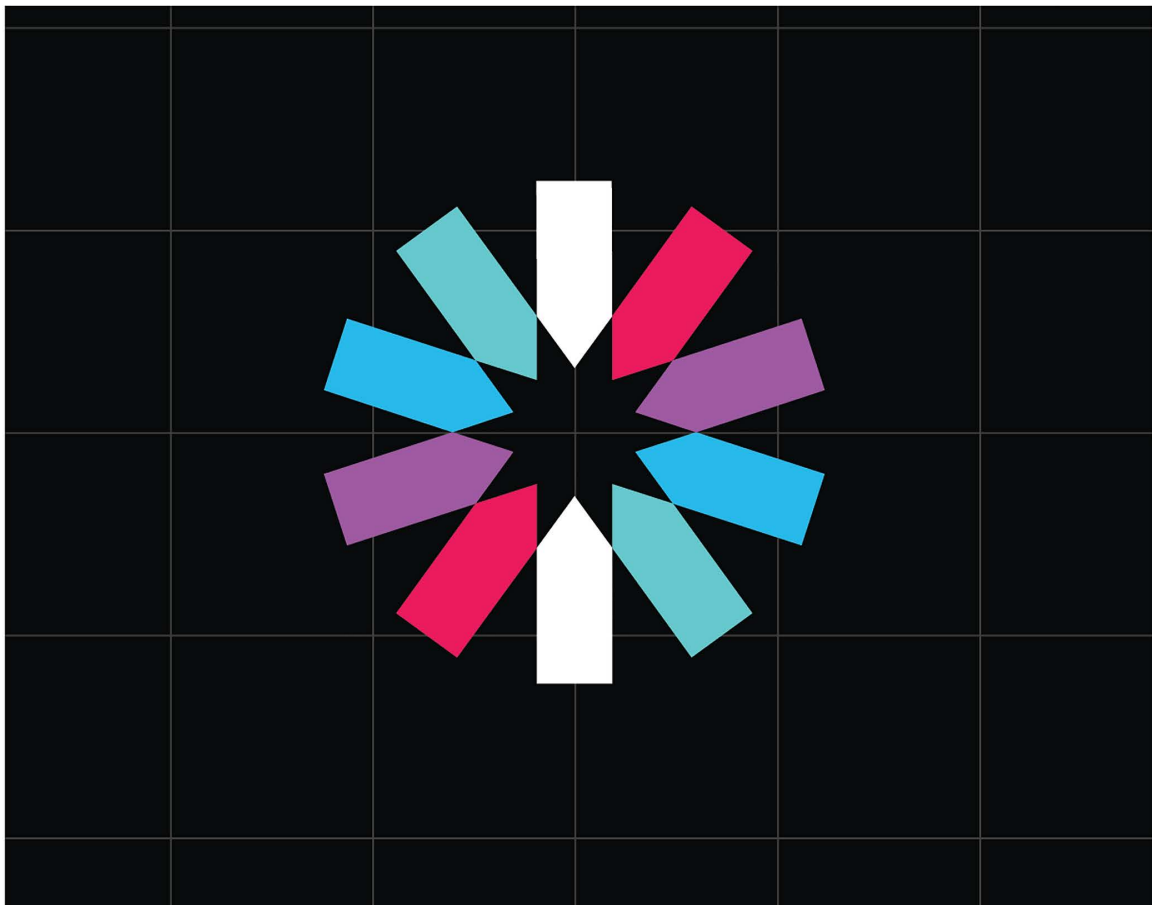

JWT HANDBOOK

By Sebastián Peyrott

Auth0
by Okta

The JWT Handbook

Sebastián E. Peyrott, Auth0 Inc.

Version 0.14.2, 2024

Contents

Special Thanks

In no special order: **Prosper Otemuyiwa** (for providing the federated identity example from chapter 2), **Diego Poza** (for reviewing this work and keeping my hands free while I worked on it), **Matías Woloski** (for reviewing the hard parts of this work), **Martín Gontovnikas** (for putting up with my requests and doing everything to make work amenable), **Bárbara Mercedes Muñoz Cruzado** (for making everything look nice), **Alejo Fernández** and **Víctor Fernández** (for doing the frontend and backend work to distribute this handbook), **Sergio Fruto** (for going out of his way to help teammates), **Federico Jack** (for keeping everything running and still finding the time to listen to each and everyone).

Chapter 1

Introduction

JSON Web Token, or JWT ("jot") for short, is a standard for *safely* passing *claims* in space constrained environments. It has found its way into all[1] major[2] web[3] frameworks[4]. Simplicity, compactness and usability are key features of its architecture. Although much more complex systems[5] are still in use, JWTs have a broad range of applications. In this little handbook, we will cover the most important aspects of the architecture of JWTs, including their binary representation and the algorithms used to construct them, while also taking a look at how they are commonly used in the industry.

1.1 What is a JSON Web Token?

A JSON Web Token looks like this (newlines inserted for readability):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

While this looks like gibberish, it is actually a very **compact**, **printable** representation of a series of *claims*, along with a **signature** to verify its authenticity.

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

{
```

[1] https://github.com/auth0/express-jwt
[2] https://github.com/nsarno/knock
[3] https://github.com/tymondesigns/jwt-auth
[4] https://github.com/jpadilla/django-jwt-auth
[5] https://en.wikipedia.org/wiki/Security_Assertion_Markup_Language

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

Claims are *definitions* or *assertions* made about a certain party or object. Some of these claims and their meaning are defined as part of the JWT spec. Others are user defined. The magic behind JWTs is that they standardize certain claims that are useful in the context of some common operations. For example, one of these common operations is establishing the identity of certain party. So one of the standard claims found in JWTs is the *sub* (from "subject") claim. We will take a deeper look at each of the standard claims in chapter 3.

Another key aspect of JWTs is the possiblity of signing them, using JSON Web Signatures (JWS, RFC 7515[6]), and/or encrypting them, using JSON Web Encryption (JWE, RFC 7516[7]). Together with JWS and JWE, JWTs provide a powerful, secure solution to many different problems.

1.2 What problem does it solve?

Although the main purpose of JWTs is to transfer claims between two parties, arguably the most important aspect of this is the *standardization effort* in the form of a *simple, optionally validated and/or encrypted, container format.* Ad hoc solutions to this same problem have been implemented both privately and publicly in the past. Older standards[8] for establishing claims about certain parties are also available. What JWT brings to the table is a *simple*, useful, standard container format.

Although the definition given is a bit abstract so far, it is not hard to imagine how they can be used: login systems (although other uses are possible). We will take a closer look at practical applications in chapter 2. Some of these applications include:

- Authentication
- Authorization
- Federated identity
- Client-side sessions ("stateless" sessions)
- Client-side secrets

1.3 A little bit of history

The JSON Object Signing and Encryption group (JOSE) was formed in the year 2011[9]. The group's objective was to "*standardize the mechanism for integrity protection (signature and MAC) and encryption as well as the format for keys and algorithm identifiers to support interoperability of security services for protocols that use JSON*". By year 2013 a series of drafts, including a cookbook

[6] https://tools.ietf.org/html/rfc7515
[7] https://tools.ietf.org/html/rfc7516
[8] https://en.wikipedia.org/wiki/Security_Assertion_Markup_Language
[9] https://datatracker.ietf.org/wg/jose/history/

with different examples of the use of the ideas produced by the group, were available. These drafts would later become the JWT, JWS, JWE, JWK and JWA RFCs. As of year 2016, these RFCs are in the standards track process and errata have not been found in them. The group is currently inactive.

The main authors behind the specs are Mike Jones[10], Nat Sakimura[11], John Bradley[12] and Joe Hildebrand[13].

[10] http://self-issued.info/
[11] https://nat.sakimura.org/
[12] https://www.linkedin.com/in/ve7jtb
[13] https://www.linkedin.com/in/hildjj

Chapter 2

Practical Applications

Before taking a deep dive into the structure and construction of a JWT, we will take a look at several practical applications. This chapter will give you a sense of the complexity (or simplicity) of common JWT-based solutions used in the industry today. All code is available from public repositories[1] for your convenience. Be aware that the following demonstrations are *not* meant to be used in production. Test cases, logging, and security best practices are all essential for production-ready code. These samples are for educational purposes only and thus remain simple and to the point.

2.1 Client-side/Stateless Sessions

The so-called *stateless* sessions are in fact nothing more than client-side data. The key aspect of this application lies in the use of *signing* and possibly *encryption* to authenticate and protect the contents of the session. Client-side data is subject to *tampering*. As such it must be handled with great care by the backend.

JWTs, by virtue of JWS and JWE, can provide various types of signatures and encryption. Signatures are useful to *validate* the data against tampering. Encryption is useful to *protect* the data from being read by third parties.

Most of the time sessions need only be signed. In other words, there is no security or privacy concern when data stored in them is read by third parties. A common example of a claim that can usually be safely read by third parties is the *sub* claim ("subject"). The subject claim usually identifies one of the parties to the other (think of user IDs or emails). It is not a requirement that this claim be *unique*. In other words, additional claims may be required to uniquely identify a user. This is left to the users to decide.

A claim that may not be appropriately left in the open could be an "items" claim representing a user's shopping cart. This cart might be filled with items that the user is about to purchase and

[1] https://github.com/auth0/jwt-handbook-samples

thus are associated to his or her session. A third party (a client-side script) might be able to harvest these items if they are stored in an unencrypted JWT, which could raise privacy concerns.

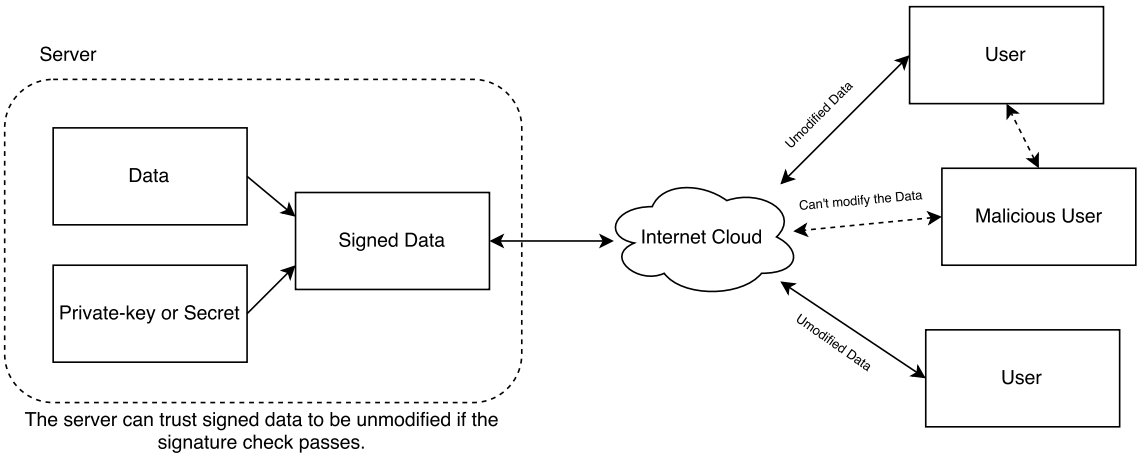

Figure 2.1: Client-side Signed Data

2.1.1 Security Considerations

2.1.1.1 Signature Stripping

A common method for attacking a signed JWT is to simply remove the signature. Signed JWTs are constructed from three different parts: the header, the payload, and the signature. These three parts are encoded separately. As such, it is possible to remove the signature and then *change* the header to claim the JWT is *unsigned*. Careless use of certain JWT validation libraries can result in unsigned tokens being taken as valid tokens, which may allow an attacker to modify the payload at his or her discretion. This is easily solved by making sure that the application that performs the validation does not consider unsigned JWTs valid.

Figure 2.2: Signature Stripping

2.1.1.2 Cross-Site Request Forgery (CSRF)

Cross-site request forgery attacks attempt to perform requests against sites where the user is logged in by tricking the user's browser into sending a request from a different site. To accomplish this, a specially crafted site (or item) must contain the URL to the target. A common example is an `<img>` tag embedded in a malicious page with the `src` pointing to the attack's target. For instance:

```
<!-- This is embedded in another domain's site -->
<img src="http://target.site.com/add-user?user=name&grant=admin">
```

The above `<img>` tag will send a request to `target.site.com` every time the page that contains it is loaded. If the user had previously logged in to `target.site.com` and the site used a cookie to keep the session active, this cookie will be sent as well. If the target site does not implement any CSRF mitigation techniques, the request will be handled as a valid request on behalf of the user. JWTs, like any other client-side data, can be stored as cookies.

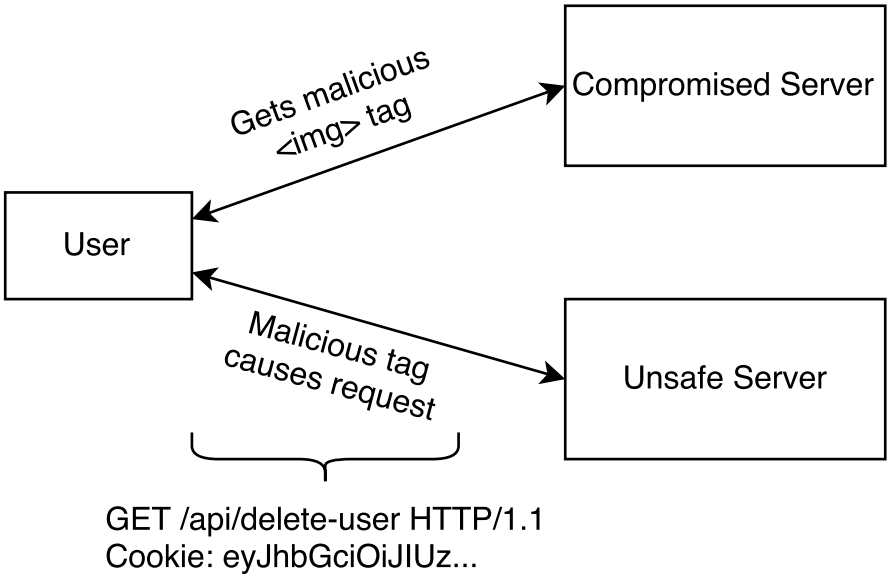

Figure 2.3: Cross-Site Request Forgery

Short-lived JWTs can help in this case. Common CSRF mitigation techniques include special headers that are added to requests only when they are performed from the right origin, per session cookies, and per request tokens. If JWTs (and session data) are not stored as cookies, CSRF attacks are not possible. Cross-site scripting attacks are still possible, though.

2.1.1.3 Cross-Site Scripting (XSS)

Cross-site scripting (XSS) attacks attempt to inject JavaScript in trusted sites. Injected JavaScript can then steal tokens from cookies and local storage. If an access token is leaked before it expires, a malicious user could use it to access protected resources. Common XSS attacks are usually caused by improper validation of data passed to the backend (in similar fashion to SQL injection attacks).

An example of a XSS attack could be related to the comments section of a public site. Every time a user adds a comment, it is stored by the backend and displayed to users who load the comments section. If the backend does not sanitize the comments, a malicious user could write a comment in such a way that it could be interpreted by the browser as a `<script>` tag. So, a malicious user could insert arbitrary JavaScript code and execute it in every user's browser, thus, stealing credentials stored as cookies and in local storage.

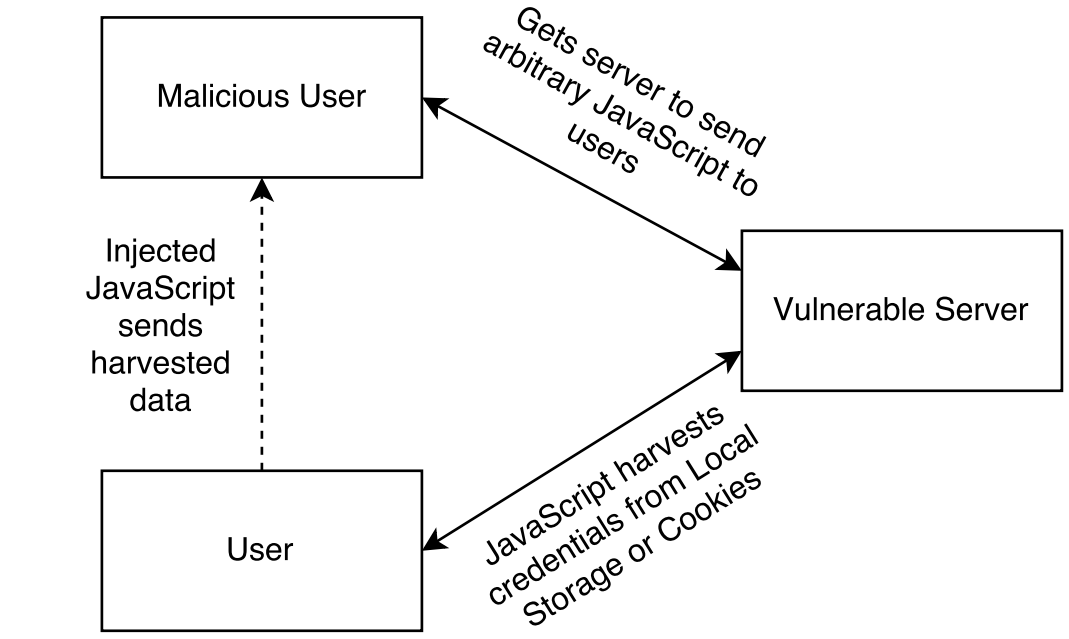

Figure 2.4: Persistent Cross Site Scripting

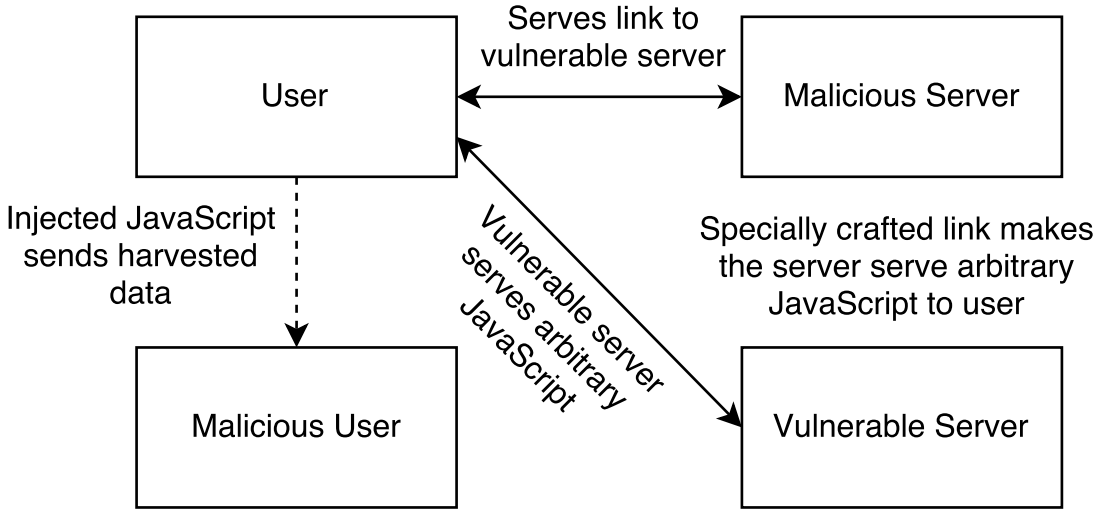

Figure 2.5: Reflective Cross Site Scripting

Mitigation techniques rely on proper validation of all data passed to the backend. In particular, any data received from clients must always be sanitized. If cookies are used, it is possible to protect them from being accessed by JavaScript by setting the HttpOnly flag[2]. The HttpOnly flag, while useful, will not protect the cookie from CSRF attacks.

2.1.2 Are Client-Side Sessions Useful?

There are pros and cons to any approach, and client-side sessions are not an exception[3]. Some applications may require big sessions. Sending this state back and forth for every request (or group of requests) can easily overcome the benefits of the reduced chattiness in the backend. A certain balance between client-side data and database lookups in the backend is necessary. This depends on the data model of your application. Some applications do not map well to client-side sessions. Others may depend entirely on client-side data. The final word on this matter is your own! Run benchmarks, study the benefits of keeping certain state client-side. Are the JWTs too big? Does this have an impact on bandwidth? Does this added bandwidth overthrow the reduced latency in the backend? Can small requests be aggregated into a single bigger request? Do these requests still require big database lookups? Answering these questions will help you decide on the right approach.

2.1.3 Example

For our example we will make a simple shopping application. The user's shopping cart will be stored client-side. In this example, there are multiple JWTs present. Our shopping cart will be one of them.

- One JWT for the ID token, a token that carries the user's profile information, useful for the UI.
- One JWT for interacting with the API backend (the access token).
- One JWT for our client-side state: the shopping cart.

Here's how the shopping cart looks when decoded:

```
{
  "items": [
    0,
    2,
    4
  ],
  "iat": 1493139659,
  "exp": 1493143259
}
```

Each item is identified by a numeric ID. The encoded and signed JWT looks like:

[2] https://www.owasp.org/index.php/HttpOnly
[3] https://auth0.com/blog/stateless-auth-for-stateful-minds/

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJpdGVtcyI6WzAsMiw0XSwiaWF0IjoxNDkzMTM5NjU5LCJleHAiOjE0OTMxNDMyNTl9.
932ZxtZzy1qhLXs932hd04J58Ihbg5_g_rIrj-Z16Js
```

To render the items in the cart, the frontend only needs to retrieve it from its cookie:

```
function populateCart() {
    const cartElem = $('#cart');
    cartElem.empty();

    const cartToken = Cookies.get('cart');
    if(!cartToken) {
        return;
    }

    const cart = jwt_decode(cartToken).items;

    cart.forEach(itemId => {
        const name = items.find(item => item.id == itemId).name;
        cartElem.append(`<li>${name}</li>`);
    });
}
```

Note that the frontend does not check the signature, it simply decodes the JWT so it can display its contents. The actual checks are performed by the backend. All JWTs are verified.

Here is the backend check for the validity of the cart JWT implemented as an Express middleware:

```
function cartValidator(req, res, next) {
    if(!req.cookies.cart) {
        req.cart = { items: [] };
    } else {
        try {
            req.cart = {
                items: jwt.verify(req.cookies.cart,
                                  process.env.AUTH0_CART_SECRET,
                                  cartVerifyJwtOptions).items
            };
        } catch(e) {
            req.cart = { items: [] };
        }
    }

    next();
}
```

When items are added, the backend constructs a new JWT with the new item in it and a new signature:

```
app.get('/protected/add_item', idValidator, cartValidator, (req, res) => {
```

```
    req.cart.items.push(parseInt(req.query.id));

    const newCart = jwt.sign(req.cart,
                             process.env.AUTH0_CART_SECRET,
                             cartSignJwtOptions);

    res.cookie('cart', newCart, {
        maxAge: 1000 * 60 * 60
    });

    res.end();

    console.log(`Item ID ${req.query.id} added to cart.`);
});
```

Note that locations prefixed by `/protected` are also protected by the API access token. This is setup using `express-jwt`:

```
app.use('/protected', expressJwt({
    secret: jwksClient.expressJwtSecret(jwksOpts),
    issuer: process.env.AUTH0_API_ISSUER,
    audience: process.env.AUTH0_API_AUDIENCE,
    requestProperty: 'accessToken',
    getToken: req => {
        return req.cookies['access_token'];
    }
}));
```

In other words, the `/protected/add_item` endpoint must first pass the access token validation step before validating the cart. One token validates access (authorization) to the API and the other token validates the integrity of the client side data (the cart).

The access token and the ID token are assigned by Auth0 to our application. This requires setting up a client[4] and an API endpoint[5] using the Auth0 dashboard[6]. These are then retrieved using the Auth0 JavaScript library, called by our frontend:

```
//Auth0 Client ID
const clientId = "t42WY87weXzepAdUlwMiHYRBQj9qWVAT";
//Auth0 Domain
const domain = "speyrott.auth0.com";

const auth0 = new window.auth0.WebAuth({
    domain: domain,
    clientID: clientId,
    audience: '/protected',
```

[4] https://manage.auth0.com/#/clients
[5] https://manage.auth0.com/#/apis
[6] https://manage.auth0.com

```
    scope: 'openid profile purchase',
    responseType: 'id_token token',
    redirectUri: 'http://localhost:3000/auth/',
    responseMode: 'form_post'
});

//(...)

$('#login-button').on('click', function(event) {
    auth0.authorize();
});
```

The `audience` claim must match the one setup for your API endpoint using the Auth0 dashboard.

The Auth0 authentication and authorization server displays a login screen with our settings and then redirects back to our application at a specific path with the tokens we requested. These are handled by our backend which simply sets them as cookies:

```
app.post('/auth', (req, res) => {
    res.cookie('access_token', req.body.access_token, {
        httpOnly: true,
        maxAge: req.body.expires_in * 1000
    });
    res.cookie('id_token', req.body.id_token, {
        maxAge: req.body.expires_in * 1000
    });
    res.redirect('/');
});
```

Implementing CSRF mitigation techniques is left as an exercise for the reader. The full example for this code can be found in the `samples/stateless-sessions` directory.

2.2 Federated Identity

Federated identity[7] systems allow different, possibly unrelated, parties to share authentication and authorization services with other parties. In other words, a user's identity is centralized. There are several solutions for federated identity management: SAML[8] and OpenID Connect[9] are two of the most common ones. Certain companies provide specialized products that centralize authentication and authorization. These may implement one of the standards mentioned above or use something completely different. Some of these companies use JWTs for this purpose.

The use of JWTs for centralized authentication and authorization varies from company to company, but the essential flow of the authorization process is:

[7] https://auth0.com/blog/2015/09/23/what-is-and-how-does-single-sign-on-work/
[8] http://saml.xml.org/saml-specifications
[9] https://openid.net/connect/

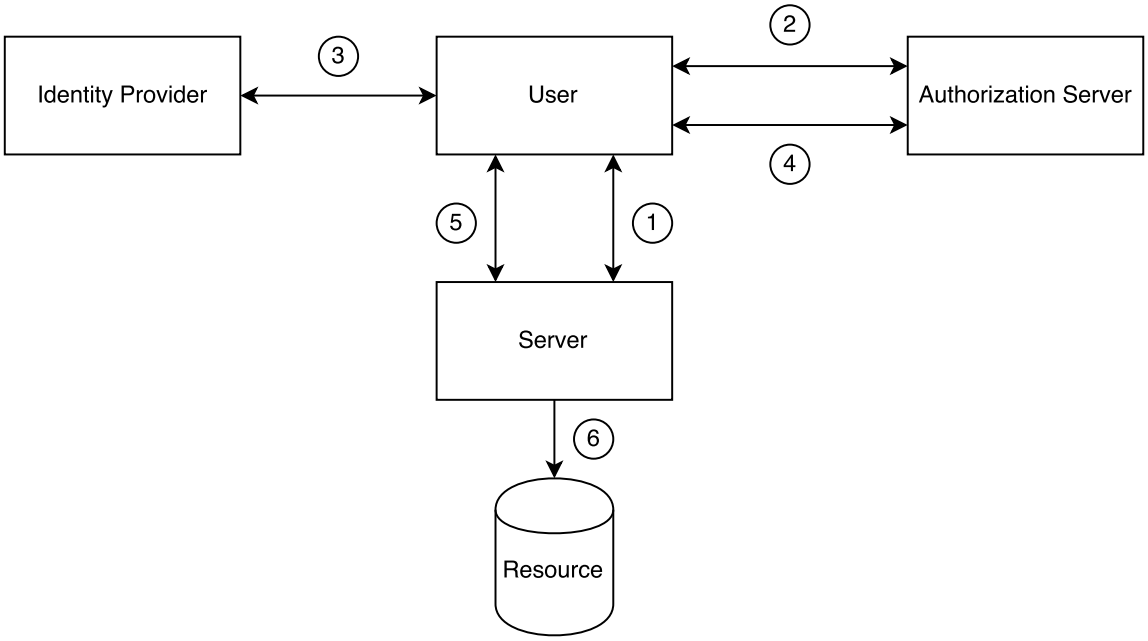

Figure 2.6: Common Federated Identity Flow

1. The user attempts to access a resource controlled by a server.
2. The user does not have the proper credentials to access the resource, so the server redirects the user to the authorization server. The authorization server is configured to let users log-in using the credentials managed by an identity provider.
3. The user gets redirected by the authorization server to the identity's provider log-in screen.
4. The user logs-in successfully and gets redirected to the authorization server. The authorization server uses the credentials provided by the identity provider to access the credentials required by the resource server.
5. The user gets redirected to the resource server by the authorization server. The request now has the correct credentials required to access the resource.
6. The user gets access to the resource successfully.

All the data passed from server to server flows through the user by being embedded in the redirection requests (usually as part of the URL). This makes transport security (TLS) and data security essential.

The credentials returned from the authorization server to the user can be encoded as a JWT. If the authorization server allows logins through an identity provider (as is the case in this example), the authorization server can be said to be providing a unified interface and unified data (the JWT) to the user.

For our example later in this section, we will use Auth0 as the authorization server and handle logins through Twitter, Facebook, and a run-of-the-mill user database.

2.2.1 Access and Refresh Tokens

Access and refresh tokens are two types of tokens you will see a lot when analyzing different federated identity solutions. We will briefly explain what they are and how they help in the context of authentication and authorization.

Both concepts are usually implemented in the context of the OAuth2 specification[10]. The OAuth2 spec defines a series of steps necessary to provide access to resources by separating access from ownership (in other words, it allows several parties with different access levels to access the same resource). Several parts of these steps are *implementation defined*. That is, competing OAuth2 implementations may not be interoperable. For instance, the actual binary format of the tokens is *not specified*. Their purpose and functionality is.

Access tokens are tokens that give those who have them access to protected resources. These tokens are usually short-lived and may have an expiration date embedded in them. They may also carry or be associated with additional information (for instance, an access token may carry the IP address from which requests are allowed). This additional data is implementation defined.

Refresh tokens, on the other hand, allow clients to request new access tokens. For instance, after an access token has expired, a client may perform a request for a new access token to the authorization server. For this request to be satisfied, a refresh token is required. In contrast to access tokens, refresh tokens are usually long-lived.

[10] https://tools.ietf.org/html/rfc6749#section-1.4

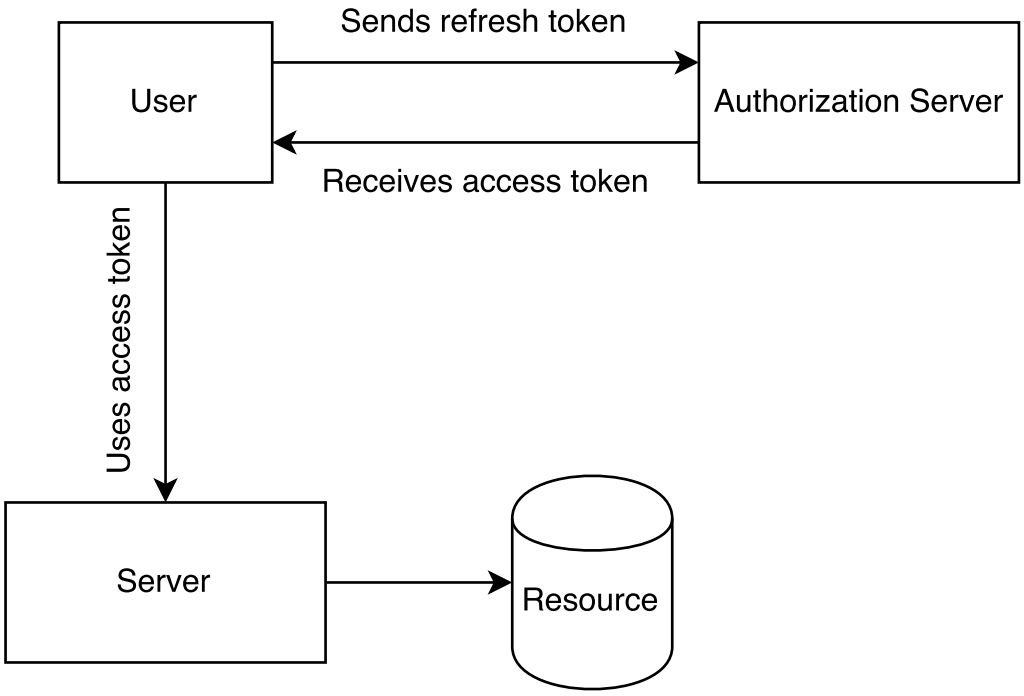

Figure 2.7: Refresh and access tokens

The key aspect of the separation between access and refresh tokens lies in the possibility of making access tokens easy to validate. An access token that carries a signature (such as a signed JWT) may be validated by the resource server on its own. There is no need to contact the authorization server for this purpose.

Refresh tokens, on the other hand, require access to the authorization server. By keeping validation separate from queries to the authorization server, better latency and less complex access patterns are possible. Appropriate security in case of token leaks is achieved by making access tokens as short-lived as possible and embedding additional checks (such as client checks) into them.

Refresh tokens, by virtue of being long-lived, must be protected from leaks. In the event of a leak, blacklisting may be necessary in the server (short-lived access tokens force refresh tokens to be used eventually, thus protecting the resource after it gets blacklisted and all access tokens are expired).

> Note: the concepts of access token and refresh token were introduced in OAuth2. OAuth 1.0 and 1.0a use the word *token* differently.

2.2.2 JWTs and OAuth2

Although OAuth2 makes no mention of the format of its tokens, JWTs are a good match for its requirements. Signed JWTs make good access tokens, as they can encode all the necessary data

to differentiate access levels to a resource, can carry an expiration date, and are signed to avoid validation queries against the authorization server. Several federated identity providers issue access tokens in JWT format.

JWTs may also be used for refresh tokens. There is less reason to use them for this purpose, though. As refresh tokens require access to the authorization server, most of the time a simple UUID will suffice, as there is no need for the token to carry a payload (it may be signed, though).

2.2.3 JWTs and OpenID Connect

OpenID Connect[11] is a standardization effort to bring typical use cases of OAuth2 under a common, well-defined spec. As many details behind OAuth2 are left to the choice of implementers, OpenID Connect attempts to provide proper definitions for the missing parts. Specifically, OpenID Connect defines an API and data format to perform OAuth2 authorization flows. Additionally, it provides an authentication layer built on top of this flow. The data format chosen for some of its parts is JSON Web Token. In particular, the ID token[12] is a special type of token that carries information about the authenticated user.

2.2.3.1 OpenID Connect Flows and JWTs

OpenID Connect defines several flows which return data in different ways. Some of this data may be in JWT format.

- **Authorization flow**: the client requests an authorization code to the authorization endpoint (`/authorize`). This code can be used againt the token endpoint (`/token`) to request an ID token (in JWT format), an access token or a refresh token.
- **Implicit flow**: the client requests tokens directly from the authorization endpoint (`/authorize`). The tokens are specified in the request. If an ID token is requested, is is returned in JWT format.
- **Hybrid flow**: the client requests both an authorization code and certain tokens from the authorization endpoint (`/authorize`). If an ID token is requested, it is returned in JWT format. If an ID token is not requested at this step, it may later by requested directly from the token endpoint (`/token`).

2.2.4 Example

For this example we will use Auth0[13] as the authorization server. Auth0 allows for different identity providers to be set dynamically. In other words, whenever a user attempts to login, changes made in the authorization server may allow users to login with different identity providers (such as Twitter, Facebook, etc). Applications need not commit to specific providers once deployed. So our example

[11] https://openid.net/connect/
[12] http://openid.net/specs/openid-connect-core-1_0.html#IDToken
[13] https://auth0.com

can be quite simple. We set up the Auth0 login screen using the Auth0.js library[14] in all of our sample servers. Once a user logs in to one server, he will also have access to the other servers (even if they are not interconnected).

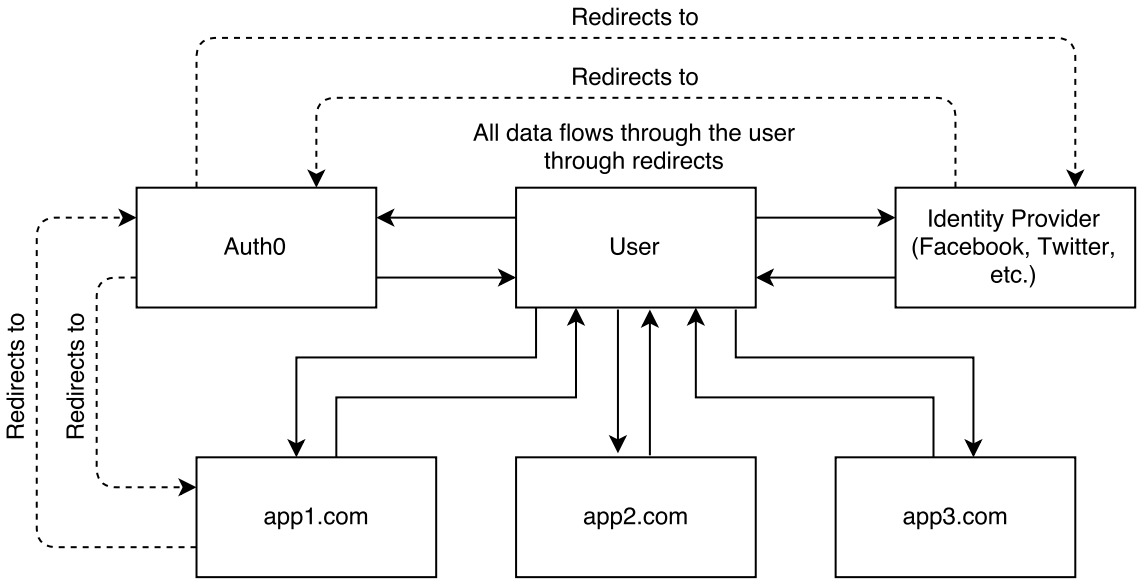

Figure 2.8: Auth0 as Authorization Server

2.2.4.1 Setting up Auth0 Lock for Node.js Applications

Setting up the Auth0 library[15] can be done as follows. We will use the same example used for the stateless sessions example:

```
const auth0 = new window.auth0.WebAuth({
    domain: domain,
    clientID: clientId,
    audience: 'app1.com/protected',
    scope: 'openid profile purchase',
    responseType: 'id_token token',
    redirectUri: 'http://app1.com:3000/auth/',
    responseMode: 'form_post'
});

// (...)
```

[14] https://github.com/auth0/auth0.js
[15] https://github.com/auth0/auth0.js

```
$('#login-button').on('click', function(event) {
    auth0.authorize({
        prompt: 'none'
    });
});
```

Note the use of the `prompt: 'none'` parameter for the `authorize` call. The `authorize` call redirects the user to the authorization server. With the `none` parameter, if the user has already given authorization for an app to use his or her credentials for access to a protected resource, the authorization server will simply redirect back to the application. This looks to the user as if he were already logged-in in the app.

