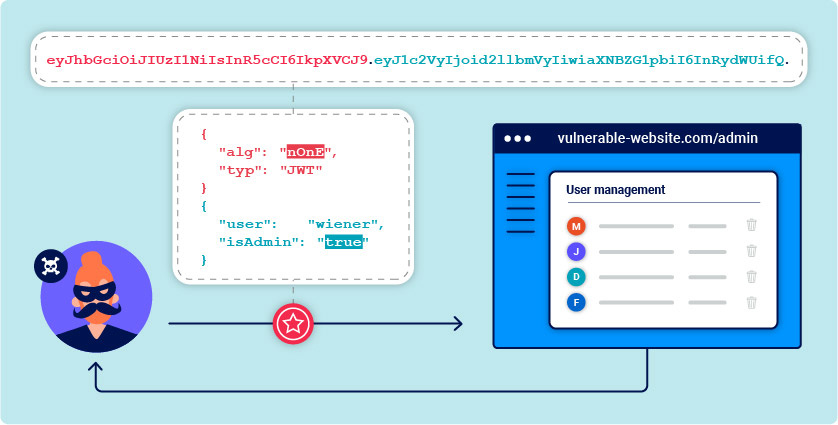
**JWT attacks**

In this section, we'll look at how design issues and flawed handling of JSON web tokens (JWTs) can leave websites vulnerable to a variety of high-severity attacks. As JWTs are most commonly used in authentication, session management, and access control mechanisms, these vulnerabilities can potentially compromise the entire website and its users.

Don't worry if you're not familiar with JWTs and how they work - we'll cover all of the relevant details as we go. We've also provided a number of deliberately vulnerable labs so that you can practice exploiting these vulnerabilities safely against realistic targets.



**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.

Unlike with classic session tokens, all of the data that a server needs is stored client-side within the JWT itself. This makes JWTs a popular choice for highly distributed websites where users need to interact seamlessly with multiple back-end servers.

**JWT format**

A JWT consists of 3 parts: a header, a payload, and a signature. These are each separated by a dot, as shown in the following example:

eyJraWQiOiI5MTM2ZGRiMy1jYjBhLTRhMTktYTA3ZS1lYWRmNWE0NGM4YjUiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJwb3J0c3dpZ2dlciIsImV4cCI6MTY0ODAzNzE2NCwibmFtZSI6IkNhcmxvcyBNb250b3lhIiwic3ViIjoiY2FybG9zIiwicm9sZSI6ImJsb2dfYXV0aG9yIiwiZW1haWwiOiJjYXJsb3NAY2FybG9zLW1vbnRveWEubmV0IiwiaWF0IjoxNTE2MjM5MDIyfQ.SYZBPIBg2CRjXAJ8vCER0LA\_ENjII1JakvNQoP-Hw6GG1zfl4JyngsZReIfqRvIAEi5L4HV0q7\_9qGhQZvy9ZdxEJbwTxRs\_6Lb-fZTDpW6lKYNdMyjw45\_alSCZ1fypsMWz\_2mTpQzil0lOtps5Ei\_z7mM7M8gCwe\_AGpI53JxduQOaB5HkT5gVrv9cKu9CsW5MS6ZbqYXpGyOG5ehoxqm8DL5tFYaW3lB50ELxi0KsuTKEbD0t5BCl0aCR2MBJWAbN-xeLwEenaqBiwPVvKixYleeDQiBEIylFdNNIMviKRgXiYuAvMziVPbwSgkZVHeEdF5MQP1Oe2Spac-6IfA

The header and payload parts of a JWT are just base64url-encoded JSON objects. The header contains metadata about the token itself, while the payload contains the actual "claims" about the user. For example, you can decode the payload from the token above to reveal the following claims:

{

"iss": "mrwebsecure",

"exp": 1648037164,

"name": "Carlos Montoya",

"sub": "carlos",

"role": "blog\_author",

"email": "carlos@carlos-montoya.net",

"iat": 1516239022

}

In most cases, this data can be easily read or modified by anyone with access to the token. Therefore, the security of any JWT-based mechanism is heavily reliant on the cryptographic signature.

