The Role of Cryptography in Securing the JWT Ecosystem in Real-World Systems

Arun Neelan

Independent Researcher
PA, USA
arunneelan@yahoo.co.in

Abstract
This review paper provides a comprehensive overview of cryptography and its role in JSON Web Tokens (JWT), focusing on secure information exchange in modern web applications and microservices architectures. It introduces key cryptographic concepts, including symmetric and asymmetric encryption, hashing, and digital signatures. It then explores the structure and functionality of JWT and its use in authentication and authorization. The paper also explores JSON Web Signature (JWS) and JSON Web Encryption (JWE), detailing their processes for digital signing, verification, and secure data encryption. Emphasizing the importance of strong encryption, proper key management, and secure transport protocols, the review underscores the need for continuous updates to cryptographic practices to maintain the integrity, confidentiality, and authenticity of information in modern systems, and understanding these concepts becomes increasingly critical for developers, system architects, and security professionals alike.

Keywords: Cryptography, Cryptographic Algorithms, JWT, JWT Encryption, JWE, JWT Signature, JWS, Digital Signature, API Security, Data Integrity, Data Confidentiality

I. INTRODUCTION

In today's interconnected digital world and evolving, vast amounts of sensitive data are exchanged across networks, ensuring the confidentiality, integrity, and authenticity of that data has never been more crucial. The rapid growth of web applications, microservices, and API-based communication frameworks has elevated the need for robust security protocols to safeguard information during transmission. Cryptography, which involves using mathematical techniques to protect information, is central to these security measures. It ensures that data is encrypted, its authenticity is verified, and it is protected from unauthorized access or tampering. From password protection to secure messaging, cryptography plays a vital role in protecting information across various digital platforms.

One of the most widely adopted cryptographic tools for securing communication on the web is the use of JSON Web Tokens (JWT). JWTs provide a compact, URL-safe way to transmit information between parties as a JSON object. [1] These tokens are widely used for authentication and authorization purposes, forming the backbone of secure user login systems, Single Sign-On (SSO) services, and access control in microservices architectures. To understand the effectiveness and security of JWTs, it is crucial to grasp the underlying cryptographic principles that support them, including encryption methods, digital signatures, and key management practices.

This paper explores the role of cryptography in securing JWTs, with a focus on JSON Web Signatures (JWS) and JSON Web Encryption (JWE). It will examine the cryptographic techniques that ensure data

security and authenticity in JWTs and discuss best practices for implementing these mechanisms in real-world applications.

II. CRYPTOGRAPHY OVERVIEW

To understand JWT and how it works, it's essential to have a basic understanding of cryptography. Cryptography is the process of converting information into a secret code, ensuring that only the intended person can read it. It's like locking a message in a box and giving the key only to the person meant to open it. It helps protect our data, such as passwords and messages, from being accessed by others.

Types of Cryptography include:

A. Symmetric Key Cryptography

In symmetric encryption, it’s the same key that’s used for both encryption and decryption. Since it’s the same key used for both encryption and decryption, it must be kept secret and shared between parties in a secure manner. It’s faster and more efficient for large amounts of data.

