AhnLab Cyber Threat Intelligence Report

TLP : CLEAR

Analysis Report

A Shadow of JWT

Korean e-commerce giant breached due to unmanaged JWT

AhnLab SEcurity intelligence Center (ASEC)

2026. 01

AhnLab

Table of Contents

Summary

At the end of 2025, a massive personal data breach involving South Korea's largest e-commerce platform came to light. As of December 2025, the scale of the leak was reported to exceed 30 million. Behind this security incident lay the poor management of the company's JWT (JSON Web Token) authentication system. A JWT signing key belonging to a former employee was not reset even after their departure, and the attacker was able to use this valid signing key to access customer data without restriction.

JWT-based authentication, which lies at the center of this incident, has become a standard in modern web and mobile applications. While it offers the convenience of stateless authentication, improper management can render it a single point of failure, potentially collapsing the entire authentication system.

This report introduces the concept of JWTs and their authentication mechanisms, then analyzes major vulnerabilities, and finally presents practical security strategies for prevention and mitigation.

About JWT

JWT is a web standard (RFC 7519) and a lightweight, compact and self-contained authentication mechanism that securely transmits information between two parties using JSON objects. A JWT authentication token contains all required information within itself, enabling user authentication and authorization without database lookups on the server.

Its stateless structure reduces the burden of storing and synchronizing server-side sessions, and it can be easily transmitted via HTTP headers or URL parameters. Thanks to its scalability in distributed environments, JWT has become a core authentication mechanism in modern web and microservices architectures.

1. Structure and Component

A JWT consists of three Base64Url-encoded parts - header, payload, and signature - concatenated and separated by dots (.).

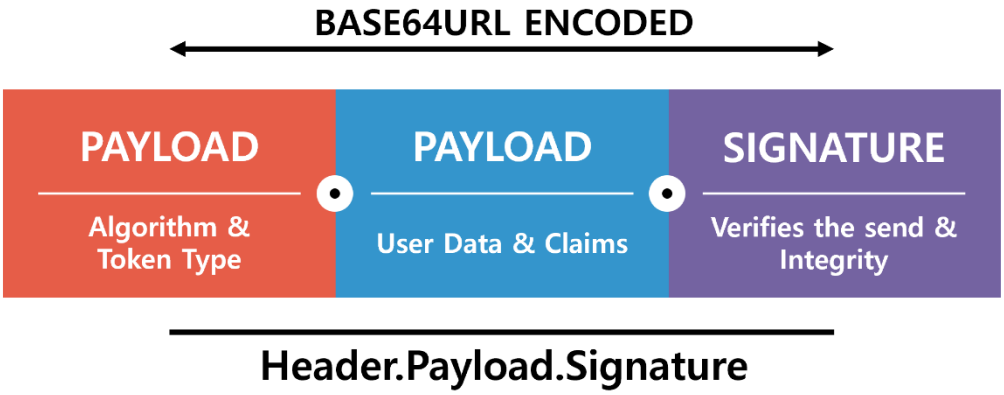

Figure 1. Structure of JWT

1) Header: metadata

The header is the Base64Url-encoded JSON object that contains the token's metadata. It mandatorily includes the typ (Type, typically JWT) and alg (Algorithm, the cryptographic algorithm used for signing, e.g., HS256, RS256) claims.

```
{
  "alg": "HS256",   // signature algorithm
  "typ": "JWT"      // token type
}
```

Table 1. Header of JWT

2) Payload: claims

The payload contains the authentication, authorization, and other attribute information to be transmitted in the form of a JSON object. These pieces of information are referred to as claims. Like the header, the payload is Base64Url-encoded.

- **Types of claims:**
  Claims are categorized into three types based on their role: registered, public, and private. Registered claims include standard fields used for token validation, such as expiration time (exp), issued-at time (iat), issuer (iss), and audience (aud).
- **Security considerations:**
  Since the header and payload are only encoded, not encrypted, their contents can be easily decoded if the payload is obtained. Therefore, sensitive personally identifiable information (PII), such as email addresses or payment details, must not be included in the payload.

```
{
  "sub": "user123",              // user ID
  "name": "John Doe",            // user name
  "role": "admin",               // privilege
  "iat": 1516239022,             // issued-at time
  "exp": 1516242622              // expiration time
}
```

Table 2. Payloads of JWT

3) Signature: ensuring integrity

The signature is the core element that guarantees the token's integrity and authenticity. It is calculated by applying a cryptographic signing algorithm to the encoded header and payload, combined with a secret key stored on the server (HS256) or a private key (RS256). The server recalculates the signature and compares it with the received one to determine whether the token has been tampered with. An attacker who does not possess the secret

signing key cannot generate a valid signature.

```
HMACSHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  SECRET_KEY
)
```

Table 2. Signature of JWT

2. Session-based vs. JWT

Figure 2 presents a comparison between session-based authentication and JWT authentication.