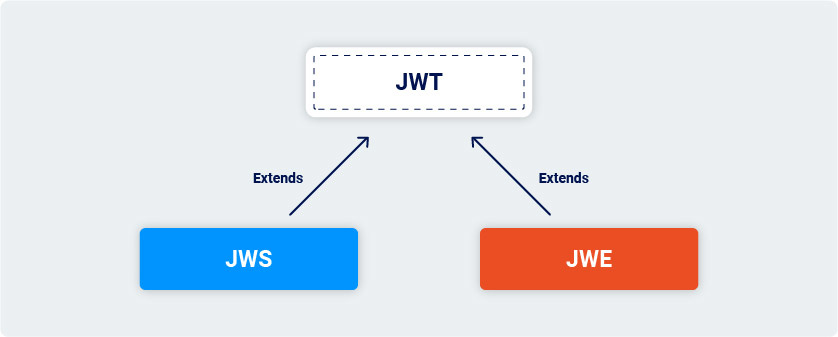
**JWT signature**

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 with since it was issued:

* 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 specification is actually very limited. It only defines a format for representing information ("claims") as a JSON object that can be transferred between two parties. In practice, JWTs aren't really used as a standalone entity. 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.

**Note**

For simplicity, throughout these materials, "JWT" refers primarily to JWS tokens, although some of the vulnerabilities described may also apply to JWE tokens.

**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. Even if the signature is robustly verified, whether it can truly be trusted relies heavily on the server's secret key remaining a secret. 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**

By design, servers don't usually store any information about the JWTs that they issue. Instead, each token is an entirely self-contained entity. This has several advantages, but also introduces a fundamental problem - the server doesn't actually know anything about the original contents of the token, or even what the original signature was. Therefore, 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.

For example, consider a JWT containing the following claims:

{

"username": "carlos",

"isAdmin": false

}

If the server identifies the session based on this username, modifying its value might enable an attacker to impersonate other logged-in users. Similarly, if the isAdmin value is used for access control, this could provide a simple vector for privilege escalation.

In the first couple of labs, you'll see some examples of how these vulnerabilities might look in real-world applications.

**Accepting arbitrary signatures**

JWT libraries typically provide one method for verifying tokens and another that just decodes them. For example, the Node.js library jsonwebtoken has verify() and decode().

Occasionally, developers confuse these two methods and only pass incoming tokens to the decode() method. This effectively means that the application doesn't verify the signature at all.

**Accepting tokens with no signature**

Among other things, 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.

{

"alg": "HS256",

"typ": "JWT"

}

This is inherently flawed because the server has no option but to implicitly trust user-controllable input from the token which, at this point, hasn't been verified at all. In other words, an attacker can directly influence how the server checks whether the token is trustworthy.

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. However, as this kind of filtering relies on string parsing, you can sometimes bypass these filters using classic obfuscation techniques, such as mixed capitalization and unexpected encodings.

**Note**

Even if the token is unsigned, the payload part must still be terminated with a trailing dot.

**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**

We recommend using hashcat to brute-force secret keys. You can [install hashcat manually](https://hashcat.net/wiki/doku.php?id=frequently_asked_questions#how_do_i_install_hashcat), but it also comes pre-installed and ready to use on Kali Linux.

**Note**

If you're using the pre-built VirtualBox image for Kali rather than the bare metal installer version, this may not have enough memory allocated to run hashcat.

You just need a valid, signed JWT from the target server and a wordlist of well-known secrets. You can then run the following command, passing in the JWT and wordlist as arguments:

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:

<jwt>:<identified-secret>

**Note**

If you run the command more than once, you need to include the --show flag to output the results.

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.

Once you have identified the secret key, you can use it to generate a valid signature for any JWT header and payload that you like. For details on how to re-sign a modified JWT in Burp Suite, see Editing JWTs.

If the server uses an extremely weak secret, it may even be possible to brute-force this character-by-character rather than using a 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.

As you can see, 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.

**JWK**

A JWK (JSON Web Key) is a standardized format for representing keys as a JSON object.

You can see an example of this in the following JWT header:

{

"kid": "ed2Nf8sb-sD6ng0-scs5390g-fFD8sfxG",

"typ": "JWT",

"alg": "RS256",

"jwk": {

"kty": "RSA",

"e": "AQAB",

"kid": "ed2Nf8sb-sD6ng0-scs5390g-fFD8sfxG",

"n": "yy1wpYmffgXBxhAUJzHHocCuJolwDqql75ZWuCQ\_cb33K2vh9m"

}

}

**Public and private keys**

In case you're not familiar with the terms "public key" and "private key", we've covered this as part of our materials on algorithm confusion attacks. For more information,

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 exploit this behavior by signing a modified JWT using your own RSA private key, then embedding the matching public key in the jwk header.

Although 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. The extension's built-in attack takes care of this step for you.

**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.

**JWK Set**

A JWK Set is a JSON object containing an array of JWKs representing different keys. You can see an example of this below.

{

"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. We covered some examples of these in our topic on SSRF.