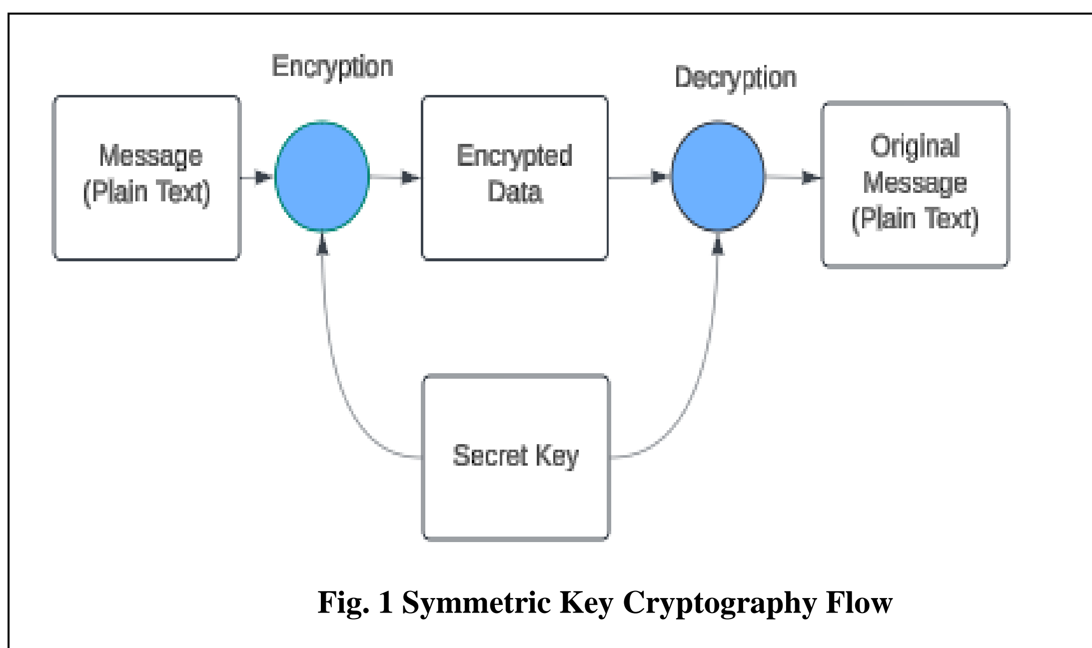

Fig. 1 Symmetric Key Cryptography Flow

B. Asymmetric Key Cryptography

In asymmetric encryption, two distinct keys are used: a public key for encryption and a private key for decryption. The owner securely stores the private key and shares the public key with clients. Clients use the public key to encrypt data before sending it to the owner, ensuring only the owner can decrypt and access the data with the corresponding private key. This method can also be applied to exchange symmetric keys securely.

Clients usually share public keys in JWKS (JSON Web Key Set) format. This format is an array of JWKs (JSON Web Keys), which represent cryptographic public keys in JSON format, and are used for verifying signatures or encrypting data.

Verifying Signatures - The public keys are used to verify the authenticity of a signature, ensuring the data hasn't been altered and was signed by the correct sender.

Encrypting Data - The public keys are used to encrypt data, ensuring that only the intended recipient with the matching private key can decrypt and access the original data.

A JWK is a JSON object representing a cryptographic key, typically a public key, in a structured format. The key includes various attributes that define its type, algorithm, and other relevant details for its use. The table below lists some key fields that are part of a JWK. [2]

Attr Name	*Description*	*Sample*
kty	Key type.	RSA
kid	A unique identifier for the key allows clients to select the correct key when multiple public keys are available.	123456789
alg	The algorithm used.	RS256
use	The intended use of the key.	sig for Signing; enc for encryption.
n	The modulus in RSA encryption, denoted as n, plays a key role in determining the encryption's strength. The size of n directly impacts security: larger values make it harder to break the encryption by factoring n into its prime components.	323
e	The public exponent for the RSA key.	17

Table 1. JWK Fields

C. Hashing

Hashing is a process that transforms data into a fixed-length string of characters, known as a hash, using a mathematical algorithm. It's a one-way process, meaning the original data cannot be retrieved from the hash. Hashing is commonly used to verify data integrity and securely store passwords. For added security, a salt (a random string) is often added to the data before hashing, making it more resistant to attacks like rainbow table attacks. [3]

