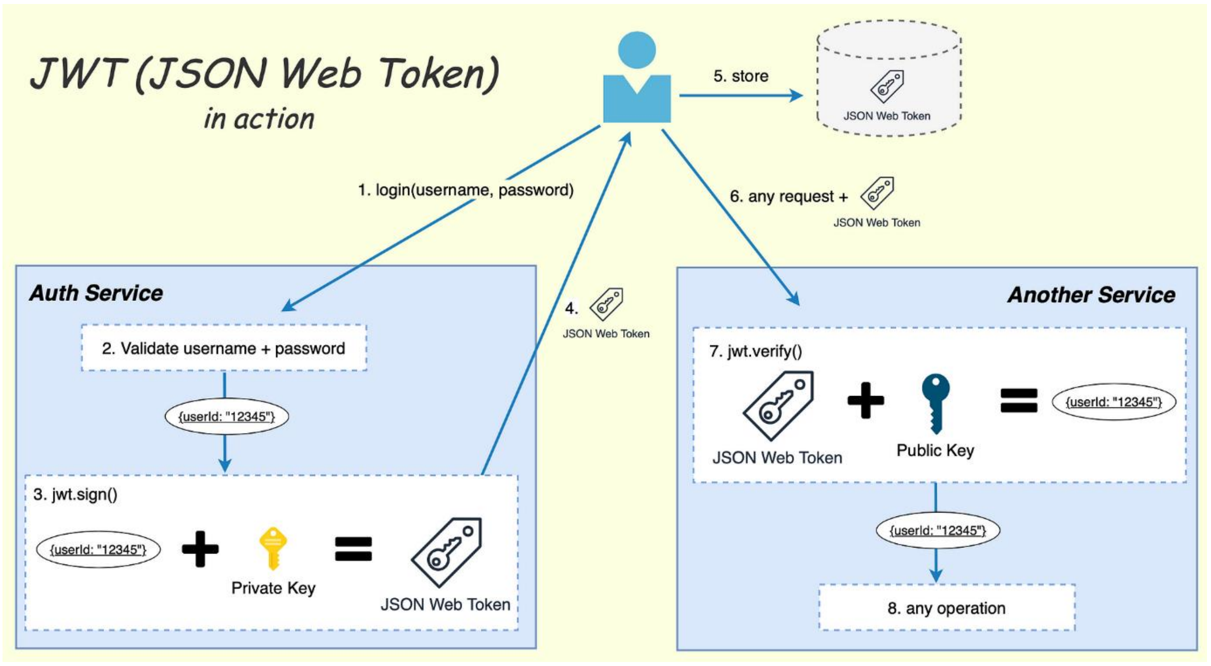
JWT HACKING

JSON Web Token (JWT)

Definition

A JSON Web Token (JWT) is a compact, URL-safe, and self-contained token format used to securely transmit information between two parties (usually a client and a server). It is often used for authentication and authorization mechanisms, especially in web applications.

JWT is a token that is digitally signed, ensuring the authenticity of the token's content. It can also be encrypted to protect sensitive information. The token typically contains claims about the user or session and is used to verify the user's identity without needing to query a database on every request.



Structure of a JWT

A JWT consists of three parts, separated by dots (.), and encoded in Base64:

1. Header
2. Payload
3. Signature

A JWT looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ
SI6IkpvbiBEb2UiLCJpYXQiOjE1MTYyMzkwMjJ9.SflKxwRJSMeKKF2QT4fwpMeJf3
6POk6yJV_adQssw5c
```

1. Header

The header typically consists of two parts:

- Type of the token (always JWT).
- Signing algorithm used (e.g., HMAC, SHA256, RSA).

Example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. Payload

The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims:

- Registered claims: Predefined claims with common uses, such as iss (issuer), exp (expiration time), sub (subject), and aud (audience).
- Public claims: Claims that can be defined by the user, like name, email, or role.
- Private claims: Custom claims that are used to share information between parties that agree on their usage.

Example:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

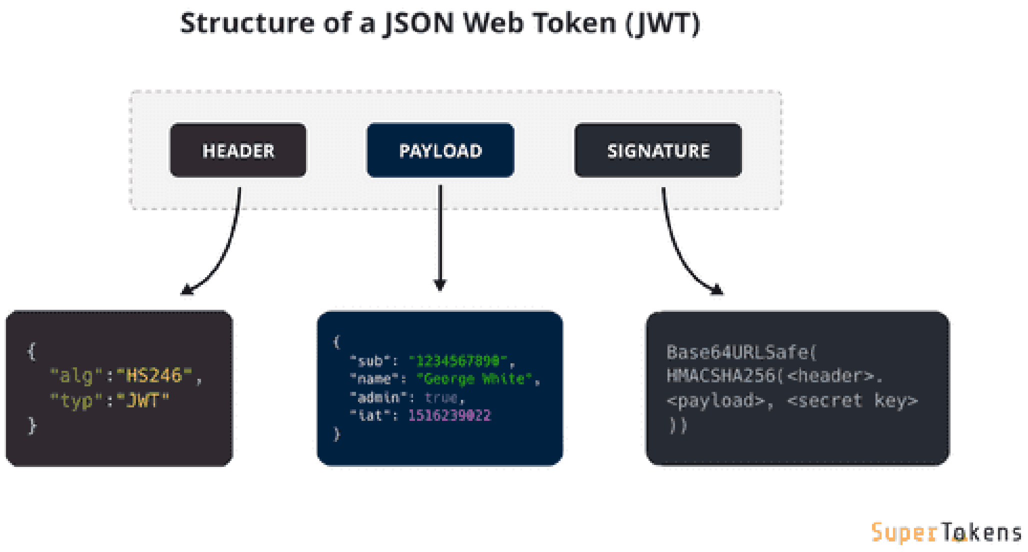
3. Signature

The signature is used to verify that the token has not been altered and to ensure the authenticity of the sender. To create the signature, the encoded header, payload, and a secret key are combined and passed through a cryptographic hashing algorithm.

Example (using HMAC SHA256):

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

If the token is signed with an asymmetric algorithm (e.g., RSA), a private key is used to sign the token, and a public key is used to verify the signature.



```
{
  "alg":"HS246",
  "typ":"JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "George White",
  "admin": true,
  "iat": 1516239022
}
```

```
Base64URLSafe(
HMACSHA256(<header>.
<payload>, <secret key>
))
```

How JWT Works in Authentication

1. **Client Authentication:**
   - When a user logs in, the server verifies their credentials and generates a JWT.
   - The JWT is sent back to the client (usually in an HTTP response).

2. **Client Stores the JWT:**
    - The client stores the JWT (usually in local storage or a cookie).
3. **Subsequent Requests:**
    - For every subsequent request to the server, the client includes the JWT (usually in the HTTP Authorization header).
    - Example:

```
Authorization: Bearer <JWT>
```

4. **Server Verifies the JWT:**
    - The server verifies the signature of the JWT using a secret or public key.
    - If valid, the server extracts the claims (e.g., user identity, roles) from the token to grant or deny access.
5. **Token Expiration:**
    - JWTs often include an exp (expiration) claim. After the expiration time, the token becomes invalid, and the user must log in again.

Benefits of JWT

1. Stateless: JWTs are self-contained and do not require the server to store session data, making them ideal for distributed applications (e.g., microservices).
2. Compact: JWTs are lightweight and can easily be transmitted via URL, HTTP headers, or inside request bodies.
3. Secure: JWTs are signed, ensuring their integrity. They can also be encrypted for additional confidentiality.
4. Easy to Use: JWTs are easy to generate and verify using common libraries in most programming languages.

Use Cases

1. **Authentication:**
   - JWT is often used to authenticate users by sending a token that represents the user's identity. This is the basis of many single sign-on (SSO) mechanisms.

2. **Authorization:**
   - After a user is authenticated, each subsequent request can include the JWT to verify their identity and access level without needing to re-authenticate on each request.
3. **Information Exchange:**
   - JWTs can be used to securely transmit information between parties. Since they can be signed, the parties involved can verify that the data has not been tampered with.

Security Considerations

1. Keep Secrets Safe: If using symmetric algorithms (e.g., HMAC), ensure that the secret key is securely stored.
2. Use HTTPS: Always send JWTs over HTTPS to prevent man-in-the-middle (MITM) attacks.
3. Short Expiration Times: Ensure tokens have a reasonable expiration time to limit the impact of token theft.
4. Revoking Tokens: Since JWTs are stateless, they cannot be easily revoked. Use mechanisms like blacklists or short-lived tokens to mitigate this risk.

JWT Token Hacking and Types of Attacks

While JSON Web Tokens (JWT) are designed to provide secure authentication and data exchange, they are not immune to attacks, especially if improperly implemented. Attackers can exploit weaknesses in the handling, creation, or validation of JWTs to gain unauthorized access, manipulate data, or impersonate users.

Common JWT Attacks

1. None Algorithm Attack (Algorithm Confusion)

One of the most notorious vulnerabilities in JWT implementation is improper use of the "none" algorithm in the token's header. The "none" algorithm is meant to indicate that the JWT should not be signed.

Vulnerability

- If the server incorrectly accepts tokens with the "none" algorithm, an attacker can modify the token's payload and send it back to the server with no signature. Since no validation occurs, the server may accept the modified token as legitimate.

Example:

- Original JWT header:

```
{
 "alg": "HS256",
 "typ": "JWT"
}
```

- Modified JWT header with "none":