**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. For example, they might use the kid parameter to point to a particular entry in a database, or even the name of a file.

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.

{

"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.

You could theoretically do this with any file, but one of the simplest methods is to use /dev/null, which is present on most Linux systems. As this is an empty file, reading it returns an empty string. Therefore, signing the token with a empty string will result in a valid signature.

**Note**

If you're using the JWT Editor extension, note that this doesn't let you sign tokens using an empty string. However, due to a bug in the extension, you can get around this by using a Base64-encoded null byte.

If the server stores its verification keys in a database, the kid header parameter is also a potential vector for SQL injection attacks.

**Other interesting JWT header parameters**

The following header parameters may also be interesting for attackers:

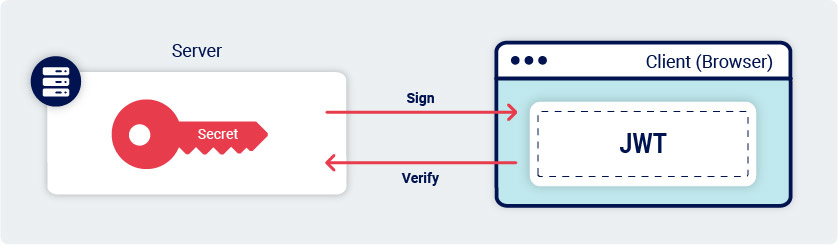
* cty (Content Type) - Sometimes used to declare a media type for the content in the JWT payload. This is usually omitted from the header, but the underlying parsing library may support it anyway. If you have found a way to bypass signature verification, you can try injecting a cty header to change the content type to text/xml or application/x-java-serialized-object, which can potentially enable new vectors for XXE and deserialization attacks.
* x5c (X.509 Certificate Chain) - Sometimes used to pass the X.509 public key certificate or certificate chain of the key used to digitally sign the JWT. This header parameter can be used to inject self-signed certificates, similar to the jwk header injection attacks discussed above. Due to the complexity of the X.509 format and its extensions, parsing these certificates can also introduce vulnerabilities. Details of these attacks are beyond the scope of these materials, but for more details, check out CVE-2017-2800 and CVE-2018-2633.

# Algorithm confusion attacks

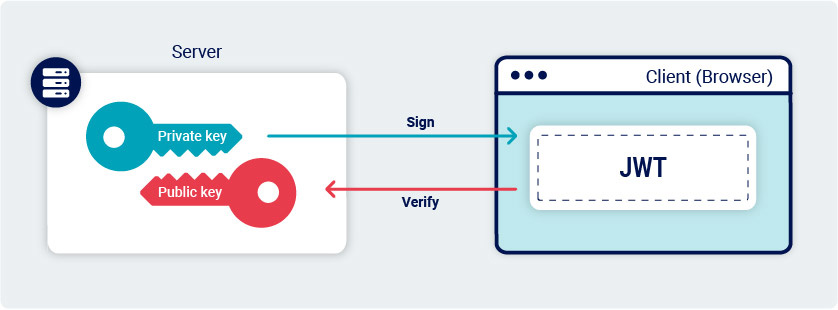
Algorithm confusion attacks (also known as key confusion attacks) occur when an attacker is able to force the server to verify the signature of a JSON web token (JWT) using a different algorithm than is intended by the website's developers. If this case isn't handled properly, this may enable attackers to forge valid JWTs containing arbitrary values without needing to know the server's secret signing key.

## Symmetric vs asymmetric algorithms

JWTs can be signed using a range of different algorithms. Some of these, such as HS256 (HMAC + SHA-256) use a "symmetric" key. This means that the server uses a single key to both sign and verify the token. Clearly, this needs to be kept secret, just like a password.



Other algorithms, such as RS256 (RSA + SHA-256) use an "asymmetric" key pair. This consists of a private key, which the server uses to sign the token, and a mathematically related public key that can be used to verify the signature.



As the names suggest, the private key must be kept secret, but the public key is often shared so that anybody can verify the signature of tokens issued by the server.

## How do algorithm confusion vulnerabilities arise?

Algorithm confusion vulnerabilities typically arise due to flawed implementation of JWT libraries. Although the actual verification process differs depending on the algorithm used, many libraries provide a single, algorithm-agnostic method for verifying signatures. These methods rely on the alg parameter in the token's header to determine the type of verification they should perform.