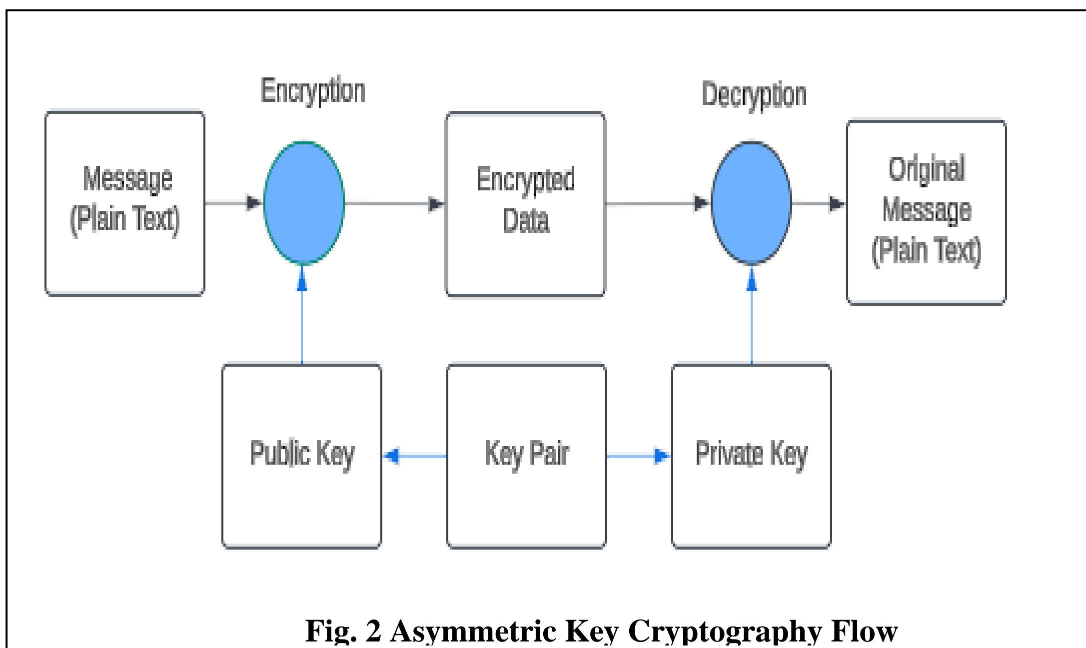

Fig. 2 Asymmetric Key Cryptography Flow

D. Digital Signatures

It's a method used to verify both authenticity and integrity, which will be explained in more detail in the following sections.

III. JSON WEB TOKENS (JWT)

As outlined in [1], JWT (JSON Web Token) is a compact, URL-safe format used to securely exchange information as a JSON object. It is commonly used for authentication and authorization, allowing data (claims) to be transmitted in a way that can be verified and trusted, as the information is digitally signed.

A JWT typically consists of three parts, separated by periods (.). The format is as follows:

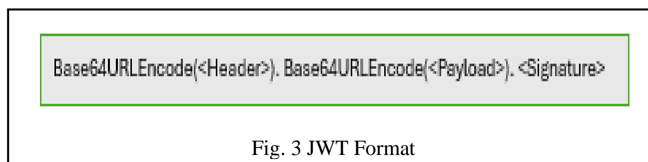

Fig. 3 JWT Format

The breakdown of JWT is as given below:

A. JOSE Header

The header is a JSON object that contains metadata about the algorithm used to sign or encrypt the token, along with other relevant options. It helps the recipient understand how to properly process the token.

JOSE Header Name	***Description***
alg	Indicates the cryptographic algorithm used to sign or verify the JWT's signature.
typ	Denotes that this is a JWT.
kid	Specifies the key id that can be used to identify the correct key for signature verification.

Table 2. JOSE Headers

B. JWT Claims (Payload)

JWT claims are the payload that represent user data, authorization information, and other relevant application details. Custom claims can also be included to store application-specific information.

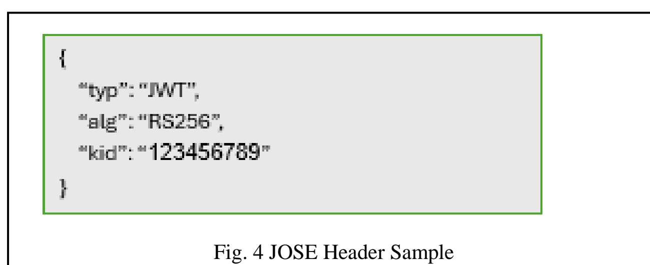

Fig. 4 JOSE Header Sample

All the fields below are optional.

Claim Name	*Description*
iss	The issuer claim identifies the principal that issued the JWT.
sub	The subject claim identifies the principal that is the subject of the JWT.
aud	The audience claim identifies the recipients that the JWT is intended for.
exp	The expiration time identifies the expiration time on or after which the JWT must not be accepted for processing.
nbf	The not before claim identifies the time before which the JWT must not be accepted for processing.
iat	The issued at claim identifies the time at which the JWT was issued.
jti	The JWT ID claim provides a unique identifier for the JWT.

