ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/azrajabeen-mohamed-ali/411

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 1 & Iss: 1

Research Article

The Power of Cryptography: Hashing and Encryption for Data Protection

Azra Jabeen Mohamed Ali*

Independent Researcher, California, USA

Citation: Ali AZM. The Power of Cryptography: Hashing and Encryption for Data Protection. *J Artif Intell Mach Learn & Data Sci 2023*, 1(1), 1857-1861. DOI: doi.org/10.51219/JAIMLD/azrajabeen-mohamed-ali/411

Received: 02 February, 2023; **Accepted:** 18 February, 2023; **Published:** 20 February, 2023

***Corresponding author:** AzraJabeen Mohamed Ali, Independent researcher, California, USA, E-mail: Azra.jbn@gmail.com

A B S T R A C T

This paper explores the foundational techniques of hashing and encryption, which are essential for ensuring data protection and integrity in various applications, from secure communications to safeguarding stored data. In today's digital age, securing sensitive information is paramount and cryptography plays a critical role in achieving this goal. Hashing is used to convert input data into fixed-size hashes, providing a way to verify data integrity without exposing the original data, while encryption ensures that information is transformed into unreadable formats, only accessible by authorized parties. Through a detailed examination of popular cryptographic algorithms, such as SHA-256 for hashing and AES for encryption, the paper highlights how these methods are applied in real-world scenarios to protect privacy, prevent unauthorized access and guarantee data authenticity. Additionally, the paper discusses the strengths and weaknesses of each technique, their respective use cases and the evolving challenges in the cryptographic landscape, including computational threats and the potential impact of quantum computing. Ultimately, this study underscores the significance of hashing and encryption as cornerstones of modern cybersecurity practices, offering solutions to safeguard digital assets in an increasingly interconnected world.

Keywords: Cryptography, Encryption, Hashing, Security, Data protection, Algorithms, Decryption, decode, ciphertext

1. Introduction

1.1. Cryptography

Cryptography is the study and application of protecting data and communications from outside influence or manipulation. Data confidentiality, integrity and authenticity are safeguarded through the development of algorithms, protocols and systems. From data storage and digital signatures to online banking and secure messaging, cryptography is a fundamental component of contemporary information security.

There is no way for entities to communicate securely over public networks like the Internet. Unauthorized third parties may be able to view or even alter communications across these networks. In addition to providing a safe method of communicating over otherwise insecure channels, cryptography also helps prevent data from being viewed and offers methods for determining whether it has been altered. For instance, a cryptographic technique can be used to encrypt data, which can then be delivered in an encrypted state and decrypted by the intended recipient. It will be challenging to decode encrypted data if it is intercepted by a third party.

In C#, a namespace is used to organize code into groups, making it easier to manage and maintain. When working with cryptography in C#, the System, Security. Cryptography namespace is used, which contains classes that provide cryptographic functionality, such as hashing, encryption and digital signatures.

Fundamental principles of cryptography:

1. **Confidential:** Ensuring that only those with permission can access information.
2. **Integrity:** Ensuring that data is not changed while being sent or stored.

3. **Authentication:** Verifying the identity of the sender or receiver of the data.
4. **Non-repudiation:** Ensuring that a party cannot deny the authenticity of their message.

1.2. Core Cryptographic Mechanism components

The core cryptographic mechanism components are Encryption, Hashing, Digital signatures, Public key Infrastructure (PKI), Cryptographic Protocols.

1.3. Encryption

Encryption in cryptography is the process of converting plaintext (readable data) into ciphertext (unreadable data) using a specific algorithm and a key. Encryption serves to safeguard data secrecy by limiting access and comprehension to only those who are permitted. The foundation of contemporary cryptography is encryption, which guarantees the security of private data, communications and passwords while they are being transmitted or stored. It is classified into two types

1. Symmetric encryption is also known as Secret Key Encryption or Private Key Encryption or Single Key Encryption.
2. Asymmetric Encryption also known as Public Key Encryption or Two Key Encryption.

1.4. Symmetric Encryption

Symmetric Encryption uses the same single secret key for encryption and decryption. Because of this, it is known as symmetric encryption and its methods are far faster than public-key algorithms. The sender and receiver must both possess the same secret key, which must be kept confidential. AES (Advanced Encryption Standard), DES (Data Encryption Standard), 3DES (Triple DES), RC4, Blowfish algorithm classes are used to implement symmetric encryption. It is necessary to create a key and an initialization vector (IV) for symmetric algorithms. This key must be secret so it would be encrypted and IV does not need to be secret so it can be sent as plain text. When a new instance of one of the managed symmetric cryptographic classes is created using the parameterless Create () function, a new key and IV are automatically generated. Anybody who is permitted to decode data needs to have the same key, IV and algorithm.