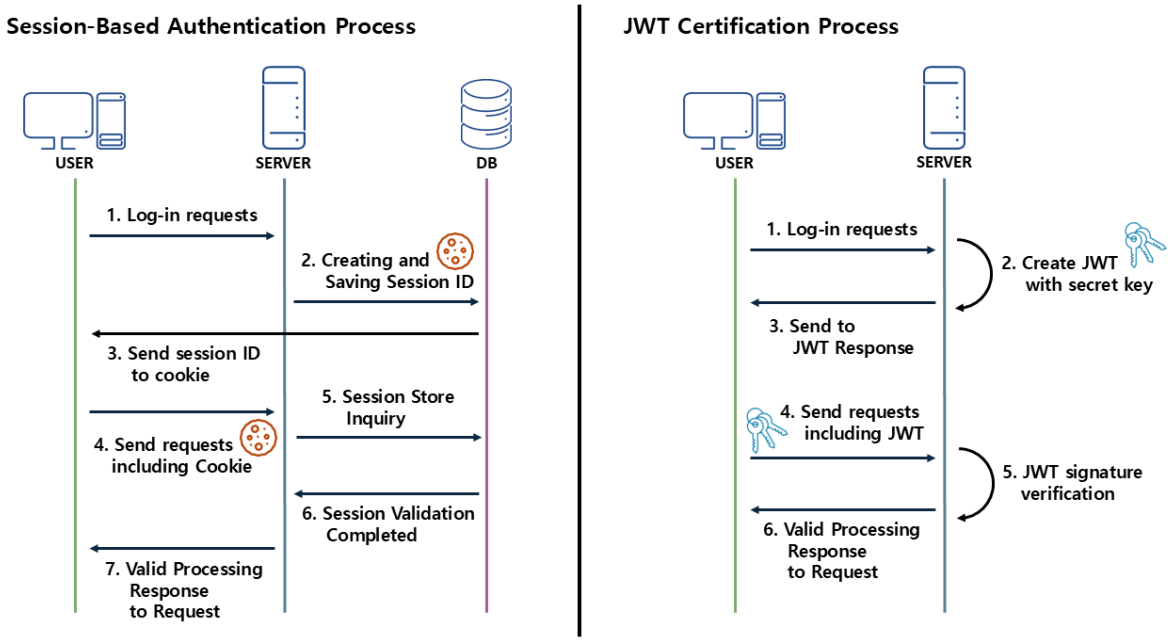

Figure 2. Comparison between session-based and JWT authentication

JWT authentication gained widespread adoption because it addresses the shortcomings of session-based authentication while offering greater convenience.

Session-based authentication

Session-based authentication has four major drawbacks:

- It follows a stateful architecture in which the server must manage session state.
- Session sharing (e.g., centralized session stores, sticky sessions) is required when horizontally scaling servers.
- There are additional load and operational costs for session storage systems

(databases, Redis, etc.).

- Dependence on a central session store reduces flexibility in microservices and distributed architectures.

JWT authentication

In contrast, JWT authentication mitigates many of the drawbacks of session-based authentication:

- Its stateless architecture enables basic authentication through signature verification alone without a separate session store.
- No session synchronization is required between servers or instances, making horizontal scaling easier.
- Each service can validate tokens independently, making JWT well-suited for microservices architectures.
- Tokens can include user identifiers, permissions, and expiration times, reducing dependence on server-side state.

JWT-related Vulnerabilities

JWT vulnerabilities typically arise when there are errors in the implementation of signature verification logic, or when the server blindly trusts user-controllable header parameters. The stateless nature of JWT - one of its key advantages - can paradoxically become a critical weakness if operational design is inadequate.