Table 3. JWT Claims


```
{
  "iss": "https://mydomain.com",
  "aud": "myaud.com",
  "exp": 1686218321
}
```


Fig. 5 JWT Claims Sample


```
ew0KICAgIOKAnHR5cOKAnTog4oCcSldU4oCdLA0KICAgIOKAnGFsZ-
.
ew0KICAgIOKAnGlzc-KAnTog4oCcYWJj4oCdLA0KICAgIOKAnGF1ZOKAnTog4oCcbXlhd
.
TTwOPdpiNu-
c0N9uAIBVVAZKDVZMPGGY1PUkMp_PE34azdSkimFWdShYZ8EG37EZQYyqStaClVD3AVHe0O
lFQSY9KfK2mSz2dQNi-
```


Fig. 6 JWT Sample

C. Signature

JSON Web Signature (JWS) is a standard for securely transmitting information as a JSON object. It enables the sender to digitally sign the data, ensuring its authenticity, integrity, and, in some cases, confidentiality.

IV. JSON WEB SIGNATURE PROCESS

A. Digital Signing

The signature is created by signing the combination of the base64url-encoded JOSE header and base64url-encoded JWT claims. The secret key is used for HS256 (HMAC with SHA-256, while for RS256 (RSA with SHA-256), the private key associated with the kid and alg in the JOSE header is used. This process ensures the integrity and authenticity of the data, confirming that it hasn't been tampered with and proving it came from a trusted source.

Signing a message involves

- Hashing the message with a cryptographic hash function (e.g., SHA-256) available in the alg of the header. Message that will be hashed: base64UrlEncode(header) + "." + base64UrlEncode(payload)
- Encrypting the resulting hash with a private key, creates a digital signature.

Thus a JWT is formed by combining base64url-encoded JOSE header, base64url-encoded JWT claims, and the signature: base64url(JOSE Header).base64url(JWT Claims).Signature, and it's this JWT that's sent to the recipient. [4] Libraries are available to generate JWTs for a given message.

The JWT can be decoded to verify that it contains the correct header and payload. The signed message (part 3) is sent to the recipient along with the original message (parts 1 and 2).

B. Signature Verification

The signature verification process in JWT (JSON Web Token) is essential for confirming the token's integrity and authenticity. When a server or recipient receives a JWT, they need to ensure that the signature is valid and that the data in the token remains unchanged. This helps guarantee that the token hasn't been tampered with and that a trusted source issued it.

Verification involves

Base64Url decode the JWT to ensure it has the correct header and payload.

- Split the JWT to obtain the header, payload and signature separately
- Base64Url the values individually to obtain the actual header, payload and signature separately.
- Validate that the header and payload have the intended data.

Validate the signature [5]

- Symmetric Cryptography (HMAC): To validate the signature, use the secret key to generate a new signature from the original message (header + payload). Compare this newly generated signature with the one stored in the JWT. If they match, the JWT is valid.
- Asymmetric Cryptography (RSA): Use the public key to verify the JWT's signature. The public key will check if the signature was created using the corresponding private key, based on the algorithm specified in the JWT header (e.g., RS256). If the verification passes, the JWT is authentic.

The implementation flow for the digital signature in access token retrieval is outlined below.

Flow Overview

1. The client registers with the Authorization Server by submitting its initial JWKS (JSON Web Key Set) and/or the associated JWKS URL. This allows the Authorization Server to retrieve the latest public keys from the URL when the client's keys are about to expire.