In our example, there are two apps: `app1.com` and `app2.com`. Once a user has authorized both apps (which happens only once: the first time the user logs-in), any subsequent logins to any of both apps will also allow the other app to login without presenting any login screens.

To test this, see the `README` file for the example located in the `samples/single-sign-on-federated-identity` directory to set up both applications and run them. Once both are running, go to app1.com:3000[16] and app2.com:3001[17] and login. Then logout from both apps. Now attempt to login to one of them. Then go back to the other one and login. You will notice the login screen will be absent in both apps. The authorization server remembers previous logins and can issue new access tokens when requested by any of those apps. Thus, as long as the user has an authorization server session, he or she is already logged-in to both apps.

Implementing CSRF mitigation techniques is left as en exercise for the reader.

[16] http://app1.com:3000
[17] http://app2.com:3001

Chapter 3

JSON Web Tokens in Detail

As described in chapter 1, all JWTs are constructed from three different elements: the header, the payload, and the signature/encryption data. The first two elements are JSON objects of a certain structure. The third is dependent on the algorithm used for signing or encryption, and, in the case of *unencrypted* JWTs it is omitted. JWTs can be encoded in a *compact representation* known as *JWS/JWE Compact Serialization*.

> The JWS and JWE specifications define a third serialization format known as *JSON Serialization*, a non-compact representation that allows for multiple signatures or recipients in the same JWT. Is is explained in detail in chapters 4 and 5.

The compact serialization is a Base64[1] URL-safe encoding of the UTF-8[2] bytes of the first two JSON elements (the header and the payload) and the data, as required, for signing or encryption (which is not a JSON object itself). This data is Base64-URL encoded as well. These three elements are separated by dots (".").

> JWT uses a variant of Base64 encoding that is safe for URLs. This encoding basically substitutes the "+" and "/" characters for the "-" and "_" characters, respectively. Padding is removed as well. This variant is known as base64url[3]. Note that all references to Base64 encoding in this document refer to this variant.

The resulting sequence is a printable string like the following (newlines inserted for readability):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Notice the dots separating the three elements of the JWT (in order: the header, the payload, and the signature).

In this example the decoded header is:

[1] https://en.wikipedia.org/wiki/Base64
[2] https://en.wikipedia.org/wiki/UTF-8
[3] https://tools.ietf.org/html/rfc4648#section-5

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

The decoded payload is:

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

And the secret required for verifying the signature is `secret`.

> JWT.io[4] is an interactive playground for learning more about JWTs. Copy the token from above and see what happens when you edit it.

3.1 The Header

Every JWT carries a header (also known as the *JOSE header*) with claims about itself. These claims establish the algorithms used, whether the JWT is signed or encrypted, and in general, how to parse the rest of the JWT.

According to the type of JWT in question, more fields may be mandatory in the header. For instance, encrypted JWTs carry information about the cryptographic algorithms used for key encryption and content encryption. These fields are not present for unencrypted JWTs.

The only mandatory claim for an *unencrypted* JWT header is the `alg` claim:

- **alg**: the main algorithm in use for signing and/or decrypting this JWT.

For unencrypted JWTs this claim must be set to the value `none`.

Optional header claims include the `typ` and `cty` claims:

- **typ**: the media type[5] of the JWT itself. This parameter is only meant to be used as a help for uses where JWTs may be mixed with other objects carrying a JOSE header. In practice, this rarely happens. When present, this claim should be set to the value `JWT`.
- **cty**: the content type. Most JWTs carry specific claims plus arbitrary data as part of their payload. For this case, the content type claim *must not* be set. For instances where the payload is a JWT itself (a nested JWT), this claim *must* be present and carry the value `JWT`. This tells the implementation that further processing of the nested JWT is required. Nested JWTs are rare, so the `cty` claim is rarely present in headers.

So, for unencrypted JWTs, the header is simply:

[4] https://jwt.io
[5] http://www.iana.org/assignments/media-types/media-types.xhtml

```
{
    "alg": "none"
}
```

which gets encoded to:

```
eyJhbGciOiJub25lIn0
```

> It is possible to add additional, user-defined claims to the header. This is generally of limited use, unless certain user-specific metadata is required in the case of encrypted JWTs before decryption.

3.2 The Payload

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

The payload is the element where all the interesting user data is usually added. In addition, certain claims defined in the spec may also be present. Just like the header, the payload is a JSON object. No claims are mandatory, although specific claims have a definite meaning. The JWT spec specifies that claims that are not understood by an implementation should be ignored. The claims with specific meanings attached to them are known as *registered claims*.

3.2.1 Registered Claims

- **iss**: from the word *issuer*. A case-sensitive string or URI that uniquely identifies the party that issued the JWT. Its interpretation is application specific (there is no central authority managing issuers).
- **sub**: from the word *subject*. A case-sensitive string or URI that uniquely identifies the party that this JWT carries information about. In other words, the claims contained in this JWT are statements about this party. The JWT spec specifies that this claim must be unique in the context of the issuer or, in cases where that is not possible, globally unique. Handling of this claim is application specific.
- **aud**: from the word *audience*. Either a single case-sensitive string or URI or an array of such values that uniquely identify the intended recipients of this JWT. In other words, when this claim is present, the party reading the data in this JWT must find itself in the *aud* claim or disregard the data contained in the JWT. As in the case of the *iss* and *sub* claims, this claim is application specific.
- **exp**: from the word *expiration* (time). A number representing a specific date and time in the format "seconds since epoch" as defined by POSIX[6]. This claims sets the exact moment from

[6] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15

which this JWT is considered *invalid*. Some implementations may allow for a certain skew between clocks (by considering this JWT to be valid for a few minutes after the expiration date).

- **nbf**: from *not before* (time). The opposite of the *exp* claim. A number representing a specific date and time in the format "seconds since epoch" as defined by POSIX[7]. This claim sets the exact moment from which this JWT is considered *valid*. The current time and date must be equal to or later than this date and time. Some implementations may allow for a certain skew.
- **iat**: from *issued at* (time). A number representing a specific date and time (in the same format as *exp* and *nbf*) at which this JWT was issued.
- **jti**: from *JWT ID*. A string representing a unique identifier for this JWT. This claim may be used to differentiate JWTs with other similar content (preventing replays, for instance). It is up to the implementation to guarantee uniqueness.

As you may have noticed, all names are short. This complies with one of the design requirements: to keep JWTs as small as possible.

String or URI: according to the JWT spec, a URI is interpreted as any string containing a `:` character. It is up to the implementation to provide valid values.

3.2.2 Public and Private Claims

All claims that are not part of the *registered claims* section are either **private** or **public** claims.

- **Private** claims: are those that are defined by *users* (consumers and producers) of the JWTs. In other words, these are ad hoc claims used for a particular case. As such, care must be taken to prevent collisions.
- **Public** claims: are claims that are either *registered* with the IANA JSON Web Token Claims registry[8] (a registry where users can register their claims and thus prevent collisions), or named using a collision resistant name (for instance, by prepending a namespace to its name).

In practice, most claims are either registered claims or private claims. In general, most JWTs are issued with a specific purpose and a clear set of potential users in mind. This makes the matter of picking collision resistant names simple.

Just as in the JSON parsing rules, duplicate claims (duplicate JSON keys) are handled by keeping only the last occurrence as the valid one. The JWT spec also makes it possible for implementations to consider JWTs with duplicate claims as *invalid*. In practice, if you are not sure about the implementation that will handle your JWTs, take care to avoid duplicate claims.

[7] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15
[8] https://tools.ietf.org/html/rfc7519#section-10.1

3.3 Unsecured JWTs

With what we have learned so far, it is possible to construct unsecured JWTs. These are the simplest JWTs, formed by a simple (usually static) header:

```
{
    "alg": "none"
}
```

and a user defined payload. For instance:

```
{
  "sub": "user123",
  "session": "ch72gsb320000udocl363eofy",
  "name": "Pretty Name",
  "lastpage": "/views/settings"
}
```

As there is no signature or encryption, this JWT is encoded as simply two elements (newlines inserted for readability):

```
eyJhbGciOiJub25lIn0.
eyJzdWIiOiJ1c2VyMTIzIiwic2Vzc2lvbiI6ImNoNzJnc2IzMjAwMDB1ZG9jbDM2M
2VvZnkiLCJuYW1lIjoiUHJldHR5IE5hbWUiLCJsYXN0cGFnZSI6Ii92aWV3cy9zZXR0aW5ncyJ9.
```

An unsecured JWT like the one shown above may be fit for client-side use. For instance, if the session ID is a hard-to-guess number, and the rest of the data is only used by the client for constructing a view, the use of a signature is superfluous. This data can be used by a single-page web application to construct a view with the "pretty" name for the user without hitting the backend while he gets redirected to his last visited page. Even if a malicious user were to modify this data he or she would gain nothing.

> Note the trailing dot (`.`) in the compact representation. As there is no signature, it is simply an empty string. The dot is still added, though.

In practice, however, unsecured JWTs are rare.

3.4 Creating an Unsecured JWT

To arrive at the compact representation from the JSON versions of the header and the payload, perform the following steps:

1. Take the header as a byte array of its UTF-8 representation. The JWT spec *does not* require the JSON to be minified or stripped of meaningless characters (such as whitespace) before encoding.
2. Encode the byte array using the Base64-URL algorithm, removing trailing equal signs (`=`).
3. Take the payload as a byte array of its UTF-8 representation. The JWT spec *does not* require the JSON to be minified or stripped of meaningless characters (such as whitespace) before encoding.

4. Encode the byte array using the Base64-URL algorithm, removing trailing equal signs (=).
5. Concatenate the resulting strings, putting first the header, followed by a "." character, followed by the payload.

Validation of both the header and the payload (with respect to the presence of required claims and the correct use of each claim) must be performed before encoding.

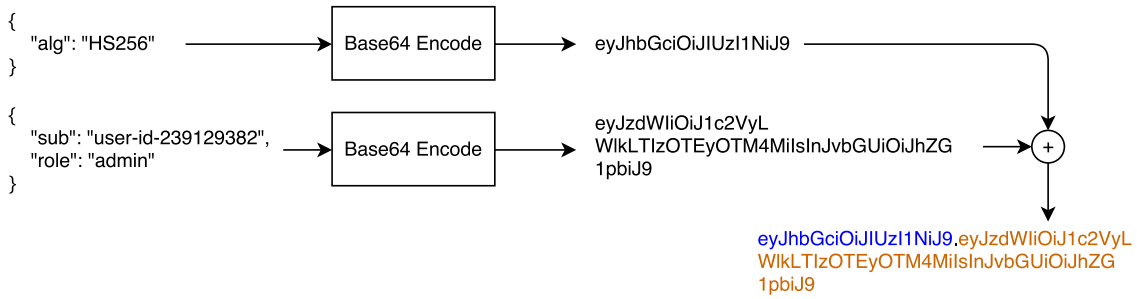

Figure 3.1: Compact Unsecured JWT Generation

3.4.1 Sample Code

```
// URL-safe variant of Base64
function b64(str) {
    return new Buffer(str).toString('base64')
                          .replace(/=/g, '')
                          .replace(/\+/g, '-')
                          .replace(/\//g, '_');
}

function encode(h, p) {
    const headerEnc = b64(JSON.stringify(h));
    const payloadEnc = b64(JSON.stringify(p));
    return `${headerEnc}.${payloadEnc}`;
}
```

The full example is in file `coding.js` of the accompanying sample code.

3.5 Parsing an Unsecured JWT

To arrive at the JSON representation from the compact serialization form, perform the following steps:

1. Find the first period "." character. Take the string before it (not including it.)

2. Decode the string using the Base64-URL algorithm. The result is the JWT header.
3. Take the string after the period from step 1.
4. Decode the string using the Base64-URL algorithm. The result is the JWT payload.

The resulting JSON strings may be "prettified" by adding whitespace as necessary.

3.5.1 Sample Code

```
function decode(jwt) {
    const [headerB64, payloadB64] = jwt.split('.');
    // These supports parsing the URL safe variant of Base64 as well.
    const headerStr = new Buffer(headerB64, 'base64').toString();
    const payloadStr = new Buffer(payloadB64, 'base64').toString();
    return {
        header: JSON.parse(headerStr),
        payload: JSON.parse(payloadStr)
    };
}
```

The full example is in file `coding.js` of the accompanying sample code.

Chapter 4

JSON Web Signatures

JSON Web Signatures are probably the single most useful feature of JWTs. By combining a simple data format with a well-defined series of signature algorithms, JWTs are quickly becoming the ideal format for safely sharing data between clients and intermediaries.

The purpose of a signature is to allow one or more parties to establish the *authenticity* of the JWT. Authenticity in this context means the data contained in the JWT has not been tampered with. In other words, any party that can perform a *signature check* can rely on the contents provided by the JWT. It is important to stress that a signature does not prevent other parties from *reading* the contents inside the JWT. This is what encryption is meant to do, and we will talk about that later in chapter 5.

The process of checking the signature of a JWT is known as *validation* or *validating* a token. A token is considered valid when all the restrictions specified in its header and payload are satisfied. This is a *very important* aspect of JWTs: implementations are required to check a JWT up to the point specified by both its header and its payload (and, additionally, whatever the user requires). So, a JWT may be considered valid *even if it lacks a signature* (if the header has the *alg* claim set to `none`). Additionally, even if a JWT has a valid signature, it may be considered invalid for other reasons (for instance, it may have expired, according to the *exp* claim). A common attack against signed JWTs relies on stripping the signature and then changing the header to make it an unsecured JWT. It is the responsibility of the user to make sure JWTs are validated according to their own requirements.

Signed JWTs are defined in the JSON Web Signature spec, RFC 7515[1].

4.1 Structure of a Signed JWT

We have covered the structure of a JWT in chapter 3. We will review it here and take special note of its signature component.

[1] https://tools.ietf.org/html/rfc7515

A signed JWT is composed of three elements: the header, the payload, and the signature (newlines inserted for readability):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

The process for decoding the first two elements (the header and the payload) is identical to the case of unsecured JWTs. The algorithm and sample code can be found at the end of chapter 3.

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

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

Signed JWTs, however, carry an additional element: the signature. This element appears after the last dot (`.`) in the compact serialization form.

There are several types of signing algorithms available according to the JWS spec, so the way these octets are interpreted varies. The JWS specification requires a single algorithm to be supported by all conforming implementations:

- HMAC using SHA-256, called `HS256` in the JWA spec.

The specification also defines a series of *recommended* algorithms:

- RSASSA PKCS1 v1.5 using SHA-256, called `RS256` in the JWA spec.
- ECDSA using P-256 and SHA-256, called `ES256` in the JWA spec.

  JWA is the JSON Web Algorithms spec, RFC 7518[2].

These algorithms will be explained in detail in chapter 7. In this chapter, we will focus on the practical aspects of their use.

The other algorithms supported by the spec, in optional capacity, are:

- HS384, HS512: SHA-384 and SHA-512 variations of the HS256 algorithm.
- RS384, RS512: SHA-384 and SHA-512 variations of the RS256 algorithm.
- ES384, ES512: SHA-384 and SHA-512 variations of the ES256 algorithm.
- PS256, PS384, PS512: RSASSA-PSS + MGF1 with SHA256/384/512 variants.

These are, essentially, variations of the three main required and recommended algorithms. The meaning of these acronyms will become clearer in chapter 7.

[2] https://tools.ietf.org/html/rfc7518

4.1.1 Algorithm Overview for Compact Serialization

In order to discuss these algorithms in general, let's first define some functions in a JavaScript 2015 environment:

- **base64:** a function that receives an array of octets and returns a new array of octets using the Base64-URL algorithm.
- **utf8:** a function that receives text in any encoding and returns an array of octets with UTF-8 encoding.
- **JSON.stringify:** a function that takes a JavaScript object and serializes it to string form (JSON).
- **sha256:** a function that takes an array of octets and returns a new array of octets using the SHA-256 algorithm.
- **hmac:** a function that takes a SHA function, an array of octets and a secret and returns a new array of octets using the HMAC algorithm.
- **rsassa:** a function that takes a SHA function, an array of octets and the private key and returns a new array of octets using the RSASSA algorithm.