In traditional session-based authentication, forcing a logout is immediately effective once the server deletes the session. With JWT, however, the client retains the token, making it difficult for the server to revoke it instantly. Without a separate revocation mechanism, a token may remain valid until its expiration time. Even if an employee's privileges are changed or an account is deactivated, previously issued JWTs may still allow API access as long as they have not expired. Even when an account is compromised and a password is reset, any JWTs already obtained by an attacker can remain valid until expiration.

If a signing key is leaked, whether the system uses a symmetric-key algorithm such as HS256 or a public-key–based algorithm such as RS256, an attacker can generate JWTs that are syntactically and cryptographically valid using the same signing key as the server. If such forged tokens pass signature verification, the server and detection systems process the requests as legitimate traffic. As a result, malicious behaviors may only be detected after a large-scale data breach has already occurred. This poorly managed JWT and its vulnerabilities were the starting point of the Korean e-commerce platform's personal data breach incident.

Let's examine the major vulnerabilities associated with JWTs

1. Signature Verification Bypass

1) HS/RS algorithm confusion vulnerability (CVE-2015-9235)

JWT library for Node.js prior to v4.2.2 suffers from an algorithm confusion issue, in which tokens signed with a symmetric algorithm (HS) are accepted even when verification should be performed using an asymmetric algorithm (RS/ES). An attacker can abuse the public key used by the server for verification as an HMAC secret key, forge a token with an arbitrary payload, and bypass authentication or escalate privileges.

2) alg=none acceptance (CVE-2021-22160 / CVE-2022-23540 / CVE-2025-61152)

Some JWT implementations fail to verify signatures for tokens that include alg: "none", or allow none when verification options are not specified; this leads to signature verification bypass. Apache Pulsar was reported to skip signature verification when alg=none is used. The JWT library allowed verification bypass by processing 'none' as the default when algorithms were not explicitly specified in jwt.verify(). Python-jose had an issue where tokens with alg=none were accepted without signature verification.

3) JWT claim tampering (CVE-2022-39227)

Python-jwt prior to v3.3.4 suffers from inconsistencies between JWT compact serialization and JSON serialization processing, which cause discrepancies between the claims that pass signature verification and the claims actually used by the application. An attacker can manipulate claims and bypass authentication by reusing an existing token's signature without knowing the secret key. This vulnerability was rated critical with a CVSS score of 9.1.

2. Algorithm Confusion

1) JWT library vulnerability (CVE-2023-48238)

JWT library for Node.js prior to v3.1.1 has a flaw in which the algorithm used for verification is derived directly from the alg field in the JWT header. If a server uses RS256 and an attacker obtains the public key, the attacker can set the alg parameter to HS256 and misuse the public key as an HMAC secret key, allowing token forgery without the private key.

2. cjwt library vulnerability (CVE-2024-54150)

Version 2.2.0 of the C-based JWT library cjwt may fail to distinguish between signature schemes during token verification, leading to algorithm confusion. In configurations where the verification logic does not safely enforce the token's algorithm value and key type, an attacker can cause systems to process a public key as an HMAC key. This enables forged tokens to pass verification without the private key. This vulnerability was fixed in version 2.3.0.

3. Key Lookup Parameter Injection

1) kid parameter path search and SQL injection

If a server directly uses the kid value in file paths or database lookup queries, an adversary can inject path search strings (../../dev/null) or SQL statements. This can force the server to select arbitrary keys or disrupt the key lookup logic. As a result, token verification may be performed with an incorrect key, or sensitive information about the key store may be exposed.

4. Key Management Failure

1) Hardcoded secret keys (CVE-2025-7079 / CVE-2025-6950)

When signing keys are hardcoded into source code or firmware, attackers who obtain these values can forge arbitrary JWTs and bypass authentication. CVE-2025-7079 is an example in which the string "bluebell-plus" was hardcoded in the jwt.go file of bluebell-plus. CVE-2025-6950 reports the use of hardcoded JWT signing keys in Moxa network security appliances and routers.

2) iss claim format violation (CVE-2025-30144)

Fast-jwt prior to v5.0.6 contains a validation flaw that allows the iss claim, which must be a string under RFC 7519, to be a string array. An attacker can bypass verification logic by crafting an iss array that mixes legitimate and malicious issuers.