2. The Authorization Server generates and provides the client with a unique client ID and the token request URL.
3. The Authorization Server fetches the latest JWKS either when the previously obtained set is about to expire or at regular intervals.
4. The client provides the latest JWKS in response.
5. The client creates a JWT (JSON Web Token) to request an access token. JWT Format = Base64UrlEncode(JOSE Header). Base64UrlEncode (JWT Claims). Base64UrlEncode(Signature)
6. The client sends the generated JWT to the Authorization Server as part of the token request. The recommended approach is to include the JWT in the Authorization header using the Bearer scheme: Authorization: Bearer <jwt>
7. The Authorization Server verifies the JWT's signature to ensure a trusted party issued it. This process involves using libraries that accept the public key and algorithm for validation. If the JWT has been tampered with, the Authorization Server returns an error.
8. The Authorization Server validates the claims within the JWT, checking attributes such as the intended audience, expiration time, and others. The Authorization Server returns an error if any claim contains invalid data, such as an expired timestamp.
9. Once the verification is successful, the Authorization Server generates and returns an access token to the client.
10. The client stores the access token in a cache and reuses it for subsequent requests until it expires. Once it is expired, the client will request a new access token.

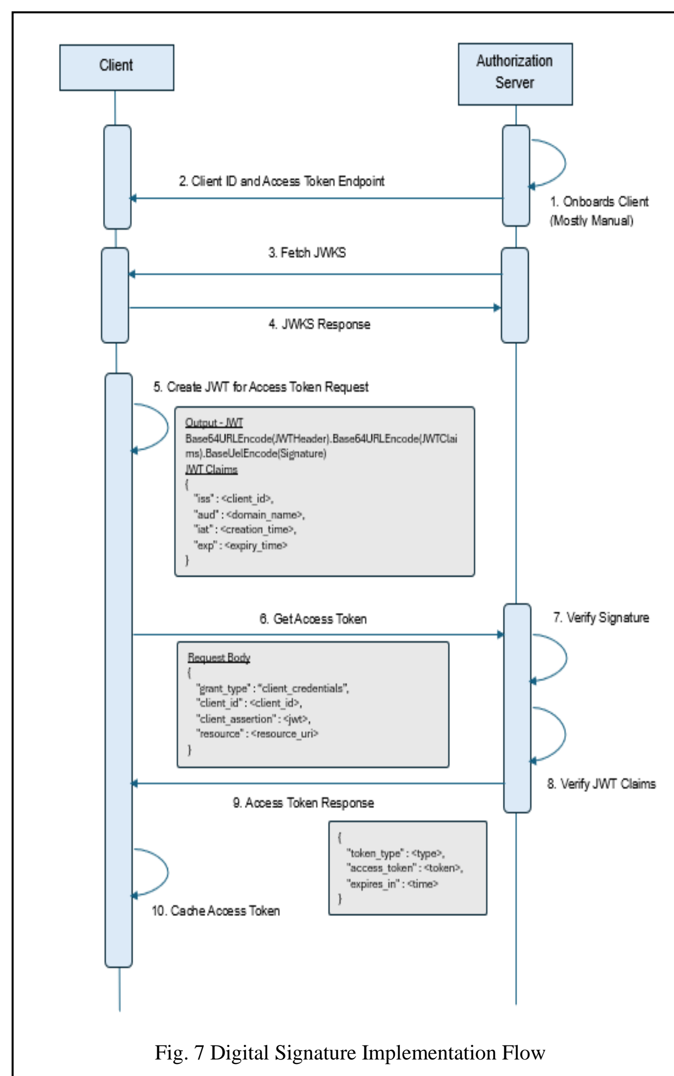

Fig. 7 Digital Signature Implementation Flow

The above example uses asymmetric cryptography (RSA or ECDSA algorithms), where different keys are used for signing and verification. However, symmetric cryptography (HMAC) can also be employed, where the same key is used for both signing and verification.

C. Security Considerations and Best Practices

- *Use Strong Signing Algorithms:* When signing a JSON Web Signature (JWS), it is essential to use strong and widely recognized cryptographic algorithms, such as RS256 (RSA with SHA-256) or ES256 (ECDSA with P-256 and SHA-256), which provide proven security and are broadly trusted. Weak or deprecated algorithms, especially those involving static keys or the 'none' algorithm, should be strictly avoided, as they leave the JWT unsigned and vulnerable to tampering. [1], [6] The 'alg' claim must always reference a valid and secure signing algorithm, and any JWT that employs the 'none' algorithm should be rejected to protect against potential exploitation. In scenarios where symmetric cryptography is required, the secret key must be sufficiently long, random, and securely managed to maintain system integrity and prevent unauthorized access. Stay up to date with cryptographic research to avoid using deprecated or insecure algorithms.
- *Validate Signature:* Always validate the signature of incoming JWS tokens to ensure the payload has not been tampered with. This can be done using the public key corresponding to the private key that signed the token. Data should never be trusted without proper validation, as it is crucial for maintaining security and authenticity.
- *Avoid sensitive data in JWT Payload:* Sensitive information, such as passwords or personal data, should not be placed in the JWT payload, as it is base64-encoded and can be easily decoded, even though its integrity is protected if signed. Storing sensitive data outside the JWT is essential, as anyone with access to the token can inspect the payload, even if the signature prevents modification. Alternative mechanisms, such as encryption, should be used to ensure confidentiality, as will be discussed in the following sections.

