# Securing APIs with JWT Tokens: A Detailed Guide

Shubham Kumar
imshubhamkumar

## 1 Introduction

In the era of rapid digitalization and web-based applications, securing APIs (Application Programming Interfaces) is of paramount importance. APIs, which serve as interfaces between different software systems, often need to handle sensitive data and thus require robust security mechanisms.

JSON Web Tokens (JWTs) have emerged as a standard tool for securing information transmission between parties in the Java ecosystem. JWT is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. The compact size of JWTs makes them especially suitable for single sign-on (SSO) scenarios and the likes, where they can be easily passed around in HTTP headers and URL parameters.

JWTs are designed in a way that they can be digitally signed or encrypted. When signed using a secret key, the receiver can verify whether the token was issued by a trusted party. This is particularly useful in scenarios where you want to authenticate the client making the request to your Java application.

JWTs encapsulate a series of claims in a JSON object. These claims are statements about an entity (typically, the user) and additional metadata. The server can use these claims to validate the identity of the client and vice-versa, ensuring a secure information exchange.

Java provides extensive support for JWTs through libraries like JJWT (Java JWT: JSON Web Token for Java and Android) and others. These libraries provide a fluent API to create, decode, verify and sign JWTs in Java applications, thus simplifying the process of securing APIs and managing user authentication.

In the subsequent sections of this document, we will detail the workflow of JWTs, discuss how to maximize API security using JWTs and associated best practices, and provide Java code examples for better understanding. This guide is intended for Java developers and assumes basic familiarity with Java, JWTs, and API development.

## 2 JWT Workflow

The JWT workflow in a typical Java web application can be broken down into the following steps:

1. **User Authentication:** The process begins when the user logs in with their credentials (username and password). The server then verifies these credentials against a database or another kind of user store.

2. **Token Generation:** Once the user is authenticated, the server generates a JWT. This is typically accomplished using a library such as JJWT in Java. The server places a set of claims in the JWT, which may include the user's ID, username, roles, and any other information that should be transmitted. The

JWT is then signed using a secret key known only to the server. This ensures that the token can't be altered in transit without the server knowing.

```
String jwt = Jwts.builder().setSubject("username")
                .claim("roles", "user")
                .setIssuedAt(new Date())
                .signWith(SignatureAlgorithm.HS256, "secretkey")
                .compact();

```

3. **Token Transmission:** The server sends the generated JWT back to the client. The token can be sent in the body of the response, in a cookie, or even as a parameter in the URL (though this is less secure and not generally recommended).

4. **Token Storage:** Upon receiving the JWT, the client stores it in a secure manner. This could be in memory, local storage, or secure HttpOnly cookies, depending on the requirements of the application.

5. **Token Usage:** For subsequent requests to the server, the client includes the JWT in the Authorization header of the HTTP request. The server can then authenticate the user's requests without requiring them to repeatedly supply their username and password.

6. **Token Verification:** Upon receiving a request with a JWT, the server first verifies the token. This involves decoding the JWT and checking the signature using the server's secret key. If the token has been tampered with or was issued by an unknown party, the signature will not match and the server will know to reject the request.

```
Jws<Claims> jws;
try {
    jws = Jwts.parserBuilder()
        .setSigningKey("secretkey")
        .build()
        .parseClaimsJws(jwt);
} catch (JwtException ex) {
    // Token is invalid or not trusted
}

```

7. **Request Processing:** If the JWT is valid, the server processes the request. The server can trust the claims in the JWT (like the user's ID and roles) and use this information to perform the necessary operations. For example, it can use the user's ID to fetch user-specific data, or check the user's roles to authorize certain actions.

This JWT-based authentication workflow ensures that the server remains stateless (it doesn't need to remember any user state between requests) while still providing a secure method for users to authenticate and interact with the server.

# 3 Maximizing API Security

Maximizing API security is a multi-faceted process. While JWTs provide one layer of security, they are just one tool in the toolbox. Here are some additional strategies, along with Java-oriented examples, that can be employed to maximize the security of your APIs:

## 3.1 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is the secure version of HTTP, the protocol over which data is sent between a browser and the website that it is connected to. The 'S' at the end of HTTP stands for

'Secure', which means that all communications between the browser and the website are encrypted. HTTPS uses the SSL/TLS protocol to provide this encryption layer.

For Java-based web services, the standard way to handle HTTPS is by using Java Secure Socket Extension (JSSE). This includes managing certificates, creating SSL contexts, and handling the handshake between client and server.