3) Insider abuse due to failure in signing key rotation and revocation

If an organization operates JWT signing keys for long periods without performing key rotation or revocation when personnel changes or privileges are adjusted, leaked keys can continue to be used to generate valid JWTs. As long as signature verification succeeds, servers and monitoring systems may misinterpret such requests as legitimate traffic. This can delay detection, as incidents may not be identified until after external reports or investigations.

Security Measures

Organizations must strengthen server-side verification logic and establish real-time intrusion detection capabilities to protect against all JWT-related breaches, including internal key leakage.

1. Secure Key Management System and HS256 Replacement

1) Using HSM or cloud-based KMS

A core strategy for mitigating insider threats is to isolate the signing key from the application server's file system and memory. Signing keys (secret or private keys) should be stored in a hardware security module (HSM) or a cloud-based key management service (KMS). The application server should never directly process key values; instead, it should invoke signing operations via APIs. This approach physically prevents insiders from obtaining key data even if they gain access to the server.

2) Transitioning to asymmetric keys (RS256) and enforcing key rotation

Organizations should prefer to use asymmetric key schemes (RS256) with a public/private key structure over single shared symmetric keys (HS256) to reduce insider threats and key exposure risks. The private key is securely stored within the HSM/KMS, while the public key can be distributed externally. As long as the private key remains confidential, adversaries cannot forge valid tokens. In addition, organizations should define key lifetimes and rotation schedules, assuming potential compromise. Also, they must establish operational procedures for regular key rotation and revocation.

2. Strict Server-Side Token Validation

When using JWT libraries, developers must explicitly block all potential risks identified by the standard specification.

1) Enforcing an algorithm whitelist

During JWT verification, the server must explicitly pin the allowed algorithms (e.g., RS256) in the library configuration. This prevents unsafe options such as alg=none and neutralizes algorithm confusion attack attempts.

2) Strictly validating critical claims

Organizations must thoroughly validate key claims such as exp (expiration), iss (issuer), and aud (audience). They should prevent the use of expired tokens via exp validation when receiving an access token.

3) Safely using key reference parameters

Dynamic key reference mechanisms such as kid, jku, and jwk should be avoided as they enable injection attacks. If their use is unavoidable, apply strong whitelist-based input filtering to the kid value, including detection and blocking of path traversal and SQL injection patterns.

3. Token Lifetime Management and Revocation

1) Access token lifetime management

To minimize impact if compromised, access tokens should have a very short expiration (exp), ideally within a few minutes. Long-lived sessions should be handled separately using refresh tokens. Protect refresh tokens with policies such as rotation and one-time use to prevent reuse.

2) Real-time token revocation

In cases of key leakage, build a centralized revocation list to invalidate tokens immediately. Each API request should verify the token's jti claim against the list and block the request if it is present. Manage revocation entries with TTL (Time To Live) so they expire automatically.

4. Enhancing Intrusion Detection and Monitoring

1) Detailed logging and SIEM integration

Log JWT verification failures in detail (e.g., invalid signatures, algorithm mismatches, claim errors) and integrate these logs into a SIEM system for monitoring and response.

2) Automated anomaly detection

Various anomalies may occur, such as excessive API calls using the same token within a short time frame, geographically abnormal access patterns, or the presence of tokens with unusually long expiration times. Organizations can address them by building systems that immediately detect such anomalies and generate alerts.

Conclusion

If organizations do not properly manage the security of a JWT-based authentication system, their core assets can be compromised on a massive scale. In particular, failures in signing key management can become a single point of failure that undermines the entire authentication framework. We recently witnessed it in the South Korean e-commerce platform incident.

Therefore, organizations should prioritize isolating signing keys within HSMs or KMSs and adopting asymmetric keys (RS256) along with robust key rotation policies. At the same time, they must implement strict verification logic - including algorithm pinning, validation of critical claims, and defenses against key reference input attacks - to prevent breaches from occurring. Finally, by building a security framework that integrates log-based detection with automated anomaly detection, organizations can identify signs of compromise early and reduce potential damage.

Analysis Report

A Shadow of JWT

AhnLab, Inc.
220, Pangyoyeok-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, 13493, Korea
Website: www.ahnlab.com
Purchase Inquiry: global.sales@ahnlab.com

AhnLab