```
{
 "alg": "none",
 "typ": "JWT"
}
```

How the Attack Works:

- The attacker can decode the JWT, modify the payload (e.g., change user roles or privileges), change the algorithm to "none," and send the altered token back to the server. If the server doesn't verify the signature, it will accept the malicious token.

Mitigation:

- Always configure the server to reject JWT tokens using the "none" algorithm. Only allow JWTs signed with valid algorithms (e.g., HMAC, RSA).

2. Brute-Forcing the Secret Key (HMAC Secret Key Attack)

JWTs signed with the HMAC algorithm (e.g., HS256) rely on a secret key shared between the client and server. If this key is weak, attackers can use brute force techniques to guess the key, allowing them to forge valid tokens.

Vulnerability:

- If a weak or easily guessable secret key is used (e.g., "password123"), an attacker can use brute-force or dictionary attacks to find the key and then sign their own JWT tokens.

How the Attack Works:

- The attacker captures a JWT, and then uses a brute force tool to guess the secret key. Once the key is discovered, they can modify the token payload (e.g., change roles or permissions) and sign the new token with the guessed key. The server, unable to detect that the key was compromised, will accept the forged token.

Mitigation:

- Use a strong, complex secret key that is difficult to brute-force.
- Rotate keys periodically to reduce the window of opportunity for attackers.

3. JWT Signature Replay Attack

JWT tokens are often used in stateless authentication, meaning they are not stored on the server after being issued. This makes JWTs vulnerable to replay attacks, where an attacker intercepts and reuses a valid JWT token to authenticate themselves.

<u>Vulnerability:</u>

- JWTs, especially long-lived tokens, can be intercepted and reused by an attacker, potentially gaining unauthorized access to sensitive data or resources.

<u>How the Attack Works:</u>

- An attacker captures a legitimate JWT (e.g., through a man-in-the-middle attack or a vulnerable client). Since the JWT is stateless, the server has no way of knowing if the token has been used before. The attacker can reuse the token to access protected resources as long as the token is valid.

<u>Mitigation:</u>

- Implement short expiration times (exp claim) for JWTs, forcing users to refresh tokens regularly.
- Use one-time-use tokens or session management techniques to prevent token replay.
- Implement token revocation mechanisms (e.g., blacklist or maintain server-side sessions) for critical operations.