The following pseudo-code shows a simplified example of what the declaration for this generic verify() method might look like in a JWT library:

function verify(token, secretOrPublicKey){

algorithm = token.getAlgHeader();

if(algorithm == "RS256"){

// Use the provided key as an RSA public key

} else if (algorithm == "HS256"){

// Use the provided key as an HMAC secret key

}

}

Problems arise when website developers who subsequently use this method assume that it will exclusively handle JWTs signed using an asymmetric algorithm like RS256. Due to this flawed assumption, they may always pass a fixed public key to the method as follows:

publicKey = <public-key-of-server>;

token = request.getCookie("session");

verify(token, publicKey);

In this case, if the server receives a token signed using a symmetric algorithm like HS256, the library's generic verify() method will treat the public key as an HMAC secret. This means that an attacker could sign the token using HS256 and the public key, and the server will use the same public key to verify the signature.

#### Note

The public key you use to sign the token must be absolutely identical to the public key stored on the server. This includes using the same format (such as X.509 PEM) and preserving any non-printing characters like newlines. In practice, you may need to experiment with different formatting in order for this attack to work.

## Performing an algorithm confusion attack

An algorithm confusion attack generally involves the following high-level steps:

1. Obtain the server's public key
2. Convert the public key to a suitable format
3. Create a malicious JWT with a modified payload and the alg header set to HS256.
4. Sign the token with HS256, using the public key as the secret.

In this section, we'll walk through this process in more detail, demonstrating how you can perform this kind of attack using Burp Suite.

### Step 1 - Obtain the server's public key

Servers sometimes expose their public keys as JSON Web Key (JWK) objects via a standard endpoint mapped to /jwks.json or /.well-known/jwks.json, for example. These may be stored in an array of JWKs called keys. This is known as a JWK Set.

{

"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"

}

]

}

Even if the key isn't exposed publicly, you may be able to extract it from a pair of existing JWTs.

### Step 2 - Convert the public key to a suitable format

Although the server may expose their public key in JWK format, when verifying the signature of a token, it will use its own copy of the key from its local filesystem or database. This may be stored in a different format.

In order for the attack to work, the version of the key that you use to sign the JWT must be identical to the server's local copy. In addition to being in the same format, every single byte must match, including any non-printing characters.

For the purpose of this example, let's assume that we need the key in X.509 PEM format. You can convert a JWK to a PEM using the JWT Editor extension in Burp as follows:

1. With the extension loaded, in Burp's main tab bar, go to the **JWT Editor Keys** tab.
2. Click **New RSA** Key. In the dialog, paste the JWK that you obtained earlier.
3. Select the **PEM** radio button and copy the resulting PEM key.
4. Go to the **Decoder** tab and Base64-encode the PEM.
5. Go back to the **JWT Editor Keys** tab and click **New Symmetric Key**.
6. In the dialog, click **Generate** to generate a new key in JWK format.
7. Replace the generated value for the k parameter with a Base64-encoded PEM key that you just copied.
8. Save the key.

### Step 3 - Modify your JWT

Once you have the public key in a suitable format, you can modify the JWT however you like. Just make sure that the alg header is set to HS256.

### Step 4 - Sign the JWT using the public key

Sign the token using the HS256 algorithm with the RSA public key as the secret.

## Deriving public keys from existing tokens

In cases where the public key isn't readily available, you may still be able to test for algorithm confusion by deriving the key from a pair of existing JWTs. This process is relatively simple using tools such as jwt\_forgery.py. You can find this, along with several other useful scripts, on the rsa\_sign2n GitHub repository.

We have also created a simplified version of this tool, which you can run as a single command:

docker run --rm -it portswigger/sig2n <token1> <token2>

#### Note

You need the Docker CLI to run either version of the tool. The first time you run this command, it will automatically pull the image from Docker Hub, which may take a few minutes.

This uses the JWTs that you provide to calculate one or more potential values of n. Don't worry too much about what this means - all you need to know is that only one of these matches the value of n used by the server's key. For each potential value, our script outputs:

* A Base64-encoded PEM key in both X.509 and PKCS1 format.
* A forged JWT signed using each of these keys.

To identify the correct key, use Burp Repeater to send a request containing each of the forged JWTs. Only one of these will be accepted by the server. You can then use the matching key to construct an algorithm confusion attack.

**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.

**Additional best practice for JWT handling**

Although not strictly necessary to avoid introducing vulnerabilities, we recommend adhering to the following best practice when using JWTs in your applications:

* Always set an expiration date for any tokens that you issue.
* Avoid sending tokens in URL parameters where possible.
* Include the aud (audience) claim (or similar) to specify the intended recipient of the token. This prevents it from being used on different websites.
* Enable the issuing server to revoke tokens (on logout, for example).