Other best practices are covered at the end of this paper, as they apply not only to JWS but also to JWE.

V. JSON WEB ENCRYPTION PROCESS

As outlined in [2], a JSON Web Encryption (JWE) is a compact and self-contained way to represent encrypted content using JSON-based data structures. It is widely used for securely transmitting sensitive information between parties.

JWE involves encrypting the payload of a message, which is defined in the [2]. It includes a header, an encrypted key, an initialization vector, the ciphertext, and an authentication tag. These elements together form the encrypted message.

Base64URLEncode(<Header>). Base64URLEncode(<Payload>). Base64URLEncode(<EncryptedKey>).
Base64URLEncode(<IV>). Base64URLEncode(<CipherText>). Base64URLEncode(<AuthorizationTag>)

Fig. 8 JWE Token Sample

The breakdown of JWE is as given below:

Header: This is a JSON object that typically includes information such as the encryption algorithm used (e.g., alg and enc).

Encrypted Key: The symmetric encryption key (CEK) that was used to encrypt the payload, which is encrypted with the recipient's public key.
Initialization Vector (IV): A random value used in encryption (for algorithms like AES) to ensure that encrypting the same data multiple times yields different ciphertexts.
Ciphertext: This is the encrypted payload.
Authentication Tag: This is an optional field that is base64url-encoded and used to verify the integrity and authenticity of the message.

A. *JWE Token Creation*

The JWE creation process involves generating a random Content Encryption Key (CEK) to encrypt the payload using a symmetric encryption algorithm (e.g., AES). A random Initialization Vector (IV) is used to ensure unique encryption each time. The CEK is then encrypted using the recipient's public key with an asymmetric algorithm (e.g., RSA). A header is created that mentions the encryption algorithms, and all components (header, encrypted key, IV, ciphertext, and authentication tag) are base64url-encoded and concatenated to form the final JWE string for transmission. [7]

The JWE string is sent to the recipient for processing, usually in a header or payload.

B. *JWE Token Decryption*

The JWE decryption process begins by splitting the JWE string into its components and Base64Url decoding them: header, encrypted key, IV, ciphertext, and authentication tag. The recipient uses their private key to decrypt the encrypted key, retrieving the Content Encryption Key (CEK). With the CEK and the IV, the recipient can decrypt the ciphertext using the appropriate symmetric algorithm (e.g., AES). [7] If an authentication tag is present, it is used to verify the integrity of the decrypted data. The result is the original payload, now accessible to the recipient.

The JWE implementation flow for request processing is illustrated in the diagram below.

