JWT attacks

Design issues and flawed handling of JSON web tokens (JWTs) can leave websites vulnerable to a variety of high-severity attacks.

JWTs are most commonly used in authentication, session management, and access control mechanisms, these vulnerabilities can potentially compromise the entire website and its users.

What are JWTs?

JSON web tokens (JWTs) are a standardized format for sending cryptographically signed JSON data between systems. They can theoretically contain any kind of data, but are most commonly used to send information ("claims") about users as part of authentication, session handling, and access control mechanisms.

JWT format

A JWT consists of 3 parts: a header, a payload, and a signature.

```
eyJraWQiOiI5MTM2ZGRiMy1jYjBhLTRhMTktYTA3ZS1lYWRmNWE0NGM4YjUiLCJhbGciOiJSUzI1NiJ9.eyJpc3
MiOiJwb3J0c3dpZ2dlciIsImV4cCI6MTY0ODAzNzE2NCwibmFtZSI6IkNhcmxvcyBNb250b3lhIiwic3ViIjoiY
2FybG9zIiwicm9sZSI6ImJsb2dfYXV0aG9yIiwiZW1haWwiOiJjYXJsb3NAY2FybG9zLW1vbnRveWEubmV0Iiwi
aWF0IjoxNTE2MjM5MDIyfQ.SYZBPIBg2CRjXAJ8vCER0LA_ENjII1JakvNQoP-Hw6GG1zfl4JyngsZReIfqRvIA
Ei5L4HV0q7_9qGhQZvy9ZdxEJbwTxRs_6Lb-fZTDpW6lKYNdMyjw45_alSCZ1fypsMWz_2mTpQzil0lOtps5Ei_
z7mM7M8gCwe_AGpI53JxduQOaB5HkT5gVrv9cKu9CsW5MS6ZbqYXpGyOG5ehoxqm8DL5tFYaW3lB50ELxi0KsuT
KEbD0t5BCl0aCR2MBJWAbN-xeLwEenaqBiwPVvKixYleeDQiBEIylFdNNIMviKRgXiYuAvMziVPbwSgkZVHeEdF
5MQP1Oe2Spac-6IfA
```

The header and payload parts of a JWT are just base64url-encoded JSON objects.

The header contains metadata about the token itself

The payload contains the actual "claims" about the user.

```bash
#Header
{
  "kid": "9136ddb3-cb0a-4a19-a07e-eadf5a44c8b5",
  "alg": "RS256"
}
#Payload
{
  "iss": "portswigger",
  "exp": 1648037164,
  "name": "Carlos Montoya",
  "sub": "carlos",
  "role": "blog_author",
  "email": "carlos@carlos-montoya.net",
```

```
  "iat": 1516239022
}
#Signature
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
)
```

JWT signature

The server that issues the token typically generates the signature by hashing the header and payload. In some cases, they also encrypt the resulting hash. Either way, this process involves a secret signing key. This mechanism provides a way for servers to verify that none of the data within the token has been tampered

- As the signature is directly derived from the rest of the token, changing a single byte of the header or payload results in a mismatched signature
- Without knowing the server's secret signing key, it shouldn't be possible to generate the correct signature for a given header or payload

JWT vs JWS vs JWE

The JWT spec is extended by both the JSON Web Signature (JWS) and JSON Web Encryption (JWE) specifications, which define concrete ways of actually implementing JWTs.

In other words, a JWT is usually either a JWS or JWE token. When people use the term "JWT", they almost always mean a JWS token. JWEs are very similar, except that the actual contents of the token are encrypted rather than just encoded.

What are JWT attacks?

JWT attacks involve a user sending modified JWTs to the server in order to achieve a malicious goal. Typically, this goal is to bypass authentication and access controls by impersonating another user who has already been authenticated.

What is the impact of JWT attacks?

The impact of JWT attacks is usually severe. If an attacker is able to create their own valid tokens with arbitrary values, they may be able to escalate their own privileges or impersonate other users, taking full control of their accounts.

How do vulnerabilities to JWT attacks arise?

JWT vulnerabilities typically arise due to flawed JWT handling within the application itself. The various specifications related to JWTs are relatively flexible by design, allowing website

developers to decide many implementation details for themselves. This can result in them accidentally introducing vulnerabilities even when using battle-hardened libraries.

These implementation flaws usually mean that the signature of the JWT is not verified properly. This enables an attacker to tamper with the values passed to the application via the token's payload.

If this key is leaked in some way, or can be guessed or brute-forced, an attacker can generate a valid signature for any arbitrary token, compromising the entire mechanism

Exploiting flawed JWT signature verification

if the server doesn't verify the signature properly, there's nothing to stop an attacker from making arbitrary changes to the rest of the token.

Accepting tokens with no signature

the JWT header contains an `alg` parameter. This tells the server which algorithm was used to sign the token and, therefore, which algorithm it needs to use when verifying the signature.

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

JWTs can be signed using a range of different algorithms, but can also be left unsigned. In this case, the `alg` parameter is set to `none`, which indicates a so-called "unsecured JWT". Due to the obvious dangers of this, servers usually reject tokens with no signature

Brute-forcing secret keys

Some signing algorithms, such as HS256 (HMAC + SHA-256), use an arbitrary, standalone string as the secret key. Just like a password, it's crucial that this secret can't be easily guessed or brute-forced by an attacker. Otherwise, they may be able to create JWTs with any header and payload values they like, then use the key to re-sign the token with a valid signature.