A unique stream class known as a Crypto Stream is used with the managed symmetric cryptography classes to encrypt data that is read into the stream. A managed stream class, a class that implements the ICryptoTransform interface (derived from a class that implements a cryptographic algorithm) and a CryptoStreamMode enumeration that specifies the kind of access allowed to the CryptoStream are used to initialize the CryptoStream class. Any class that inherits from the Stream class, such as FileStream, MemoryStream and NetworkStream, can be used to initialize the CryptoStream class. These classes allows us to symmetrically encrypt a range of stream objects.

Below code (Figure1) demonstrates how to encrypt a plaintext string using the AES algorithm and a password as the key.

1.4.1. AES Encryption: The Aes.Create() method is used to create an AES encryption object. The Key is a 16-byte key, padded with PadRight(16) to ensure it's the correct size for AES-128 encryption. Typically, AES supports key sizes of 128, 192 or 256 bits and here we use 128 bits (16 bytes). The IV (Initialization Vector) is set to a zeroed byte array of 16 bytes. In real applications, it's recommended to use a random IV for enhanced security. GenerateKey and GenerateIV methods are available to generate multiple keys and IV.

```
static string Encrypt(string plainText, string password)
{
    using (Aes aesAlg = Aes.Create()) // Create an instance of AES encryption
    {
        aesAlg.Key = Encoding.UTF8.GetBytes(password.PadRight(16)); // AES requires 16 bytes key
        aesAlg.IV = new byte[16]; // Initialization vector

        // Create an encryptor from the key and IV
        ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
        // Use a MemoryStream to hold the encrypted data
        using (MemoryStream ms = new MemoryStream())
        {
            // Use CryptoStream for encryption
            using (CryptoStream cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write))
            {
                // StreamWriter to write the data to the CryptoStream
                using (StreamWriter sw = new StreamWriter(cs))
                {
                    sw.Write(plainText);// Write the plaintext to the stream to be encrypted
                }
            }
            return Convert.ToBase64String(ms.ToArray());
        }
    }
}
```

Figure 1.

1.4.2. CryptoStream: This stream encrypts data in real-time as it's being written. It reads plaintext and writes ciphertext (unreadable data) in the underlying MemoryStream.

1.4.3. Base64 Encoding: The result is a byte [] array (ciphertext), which is then converted to a Base64 string. Base64 encoding is used because it ensures the encrypted data can be represented as readable text, suitable for storage or transmission.

Benefits of Symmetric Encryption:

1. **Speed and Efficiency:** In general, symmetric encryption algorithms such as AES, DES and 3DES are quicker and more effective than asymmetric encryption, particularly when handling big data sets.
2. **Security with Key Management:** Symmetric encryption can offer robust secrecy and privacy when combined with a secure key management system (for example, via a secure key exchange procedure).
3. **Lower Computational Load:** Symmetric encryption puts less computational load on devices than asymmetric encryption since it employs fewer mathematical procedures.

1.5. Challenges of Symmetric Encryption

1.5.1. Key Distribution Issue: The requirement that communicating parties safely exchange and maintain the secret key is the main drawback of symmetric encryption. The encryption is rendered unsafe if the key is compromised.

1.6. Symmetric Decryption

Below code (Figure 2) demonstrates the Decrypt method with the implementation of AES decryption in C#. It decrypts a cipherText (which is assumed to be in Base64 format) using a given password. The method converts the Base64-encoded cipherText back into a byte array using Convert.FromBase64String (). The byte array is passed into a MemoryStream that acts as a buffer for the encrypted data. The CryptoStream is used to decrypt the data, transforming the byte data into readable plaintext. A StreamReader reads the decrypted data from the CryptoStream and returns it as a string.

1.7. Asymmetric Encryption

Asymmetric encryption is also known by several other names, including Public Key Encryption, Two-Key Encryption,