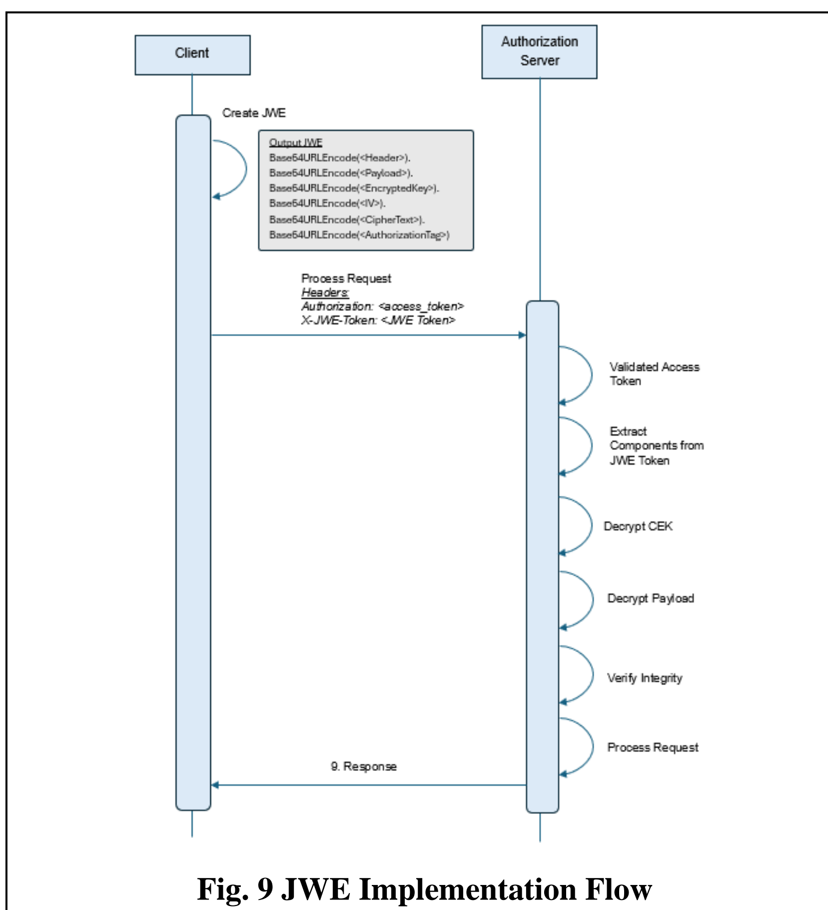

Fig. 9 JWE Implementation Flow

C. *Security Considerations and Best Practices*

- *Use Strong Encryption Algorithms:* AES (Advanced Encryption Standard) is considered the most secure symmetric encryption algorithm, with AES-GCM (Galois/Counter Mode) being the recommended mode for authenticated encryption, ensuring both confidentiality and integrity of the ciphertext. AES-CBC (Cipher Block Chaining) may be used when authenticated encryption is not required, but it necessitates careful management of Initialization Vectors (IVs) to maintain security. For asymmetric encryption in key exchange, RSA is commonly employed, but it must be used with key sizes of at least 2048 bits to maintain security, as smaller sizes, such as 1024 bits, are now deemed insecure. [7] Elliptic Curve Cryptography (ECC) algorithms, such as ECDH (Elliptic Curve Diffie-Hellman), are suitable for key exchange in JWE, offering strong security with smaller key sizes, like P-256 (256-bit curve), while providing computational efficiency. It's essential to stay up to date with cryptographic research to avoid using deprecated or insecure algorithms.

VI. SECURITY CONSIDERATIONS AND BEST PRACTICES COMMON TO BOTH JWS AND JWE TOKENS

A. *Use Security Transport (TLS/HTTPS)*

To ensure the security of both JWS (JSON Web Signature) and JWE (JSON Web Encryption) tokens, it is critical to always use HTTPS (TLS/SSL) for their transmission across the network. Transmitting tokens over unencrypted channels exposes them to risks such as interception, tampering, and man-in-the-middle attacks. [8] While JWS ensures data integrity and authentication by signing the payload, and JWE provides encryption to protect the confidentiality of the payload, HTTPS secures the communication channel between the client and server. This combination of cryptographic techniques and secure transport guarantees that both JWS and JWE tokens are protected from unauthorized access or alteration during transit, enabling secure, efficient communication in web applications or microservices architectures.

B. *Proper Key Management*

Effective key management is fundamental to the security of both JWS (JSON Web Signature) and JWE (JSON Web Encryption) tokens. Cryptographic keys must be stored securely using trusted solutions, such as Hardware Security Modules (HSMs) or Cloud Key Management Services (KMS), to safeguard against unauthorized access. To prevent exposure, private keys used for asymmetric encryption (e.g., RSA, ECDSA) and signing must be rigorously protected. For symmetric encryption (e.g., AES) in JWE and symmetric signing (e.g., HMAC) in JWS, it is essential to use strong, randomly generated keys, with a policy of regular key rotation to reduce the risk of compromise. Automated key rotation mechanisms and proper retirement of obsolete keys are crucial to maintaining robust security over time. [9] Additionally, setting expiration times for both JWS and JWE tokens ensures that, its validity is time-limited even if a key or token is compromised, its validity is time-limited, effectively minimizing the potential for exploitation and strengthening overall system resilience.

