.

Given the dynamic nature of cybersecurity threats and the evolving tactics employed by threat actors, it's essential for organizations to stay vigilant and implement robust security measures to detect and mitigate PowerShell-based attacks. This includes implementing security controls such as PowerShell logging, script block logging, application whitelisting, and behavior-based anomaly detection to detect and prevent malicious PowerShell activity.

Appendix:

Independent Research:

https://auth0.com/blog/encoding-encryption-hashing/

https://danielmiessler.com/p/encoding-encryption-hashing-obfuscation/

https://www.okta.com/identity-101/hashing-vs-encryption/

https://auth0.com/blog/how-secure-are-encryption-hashing-encoding-and-obfuscation/

https://en.wikipedia.org/wiki/Cryptography