Here is an example of how to create a simple HTTPS server in Java using HttpsServer provided by the com.sun.net library.

```java
import com.sun.net.httpserver.*;

import javax.net.ssl.*;
import java.io.*;
import java.net.InetSocketAddress;
import java.security.*;

public class SimpleHTTPSServer {

    public static void main(String[] args) throws Exception {

        // Setup the socket address
        InetSocketAddress address = new InetSocketAddress(8000);

        // Initialise the HTTPS server
        HttpsServer httpsServer = HttpsServer.create(address, 0);
        SSLContext sslContext = SSLContext.getInstance("TLS");

        // Initialise the keystore
        char[] password = "password".toCharArray();
        KeyStore ks = KeyStore.getInstance("JKS");
        FileInputStream fis = new FileInputStream("testkey.jks");
        ks.load(fis, password);

        // Setup the key manager factory
        KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
        kmf.init(ks, password);

        // Setup the trust manager factory
        TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
        tmf.init(ks);

        // Setup the HTTPS context and parameters
        sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
        httpsServer.setHttpsConfigurator(new HttpsConfigurator(sslContext) {
            public void configure(HttpsParameters params) {
                try {
                    // Initialise the SSL context
                    SSLContext c = getSSLContext();
                    SSLEngine engine = c.createSSLEngine();
                    params.setNeedClientAuth(false);
                    params.setCipherSuites(engine.getEnabledCipherSuites());
                    params.setProtocols(engine.getEnabledProtocols());

                    // Get the default parameters
                    SSLParameters defaultSSLParameters = c.getDefaultSSLParameters();
                    params.setSSLParameters(defaultSSLParameters);
                } catch (Exception ex) {
                    System.out.println("Failed to create HTTPS port");
                }
            }
        });
        httpsServer.createContext("/test", new MyHandler());
        httpsServer.setExecutor(null); // creates a default executor
        httpsServer.start();
    }

    static class MyHandler implements HttpHandler {
        public void handle(HttpExchange t) throws IOException {
```

```
            String response = "Welcome to the secure server!";
            t.sendResponseHeaders(200, response.length());
            OutputStream os = t.getResponseBody();
            os.write(response.getBytes());
            os.close();
        }
    }
}
```

## 3.2 Token Storage

The way in which the client stores the JWT is also critical. Storing the JWT in an unsecured manner can lead to token theft. In JavaScript web applications, for example, it's often best to store the JWT in a secure HttpOnly cookie.

## 3.3 Token Expiry

Implementing short-lived tokens and a token refresh strategy can limit the potential misuse of a compromised token. In Java, you can set the expiration of a JWT using the setExpiration() method.

```
Date now = new Date();
Date expiry = new Date(now.getTime() + 3600000); // 1 hour expiry

String jwt = Jwts.builder()
    .setSubject("username")
    .setIssuedAt(now)
    .setExpiration(expiry)
    .signWith(SignatureAlgorithm.HS256, "secretkey")
    .compact();
```

## 3.4 Secure Cookies

A cookie is a small piece of data stored in the user's web browser while the user is browsing a particular website. Cookies are designed to be a reliable mechanism for websites to remember stateful information, such as items added in a shopping cart, or to record the user's browsing activity.

However, cookies can also pose a security risk if not handled properly, as they can be used to track and collect personal information about users without their consent. Furthermore, cookies can be stolen and used to impersonate users, leading to potential security breaches.

To mitigate these risks, secure cookies are often used in sensitive applications. There are several key attributes that make a cookie secure:

### 3.4.1 Secure Attribute

The secure attribute is an option that can be set by the server when sending a new cookie to the user's browser. It instructs the browser to only send the cookie over an encrypted HTTPS connection. If a browser receives a cookie with the secure attribute, it will only include that cookie in an HTTP request if the request is transmitted over a secure (HTTPS) connection.

### 3.4.2 HttpOnly Attribute

The HttpOnly attribute is another option that can be set by the server when sending a new cookie. It instructs the browser to prevent access to the cookie from client-side scripts. This makes the cookie less likely to be stolen through cross-site scripting (XSS) attacks.

### 3.4.3 SameSite Attribute

The SameSite attribute prevents the browser from sending the cookie along with cross-site requests, which provides some protection against cross-site request forgery (CSRF) attacks.

Here's how you can set a secure cookie in Java:

```
Cookie cookie = new Cookie("key", "value");
cookie.setSecure(true);
cookie.setHttpOnly(true);
cookie.setPath("/");
response.addCookie(cookie);
```

In this code snippet, 'response' is an instance of 'HttpServletResponse'. The 'setSecure(true)' and 'setHttpOnly(true)' calls ensure that the cookie is only sent over HTTPS and cannot be accessed through client-side scripts, respectively.