For HMAC-based signing algorithms:

```
const encodedHeader = base64(utf8(JSON.stringify(header)));
const encodedPayload = base64(utf8(JSON.stringify(payload)));
const signature = base64(hmac(`${encodedHeader}.${encodedPayload}`,
                              secret, sha256));
const jwt = `${encodedHeader}.${encodedPayload}.${signature}`;
```

For public-key signing algorithms:

```
const encodedHeader = base64(utf8(JSON.stringify(header)));
const encodedPayload = base64(utf8(JSON.stringify(payload)));
const signature = base64(rsassa(`${encodedHeader}.${encodedPayload}`,
                                privateKey, sha256));
const jwt = `${encodedHeader}.${encodedPayload}.${signature}`;
```

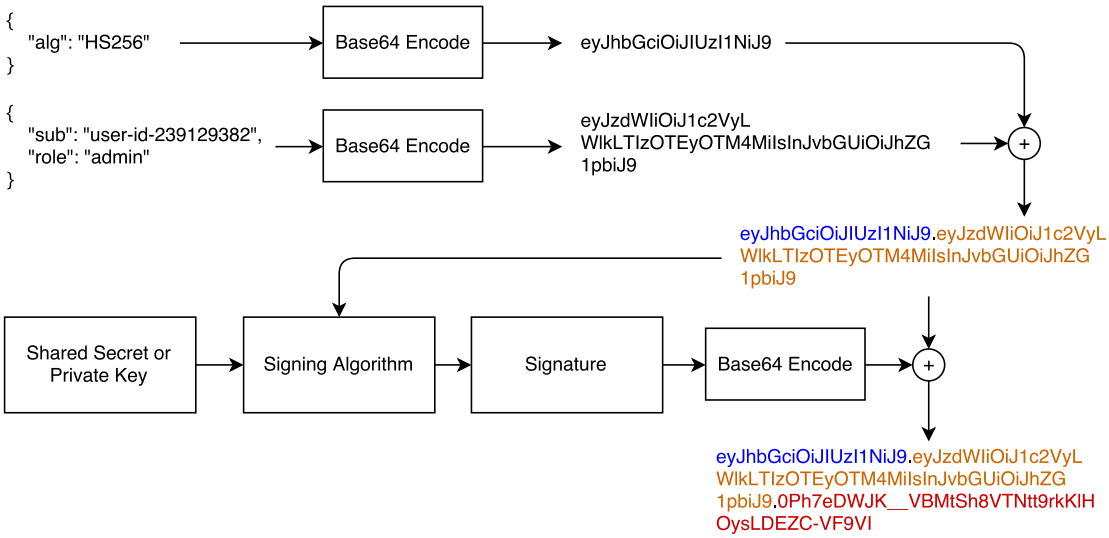

Figure 4.1: JWS Compact Serialization

The full details of these algorithms are shown in chapter 7.

4.1.2 Practical Aspects of Signing Algorithms

All signing algorithms accomplish the same thing: they provide a way to establish the authenticity of the data contained in the JWT. How they do that varies.

Keyed-Hash Message Authentication Code (HMAC) is an algorithm that combines a certain payload with a *secret* using a cryptographic hash function[3]. The result is a code that can be used to verify a message *only* if both the generating and verifying parties know the secret. In other words, **HMACs allow messages to be verified through shared secrets**.

The cryptographic hash function used in HS256, the most common signing algorithm for JWTs, is SHA-256. SHA-256 is explained in detail in chapter 7. Cryptographic hash functions take a message of arbitrary length and produce an output of fixed length. The same message will always produce the same output. The *cryptographic* part of a hash function makes sure that it is mathematically infeasible to recover the original message from the output of the function. In this way, cryptographic hash functions are *one-way functions* that can be used to identify messages without actually sharing the message. A slight variation in the message (a single byte, for instance) will produce an entirely different output.

RSASSA is a variation of the RSA algorithm[4] (explained in chapter 7) adapted for signatures. RSA is a public-key algorithm. Public-key algorithms generate split keys: one public key and one private

[3] https://en.wikipedia.org/wiki/Cryptographic_hash_function
[4] https://en.wikipedia.org/wiki/RSA_%28cryptosystem%29

key. In this specific variation of the algorithm, the private key can be used both to create a signed message and to verify its authenticity. The public key, in contrast, can only be used to verify the authenticity of a message. Thus, this scheme allows for the secure distribution of a **one-to-many** message. Receiving parties can verify the authenticity of a message by keeping a copy of the public key associated with it, but they cannot create new messages with it. This allows for different usage scenarios than shared-secret signing schemes such as HMAC. With HMAC + SHA-256, any party that can verify a message can also *create new messages*. For example, if a legitimate user turned malicious, he or she could modify messages without the other parties noticing. With a public-key scheme, a user who turned malicious would only have the public key in his or her possession and so could not create new signed messages with it.

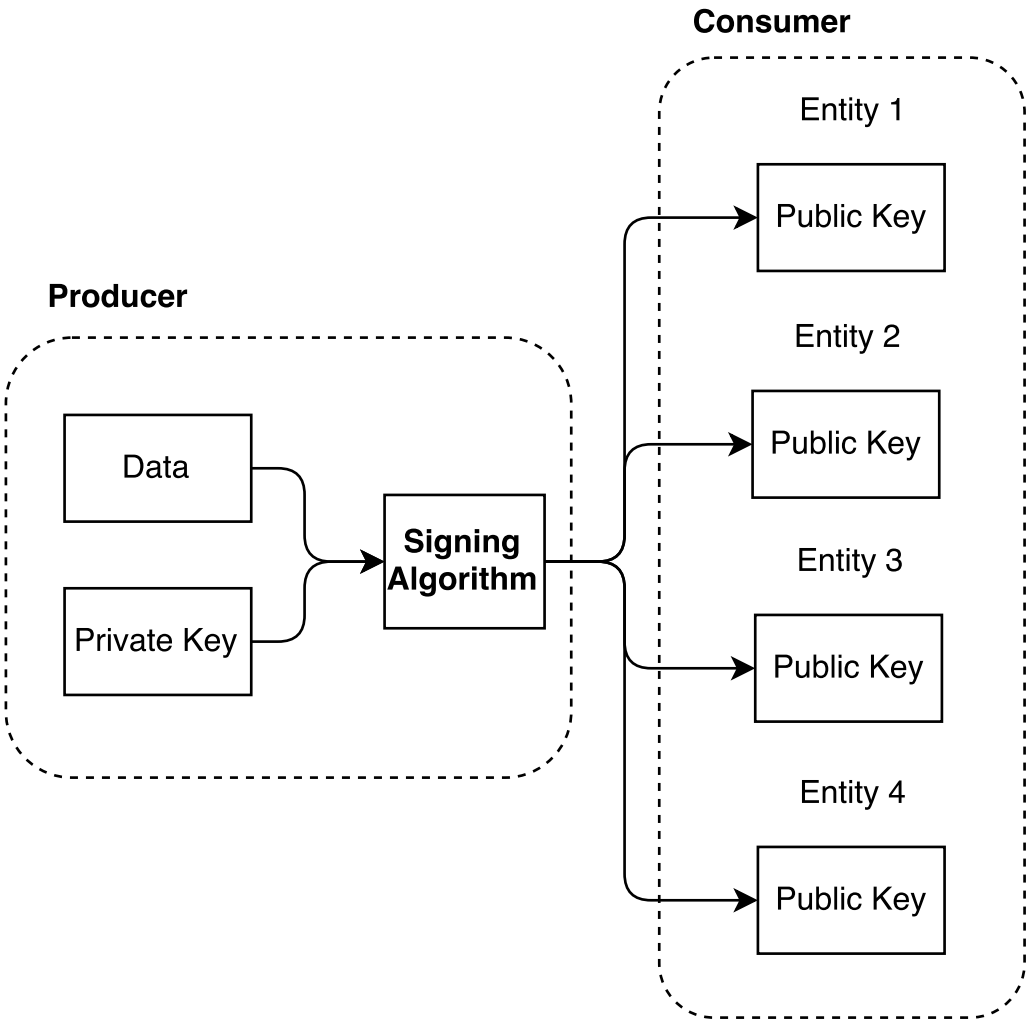

Figure 4.2: One-to-many signing

Public-key cryptography[5] allows for other usage scenarios. For instance, using a variation of the same RSA algorithm, it is possible to encrypt messages by using the public key. These messages can only be decrypted using the private key. This allows a **many-to-one** secure communications channel to be constructed. This variation is used for encrypted JWTs, which are discussed in

```
<div id="chapter5"></div>
```

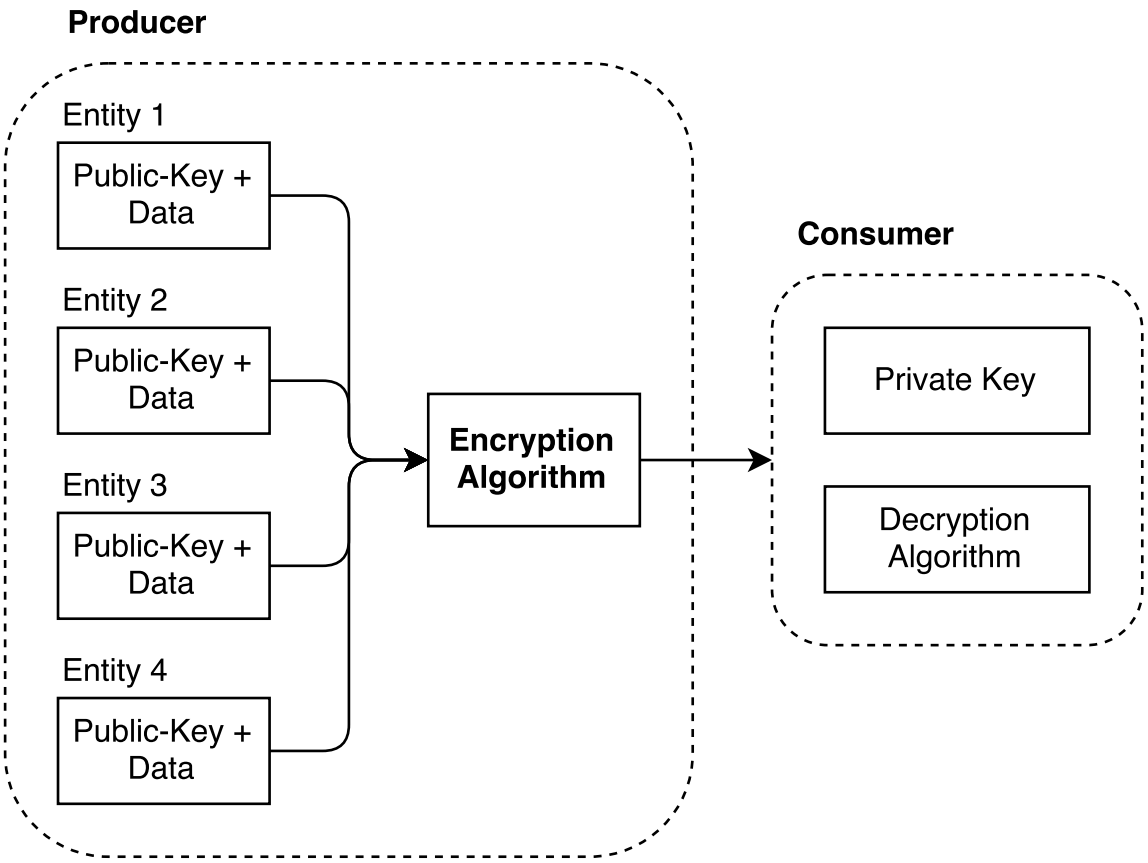

Figure 4.3: Many-to-one encryption

Elliptic Curve Digital Signature Algorithm (ECDSA)[6] is an alternative to RSA. This algorithm also generates a public and private key pair, but the mathematics behind it are different. This difference allows for lesser hardware requirements than RSA for similar security guarantees.

We will study these algorithms in more detail in chapter 7.

[5] https://en.wikipedia.org/wiki/Public-key_cryptography
[6] https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm

4.1.3 JWS Header Claims

JWS allows for special use cases that force the header to carry more claims. For instance, for public-key signing algorithms, it is possible to embed the URL to the public key as a claim. What follows is the list of registered header claims available for JWS tokens. All of these claims are *in addition* to those available for unsecured JWTs, and are optional depending on how the signed JWT is meant to be used.

- **jku**: JSON Web Key (JWK) Set URL. A URI pointing to a set of JSON-encoded public keys used to sign this JWT. Transport security (such as TLS for HTTP) must be used to retrieve the keys. The format of the keys is a JWK Set (see chapter 6).
- **jwk**: JSON Web Key. The key used to sign this JWT in JSON Web Key format (see chapter 6).
- **kid**: Key ID. A user-defined string representing a single key used to sign this JWT. This claim is used to signal key signature changes to recipients (when multiple keys are used).
- **x5u**: X.509 URL. A URI pointing to a set of X.509 (a certificate format standard) public certificates encoded in PEM form. The first certificate in the set must be the one used to sign this JWT. The subsequent certificates each sign the previous one, thus completing the certificate chain. X.509 is defined in RFC 5280[7]. Transport security is required to transfer the certificates.
- **x5c**: X.509 certificate chain. A JSON array of X.509 certificates used to sign this JWS. Each certificate must be the Base64-encoded value of its DER PKIX representation. The first certificate in the array must be the one used to sign this JWT, followed by the rest of the certificates in the certificate chain.
- **x5t**: X.509 certificate SHA-1 fingerprint. The SHA-1 fingerprint of the X.509 DER-encoded certificate used to sign this JWT.
- **x5t#S256**: Identical to **x5t**, but uses SHA-256 instead of SHA-1.
- **typ**: Identical to the **typ** value for unencrypted JWTs, with additional values "JOSE" and "JOSE+JSON" used to indicate compact serialization and JSON serialization, respectively. This is only used in cases where similar JOSE-header carrying objects are mixed with this JWT in a single container.
- **crit**: from *critical*. An array of strings with the names of claims that are present in this same header used as implementation-defined extensions that must be handled by parsers of this JWT. It must either contain the names of claims or not be present (the empty array is not a valid value).

4.1.4 JWS JSON Serialization

The JWS spec defines a different type of serialization format that is not compact. This representation allows for multiple signatures in the same signed JWT. It is known as *JWS JSON Serialization*.

[7] https://tools.ietf.org/html/rfc5280

In JWS JSON Serialization form, signed JWTs are represented as printable text with JSON format (i.e., what you would get from calling `JSON.stringify` in a browser). A topmost JSON object that carries the following key-value pairs is required:

- **payload**: a Base64 encoded string of the actual JWT payload object.
- **signatures**: an array of JSON objects carrying the signatures. These objects are defined below.

In turn, each JSON object inside the **signatures** array must contain the following key-value pairs:

- **protected**: a Base64 encoded string of the JWS header. Claims contained in this header are protected by the signature. This header is required only if there are no unprotected headers. If unprotected headers are present, then this header may or may not be present.
- **header**: a JSON object containing header claims. This header is unprotected by the signature. If no protected header is present, then this element is mandatory. If a protected header is present, then this element is optional.
- **signature**: A Base64 encoded string of the JWS signature.

In contrast to compact serialization form (where only a protected header is present), JSON serialization admits two types of headers: **protected** and **unprotected**. The protected header is validated by the signature. The unprotected header is not validated by it. It is up to the implementation or user to pick which claims to put in either of them. At least one of these headers must be present. Both may be present at the same time as well.

When both protected and unprotected headers are present, the actual JOSE header is built from the union of the elements in both headers. No duplicate claims may be present.

The following example is taken from the JWS RFC[8]:

```
{
  "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogIm
              h0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290IjpOcnVlfQ",
  "signatures": [
    {
      "protected": "eyJhbGciOiJSUzI1NiJ9",
      "header": { "kid": "2010-12-29" },
      "signature":
        "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AA
        uHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAyn
        RFdiuB--f_nZLgrnbyTyWzO5vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB
        _eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqvhJ1phCnvWh6
        IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrBp0igcN_IoypGlU
        PQGe77Rw"
    },
    {
      "protected": "eyJhbGciOiJFUzI1NiJ9",
      "header": { "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
```

[8] https://tools.ietf.org/html/rfc7515#appendix-A.6

```
      "signature": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDx
                    w5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
    }
  ]
}
```

This example encodes two signatures for the same payload: a RS256 signature and an ES256 signature.

4.1.4.1 Flattened JWS JSON Serialization

JWS JSON serialization defines a simplified form for JWTs with only a single signature. This form is known as *flattened JWS JSON serialization*. Flattened serialization removes the *signatures* array and puts the elements of a single signature at the same level as the *payload* element.

For example, by removing one of the signatures from the previous example, a flattened JSON serialization object would be:

```
{
  "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQog
              Imh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
  "protected": "eyJhbGciOiJFUzI1NiJ9",
  "header": { "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
  "signature": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFC
                gfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
}
```

4.2 Signing and Validating Tokens

The algorithms used for signing and validating tokens are explained in detail in chapter 7. Using signed JWTs is simple enough in practice that you could apply the concepts explained so far to use them effectively. Furthermore, there are good libraries you can use to implement them conveniently. We will go over the required and recommended algorithms using the most popular of these libraries for JavaScript. Examples of other popular languages and libraries can be found in the accompanying code.

The following examples all make use of the popular `jsonwebtoken` JavaScript library.

```
import jwt from 'jsonwebtoken'; //var jwt = require('jsonwebtoken');

const payload = {
  sub: "1234567890",
  name: "John Doe",
  admin: true
};
```

4.2.1 HS256: HMAC + SHA-256

HMAC signatures require a shared secret. Any string will do:

```
const secret = 'my-secret';

const signed = jwt.sign(payload, secret, {
    algorithm: 'HS256',
    expiresIn: '5s' // if ommited, the token will not expire
});
```

Verifying the token is just as easy:

```
const decoded = jwt.verify(signed, secret, {
    // Never forget to make this explicit to prevent
    // signature stripping attacks
    algorithms: ['HS256'],
});
```

The `jsonwebtoken` library checks the validity of the token based on the signature and the expiration date. In this case, if the token were to be checked after 5 seconds of being created, it would be considered invalid and an exception would be thrown.

4.2.2 RS256: RSASSA + SHA256

Signing and verifying RS256 signed tokens is just as easy. The only difference lies in the use of a private/public key pair rather than a shared secret. There are many ways to create RSA keys. OpenSSL is one of the most popular libraries for key creation and management:

```
# Generate a private key
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits:2048
# Derive the public key from the private key
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Both PEM files are simple text files. Their contents can be copied and pasted into your JavaScript source files and passed to the `jsonwebtoken` library.

```
// You can get this from private_key.pem above.
const privateRsaKey = `<YOUR-PRIVATE-RSA-KEY>`;

const signed = jwt.sign(payload, privateRsaKey, {
    algorithm: 'RS256',
    expiresIn: '5s'
});

// You can get this from public_key.pem above.
const publicRsaKey = `<YOUR-PUBLIC-RSA-KEY>`;

const decoded = jwt.verify(signed, publicRsaKey, {
```

```
    // Never forget to make this explicit to prevent
    // signature stripping attacks.
    algorithms: ['RS256'],
});
```

4.2.3 ES256: ECDSA using P-256 and SHA-256

ECDSA algorithms also make use of public keys. The math behind the algorithm is different, though, so the steps to generate the keys are different as well. The "P-256" in the name of this algorithm tells us exactly which version of the algorithm to use (more details about this in chapter 7). We can use OpenSSL to generate the key as well:

```
# Generate a private key (prime256v1 is the name of the parameters used
# to generate the key, this is the same as P-256 in the JWA spec).
openssl ecparam -name prime256v1 -genkey -noout -out ecdsa_private_key.pem
# Derive the public key from the private key
openssl ec -in ecdsa_private_key.pem -pubout -out ecdsa_public_key.pem
```

If you open these files you will note that there is much less data in them. This is one of the benefits of ECDSA over RSA (more about this in chapter 7). The generated files are in PEM format as well, so simply pasting them in your source will suffice.

