**Description**

Broken JSON Web Token (JWT) attacks are a type of API security vulnerability that fall under the broad OWASP Top 10 Broken Authentication category of security risks. They occur when JWT authentication mechanisms fail, enabling malicious actors to craft tokens and impersonate the user of a web application.

JWT is an open internet standard representing claims to be transferred securely, with optional JSON Web Signature (JWS) and JSON Web Encryption (JWE), between parties as a JSON payload. JWTs have become ubiquitous in their use, being widely implemented in commercial applications due to their lightweight nature and digital signature enabling verification.

This widespread adoption of JWTs as a de facto session and token-based authentication standard is arguably justified if implemented properly and where appropriate (in many cases, they may be less fit for purpose than more traditional session management systems). However, they often aren't implemented correctly and thus persist as a favoured entry point for malicious actors.

**Impact**

An attacker executing a successful Broken JWT attack will enjoy the same privileges on the web application as the user he / she is impersonating. Depending on the level of access and the setup of the application, an attacker could gain complete access to all data in the web application, assume administrator rights, and compromise the confidentiality, integrity, and availability of the application.

**Scenario**

A JWT has the following form:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImpvaG4uZG9lIn0.zTHtFkPafeTIbVxBnG5f7PREr5DCStRY-Xtd6\_9iA3k

It comprises three parts separated by ., each one encoded in Base64. The first part is the header that contains the algorithm and the token type:

{

"alg": "HS256",

"typ": "JWT"

}

The second part contains non-sensitive arbitrary data chosen by the web application, for example:

{

"username": "john.doe"

}

The last part is the *signature*, that is the hashing of the first two parts using a *secret* as a key. For example, in the above JWT, the secret is $uper$secret, thus:

BASE64(HMACSHA256('eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImpvaG4uZG9lIn0', '$uper$secret')) => jxWz0c/9YC3INAlpUGegBtTgmDUA12Krsjtr1P2qgbw

There are numerous attack vectors malicious actors can choose from when attempting to break JWTs, including:

* None Algorithm Attack: an attacker modifies or creates a token by setting alg (the signing algorithm) to None in the header. Since 'alg' is 'None', this is a valid JWS.
* HMAC Algorithm Attack: an attacker creates the token by setting the signing algorithm to a symmetric HS256 instead of an asymmetric RS256, leading the API to blindly verify the token using the HS256 algorithm using the public key as a secret key. Since the public key is known, the attacker can correctly forge valid JWTs.
* Developers may not even validate the signature at all, trusting the signature passed in the Header.

There are usually third-party libraries that implement the JWT authentication mechanism; however, developers are discouraged from crafting their own ad-hoc solutions due to complexity and potential for error. Yet it is still possible to misconfigure the library and introduce vulnerabilities.

The most common mistake is to inadvertently disclose, or not choose a strong enough JWT secret.

**Prevention**

Developers can effectively mitigate algorithm vulnerabilities by strictly adhering to several precautions:

* Developers must never let the header of the JWT alone drive verification.
* Developers must know how the algorithms are structured.
* Developers must ensure an appropriate, strong key size is used.

Developers must use a JWT library that is not exposed to known vulnerabilities.

In cases where there is no built-in token revocation by the user, a token deny list can be implemented which mirrors the application of a traditional session management cookie.

**Testing**

Verify that stateless session tokens use digital signatures, encryption, and other countermeasures to protect against tampering, enveloping, replay, null cipher, and key substitution attacks.

* **OWASP ASVS**: [3.5.3, 3.2.4](https://github.com/OWASP/ASVS/releases/download/v4.0.2_release/OWASP.Application.Security.Verification.Standard.4.0.2-en.pdf)
* **OWASP Testing Guide**: [API Testing - (Token-Based) Authentication](https://github.com/OWASP/wstg/blob/master/Testing_for_APIs.md#specific-testing--token-based-authentication)

**.NET**

**Prevention**

ASP.NET Core natively provides JWT authentication through the middleware [Microsoft.AspNetCore.Authentication.JwtBearer](https://www.nuget.org/packages/Microsoft.AspNetCore.Authentication.JwtBearer/) package.

The JWT bearer authentication can be enabled in the app's Startup.Configure method by using the UseJwtBearerAuthentication extension method, or by setting the scheme to JwtBearerDefaults.AuthenticationScheme using services.AddAuthentication().

The authentication options TokenValidationParameters describe how JWT tokens will be validated. The class supports a large number of parameters, but minimal secure settings are illustrated in the code snippet below:

**var** tokenValidationParameters = **new** TokenValidationParameters

{

RequireSignedTokens = true,

IssuerSigningKey = **new** SymmetricSecurityKey(key),

ValidateIssuer = false,

ValidateLifetime = true,

ValidateAudience = false

};

* RequireSignedTokens indicates whether a SecurityToken can be considered valid if not signed. Set to true to enable basic validation for JWT tokens.
* IssuerSigningKey is the key used for validating incoming JWT tokens. By specifying a key here, the token can be validated without assistance from the issuing server.
* ValidateIssuer enforces issuer validation.
* ValidateLifetime enforces lifetime validation.
* ValidateAudience enforces audience validation.

It's vital that if a symmetric key is used as IssuerSigningKey, it must be a strong secret with sufficient entropy to avoid being compromised by attackers forging valid tokens.

**References**

[Auth0 - Critical vulnerabilities in JSON Web Token libraries](https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/)  
[Auth0 - JWT Debugger](https://jwt.io/#debugger-io)  
[MSDN - JWT Validation and Authorization in ASP.NET Core](https://devblogs.microsoft.com/aspnet/jwt-validation-and-authorization-in-asp-net-core/)