Public/Private Key Encryption. A public key for encryption and a private key for decryption is the two keys used in asymmetric encryption. The private key is kept safe, but the public key is readily exchanged. Many contemporary cryptographic systems, including those that protect communications over the internet, are based on this encryption technique. No need for the sender and receiver to share the same secret key. RSA, ECC (Elliptic Curve Cryptography), DSA (Digital Signature Algorithm) are used to implement asymmetric encryption.

```
static string Decrypt(string cipherText, string password)
{
    using (Aes aesAlg = Aes.Create())
    {
        aesAlg.Key = Encoding.UTF8.GetBytes(password.PadRight(16)); // AES requires 16 bytes key
        aesAlg.IV = new byte[16]; // Initialization vector

        // Create the decryptor using the key and IV
        ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);

        // Convert the Base64 cipherText to byte array
        using (MemoryStream ms = new MemoryStream(Convert.FromBase64String(cipherText)))
        {
            // Use CryptoStream to read and decrypt the data
            using (CryptoStream cs = new CryptoStream(ms, decryptor, CryptoStreamMode.Read))
            {
                // Read the decrypted data as a string
                using (StreamReader sr = new StreamReader(cs))
                {
                    return sr.ReadToEnd();// Return the decrypted plain text
                }
            }
        }
    }
}
```

Figure 2.

To safeguard the encrypted data, the private key is maintained secure and should never be lost or disclosed.

(Figure3) Below code demonstrates the method to encrypt a message using RSA with OAEP padding and SHA-256 as the hashing algorithm.

1.7.1. RSA Key Creation: RSA.Create() initializes a new RSA object. This method is modern and should be used in place of the older RSACryptoServiceProvider.

1.7.2. Loading the Public Key: The method rsa. ImportRSAPublicKey(Convert.FromBase64String(publicKey), out _) is used to load the public key into the RSA instance. The public key is expected to be in Base64-encoded format.

1.7.3. Encrypting the Message: Encoding.UTF8. GetBytes (message) converts the message (string) into a byte array so that it can be processed by the RSA algorithm. rsa. Encrypt(messageBytes, RSAEncryptionPadding.OaepSHA256) encrypts the byte array with the public key using OAEP padding with SHA-256.

1.7.4. RSA Encryption: The rsa.Encrypt() method encrypts the message. In this case, OAEP padding with SHA-256 is used, which is more secure than the older PKCS#1 v1.5 padding. It ensures that the encrypted data is protected against certain types of cryptographic attacks.

Base64 Public Key: The public key is assumed to be in Base64 format (publicKey).

Public Key Format: With an XML format, it is to use rsa. FromXmlString() instead of ImportRSAPublicKey(). If we are using a PEM file format, we need to use a proper PEM parser.

RSA Limitations: RSA is generally not used to encrypt large data directly due to limitations on the size of the message it can encrypt based on the key size (typically around 256 bytes for 2048-bit keys). For larger messages, RSA is often used to encrypt a symmetric key (e.g., AES key) and then the symmetric algorithm is used to encrypt the actual data.

```
static byte[] EncryptMessage(string message, string publicKey)
{
    using (RSA rsa = RSA.Create())
    {
        // Load the public key into the RSA object
        rsa.ImportRSAPublicKey(Convert.FromBase64String(publicKey), out _);

        // Encrypt the message
        byte[] messageBytes = Encoding.UTF8.GetBytes(message);
        return rsa.Encrypt(messageBytes, RSAEncryptionPadding.OaepSHA256);
    }
}
```

Figure 3.

1.8. Benefits of Asymmetric Encryption

Key Distribution: The issue of key distribution that symmetric encryption faces is resolved by asymmetric encryption. It is simple to send encrypted messages without requiring a secure channel for key exchange because the private key is kept secret while the public key can be published publicly.

Digital Signatures: The generation of digital signatures is made possible by asymmetric encryption. With the private key, a message can be signed and with the public key, the recipient can confirm the signature. By doing this, the message's integrity and authenticity are guaranteed.

Secure Communication: It enables secure communication over insecure channels, such as the internet. Public keys can be shared openly, while private keys remain confidential.

1.9. Challenges of Asymmetric Encryption

Slower Performance: Because of the intricacy of the mathematical calculations required, asymmetric encryption performs more slowly than symmetric encryption, particularly when dealing with huge data quantities.

Key Management: It still necessitates rigorous private key management even though it circumvents the symmetric encryption problem of key distribution. It is impossible to decipher encrypted data if a private key is lost.

1.10. Asymmetric Decryption

(Figure 4) Below code is an example of asymmetric decryption using RSA in C#. It decrypts an encrypted message with the RSA private key and returns the decrypted message as a string.

1.10.1. RSA Object: RSA.Create() initializes a new RSA object. The "using" block ensures the RSA object is disposed of properly once it's no longer needed.

1.10.2. Private Key import: The private key is expected to be Base64-encoded and is loaded into the RSA object using rsa. ImportRSAPrivateKey(). This method expects the private key to be in a specific format, often generated from the corresponding public key used in encryption.

1.10.3. Decryption: The encrypted message is passed in as a byte array (encryptedMessage). rsa.Decrypt() decrypts the data using OAEP padding with SHA-256, which ensures security.