```
// You can get this from private_key.pem above.
const privateEcdsaKey = `<YOUR-PRIVATE-ECDSA-KEY>`;

const signed = jwt.sign(payload, privateEcdsaKey, {
    algorithm: 'ES256',
    expiresIn: '5s'
});

// You can get this from public_key.pem above.
const publicEcdsaKey = `<YOUR-PUBLIC-ECDSA-KEY>`;

const decoded = jwt.verify(signed, publicEcdsaKey, {
    // Never forget to make this explicit to prevent
    // signature stripping attacks.
    algorithms: ['ES256'],
});
```

Refer to chapter 2 for practical applications of these algorithms in the context of JWTs.

Chapter 5

JSON Web Encryption (JWE)

While JSON Web Signature (JWS) provides a means to *validate* data, JSON Web Encryption (JWE) provides a way to keep data *opaque* to third parties. Opaque in this case means *unreadable*. Encrypted tokens cannot be inspected by third parties. This allows for additional interesting use cases.

Although it would appear that encryption provides the same guarantees as validation, with the additional feature of making data unreadable, this is not always the case. To understand why, first it is important to note that just as in JWS, JWE essentially provides two schemes: a shared secret scheme, and a public/private-key scheme.

The shared secret scheme works by having all parties know a shared secret. Each party that holds the shared secret can both **encrypt** and **decrypt** information. This is analogous to the case of a shared secret in JWS: parties holding the secret can both verify and generate signed tokens.

The public/private-key scheme, however, works differently. While in JWS the party holding the private key can sign and verify tokens, and the parties holding the public key can only verify those tokens, in JWE the party holding the private key is the only party that can **decrypt** the token. In other words, public-key holders can **encrypt** data, but only the party holding the private-key can **decrypt** (and encrypt) that data. In practice, this means that in JWE, parties holding the *public* key can introduce new data into an exchange. In contrast, in JWS, parties holding the public-key can only *verify* data but not introduce new data. In straightforward terms, JWE does not provide the same guarantees as JWS and, therefore, does not replace the role of JWS in a token exchange. JWS and JWE are complementary when public/private key schemes are being used.

A simpler way to understand this is to think in terms of producers and consumers. The producer either signs or encrypts the data, so consumers can either validate it or decrypt it. In the case of JWT signatures, the private-key is used to sign JWTs, while the public-key can be used to validate it. The producer holds the private-key and the consumers hold the public-key. Data can *only* flow from private-key holders to public-key holders. In contrast, for JWT encryption, the public-key is used to encrypt the data and the private-key to decrypt it. In this case, the data can *only* flow from public-key holders to private-key holders - public-key holders are the producers and private-key holders are the consumers:

	JWS	JWE
Producer	Private-key	Public-key
Consumer	Public-key	Private-key

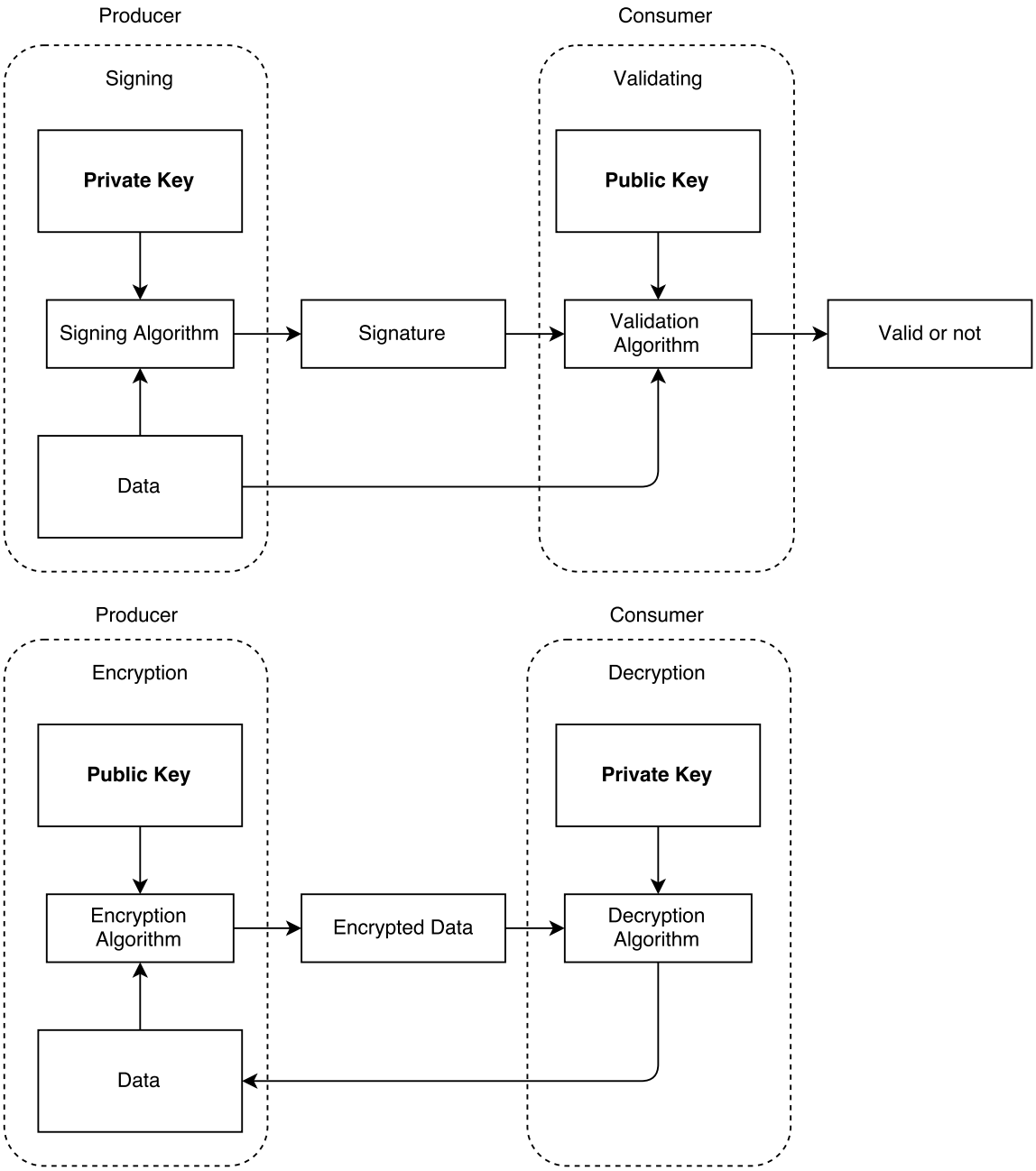

Figure 5.1: Signing vs encryption using public-key cryptography

At this point some people may ask:

> In the case of JWE, couldn't we distribute the private-key to every party that wants to send data to a consumer? Thus if a consumer can decrypt the data, he or she can be sure that it is also valid (because one cannot change data that cannot be decrypted).

Technically, it would be possible, but it wouldn't make sense. Sharing the private-key is *equivalent* to sharing the secret. So sharing the private-key in essence turns the scheme into a shared secret scheme, without the actual benefits of public-keys (remember public-keys can be derived from private-keys).

For this reason encrypted JWTs are sometimes *nested*: an encrypted JWT serves as the container for a signed JWT. This way you get the benefits of both.

> Note that all of this applies in situations where consumers are different entities from producers. If the producer is the same entity that consumes the data, then a shared-secret encrypted JWT provides the same guarantees as an encrypted *and* signed JWT.
>
> JWE encrypted JWTs, regardless of having a nested signed JWT in them or not, carry an authentication tag. This tag allows JWE JWTs to be validated. However, due to the issues mentioned above, this signature does not apply for the same use cases as JWS signatures. The purpose of this tag is to prevent padding oracle attacks[1] or ciphertext manipulation.

5.1 Structure of an Encrypted JWT

In contrast to signed and unsecured JWTs, encrypted JWTs have a different compact representation (newlines inserted for readability):

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-kFm1NJn8LE9XShH59_
i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKxGHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-
YxzZIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8Otv
zlV7elprCbuPhcCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-
cFPgwCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGljb3RoZQ.
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.
9hH0vgRfYgPnAHOd8stkvw
```

Although it may be hard to see in the example above, JWE Compact Serialization has five elements. As in the case of JWS, these elements are separated by dots, and the data contained in them is Base64-encoded.

The five elements of the compact representation are, in order:

1. **The protected header:** a header analogous to the JWS header.
2. **The encrypted key:** a symmetric key used to encrypt the ciphertext and other encrypted data. This key is derived from the actual encryption key specified by the user and thus is encrypted by it.

[1] https://en.wikipedia.org/wiki/Padding_oracle_attack

3. **The initialization vector:** some encryption algorithms require additional (usually random) data.
4. **The encrypted data (ciphertext):** the actual data that is being encrypted.
5. **The authentication tag:** additional data produced by the algorithms that can be used to *validate* the contents of the ciphertext against tampering.

As in the case of JWS and single signatures in the compact serialization, JWE supports a single encryption key in its compact form.

> Using a symmetric key to perform the actual encryption process is a common practice when using asymmetric encryption (public/private-key encryption). Asymmetric encryption algorithms are usually of high computational complexity, and thus encrypting long sequences of data (the ciphertext) is suboptimal. One way to exploit the benefits of both symmetric (faster) and asymmetric encryption is to generate a random key for a symmetric encryption algorithm, then encrypt that key with the asymmetric algorithm. This is the second element shown above, the encrypted key.
>
> Some encryption algorithms can process any data passed to them. If the ciphertext is modified (even without being decrypted), the algorithms may process it nonetheless. The authentication tag can be used to prevent this, essentially acting as a signature. This does not, however, remove the need for the nested JWTs explained above.

5.1.1 Key Encryption Algorithms

Having an encrypted encryption key means there are two encryption algorithms at play in the same JWT. The following are the encryption algorithms available for key encryption:

- **RSA variants**: RSAES PKCS #1 v1.5 (RSAES-PKCS1-v1_5), RSAES OAEP and OAEP + MGF1 + SHA-256.
- **AES variants**: AES Key Wrap from 128 to 256-bits, AES Galois Counter Mode (GCM) from 128 to 256-bits.
- **Elliptic Curve variants**: Elliptic Curve Diffie-Hellman Ephemeral Static key agreement using concat KDF, and variants pre-wrapping the key with any of the non-GCM AES variants above.
- **PKCS #5 variants**: PBES2 (password based encryption) + HMAC (SHA-256 to 512) + non-GCM AES variants from 128 to 256-bits.
- **Direct**: no encryption for the encryption key (direct use of CEK).

None of these algorithms are actually required by the JWA specification. The following are the recommended (to be implemented) algorithms by the specification:

- **RSAES-PKCS1-v1_5** (marked for removal of the recommendation in the future)
- **RSAES-OAEP** with defaults (marked to become required in the future)
- **AES-128 Key Wrap**
- **AES-256 Key Wrap**
- **Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES)** using Concat KDF (marked to become required in the future)
- **ECDH-ES + AES-128 Key Wrap**

- **ECDH-ES + AES-256 Key Wrap**

Some of these algorithms require additional header parameters.

5.1.1.1 Key Management Modes

The JWE specification defines different key management modes. These are, in essence, ways in which the key used to encrypt the payload is determined. In particular, the JWE spec describes these modes of key management:

- **Key Wrapping**: the Content Encryption Key (CEK) is encrypted for the intended recipient using a *symmetric* encryption algorithm.

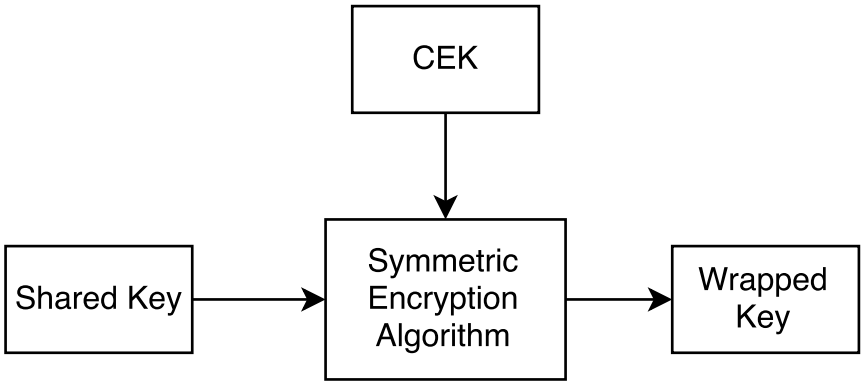

Figure 5.2: Key wrapping

- **Key Encryption**: the CEK is encrypted for the intended recipient using an *asymmetric* encryption algorithm.

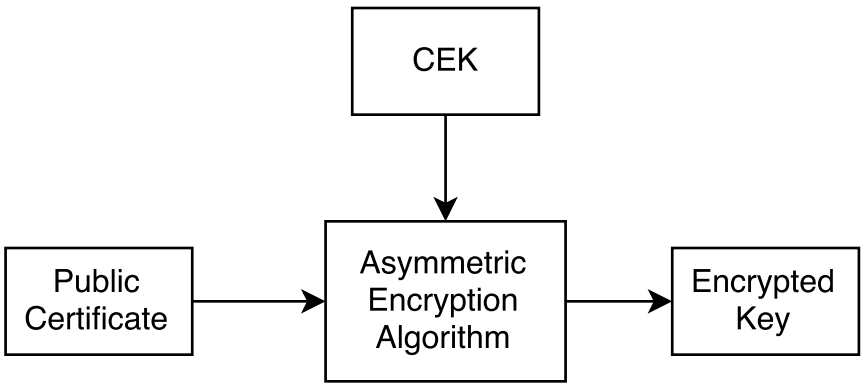

Figure 5.3: Key encryption

- **Direct Key Agreement**: a key agreement algorithm is used to pick the CEK.

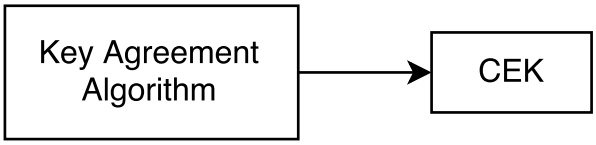

Figure 5.4: Direct key agreement

- **Key Agreement with Key Wrapping**: a key agreement algorithm is used to pick a symmetric CEK using a symmetric encryption algorithm.

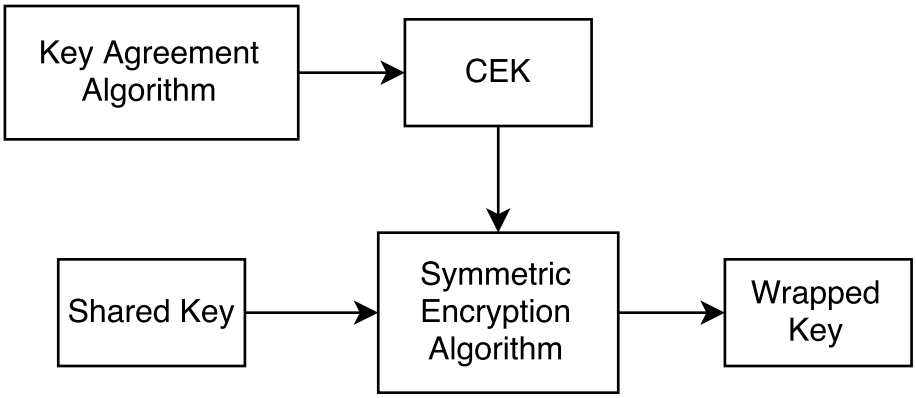

Figure 5.5: Direct key agreement

- **Direct Encryption**: a user-defined symmetric shared key is used as the CEK (no key derivation or generation).

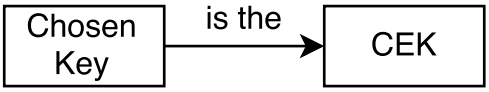

Figure 5.6: Direct key agreement

Although this constitutes a matter of terminology, it is important to understand the differences between each management mode and give each one of them a convenient name.

5.1.1.2 Content Encryption Key (CEK) and JWE Encryption Key

It is also important to understand the difference between the CEK and the JWE Encryption Key. The CEK is the actual key used to encrypt the payload: an encryption algorithm takes the CEK and the plaintext to produce the ciphertext. In contrast, the JWE Encryption Key is either the encrypted form of the CEK or an empty octet sequence (as required by the chosen algorithm). An

empty JWE Encryption Key means the algorithm makes use of an externally provided key to either directly decrypt the data (Direct Encryption) or compute the actual CEK (Direct Key Agreement).

5.1.2 Content Encryption Algorithms

The following are the content encryption algorithms, that is, the ones used to actually encrypt the payload:

- **AES CBC + HMAC SHA**: AES 128 to 256-bits with Cipher Block Chaining and HMAC + SHA-256 to 512 for validation.
- **AES GCM**: AES 128 to 256 using Galois Counter Mode.

Of these, only two are required: AES-128 CBC + HMAC SHA-256, and AES-256 CBC + HMAC SHA-512. The AES-128 and AES-256 variants using GCM are recommended.

These algorithms are explained in detail in chapter 7.

5.1.3 The Header

Just like the header for JWS and unsecured JWTs, the header carries all the necessary information for the JWT to be correctly processed by libraries. The JWE specification adapts the meanings of the registered claims defined in JWS to its own use, and adds a few claims of its own. These are the new and modified claims:

- **alg**: identical to JWS, except it defines the algorithm to be used to encrypt and decrypt the Content Encryption Key (CEK). In other words, this algorithm is used to encrypt the actual key that is later used to encrypt the content.
- **enc**: the name of the algorithm used to encrypt the content using the CEK.
- **zip**: a compression algorithm to be applied to the encrypted data before encryption. This parameter is optional. When it is absent, no compression is performed. A usual value for this is `DEF`, the common deflate algorithm[2].
- **jku**: identical to JWS, except in this case the claim points to the public-key used to encrypt the CEK.
- **jkw**: identical to JWS, except in this case the claim points to the public-key used to encrypt the CEK.
- **kid**: identical to JWS, except in this case the claim points to the public-key used to encrypt the CEK.
- **x5u**: identical to JWS, except in this case the claim points to the public-key used to encrypt the CEK.
- **x5c**: identical to JWS, except in this case the claim points to the public-key used to encrypt the CEK.

[2] https://tools.ietf.org/html/rfc1951

- **x5t**: identical to JWS, except in this case the claim points to the public-key used to encrypt the CEK.
- **x5t#S256**: identical to JWS, except in this case the claim points to the public-key used to encrypt the CEK.
- **typ**: identical to JWS.
- **cty**: identical to JWS, except this is the type of the encrypted content.
- **crit**: identical to JWS, except it refers to the parameters of this header.

Additional parameters may be required, depending on the encryption algorithms in use. You will find these explained in the section discussing each algorithm.