4. <u>JWT Claim Tampering (Manipulating Claims)</u>

The payload of a JWT is **not** encrypted (unless using JWE). This means attackers can view and potentially manipulate the claims if the signature verification is improperly implemented or ignored.

<u>Vulnerability:</u>

- An attacker can modify claims in the payload (such as user roles or permissions) and attempt to sign the token with their own key or exploit weak signature verification to bypass security checks.

<u>How the Attack Works:</u>

- The attacker decodes the JWT payload and modifies claims (e.g., changing the role from "user" to "admin"). If the server does not properly verify the JWT's signature, it will accept the altered token and grant elevated privileges.

Mitigation:

- Ensure that signature verification is always enforced.
- Avoid storing sensitive data directly in the JWT payload. Use encrypted JWTs if necessary.
- Validate all claims on the server, including iss (issuer), aud (audience), and exp (expiration time).

5. Key Confusion Attack (Using Public Key as HMAC Secret)

This attack happens in systems that use **RSA** or **ECDSA** for JWT signatures. An attacker can attempt to use the public key (intended for verifying the signature) as the HMAC secret key for signing a token.

Vulnerability:

- If the server mistakenly uses the same key for both HMAC signing and RSA verification, attackers can use the public RSA key as the HMAC secret key to forge valid tokens.

How the Attack Works:

- The attacker obtains the public RSA key, and then signs a JWT using HMAC with the public key as the secret. If the server misconfigures the signature verification process, it might accept this token as valid, despite the attacker using the public key to sign it.

Mitigation:

- Use different keys for signing and verifying in RSA/ECDSA and HMAC algorithms.
- Ensure that the server is configured to distinguish between RSA/ECDSA and HMAC-based JWTs.

6. Cross-Site Scripting (XSS) Attack

JWTs stored in client-side storage (e.g., local storage, session storage, or cookies) can be vulnerable to **Cross-Site Scripting (XSS)** attacks if an attacker injects malicious JavaScript into the web application to steal the token.

Vulnerability:

- If an attacker injects a script into the web page (via XSS), they can steal the JWT token from local storage or cookies and impersonate the user.

How the Attack Works:

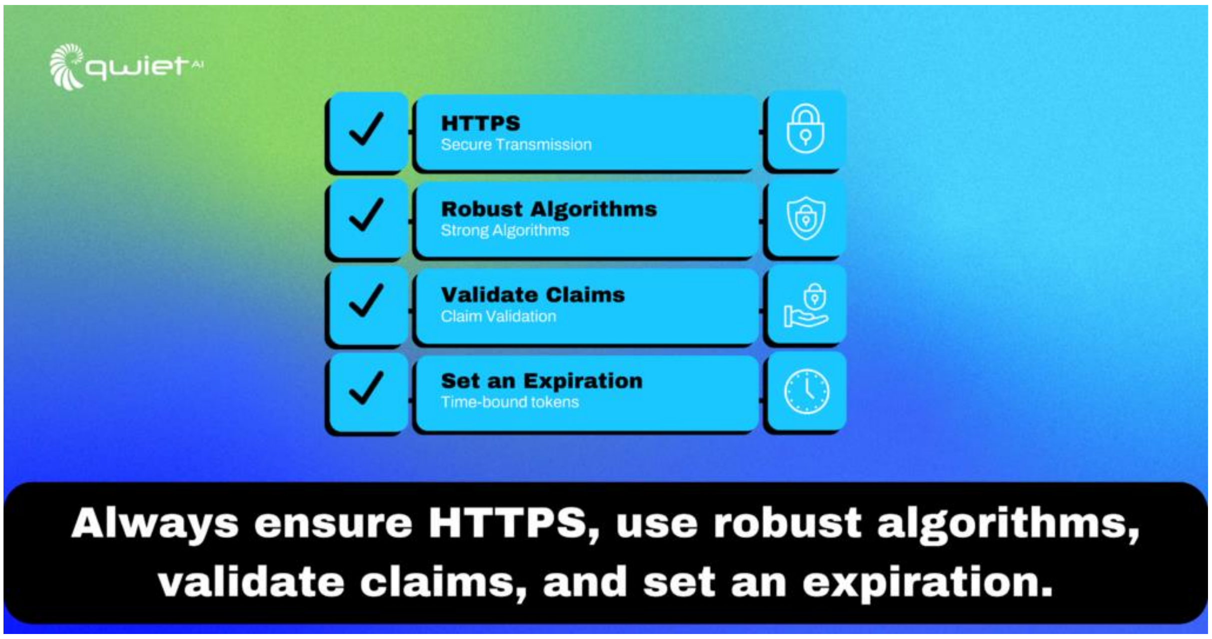