When implementing JWT applications, developers sometimes make mistakes like forgetting to change default or placeholder secrets. They may even copy and paste code snippets they find online, then forget to change a hardcoded secret that's provided as an example. In this case, it can be trivial for an attacker to brute-force a server's secret using a wordlist of well-known secrets

Brute-forcing secret keys using hashcat

```bash
hashcat -a 0 -m 16500 <jwt> <wordlist>
```

Hashcat signs the header and payload from the JWT using each secret in the wordlist, then compares the resulting signature with the original one from the server. If any of the signatures match, hashcat outputs the identified secret in the following format, along with various other details

```bash
<jwt>:<identified-secret>
```

As hashcat runs locally on your machine and doesn't rely on sending requests to the server, this process is extremely quick, even when using a huge wordlist.

JWT header parameter injections

According to the JWS specification, only the `alg` header parameter is mandatory. In practice, however, JWT headers (also known as JOSE headers) often contain several other parameters. The following ones are of particular interest to attackers.

- `jwk` (JSON Web Key) - Provides an embedded JSON object representing the key.
- `jku` (JSON Web Key Set URL) - Provides a URL from which servers can fetch a set of keys containing the correct key.
- `kid` (Key ID) - Provides an ID that servers can use to identify the correct key in cases where there are multiple keys to choose from. Depending on the format of the key, this may have a matching `kid` parameter.

These user-controllable parameters each tell the recipient server which key to use when verifying the signature. In this section, you'll learn how to exploit these to inject modified JWTs signed using your own arbitrary key rather than the server's secret

Injecting self-signed JWTs via the jwk parameter

The JSON Web Signature (JWS) specification describes an optional `jwk` header parameter, which servers can use to embed their public key directly within the token itself in JWK format.

```json
{
    "kid": "ed2Nf8sb-sD6ng0-scs5390g-fFD8sfxG",
    "typ": "JWT",
    "alg": "RS256",
    "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "kid": "ed2Nf8sb-sD6ng0-scs5390g-fFD8sfxG",
```

```
        "n": "yy1wpYmffgXBxhAUJzHHocCuJolwDqql75ZWuCQ_cb33K2vh9m"
    }
}
```

Ideally, servers should only use a limited whitelist of public keys to verify JWT signatures. However, misconfigured servers sometimes use any key that's embedded in the `jwk` parameter.

you can manually add or modify the `jwk` parameter in Burp, the JWT Editor extension provides a useful feature to help you test for this vulnerability:

1. With the extension loaded, in Burp's main tab bar, go to the **JWT Editor Keys** tab.
2. Generate a new RSA key.
3. Send a request containing a JWT to Burp Repeater.
4. In the message editor, switch to the extension-generated **JSON Web Token** tab and modify the token's payload however you like.
5. Click **Attack**, then select **Embedded JWK**. When prompted, select your newly generated RSA key.
6. Send the request to test how the server responds.

You can also perform this attack manually by adding the `jwk` header yourself. However, you may also need to update the JWT's `kid` header parameter to match the `kid` of the embedded key.

Injecting self-signed JWTs via the jku parameter

Instead of embedding public keys directly using the `jwk` header parameter, some servers let you use the `jku` (JWK Set URL) header parameter to reference a JWK Set containing the key. When verifying the signature, the server fetches the relevant key from this URL.

```json
{
    "keys": [
        {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "75d0ef47-af89-47a9-9061-7c02a610d5ab",
            "n": "o-yy1wpYmffgXBxhAUJzHHocCuJolwDqql75ZWuCQ_cb33K2vh9mk6GPM9gNN4Y_qTVX67WhsN3JvaFYw-fhvsWQ"
        },
        {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "d8fDFo-fS9-faS14a9-ASf99sa-7c1Ad5abA",
            "n": "fc3f-yy1wpYmffgXBxhAUJzHql79gNNQ_cb33HocCuJolwDqmk6GPM4Y_qTVX67WhsN3JvaFYw-dfg6DH-asAScw"
```

```
        }
    ]
}
```

JWK Sets like this are sometimes exposed publicly via a standard endpoint, such as `/.well-known/jwks.json`.

More secure websites will only fetch keys from trusted domains, but you can sometimes take advantage of URL parsing discrepancies to bypass this kind of filtering.

Injecting self-signed JWTs via the kid parameter

Servers may use several cryptographic keys for signing different kinds of data, not just JWTs. For this reason, the header of a JWT may contain a `kid` (Key ID) parameter, which helps the server identify which key to use when verifying the signature.

Verification keys are often stored as a JWK Set. In this case, the server may simply look for the JWK with the same `kid` as the token. However, the JWS specification doesn't define a concrete structure for this ID - it's just an arbitrary string of the developer's choosing.

If this parameter is also vulnerable to directory traversal, an attacker could potentially force the server to use an arbitrary file from its filesystem as the verification key.

```json
{
    "kid": "../../path/to/file",
    "typ": "JWT",
    "alg": "HS256",
    "k": "asGsADas3421-dfh9DGN-AFDFDbasfd8-anfjkvc"
}
```

This is especially dangerous if the server also supports JWTs signed using a symmetric algorithm. In this case, an attacker could potentially point the `kid` parameter to a predictable, static file, then sign the JWT using a secret that matches the contents of this file.

How to prevent JWT attacks

You can protect your own websites against many of the attacks we've covered by taking the following high-level measures:

- Use an up-to-date library for handling JWTs and make sure your developers fully understand how it works, along with any security implications. Modern libraries make it more difficult for you to inadvertently implement them insecurely, but this isn't foolproof due to the inherent flexibility of the related specifications.

- Make sure that you perform robust signature verification on any JWTs that you receive, and account for edge-cases such as JWTs signed using unexpected algorithms.
- Enforce a strict whitelist of permitted hosts for the `jku` header.
- Make sure that you're not vulnerable to path traversal or SQL injection via the `kid` header parameter