5.1.4 Algorithm Overview for Compact Serialization

At the beginning of this chapter, JWE Compact Serialization was mentioned briefly. It is basically composed of five elements encoded in printable-text form and separated by dots (.). The basic algorithm to construct a compact serialization JWE JWT is:

1. If required by the chosen algorithm (`alg` claim), generate a *random* number of the required size. It is essential to comply with certain cryptographic requirements for randomness when generating this value. Refer to RFC 4086[3] or use a cryptographically validated random number generator.
2. Determine the Content Encryption Key according to the key management mode[4]:
   - For **Direct Key Agreement**: use the key agreement algorithm and the random number to compute the Content Encryption Key (CEK).
   - For **Key Agreement with Key Wrapping**: use the key agreement algorithm with the random number to compute the key that will be used to wrap the CEK.
   - For **Direct Encryption**: the CEK is the symmetric key.
3. Determine the JWE Encrypted Key according to the key management mode:
   - For **Direct Key Agreement and Direct Encryption**: the JWE Encrypted Key is empty.
   - For **Key Wrapping, Key Encryption, and Key Agreement with Key Wrapping**: encrypt the CEK to the recipient. The result is the JWE Encrypted Key.
4. Compute an Initialization Vector (IV) of the size required by the chosen algorithm. If not required, skip this step.
5. Compress the plaintext of the content, if required (`zip` header claim).
6. Encrypt the data using the CEK, the IV, and the Additional Authenticated Data (AAD). The result is the encrypted content (JWE Ciphertext) and Authentication Tag. The AAD is only used for non-compact serializations.
7. Construct the compact representation as:

```
base64(header) + '.' +
base64(encryptedKey) + '.' +          // Steps 2 and 3
```

[3] https://tools.ietf.org/html/rfc4086
[4] 5.1.1.1

```
base64(initializationVector) + '.' + // Step 4
base64(ciphertext) + '.' +           // Step 6
base64(authenticationTag)            // Step 6
```

5.1.5 JWE JSON Serialization

In addition to compact serialization, JWE also defines a non-compact JSON representation. This representation trades size for flexibility, allowing, amongst other things, encryption of the content for multiple recipients by using several public-keys at the same time. This is analogous to the multiple signatures allowed by JWS JSON Serialization.

JWE JSON Serialization is the printable text encoding of a JSON object with the following members:

- **protected**: Base64-encoded JSON object of the header claims to be protected (validated, not encrypted) by this JWE JWT. Optional. At least this element or the unprotected header must be present.
- **unprotected**: header claims that are not protected (validated) as a JSON object (not Base64-encoded). Optional. At least this element or the protected header must be present.
- **iv**: Base64 string of the initialization vector. Optional (only present when required by the algorithm).
- **aad**: Additional Authenticated Data. Base64 string of the additional data that is protected (validated) by the encryption algorithm. If no AAD is supplied in the encryption step, this member must be absent.
- **ciphertext**: Base64-encoded string of the encrypted data.
- **tag**: Base64 string of the authentication tag generated by the encryption algorithm.
- **recipients**: a JSON array of JSON objects, each containing the necessary information for decryption by each recipient.

The following are the members of the objects in the `recipients` array:

- **header**: a JSON object of unprotected header claims. Optional.
- **encrypted_key**: Base64-encoded JWE Encrypted Key. Only present when a JWE Encrypted Key is used.

The actual header used to decrypt a JWE JWT for a recipient is constructed from the union of each header present. No repeated claims are allowed.

The format of the encrypted keys is described in chapter 6 (JSON Web Keys).

The following example is taken from RFC 7516 (JWE):

```
{
    "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
    "unprotected": { "jku":"https://server.example.com/keys.jwks" },
    "recipients":[
```

```
      {
        "header": { "alg":"RSA1_5","kid":"2011-04-29" },
        "encrypted_key":
          "UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-
          kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKx
          GHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRirb6Y5Cl_p-ko3
          YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPh
          cCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-cFPg
          wCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A"
      },
      {
        "header": { "alg":"A128KW","kid":"7" },
        "encrypted_key": "6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ"
      }
    ],
    "iv": "AxY8DCtDaGlsbGljb3RoZQ",
    "ciphertext": "KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY",
    "tag": "Mz-VPPyU4RlcuYv1IwIvzw"
}
```

This JSON Serialized JWE JWT carries a single payload for two recipients. The encryption algorithm is AES-128 CBC + SHA-256, which you can get from the protected header:

```
{
  "enc": "A128CBC-HS256"
}
```

By performing the union of all claims for each recipient, the final header for each recipient is constructed:

First recipient:

```
{
  "alg":"RSA1_5",
  "kid":"2011-04-29",
  "enc":"A128CBC-HS256",
  "jku":"https://server.example.com/keys.jwks"
}
```

Second recipient:

```
{
   "alg":"A128KW",
   "kid":"7",
   "enc":"A128CBC-HS256",
   "jku":"https://server.example.com/keys.jwks"
}
```

5.1.5.1 Flattened JWE JSON Serialization

As with JWS, JWE defines a *flat* JSON serialization. This serialization form can only be used for a single recipient. In this form, the `recipients` array is replaced by a `header` and `encrypted_key` pair or elements (i.e., the keys of a single object of the recipients array take its place).

This is the flattened representation of the example from the previous section resulting from only including the first recipient:

```
{
    "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
    "unprotected": { "jku":"https://server.example.com/keys.jwks" },
    "header": { "alg":"RSA1_5","kid":"2011-04-29" },
    "encrypted_key":
      "UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-
      kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKx
      GHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRirb6Y5Cl_p-ko3
      YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPh
      cCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-cFPg
      wCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A",
    "iv": "AxY8DCtDaGlsbGljb3RoZQ",
    "ciphertext": "KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY",
    "tag": "Mz-VPPyU4RlcuYv1IwIvzw"
}
```

5.2 Encrypting and Decrypting Tokens

The following examples show how to perform encryption using the popular node-jose[5] library. This library is a bit more complex than `jsonwebtoken` (used for the JWS examples), as it covers much more ground.

5.2.1 Introduction: Managing Keys with node-jose

For the purposes of the following examples, we will need to use encryption keys in various forms. This is managed by `node-jose` through a `keystore`. A `keystore` is an object that manages keys. We will generate and add a few keys to our keystore so that we can use them later in the examples. You might recall from JWS examples that such an abstraction was not required for the `jsonwebtoken` library. The `keystore` abstraction is an implementation detail of `node-jose`. You may find other similar abstractions in other languages and libraries.

To create an empty keystore and add a few keys of different types:

```
// Create an empty keystore
const keystore = jose.JWK.createKeyStore();
```

[5] https://github.com/cisco/node-jose#basics

```
// Generate a few keys. You may also import keys generated from external
// sources.
const promises = [
    keystore.generate('oct', 128, { kid: 'example-1' }),
    keystore.generate('RSA', 2048, { kid: 'example-2' }),
    keystore.generate('EC', 'P-256', { kid: 'example-3' }),
];
```

With `node-jose`, key generation is a rather simple matter. All key types usable with JWE and JWS are supported. In this example we create three different keys: a simple AES 128-bit key, a RSA 2048-bit key, and an Elliptic Curve key using curve P-256. These keys can be used both for encryption and signatures. In the case of keys that support public/private-key pairs, the generated key is the *private* key. To obtain the public keys, simply call:

```
var publicKey = key.toJSON();
```

The public key will be stored in JWK format.

It is also possible to import preexisting keys:

```
// where input is either a:
// *  jose.JWK.Key instance
// *  JSON Object representation of a JWK
jose.JWK.asKey(input).
        then(function(result) {
          // {result} is a jose.JWK.Key
          // {result.keystore} is a unique jose.JWK.KeyStore
        });

// where input is either a:
// *  String serialization of a JSON JWK/(base64-encoded)
//    PEM/(binary-encoded) DER
// *  Buffer of a JSON JWK/(base64-encoded) PEM/(binary-encoded) DER
// form is either a:
// * "json" for a JSON stringified JWK
// * "pkcs8" for a DER encoded (unencrypted!) PKCS8 private key
// * "spki" for a DER encoded SPKI public key
// * "pkix" for a DER encoded PKIX X.509 certificate
// * "x509" for a DER encoded PKIX X.509 certificate
// * "pem" for a PEM encoded of PKCS8 / SPKI / PKIX
jose.JWK.asKey(input, form).
        then(function(result) {
          // {result} is a jose.JWK.Key
          // {result.keystore} is a unique jose.JWK.KeyStore
        });
```

5.2.2 AES-128 Key Wrap (Key) + AES-128 GCM (Content)

AES-128 Key Wrap and AES-128 GCM are symmetric key algorithms. This means that the same key is required for both encryption and decryption. The key for "example-1" that we generated before is one such key. In AES-128 Key Wrap, this key is used to wrap a randomly generated key, which is then used to encrypt the content using the AES-128 GCM algorithm. It would also be possible to use this key directly (Direct Encryption mode).

```
function encrypt(key, options, plaintext) {
    return jose.JWE.createEncrypt(options, key)
                   .update(plaintext)
                   .final();
}

function a128gcm(compact) {
    const key = keystore.get('example-1');
    const options = {
        format: compact ? 'compact' : 'general',
        contentAlg: 'A128GCM'
    };

    return encrypt(key, options, JSON.stringify(payload));
}
```

The `node-jose` library works primarily with promises[6]. The object returned by `a128gcm` is a promise. The `createEncrypt` function can encrypt whatever content is passed to it. In other words, it is not necessary for the content to be a JWT (though most of the time it will be). It is for this reason that `JSON.stringify` must be called before passing the data to that function.

5.2.3 RSAES-OAEP (Key) + AES-128 CBC + SHA-256 (Content)

The only thing that changes between invocations of the `createEncrypt` function are the options passed to it. Therefore, it is just as easy to use a public/private-key pair. Rather than passing the symmetric key to `createEncrypt`, one simply passes either the public or the private-key (for encryption only the public key is required, though this one can be derived from the private key). For readability purposes, we simply use the private key, but in practice the public key will most likely be used in this step.

```
function encrypt(key, options, plaintext) {
    return jose.JWE.createEncrypt(options, key)
                   .update(plaintext)
                   .final();
}

function rsa(compact) {
```

[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

```
    const key = keystore.get('example-2');
    const options = {
        format: compact ? 'compact' : 'general',
        contentAlg: 'A128CBC-HS256'
    };

    return encrypt(key, options, JSON.stringify(payload));
}
```

`contentAlg` selects the actual encryption algorithm. Remember there are only two variants (with different key sizes): AES CBC + HMAC SHA and AES GCM.

5.2.4 ECDH-ES P-256 (Key) + AES-128 GCM (Content)

The API for elliptic curves is identical to that of RSA:

```
function encrypt(key, options, plaintext) {
    return jose.JWE.createEncrypt(options, key)
                   .update(plaintext)
                   .final();
}

function ecdhes(compact) {
    const key = keystore.get('example-3');
    const options = {
        format: compact ? 'compact' : 'general',
        contentAlg: 'A128GCM'
    };

    return encrypt(key, options, JSON.stringify(payload));
}
```

5.2.5 Nested JWT: ECDSA using P-256 and SHA-256 (Signature) + RSAES-OAEP (Encrypted Key) + AES-128 CBC + SHA-256 (Encrypted Content)

Nested JWTs require a bit of juggling to pass the signed JWT to the encryption function. Specifically, the signature + encryption steps must be performed manually. Recall that these steps are performed in that order: first signing, then encrypting. Although technically nothing prevents the order from being reversed, signing the JWT first prevents the resulting token from being vulnerable to signature removal attacks.

```
function nested(compact) {
    const signingKey = keystore.get('example-3');
    const encryptionKey = keystore.get('example-2');
```

```
    const signingPromise = jose.JWS.createSign(signingKey)
                                    .update(JSON.stringify(payload))
                                    .final();

    const promise = new Promise((resolve, reject) => {

        signingPromise.then(result => {
            const options = {
                format: compact ? 'compact' : 'general',
                contentAlg: 'A128CBC-HS256'
            };
            resolve(encrypt(encryptionKey, options, JSON.stringify(result)));
        }, error => {
            reject(error);
        });

    });

    return promise;
}
```

As can be seen in the example above, `node-jose` can also be used for signing. There is nothing precluding the use of other libraries (such as `jsonwebtoken`) for that purpose. However, given the necessity of `node-jose`, there is no point in adding dependencies and using inconsistent APIs.

> Performing the signing step first is only possible because JWE mandates authenticated encryption. In other words, the encryption algorithm must also perform the signing step. The reasons JWS and JWE can be combined in a useful way, in spite of JWE's authentication, were described at the beginning of chapter 5. For other schemes (i.e., for general encryption + signature), the norm is to first encrypt, then sign. This is to prevent manipulation of the ciphertext that can result in encryption attacks. It is also the reason that JWE mandates the presence of an authentication tag.

5.2.6 Decryption

Decryption is as simple as encryption. As with encryption, the payload must be converted between different data formats explicitly.

```
// Decryption test
a128gcm(true).then(result => {
    jose.JWE.createDecrypt(keystore.get('example-1'))
            .decrypt(result)
            .then(decrypted => {
                decrypted.payload = JSON.parse(decrypted.payload);
                console.log(`Decrypted result: ${JSON.stringify(decrypted)}`);
            }, error => {
                console.log(error);
```

```
            });
}, error => {
    console.log(error);
});
```

Decryption of RSA and Elliptic Curve algorithms is analogous, using the private-key rather than the symmetric key. If you have a keystore with the right `kid` claims, it is possible to simply pass the keystore to the `createDecrypt` function and have it search for the right key. So, any of the examples above can be decrypted using the exact same code:

```
jose.JWE.createDecrypt(keystore) //just pass the keystore here
        .decrypt(result)
        .then(decrypted => {
            decrypted.payload = JSON.parse(decrypted.payload);
            console.log(`Decrypted result: ${JSON.stringify(decrypted)}`);
        }, error => {
            console.log(error);
        });
```

Chapter 6

JSON Web Keys (JWK)

To complete the picture of JWT, JWS, and JWE we now come to the JSON Web Key (JWK) specification. This specification deals with the different representations for the keys used for signatures and encryption. Although there are established representations for all keys, the JWK specification aims at providing a unified representation for all keys supported in the JSON Web Algorithms (JWA) specification. A unified representation format for keys allows easy sharing and keeps keys independent from the intricacies of other key exchange formats.

JWS and JWE do support a different type of key format: X.509 certificates. These are quite common and can carry more information than a JWK. X.509 certificates can be embedded in JWKs, and JWKs can be constructed from them.

Keys are specified in different header claims. Literal JWKs are put under the `jwk` claim. The `jku` claim, on the other hand, can point to a *set* of keys stored under a URL. Both of these claims are in JWK format.

A sample JWK:

```
{
    "kty":"EC",
    "crv":"P-256",
    "x":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
    "y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
    "d":"870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",
    "use":"enc",
    "kid":"1"
}
```

6.1 Structure of a JSON Web Key

JSON Web Keys are simply JSON objects with a series of values that describe the parameters required by the key. These parameters vary according to the type of key. Common parameters are:

- **kty**: "key type". This claim differentiates types of keys. Supported types are `EC`, for elliptic curve keys; `RSA` for RSA keys; and `oct` for symmetric keys. This claim is required.
- **use**: this claim specifies the intended use of the key. There are two possible uses: `sig` (for signature) and `enc` (for encryption). This claim is optional. The same key can be used for encryption and signatures, in which case this member should not be present.
- **key_ops**: an array of string values that specifies detailed uses for the key. Possible values are: `sign`, `verify`, `encrypt`, `decrypt`, `wrapKey`, `unwrapKey`, `deriveKey`, `deriveBits`. Certain operations should not be used together. For instance, `sign` and `verify` are appropriate for the same key, while `sign` and `encrypt` are not. This claim is optional and should not be used at the same time as the `use` claim. In cases where both are present, their content should be consistent.
- **alg**: "algorithm". The algorithm intended to be used with this key. It can be any of the algorithms admitted for JWE or JWS operations. This claim is optional.
- **kid**: "key id". A unique identifier for this key. It can be used to match a key against a `kid` claim in the JWE or JWS header, or to pick a key from a set of keys according to application logic. This claim is optional. Two keys in the same key set can carry the same `kid` only if they have different `kty` claims and are intended for the same use.
- **x5u**: a URL that points to a X.509 public key certificate or certificate chain in PEM encoded form. If other optional claims are present they must be consistent with the contents of the certificate. This claim is optional.
- **x5c**: a Base64-URL encoded X.509 DER certificate or certificate chain. A certificate chain is represented as an array of such certificates. The first certificate must be the certificate referred by this JWK. All other claims present in this JWK must be consistent with the values of the first certificate. This claim is optional.
- **x5t**: a Base64-URL encoded SHA-1 thumbprint/fingerprint of the DER encoding of a X.509 certificate. The certificate this thumbprint points to must be consistent with the claims in this JWK. This claim is optional.
- **x5t#S256**: identical to the `x5t` claim, but with the SHA-256 thumbprint of the certificate.

Other parameters, such as `x`, `y`, or `d` (from the example at the opening of this chapter) are specific to the key algorithm. RSA keys, on the other hand, carry parameters such as `n`, `e`, `dp`, etc. The meaning of these parameters will become clear in chapter 7, where each key algorithm is explained in detail.

6.1.1 JSON Web Key Set

The JWK spec admits groups of keys. These are known as "JWK Sets". These sets carry more than one key. The meaning of the keys as a group and the meaning of the order of these keys is user defined.

A JSON Web Key Set is simply a JSON object with a `keys` member. This member is a JSON array of JWKs.

Sample JWK Set:

```
{
    "keys": [
        {
            "kty":"EC",
            "crv":"P-256",
            "x":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
            "y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
            "use":"enc",
            "kid":"1"
        },

        {
            "kty":"RSA",
            "n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
                4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMs
                tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
                QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbI
                SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
                w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",
            "e":"AQAB",
            "alg":"RS256",
            "kid":"2011-04-29"
        }
    ]
}
```

In this example, two public-keys are available. The first one is of elliptic curve type and is limited to *encryption* operations by the `use` claim. The second one is of RSA type and is associated with a specific algorithm (`RS256`) by the `alg` claim. This means this second key is meant to be used for *signatures*.

Chapter 7

JSON Web Algorithms

You have probably noted that there are many references to this chapter throughout this handbook. The reason is that a big part of the magic behind JWTs lies in the algorithms employed with it. Structure is important, but the many interesting uses described so far are only possible due to the algorithms in play. This chapter will cover the most important algorithms in use with JWTs today. Understanding them in depth is not necessary in order to use JWTs effectively, and so this chapter is aimed at curious minds wanting to understand the last piece of the puzzle.

7.1 General Algorithms

The following algorithms have many different applications inside the JWT, JWS, and JWE specs. Some algorithms, like Base64-URL, are used for compact and non-compact serialization forms. Others, such as SHA-256, are used for signatures, encryption, and key fingerprints.