1.10.4. Return Decrypted Message: The decrypted byte array is then converted to a UTF-8 string using Encoding.UTF8. GetString(decryptedBytes).

1.10.5. Public Key Encryption: The message is encrypted using the public key with OAEP SHA-256 padding for modern security.

1.10.6. Private Key Decryption: The encrypted message is

decrypted using the private key with the same OAEP SHA-256 padding to ensure that the decryption process matches the encryption scheme. OAEP with SHA-256 provides a more secure encryption method compared to the older PKCS#1 padding.

1.10.7. RSA Key Format: This code assumes that the public and private keys are Base64-encoded strings. If keys are in XML or PEM format, they will need to be converted to Base64 first or handled with appropriate parsers.

```
// Decrypt the message using RSA private key
static string DecryptMessage(byte[] encryptedMessage, string privateKey)
{
    using (RSA rsa = RSA.Create())
    {
        // Load the private key into the RSA object
        rsa.ImportRSAPrivateKey(Convert.FromBase64String(privateKey), out _);

        // Decrypt the message
        byte[] decryptedBytes = rsa.Decrypt(encryptedMessage, RSAEncryptionPadding.OaepSHA256);
        return Encoding.UTF8.GetString(decryptedBytes);
    }
}

static void Main()
{
    // Step 1: Generate RSA Keys (Public and Private)
    using (RSA rsa = RSA.Create())
    {
        // Export Public and Private Keys
        string publicKey = Convert.ToBase64String(rsa.ExportRSAPublicKey());
        string privateKey = Convert.ToBase64String(rsa.ExportRSAPrivateKey());

        Console.WriteLine("Public Key: ");
        Console.WriteLine(publicKey);
        Console.WriteLine("Private Key: ");
        Console.WriteLine(privateKey);

        // Step 2: Encrypt a message using the public key
        string originalMessage = "Hello, this is a secret message!";
        byte[] encryptedMessage = EncryptMessage(originalMessage, publicKey);

        Console.WriteLine("\nEncrypted Message: ");
        Console.WriteLine(Convert.ToBase64String(encryptedMessage));

        // Step 3: Decrypt the message using the private key
        string decryptedMessage = DecryptMessage(encryptedMessage, privateKey);

        Console.WriteLine("\nDecrypted Message: ");
        Console.WriteLine(decryptedMessage);
    }
}
```

Figure 4.

Table1: Comparison between Symmetric and Asymmetric.

	Symmetric Encryption	Asymmetric Encryption
Algorithm speed	Fast	Slow
No of Keys for encryption and decryption	1	2
Algorithm class used	AES (Advanced Encryption Standard), DES (Data Encryption Standard), 3DES (Triple DES), RC4, Blowfish	RSA, ECC (Elliptic Curve Cryptography), DSA (Digital Signature Algorithm)
Alias name	Single Key Encryption	Two Key Encryption

1.11. Hashing

The process of transforming an input (or "message") into a fixed-length string of characters, usually a digest, is known as hashing in cryptography. Usually, a hash function produces the outcome, which is known as the hash value or hash code. Hashing is a one-way conversion and they are frequently used to safeguard private information, including digital signatures and passwords. Hashing in C# is straightforward with the System. Security. Cryptography library. Hashing in C# is typically accomplished with the use of libraries that offer a variety of cryptographic techniques, including MD5 (Message Digest Algorithm 5) and SHA (Secure Hash Algorithm).

Common hashing Algorithms in C#:

- **SHA-256:** A member of the SHA-2 family, it produces a 256-bit hash value and its widely used.
- **SHA-512:** Another member of the SHA-2 family, it generates a 512-bit hash value and more secure than SHA1 and SHA256.
- **SHA1:** Deprecated for most cryptographic purposes due to vulnerabilities, but still supported.
- **MD5:** While still widely used, MD5 is considered cryptographically broken and unsuitable for security purposes due to vulnerabilities.

(Figure 5) Below is an example of hashing data using SHA-256 in C#

```
public static string ComputeSha256Hash(string rawData)
{
    using (SHA256 sha256Hash = SHA256.Create())
    {
        // Compute hash from the bytes of the string
        byte[] bytes = sha256Hash.ComputeHash(Encoding.UTF8.GetBytes(rawData));

        // Convert the byte array to a hexadecimal string
        StringBuilder builder = new StringBuilder();
        foreach (byte byteValue in bytes)
        {
            builder.Append(byteValue.ToString("x2"));
        }
        return builder.ToString();
    }
}
```

Figure 5.