C. *Validate Essential Claims*

Validating essential claims is crucial for both JWS (JSON Web Signature) and JWE (JSON Web Encryption) tokens. Ensure the JWT includes essential claims, such as those listed below, and validated each time.

- iss (issuer) – Who issued the JWT.
- sub (subject) – The user or entity to which the JWT is assigned.
- aud (audience) – The intended recipient(s) of the JWT.
- exp (expiration) – The expiration time of the JWT.

Proper validation of these claims helps prevent unauthorized access, misuse, and security vulnerabilities in both signed and encrypted tokens.

D. Token Revocation

A challenge with JWS tokens in stateless authentication systems is the difficulty in revoking tokens before they expire. If a token is compromised, there's no simple way to invalidate it immediately without maintaining server-side sessions. To mitigate this, implement token revocation strategies such as using a revocation list, short-lived tokens with refresh tokens, or blacklists. [10]

VII. CONCLUSION

The intersection of cryptography and JSON Web Tokens (JWT) forms the foundation for secure authentication and data exchange in modern systems. A solid understanding of the cryptographic mechanisms involved, such as symmetric and asymmetric encryption, digital signatures, and hashing, is essential for ensuring the integrity, confidentiality, and authenticity of the information exchanged between parties.

JWT provides a compact and efficient way to transmit claims, with its reliance on digital signatures and encryption ensuring robust security for web applications and microservices. However, to fully utilize its potential and enhance security, it is important to follow established best practices. This includes using strong algorithms, validating signatures, avoiding the inclusion of sensitive data in the payload, and maintaining effective key management.

In addition to these practices, ensuring secure communication via HTTPS and implementing comprehensive token management strategies, such as token revocation, is critical. Furthermore, staying informed about the latest advancements in cryptography is essential, adopting the most secure, up-to-date algorithms, and avoiding deprecated methods that could compromise system integrity. By continuously updating knowledge and systems, developers can help ensure the ongoing security and reliability of JWT-based systems in an ever-evolving technological landscape.

REFERENCES

[1] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," May 2015. doi: 10.17487/rfc7519. Available: https://doi.org/10.17487/rfc7519

[2] M. Jones and J. Hildebrand, "JSON Web Encryption (JWE)," May 2015. doi: 10.17487/rfc7516. Available: https://doi.org/10.17487/rfc7516

[3] M. A. K. Taha, "Password Hashing with Salt and Keyed-Hash Message Authentication Code (HMAC)," International Journal of Computer Applications, vol. 61, no. 19, pp. 44–47, Jan. 2013.

[4] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Signature (JWS)," May 2015. doi: 10.17487/rfc7515. Available: https://doi.org/10.17487/rfc7515

[5] C. A. G. Willison, "Web Security with JSON Web Tokens," *International Journal of Information Security Science*, vol. 5, no. 2, pp. 14-25, 2016.

[6] A. M. Meha, M. S. M. Nor, and N. M. Nordin, "A Survey on Security Issues in JSON Web Token (JWT) and Countermeasures," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 9, no. 3, pp. 301–307, 2018.

[7] A. M. S. Lee, "Practical Application of Web Encryption Mechanisms," *Journal of Web Security*, vol. 8, no. 1, pp. 12-23, 2019.

[8] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," Aug. 2008. doi: 10.17487/rfc5246. Available: https://doi.org/10.17487/rfc5246

[9] C. A. B. L. Garcia, "Cryptographic Key Management in Secure Systems," *Journal of Cryptographic Engineering*, vol. 9, no. 3, pp. 45-56, 2020.

[10] S. K. Ghosh and M. P. Singh, "Mitigating Token-based Authentication Vulnerabilities in Stateless Web Systems," *International Journal of Applied Cryptography*, vol. 5, no. 2, pp. 51-66, 2020.