7.1.1 Base64

Base64 is a binary-to-text encoding algorithm. Its main purpose is to turn a sequence of octets into a sequence of printable characters, at the cost of added size. In mathematical terms, Base64 turns a sequence of radix-256 numbers into a sequence of radix-64 numbers. The word *base* can be used in place of *radix*, hence the name of the algorithm.

> Note: Base64 is not actually used by the JWT spec. It is the *Base64-URL* variant described later in this chapter, that is used by JWT.

To understand how Base64 can turn a series of arbitrary numbers into text, it is first necessary to be familiar with text-encoding systems. Text-encoding systems map numbers to characters. Although this mapping is arbitrary and in the case of Base64 can be implementation defined, the de facto standard for Base64 encoding is RFC 4648[1].

[1] https://tools.ietf.org/rfc/rfc4648.txt

0 A	17 R	34 i	51 z
1 B	18 S	35 j	52 0
2 C	19 T	36 k	53 1
3 D	20 U	37 l	54 2
4 E	21 V	38 m	55 3
5 F	22 W	39 n	56 4
6 G	23 X	40 o	57 5
7 H	24 Y	41 p	58 6
8 I	25 Z	42 q	59 7
9 J	26 a	43 r	60 8
10 K	27 b	44 s	61 9
11 L	28 c	45 t	62 +
12 M	29 d	46 u	63 /
13 N	30 e	47 v	
14 O	31 f	48 w	(pad) =
15 P	32 g	49 x	
16 Q	33 h	50 y	

In Base64 encoding, each character represents 6 bits of the original data. Encoding is performed in groups of four encoded characters. So, 24 bits of original data are taken together and encoded as four Base64 characters. Since the original data is expected to be a sequence of 8-bit values, the 24 bits are formed by concatenating three 8-bit values from left to right.

Base64 encoding:

```
3 x 8-bit values -> 24-bit concatenated data -> 4 x 6-bit characters
```

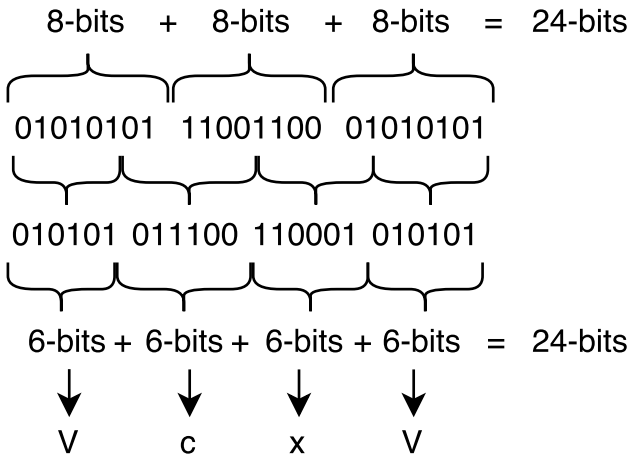

Figure 7.1: Base64 encoding

If the number of octets in the input data is not divisible by three, then the last portion of data to encode will have less than 24 bits of data. When this is the case, zeros are added to the concatenated input data to form an integral number of 6-bit groups. There are three possiblities:

1. The full 24 bits are available as input; no special processing is performed.
2. 16 bits of input are available, three 6-bit values are formed, and the last 6-bit value gets extra zeros added to the right. The resulting encoded string is padded with an extra `=` character to make it explicit that 8 bits of input were missing.
3. 8 bits of input are available, two 6-bit values are formed, and the last 6-bit value gets extra zeros added to the right. The resulting encoded string is padded with two extra `=` characters to make it explicit that 16 bits of input were missing.

The padding character (`=`) is considered optional by some implementations. Performing the steps in the opposite order will yield the original data, regardless of the presence of the padding characters.

7.1.1.1 Base64-URL

Certain characters from the standard Base64 conversion table are not URL-safe. Base64 is a convenient encoding for passing arbitrary data in text fields. Since only two characters from Base64 are problematic as part of the URL, a URL-safe variant is easy to implement. The `+` character and the `/` character are replaced by the `-` character and the `_` character.

7.1.1.2 Sample Code

The following sample implements a dumb Base64-URL encoder. The example is written with simplicity in mind, rather than speed.

```
const table = [
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
    'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',
    'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd',
    'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
    'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
    'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', '-', '_'
];

/**
 * @param input a Buffer, Uint8Array or Int8Array, Array
 * @returns a String with the encoded values
 */
export function encode(input) {
    let result = "";

    for(let i = 0; i < input.length; i += 3) {
        const remaining = input.length - i;
```

```
        let concat = input[i] << 16;
        result += (table[concat >>> (24 - 6)]);

        if(remaining > 1) {
            concat |= input[i + 1] << 8;
            result += table[(concat >>> (24 - 12)) & 0x3F];

            if(remaining > 2) {
                concat |= input[i + 2];
                result += table[(concat >>> (24 - 18)) & 0x3F] +
                          table[concat & 0x3F];
            } else {
                result += table[(concat >>> (24 - 18)) & 0x3F] + "=";
            }
        } else {
            result += table[(concat >>> (24 - 12)) & 0x3F] + "==";
        }
    }

    return result;
}
```

7.1.2 SHA

The Secure Hash Algorithm (SHA) used in the JWT specs is defined in FIPS-180[2]. It is not to be confused with the SHA-1[3] family of algorithms, which have been deprecated since 2010. To differentiate this family from the previous one, this family is sometimes called *SHA-2*.

The algorithms in RFC 4634 are SHA-224, SHA-256, SHA-384, and SHA-512. Of importance for JWT are SHA-256 and SHA-512. We will focus on the SHA-256 variant and explain its differences with regard to the other variants.

As do many hashing algorithms, SHA works by processing the input in fixed-size chunks, applying a series of mathematical operations and then accummulating the result by performing an operation with the previous iteration results. Once all fixed-size input chunks are processed, the digest is said to be computed.

The SHA family of algorithms were designed to avoid collisions and produce radically different output even when the input is only slightly changed. It is for this reason they are considered *secure*: it is computationally infeasible to find collisions for different inputs, or to compute the original input from the produced digest.

The algorithm requires a series of predefined functions:

[2] http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
[3] https://en.wikipedia.org/wiki/SHA-1

```
function rotr(x, n) {
    return (x >>> n) | (x << (32 - n));
}

function ch(x, y, z) {
    return (x & y) ^ ((~x) & z);
}

function maj(x, y, z) {
    return (x & y) ^ (x & z) ^ (y & z);
}

function bsig0(x) {
    return rotr(x, 2) ^ rotr(x, 13) ^ rotr(x, 22);
}

function bsig1(x) {
    return rotr(x, 6) ^ rotr(x, 11) ^ rotr(x, 25);
}

function ssig0(x) {
    return rotr(x, 7) ^ rotr(x, 18) ^ (x >>> 3);
}

function ssig1(x) {
    return rotr(x, 17) ^ rotr(x, 19) ^ (x >>> 10);
}
```

These functions are defined in the specification. The `rotr` function performs bitwise rotation (to the right).

Additionally, the algorithm requires the message to be of a predefined length (a multiple of 64); therefore padding is required. The padding algorithm works as follows:

1. A single binary `1` is appended to the end of the original message. For example:

```
Original message:
    01011111 01010101 10101010 00111100
Extra 1 at the end:
    01011111 01010101 10101010 00111100 1
```

2. An N number of zeroes is appended so that the resulting length of the message is the solution to this equation:

```
L = Message length in bits
0 = (65 + N + L) mod 512
```

3. Then the number of bits in the original message is appended as a 64-bit integer:

```
Original message:
```

```
    01011111 01010101 10101010 00111100
Extra 1 at the end:
    01011111 01010101 10101010 00111100 1
N zeroes:
    01011111 01010101 10101010 00111100 10000000 ...0...
Padded message:
    01011111 01010101 10101010 00111100 10000000 ...0... 00000000 00100000
```

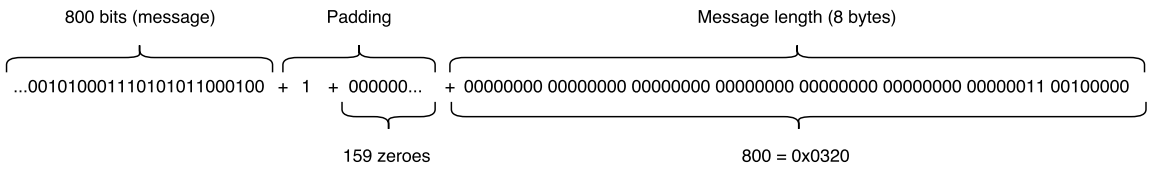

Figure 7.2: SHA padding

A simple implementation in JavaScript could be:

```
function padMessage(message) {
    if(!(message instanceof Uint8Array) && !(message instanceof Int8Array)) {
        throw new Error("unsupported message container");
    }

    const bitLength = message.length * 8;
    const fullLength = bitLength + 65; //Extra 1 + message size.
    let paddedLength = (fullLength + (512 - fullLength % 512)) / 32;
    let padded = new Uint32Array(paddedLength);

    for(let i = 0; i < message.length; ++i) {
        padded[Math.floor(i / 4)] |= (message[i] << (24 - (i % 4) * 8));
    }

    padded[Math.floor(message.length / 4)] |= (0x80 << (24 - (message.length % 4) * 8));
    // TODO: support messages with bitLength longer than 2^32
    padded[padded.length - 1] = bitLength;

    return padded;
}
```

The resulting padded message is then processed in 512-bit blocks. The implementation below follows the algorithm described in the specification step by step. All operations are performed on 32-bit integers.

```
export default function sha256(message, returnBytes) {
    // Initial hash values
    const h_ = Uint32Array.of(
```

```
    0x6a09e667,
    0xbb67ae85,
    0x3c6ef372,
    0xa54ff53a,
    0x510e527f,
    0x9b05688c,
    0x1f83d9ab,
    0x5be0cd19
);

const padded = padMessage(message);
const w = new Uint32Array(64);
for(let i = 0; i < padded.length; i += 16) {
    for(let t = 0; t < 16; ++t) {
        w[t] = padded[i + t];
    }
    for(let t = 16; t < 64; ++t) {
        w[t] = ssig1(w[t - 2]) + w[t - 7] + ssig0(w[t - 15]) + w[t - 16];
    }

    let a = h_[0] >>> 0;
    let b = h_[1] >>> 0;
    let c = h_[2] >>> 0;
    let d = h_[3] >>> 0;
    let e = h_[4] >>> 0;
    let f = h_[5] >>> 0;
    let g = h_[6] >>> 0;
    let h = h_[7] >>> 0;

    for(let t = 0; t < 64; ++t) {
        let t1 = h + bsig1(e) + ch(e, f, g) + k[t] + w[t];
        let t2 = bsig0(a) + maj(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + t1;
        d = c;
        c = b;
        b = a;
        a = t1 + t2;
    }

    h_[0] = (a + h_[0]) >>> 0;
    h_[1] = (b + h_[1]) >>> 0;
    h_[2] = (c + h_[2]) >>> 0;
    h_[3] = (d + h_[3]) >>> 0;
```

```
        h_[4] = (e + h_[4]) >>> 0;
        h_[5] = (f + h_[5]) >>> 0;
        h_[6] = (g + h_[6]) >>> 0;
        h_[7] = (h + h_[7]) >>> 0;
    }

    //(...)
}
```

The variable `k` holds a series of constants, which are defined in the specification.

The final result is in the variable `h_[0..7]`. The only missing step is to present it in readable form:

```
if(returnBytes) {
    const result = new Uint8Array(h_.length * 4);
    h_.forEach((value, index) => {
        const i = index * 4;
        result[i    ] = (value >>> 24) & 0xFF;
        result[i + 1] = (value >>> 16) & 0xFF;
        result[i + 2] = (value >>> 8)  & 0xFF;
        result[i + 3] = (value >>> 0)  & 0xFF;
    });

    return result;
} else {
    function toHex(n) {
        let str = (n >>> 0).toString(16);
        let result = "";
        for(let i = str.length; i < 8; ++i) {
            result += "0";
        }
        return result + str;
    }
    let result = "";
    h_.forEach(n => {
        result += toHex(n);
    });
    return result;
}
```

Although it works, note that the implementation above is not optimal (and does not support messages longer than 2^{32}).

Other variants of the SHA-2 family (such as SHA-512) simply change the size of the block processed in each iteration and alter the constants and their size. In particular, SHA-512 requires 64-bit math to be available. In other words, to turn the sample implementation above into SHA-512, a separate library for 64-bit math is required (as JavaScript only supports 32-bit bitwise operations and 64-bit floating-point math).

7.2 Signing Algorithms

7.2.1 HMAC

Hash-based Message Authentication Codes (HMAC)[4] make use of a cryptographic hash function (such as the SHA family discussed above) and a key to create an *authentication code* for a specific message. In other words, a HMAC-based authentication scheme takes a hash function, a message, and a secret-key as inputs and produces an authentication code as output. The strength of the cryptographic hash function ensures that the message cannot be modified without the secret key. Thus, HMACs serve both purposes of *authentication* and *data integrity*.

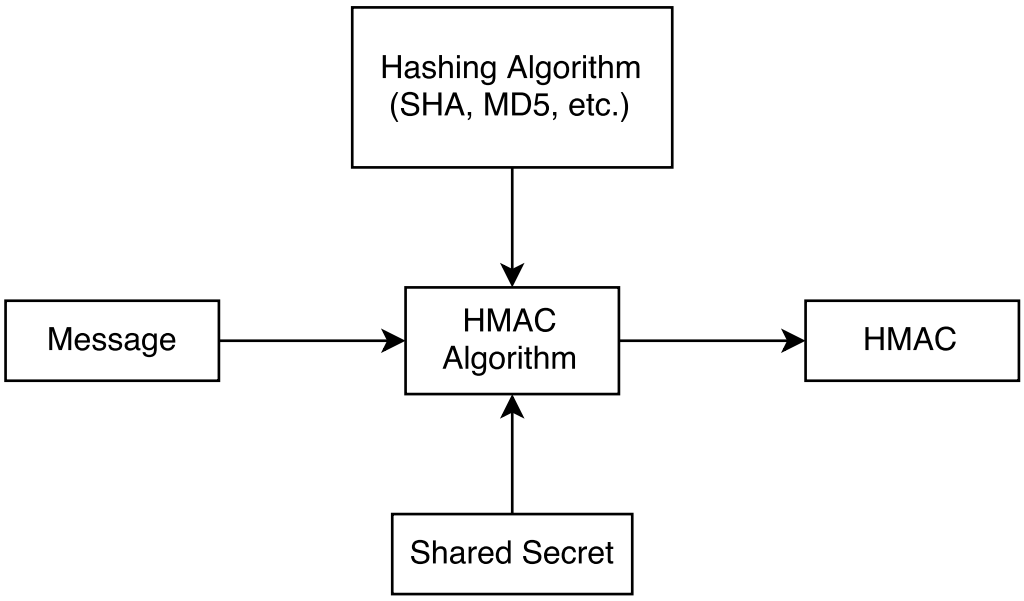

Figure 7.3: HMAC

Weak hash functions may allow malicious users to compromise the validity of the authentication code. Therefore, for HMACs to be of use, a strong hash function must be chosen. The SHA-2 family of functions is still strong enough for today's standards, but this may change in the future. MD5, a different cryptographic hash function used extensively in the past, can be used for HMACs. However, it can be vulnerable to collision and prefix attacks. Although these attacks do not necessarily make MD5 unsuitable for use with HMACs, stronger algorithms are readily available and should be considered.

The algorithm is simple enough to fit in a single line:

```
Let H be the cryptographic hash function
    B be the block length of the hash function
```

[4] https://tools.ietf.org/html/rfc2104

```
      (how many bits are processed per iteration)
    K be the secret key
    K' be the actual key used by the HMAC algorithm
    L be the length of the output of the hash function
    ipad be the byte 0x36 repeated B times
    opad be the byte 0x5C repeated B times
    message be the input message
    || be the concatenation function

HMAC(message) = H(K' XOR opad || H(K' XOR ipad || message))
```

`K'` is computed from the secret key `K` as follows:

If `K` is shorter than `B`, zeroes are appended until `K` is of `B` length. The result is `K'`. If `K` is longer than `B`, `H` is applied to `K`. The result is `K'`. If `K` is exactly `B` bytes, it is used as is (`K` is `K'`).

Here is a sample implementation in JavaScript:

```
export default function hmac(hashFn, blockSizeBits, secret, message, returnBytes) {
    if(!(message instanceof Uint8Array)) {
        throw new Error('message must be of Uint8Array');
    }

    const blockSizeBytes = blockSizeBits / 8;

    const ipad = new Uint8Array(blockSizeBytes);
    const opad = new Uint8Array(blockSizeBytes);
    ipad.fill(0x36);
    opad.fill(0x5c);

    const secretBytes = stringToUtf8(secret);
    let paddedSecret;
    if(secretBytes.length <= blockSizeBytes) {
        const diff = blockSizeBytes - secretBytes.length;
        paddedSecret = new Uint8Array(blockSizeBytes);
        paddedSecret.set(secretBytes);
    } else {
        paddedSecret = hashFn(secretBytes);
    }

    const ipadSecret = ipad.map((value, index) => {
        return value ^ paddedSecret[index];
    });
    const opadSecret = opad.map((value, index) => {
        return value ^ paddedSecret[index];
    });

    // HMAC(message) = H(K' XOR opad || H(K' XOR ipad || message))
```

```
    const result = hashFn(
        append(opadSecret,
               uint32ArrayToUint8Array(hashFn(append(ipadSecret,
                                                     message), true))),
        returnBytes);

    return result;
}
```

To verify a message against an HMAC, one simply computes the HMAC and compares the result with the HMAC that came with the message. This requires knowledge of the secret-key by all parties: those who produce the message, and those who only want to verify it.

7.2.1.1 HMAC + SHA256 (HS256)

Understanding Base64-URL, SHA-256, and HMAC are all that is needed to implement the HS256 signing algorithm from the JWS specification. With this is mind, we can now combine all the sample code developed so far and construct a fully signed JWT.

```
export default function jwtEncode(header, payload, secret) {
    if(typeof header !== 'object' || typeof payload !== 'object') {
        throw new Error('header and payload must be objects');
    }
    if(typeof secret !== 'string') {
        throw new Error("secret must be a string");
    }

    header.alg = 'HS256';

    const encHeader = b64(JSON.stringify(header));
    const encPayload = b64(JSON.stringify(payload));
    const jwtUnprotected = `${encHeader}.${encPayload}`;
    const signature = b64(uint32ArrayToUint8Array(
        hmac(sha256, 512, secret, stringToUtf8(jwtUnprotected), true)));

    return `${jwtUnprotected}.${signature}`;
}
```

Note that this function performs no validation of the header or payload (other than checking to see if they are objects). You can call this function like this:

```
console.log(jwtEncode({}, {sub: "test@test.com"}, 'secret'));
```

Paste the generated JWT in JWT.io's debugger[5] and see how it gets decoded and validated.