- **SHA256.Create():** Creates an instance of the SHA-256 algorithm. It's used to generate the hash.

1.11.1. Compute Hash(): This method takes the byte array of the input data (rawData converted to UTF-8 bytes) and returns the hash as a byte array. The length of the hash is fixed at 32 bytes (256 bits) for SHA-256.

1.11.2. Hexadecimal Conversion (x2): builder. Append(byteValue.ToString("x2")): For each byte in the hash array, this converts the byte to a two-digit hexadecimal string (x2 stands for hexadecimal format with two digits). This is done to represent the hash as a readable hexadecimal string.

1.11.3. StringBuilder: The StringBuilder class is used to efficiently append each hexadecimal byte to the result string. It's generally more efficient than using regular string concatenation inside a loop.

1.11.4. return builder.ToString();: Finally, the StringBuilder is converted into a string and returned, which is the final SHA-256 hash in hexadecimal format.

SHA-256 is one of the most secure cryptographic hash functions and is widely used for password hashing, data integrity verification and digital signatures. Using StringBuilder for constructing the hexadecimal string is optimal for performance, especially when hashing larger inputs or using this function repeatedly. This function is ready to use and can be incorporated into larger cryptographic operations, such as password storage or file integrity checks.

1.12. Benefits of Hashing

Data Integrity: Data integrity is ensured through hashing. The hash is an effective instrument for confirming the integrity of data because it will vary drastically if even a single bit of data changes. Digital signatures and file verification benefit greatly from this.

Efficiency: Hash functions are quick and low-cost to compute. Even for big datasets, they can produce a hash value fast.

Password Storage: Hashing is a popular technique for safely storing passwords. A password is hashed when it is created by the user and only the hash is saved. In the event of a data breach, the danger of exposure is decreased because the actual password is never saved.

Fixed Output Size: The hash value (e.g., SHA-256) has a fixed length, regardless of the input size. This makes handling data simpler and more reliable.

Feature	Hashing	Symmetric Encryption	Asymmetric Encryption
Security	Provides data integrity and authenticity	Ensures confidentiality of data with shared key	Secures communication without sharing a secret key
Speed	Very fast and efficient	Faster than asymmetric encryption	Slower than symmetric encryption
Use Cases	Password storage, file integrity, digital signatures	Data encryption, VPNs, secure communications	Secure communications, digital signatures, blockchain
Key Management	No keys to manage for verification	Requires secure key distribution/ management	Easy key distribution (public key), private key needs protection
Reversibility	One-way process (irreversible)	Reversible (with the same key)	Reversible (with private key for decryption)
Non-repudiation	Provides data verification	No inherent non-repudiation	Provides strong non-repudiation

Table 2: Summary of Benefits.

2. Conclusion

Cryptography is an essential technology for ensuring the security and privacy of digital information in today's interconnected world. It enables secure communication, protects sensitive data and supports trust in various systems. With ongoing advancements, cryptography continues to evolve, meeting the challenges of emerging technologies while safeguarding digital assets. The best cryptographic method to use depends on the specific security requirements and the context in which it is being applied. Often, a combination of these techniques is used to achieve robust security in modern systems (e.g., combining asymmetric encryption for key exchange with symmetric encryption for bulk data encryption).

3. References

1. https://www.c-sharpcorner.com/article/cryptography-in-net/
2. https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography?view=net-9.0
3. https://learn.microsoft.com/en-us/dotnet/standard/security/cryptographic-signatures
4. https://learn.microsoft.com/en-us/dotnet/standard/security/decrypting-data
5. https://learn.microsoft.com/en-us/dotnet/standard/security/walkthrough-creating-a-cryptographic-application
6. https://learn.microsoft.com/en-us/dotnet/standard/security/ensuring-data-integrity-with-hash-codes
7. Joseph Albahari and Ben Albahari, "C# 7.0 in a Nutshell 7Th Edition" O'Reilly Media, 2017.
8. Joseph Albahari and Ben Albahari, "C# 9.0 in a Nutshell" O'Reilly Media, 2021.
9. Joseph Albahari and Ben Albahari, "C# 10.0 in a Nutshell" O'Reilly Media, 2022.
10. Matthew Macdonald, Eric Johansen "C# Data Security Handbook" Apress Publication, 2003.
11. Rod Stephens, "C# 5.0 Programmer's Reference" Wrox Publication, 2014.
12. Marius Iulian Mihailescu, Stefania Loredana Nita "Pro Cryptography and Cryptanalysis: Creating Advanced Algorithms with C# and .NET "Apress Publication, 2020.