- The attacker finds an XSS vulnerability in the web application and injects malicious JavaScript. The script can access the JWT stored in localStorage or cookies and send it to an attacker-controlled server. The attacker can then use the stolen JWT to authenticate as the victim.

Mitigation:

- Store JWTs in **HTTP-only cookies** (prevents JavaScript access to tokens).
- Sanitize user inputs to prevent XSS attacks.
- Implement Content Security Policy (CSP) to mitigate XSS risks.

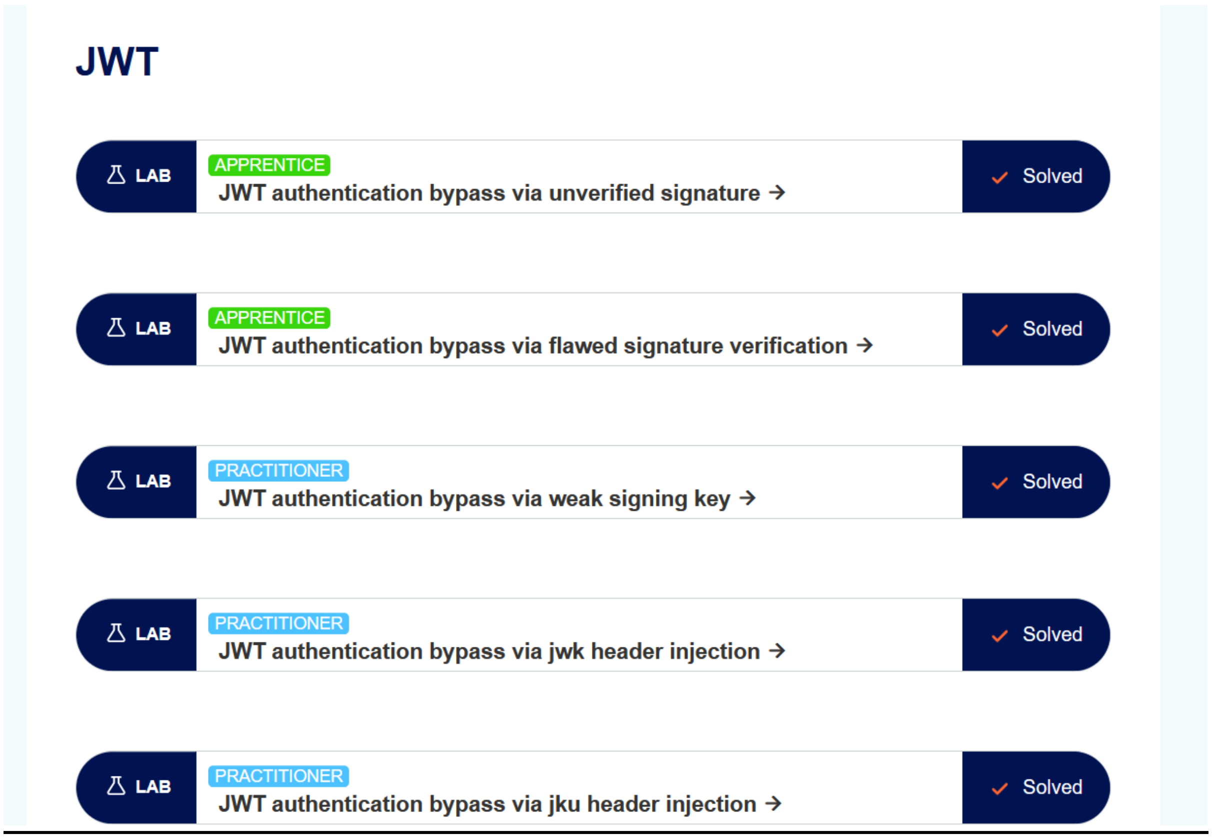
Best Practices to Secure JWTs

- **Use Strong Secrets**: Use a complex and long secret key when using HMAC algorithms to prevent brute-force attacks.
- **Short Token Expiry**: Set short expiration times (exp) for tokens, reducing the window of vulnerability if a token is stolen.
- **Signature Verification**: Always verify the token signature on the server and ensure the correct algorithm is used.
- **Encryption**: If sensitive data is being transmitted, consider encrypting the JWT using JSON Web Encryption (JWE).
- **Token Revocation**: Implement mechanisms to revoke tokens, such as blacklists, or maintain server-side sessions for critical operations.
- **HTTP-Only Cookies**: Store tokens in secure HTTP-only cookies to prevent JavaScript access and mitigate XSS risks.



PRACTICAL

https://portswigger.net/web-security/jwt/lab-jwt-authentication-bypass-via-unverified-signature



JWT

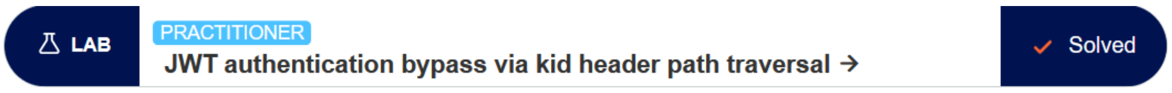
LAB APPRENTICE JWT authentication bypass via unverified signature → ✓ Solved

LAB APPRENTICE JWT authentication bypass via flawed signature verification → ✓ Solved

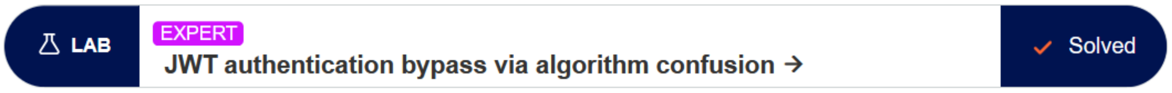
LAB PRACTITIONER JWT authentication bypass via weak signing key → ✓ Solved

LAB PRACTITIONER JWT authentication bypass via jwk header injection → ✓ Solved

LAB PRACTITIONER JWT authentication bypass via jku header injection → ✓ Solved



LAB PRACTITIONER JWT authentication bypass via kid header path traversal → ✓ Solved



LAB EXPERT JWT authentication bypass via algorithm confusion → ✓ Solved

REFERENCES

https://jwt.io/introduction

https://auth0.com/docs/secure/tokens/json-web-tokens/json-web-token-structure

https://supertokens.com/blog/what-is-jwt

https://www.geeksforgeeks.org/json-web-token-jwt/

https://fusionauth.io/articles/tokens/jwt-components-explained

https://www.invicti.com/blog/web-security/json-web-token-jwt-attacks-vulnerabilities/

https://www.qrcsolutionz.com/blog/json-token-hacking-risks-impacts-mitigation

https://www.redsentry.com/blog/decoding-jwt-vulnerabilities-a-deep-dive-into-jwt-security-risks-and-mitigation