Remember that secure cookies are not a panacea. They are one of many tools that you can use to secure your application, and they should be used in conjunction with other security best practices.

## 3.5 Encryption

Encryption is the process of converting plaintext into ciphertext, making it unreadable to anyone except those possessing the key to decrypt it. When dealing with JWTs, encryption is a critical factor to consider, especially when sensitive data is being transmitted.

In the context of JWTs, there are two types of encryption that can be applied:

### 3.5.1 Payload Encryption

This is where the data within the JWT payload is encrypted. This adds an additional layer of security as even if someone intercepts the token, they will not be able to understand the information within the payload without the decryption key. In Java, the Java Cryptography Extension (JCE) can be used for this purpose.

```
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
byte[] encryptedPayload = cipher.doFinal(payload.getBytes());
```

### 3.5.2 Token Encryption

This is where the entire JWT is encrypted. This means that all three parts of the JWT (header, payload, and signature) are encrypted. This ensures that without the decryption key, an attacker cannot determine anything about the token, not even its type.

Java JOSE + JWT library provides support for JWT encryption using a variety of algorithms.

```
JWEObject jweObject = new JWEObject(
    new JWEHeader.Builder(JWEAlgorithm.RSA_OAEP_256, EncryptionMethod.A128GCM)
        .contentType("JWT")
        .build(),
    new Payload(jwt));

jweObject.encrypt(new RSAEncrypter(publicRSAKey));

String encryptedToken = jweObject.serialize();
```

In both code snippets, 'secretKey', 'payload', 'publicRSAKey', and 'jwt' are placeholders and should be replaced with your actual values.

These methods provide an extra layer of security to your JWTs, but also come with added complexity and processing overhead. The decision to use them should be based on the sensitivity of the data you are dealing with, and the security requirements of your application.

## 3.6 Incident Response

Incident response is an organized approach to addressing and managing the aftermath of a security breach or cyberattack, also known as an IT incident, computer incident, or security incident. The goal is to handle the situation in a way that limits damage and reduces recovery time and costs.

An incident response plan includes a set of instructions to detect, respond to, and recover from network security incidents. These types of plans address issues like malware, data loss and service outages.

An incident response plan often includes the following stages:

### 3.6.1 Preparation

This involves establishing an incident response team and equipping them with the right tools and resources to effectively handle an incident. Training is also provided to ensure the team can identify signs of an incident and respond appropriately.

### 3.6.2 Identification

This stage involves detecting and acknowledging that a security incident has occurred. Effective identification often relies on monitoring, detection tools, and employee vigilance.

### 3.6.3 Containment

Once an incident is identified, steps must be taken to limit the damage and prevent further harm. This may involve isolating affected systems or networks to prevent the threat from spreading.

### 3.6.4 Eradication

After containment, the threat must be completely removed from the system. This could involve the use of antivirus tools, deletion of malicious files or reformatting infected systems.

### 3.6.5 Recovery

During recovery, systems and devices are restored to normal operation, and confidence is established that the system is no longer compromised. It's important to note, recovery may involve a phased return to normal operations as systems are confirmed to be secure.

### 3.6.6 Lessons Learned

After the incident is handled, the team should meet for a post-mortem to document what occurred during the incident, what steps were taken to resolve it, how effective those steps were, and what can be done to prevent a similar occurrence in the future.

The incident response process is a crucial aspect of any organization's security posture. It can mean the difference between a minor security incident and a catastrophic breach that results in significant data loss and damage to the organization's reputation.

# 4 Conclusion

In the world of web development, security is a paramount concern. The use of JWTs for secure API interactions has emerged as a standard practice due to its numerous advantages, such as statelessness, simplicity, and compactness. However, it's important to understand that JWTs also come with their own set of security considerations.

We've explored how to secure APIs with JWT tokens, including token encryption and the use of secure cookies. Encryption, whether at the payload or token level, can add an additional layer of security, making it difficult for attackers to decipher the contents of a token, even if they manage to intercept it.

Secure cookies, on the other hand, provide a mechanism for securely storing tokens in a user's browser. By leveraging attributes such as Secure, HttpOnly, and SameSite, we can reduce the risk of token theft through common web attacks like XSS and CSRF.

However, it's crucial to remember that these techniques are not foolproof and should not be used in isolation. They are part of a broader security strategy that should also include practices such as regular security audits, up-to-date software, secure coding practices, and user education.

In conclusion, securing APIs is a complex task that requires a deep understanding of various security principles and techniques. While JWTs offer a robust solution for securing APIs, they must be implemented and used correctly to ensure the security of your application.