This function is very similar to the one used in chapter 4 as a demonstration for the signing algorithm. From chapter 4:

[5] https://jwt.io

```
const encodedHeader = base64(utf8(JSON.stringify(header)));
const encodedPayload = base64(utf8(JSON.stringify(payload)));
const signature = base64(hmac(`${encodedHeader}.${encodedPayload}`, secret, sha256));
const jwt = `${encodedHeader}.${encodedPayload}.${signature}`;
```

Verification is just as easy:

```
export function jwtVerifyAndDecode(jwt, secret) {
    if(!isString(jwt) || !isString(secret)) {
        throw new TypeError('jwt and secret must be strings');
    }

    const split = jwt.split('.');
    if(split.length !== 3) {
        throw new Error('Invalid JWT format');
    }

    const header = JSON.parse(unb64(split[0]));
    if(header.alg !== 'HS256') {
        throw new Error(`Wrong algorithm: ${header.alg}`);
    }

    const jwtUnprotected = `${split[0]}.${split[1]}`;
    const signature =
        b64(hmac(sha256, 512, secret, stringToUtf8(jwtUnprotected), true));

    return {
        header: header,
        payload: JSON.parse(unb64(split[1])),
        valid: signature == split[2]
    };
}
```

The signature is split from the JWT and a new signature is computed. If the new signature matches the one included in the JWT, then the signature is valid.

You can use the function above as follows:

```
const secret = 'secret';
const encoded = jwtEncode({}, {sub: "test@test.com"}, secret);
const decoded = jwtVerifyAndDecode(encoded, secret);
```

This code is available in the `hs256.js` file of the samples included[6] with this handbook.

[6] https://github.com/auth0/jwt-handbook-samples

7.2.2 RSA

RSA is one of the most widely used cryptosystems today. It was developed in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman, whose initials were used to name the algorithm. The key aspect of RSA lies in its asymmetry: the key used to encrypt something is *not* the key used to decrypt it. This scheme is known as public-key encryption (PKI), were the public key is the encryption key and the private key is the decryption key.

When it comes to signatures, the private key is used to *sign* a piece of information and the public key is used to *verify* that it was signed by a specific private key (without actually knowing it).

There are variations of the RSA algorithm for both signing and encryption. We will focus on the general algorithm first, and then we will take a look at the different variations used with JWTs.

Many cryptographic algorithms, and in particular RSA, are based on the relative difficulty of performing certain mathematical operations. RSA picks the integer factorization[7] as its main mathematical tool. Integer factorization is the mathematical problem that attempts to find numbers that multiplied among themselves yield the original number as result. In other words, an integer's factors are a set of pairs of integers that when multiplied yield the original integer.

```
integer = factor_1 x factor_2
```

This problem might seem easy at first. And for small numbers, it is. Take for instance the number `35`:

```
35 = 7 x 5
```

By knowing the multiplication tables of either 7 or 5 it is easy to find two numbers that yield `35` when multiplied. A naive algorithm to find factors for an integer could be:

1. Let `n` be the number we want to factor.
2. Let `x` be a number between `2` (inclusive) and `n / 2` (inclusive).
3. Divide `n` by `x` and check whether the remainder is `0`. If it is, you have found one pair of factors (`x` and the quotient).
4. Continue performing step 3 increasing `x` by `1` in each iteratior until `x` reaches its upper bound `n / 2`. When it does, you have found all possible factors of `n`.

This is essentially the brute force approach to finding factors. As you can imagine, this algorithm is terribly inefficient.

> A better version of this algorithm is called `trial division` and sets stricter conditions for `x`. In particular, it defines `x`'s upper bound as `sqrt(n)`, and, rather than increase `x` by `1` in each iteration, it makes `x` take the value of ever bigger prime numbers. It is trivial to prove why these conditions make the algorithm more efficient while keeping it correct (though out of scope for this text).

More efficient algorithms do exist, but, as efficient as they are, even with today's computers, certain numbers are computationally infeasible to factor. The problem is compounded when certain

[7] https://en.wikipedia.org/wiki/Integer_factorization

numbers are chosen as `n`. For instance, if `n` is the result of multiplying two prime numbers[8], it is much harder to find its factors (of which those prime numbers are the only possible factors).

> If `n` were the result of multiplying two non-prime numbers, it would be much easier to find its factors. Why? Because non-prime numbers have divisors other than themselves (by defintion), and these divisors are in turn divisors of any number multiplied with them. In other words, any divisor to a factor of a number is also a factor of the number. Or, in other terms, if `n` has non-prime factors, it has more than two factors. So, if `n` is the result of multiplying two prime-numbers, it has exactly two factors (the least possible number of factors without being a prime number itself). The lesser the number of factors, the harder it is to find them.

When two different and big prime numbers are picked and then multiplied, the result is another big number (called a semiprime). But this big number has an added special property: it is really hard to factor. Even the most efficient factoring algorithms, such as the general number field sieve[9], cannot factor big numbers that are the result of multiplying big primes in reasonable time frames. To give a sense of scale, in 2009 a 768-bit number (232 decimal digits) was factored[10] after 2 years of work by a cluster of computers. Typical applications of RSA make use of 2048-bit or bigger numbers.

> Shor's algorithm[11] is a special kind of factoring algorithm that could change things drastically in the future. While most factoring algorithms are classical in nature (that is, they operate on classical computers), Shor's algorithm relies on quantum computers[12]. Quantum computers take advantage of the nature of certain quantum phenomena to speed up several classical operations. In particular, Shor's algorithm could speed up factorization, bringing its complexity into the realm of polynomial time complexity (rather than exponential). This is much more efficient than any of the current classical algorithms. It is speculated that if such algorithm were to be runnable on a quantum computer, current RSA keys would become useless. A practical quantum computer as required by Shor's algorithm has not been developed yet, but this is an acive area of research at the moment.

Although currently integer factorization is computationally infeasible for large semiprimes, there is no mathematical proof that this should be the case. In other words, in the future there might appear algorithms that solve integer factorization in reasonable timeframes. The same can be said of RSA.

With this said, we can now focus on the actual algorithm. The basic principle is captured in this expression:

$$(m^e)^d \equiv m \pmod{n}$$

Figure 7.4: RSA basic expression

[8] https://en.wikipedia.org/wiki/Prime_number
[9] https://en.wikipedia.org/wiki/General_number_field_sieve
[10] http://eprint.iacr.org/2010/006
[11] https://en.wikipedia.org/wiki/Shor%27s_algorithm
[12] https://en.wikipedia.org/wiki/Quantum_computing

It is computationally feasible to find three very large integers `e`, `d` and `n` that satisfy the equation above. The algorithm relies on the difficulty of finding `d` when all other numbers are known. In other words, this expression can be turned into a one-way function. `d` can then be considered the private-key, while `n` and `e` are the public key.

7.2.2.1 Choosing e, d and n

1. Choose two distinct prime numbers `p` and `q`.
   - A cryptographically secure random number generator should be used to pick candidates for `p` and `q`. An insecure RNG can result in an attacker finding one of these numbers.
   - As there is no way to randomly generate prime numbers, after two random numbers are picked, they should pass a primality test. Deterministic primality checks can be expensive, so some implementations rely on probabilistic methods. How probable it is to find a false prime needs to be considered.
   - `p` and `q` should be similar in magnitude but not identical, and should differ in length by a few digits.
2. `n` is the result of `p` times `q`. This is the modulus from the equation above. Its number of bits is the key length of the algorithm.
3. Compute Euler's totient function[13] for `n`. Since `n` is a semiprime number, this is as simple as: `n - (p + q - 1)`. We will call this value `phi(n)`.
4. Choose an `e` that meets the following criteria:
   - `1 < e < phi(n)`
   - `e` and `phi(n)` should be coprime
5. Pick a `d` that satisfies the following expression:

$$d \equiv e^{-1} (mod\, \varphi(n))$$

Figure 7.5: RSA basic expression

The public key is composed of values `n` and `e`. The private key is composed of values `n` and `d`. Values `p`, `q` and `phi(n)` should be discarded or kept secret, as they can be used to help in finding `d`.

From the equations above, it is evident that `e` and `d` are mathematically symmetric. We can rewrite the equation from step 5 as:

$$d \,.\, e \equiv 1 (mod\, \varphi(n))$$

Figure 7.6: Symmetry between e and d

So now you are probably wondering how RSA is safe if we publish the values `e` and `n`; could't we use those values to find `d`? The thing about modular arithmetic is that there are multiple possible solutions. As long as the `d` we pick satisfies the equation above, any value is valid. The bigger the value, the harder it is to find it. So, RSA works as long as only one of the values `e` or `d` is known to

[13] https://en.wikipedia.org/wiki/Euler%27s_totient_function#Computing_Euler.27s_totient_function

public parties. This is also the reason one of those values is picked: the public value can be chosen to be as small as possible. This speeds up computation without compromising the safety of the algorithm.

7.2.2.2 Basic Signing

Signing in RSA is performed as follows:

1. A message digest is produced from the message to be signed by a hash function.
2. This digest is then raised to the power of `d` modulo `n` (which is part of the private key).
3. The result is attached to the message as the signature.

When a recipient holding the public key wants to verify the authenticity of the message, he or she can reverse the operation as follows:

1. The signature is raised to the power of `e` modulo `n`. The resulting value is the reference digest value.
2. A message digest is produced from the message using the same hash function as in the signing step.
3. The results from step 1 and 2 are compared. If they match, the signing party must be in possession of the private key.

This signature/verification scheme is known as "signature scheme with appendix" (SSA). This scheme requires the original message to be available to verify the message. In other words, they do not allow message recovery from the signature (message and signature remain separate).

7.2.2.3 RS256: RSASSA PKCS1 v1.5 using SHA-256

Now that we have a basic notion of how RSA works, we can focus on a specific variant: PKCS#1 RSASSA v1.5 using SHA-256, also known as RS256 in the JWA specification.

The Public Key Cryptography Standard #1 (PKCS #1)[14] specification defines a series of primitives, formats and encryption schemes based on the RSA algorithm. These elements work together to provide a detailed implementation of RSA usable in modern computing platforms. RSASSA is one of the schemes defined in it, and it allows the use of RSA for signatures.

7.2.2.3.1 Algorithm

To produce a signature:

1. Apply the **EMSA-PKCS1-V1_5-ENCODE** primitive to the message (an array of octets). The result is the **encoded message**. This primitive makes use of a hash function (usually a SHA family hash function such as SHA-256). This primitive accepts an expected encoded message length. In this case, it will be the length in octets of the RSA number `n` (the key length).

[14] https://www.ietf.org/rfc/rfc3447.txt

2. Apply the **OS2IP** primitive to the encoded message. The result is the **integer message representative**. OS2IP is the acronym for "Octet-String to Integer Primitive".
3. Apply the **RSASP1** primitive to the integer message representative using the private key. The result is the **integer signature representative**.
4. Apply the **I2OSP** primitive to convert the integer signature representative to an array of octets (the **signature**). I2OSP is the acronym for "Integer to Octet-String Primitive".

A possible implementation in JavaScript, given the primitives mentioned above, could look like:

```
/**
 * Produces a signature for a message using the RSA algorithm as defined
 * in PKCS#1.
 * @param {privateKey} RSA private key, an object with
 *                     three members: size (size in bits), n (the modulus) and
 *                     d (the private exponent), both bigInts
 *                     (big-integer library).
 * @param {hashFn} the hash function as required by PKCS#1,
 *                 it should take a Uint8Array and return a Uint8Array
 * @param {hashType} A symbol identifying the type of hash function passed.
 *                   For now, only "SHA-256" is supported. See the "hashTypes"
 *                   object for possible values.
 * @param {message} A String or Uint8Array with arbitrary data to sign
 * @return {Uint8Array} The signature as a Uint8Array
 */
export function sign(privateKey, hashFn, hashType, message) {
    const encodedMessage =
        emsaPkcs1v1_5(hashFn, hashType, privateKey.size / 8, message);
    const intMessage = os2ip(encodedMessage);
    const intSignature = rsasp1(privateKey, intMessage);
    const signature = i2osp(intSignature, privateKey.size / 8);
    return signature;
}
```

To verify a signature:

1. Apply the **OS2IP** primitive to the signature (an array of octets). This is the **integer signature representative**.
2. Apply the **RSAVP1** primitive to the previous result. This primitive also takes the public key as input. This is the **integer message representative**.
3. Apply the **I2OSP** primitive to the previous result. This primitive takes an expected size as input. This size should match the length of the key's modulus in number of octets. The result is the **encoded message**.
4. Apply the **EMSA-PKCS1-V1_5-ENCODE** primitive to the message that is to be verified. The result is another **encoded message**. This primitive makes use of a hash function (usually a SHA family hash function such as SHA-256). This primitive accepts an expected encoded message length. In this case, it will be the length in octets of the RSA number `n` (the key length).
5. Compare both encoded messages (from steps 3 and 4). If they match, the signature is valid,

otherwise it is not.

In JavaScript:

```
/**
 * Verifies a signature for a message using the RSASSA algorithm as defined
 * in PKCS#1.
 * @param {publicKey} RSA private key, an object with
 *                    three members: size (size in bits), n (the modulus) and
 *                    e (the public exponent), both bigInts
 *                    (big-integer library).
 * @param {hashFn} the hash function as required by PKCS#1,
 *                 it should take a Uint8Array and return a Uint8Array
 * @param {hashType} A symbol identifying the type of hash function passed.
 *                   For now, only "SHA-256" is supported. See the "hashTypes"
 *                   object for possible values.
 * @param {message} A String or Uint8Array with arbitrary data to verify
 * @param {signature} A Uint8Array with the signature
 * @return {Boolean} true if the signature is valid, false otherwise.
 */
export function verifyPkcs1v1_5(publicKey,
                                hashFn,
                                hashType,
                                message,
                                signature) {
    if(signature.length !== publicKey.size / 8) {
        throw new Error('invalid signature length');
    }

    const intSignature = os2ip(signature);
    const intVerification = rsavp1(publicKey, intSignature);
    const verificationMessage = i2osp(intVerification, publicKey.size / 8);

    const encodedMessage =
        emsaPkcs1v1_5(hashFn, hashType, publicKey.size / 8, message);

    return uint8ArrayEquals(encodedMessage, verificationMessage);
}
```

7.2.2.3.1.1 EMSA-PKCS1-v1_5 primitive

This primitive takes three elements:

- The message
- The intended length of the result
- And the hash function to use (which must be one of the options from step 2)

1. Apply the selected hash function to the message.

2. Produce the DER encoding for the following ASN.1 structure:

```
DigestInfo ::= SEQUENCE {
    digestAlgorithm DigestAlgorithm,
    digest OCTET STRING
}
```

Where `digest` is the result from step 1 and `DigestAlgorithm` is one of:

```
DigestAlgorithm ::=
    AlgorithmIdentifier { {PKCS1-v1-5DigestAlgorithms} }

PKCS1-v1-5DigestAlgorithms    ALGORITHM-IDENTIFIER ::= {
    { OID id-md2 PARAMETERS NULL    }|
    { OID id-md5 PARAMETERS NULL    }|
    { OID id-sha1 PARAMETERS NULL   }|
    { OID id-sha256 PARAMETERS NULL }|
    { OID id-sha384 PARAMETERS NULL }|
    { OID id-sha512 PARAMETERS NULL }
}
```

3. If the requested length of the result is less than the result of step 3 plus `11` (reqLength < step2Length + 11), then the primitive fails to produce the result and outputs an error message ("intended encoded message length too short").
4. Repeat the octet `0xFF` the following number of times: `requested length + step2Length - 3`. This array of octets is called `PS`.
5. Produce the final encoded message (EM) as (`||` is the concatenation operator):

```
EM = 0x00 || 0x01 || PS || 0x00 || step2Result
```

ASN.1 OIDs are usually defined in their own specifications. In other words, you will not find the SHA-256 OID in the PKCS#1 spec. SHA-1 and SHA-2 OIDs are defined in RFC 3560[15].

7.2.2.3.1.2 OS2IP primitive

The OS2IP primitive takes an array of octets and outputs an integer representative.

- Let X_1, X_2, ..., X_n be the octets from first to last of the input.
- Compute the result as:

$$result = X_1 \cdot 256^{n-1} + X_2 \cdot 256^{n-2} + ... + X_{n-1} \cdot 256 + X_n$$

Figure 7.7: OS2IP result

7.2.2.3.1.3 RSASP1 primitive

[15] https://tools.ietf.org/html/rfc3560.html

The RSASP1 primitive takes the private key and a message representative and produces a signature representative.

- Let `n` and `d` be the RSA numbers for the private key.
- Let `m` be the message representative.

1. Check the message representative is in range: between 0 and `n - 1`.
2. Compute the result as follows:

$$s = m^d \mod (n)$$

Figure 7.8: RSASP1 result

> PKCS#1 defines an alternative, computationally convenient way of storing the private key: rather than keeping `n` and `d` in it, a combination of different precomputed values for certain operations are stored. These values can be directly used in certain operations and can speed up computations significantly. Most private keys are stored this way. Storing the private key as `n` and `d` is valid, though.

7.2.2.3.1.4 RSAVP1 primitive

The RSAVP1 primitive takes a public key and an integer signature representative and produces an integer message representative.

- Let `n` and `e` be the RSA numbers for the public key.
- Let `s` be the integer signature representative.

1. Check the message representative is in range: between 0 and `n - 1`.
2. Compute the result as follows:

$$m = s^e \mod (n)$$

Figure 7.9: RSAVP1 result

7.2.2.3.1.5 I2OSP primitive

The I2OSP primitive takes an integer representative and produces an array of octets.

- Let `len` be the expected length of the array of octets.
- Let `x` be the integer representative.

1. If x > 256^{len} then the integer is too large and the arguments are wrong.
2. Compute the base-256 representation of the integer:

$$x = x_1 \cdot 256^{len-1} + x_2 \cdot 256^{len-2} + ... + x_{len-1} \cdot 256 + x_{len}$$

Figure 7.10: I2OSP decomposition

3. Take each x_{len-i} factor from each term in order. These are the octets for the result.

7.2.2.3.2 Sample code

Since RSA requires arbitrary precision arithmetic, we will be using the big-integer[16] JavaScript library.

The `OS2IP` and `I2OSP` primitives are rather simple:

```
function os2ip(bytes) {
    let result = bigInt();

    bytes.forEach((b, i) => {
        // result += b * Math.pow(256, bytes.length - 1 - i);
        result = result.add(
            bigInt(b).multiply(
                bigInt(256).pow(bytes.length - i - 1)
            )
        );
    });

    return result;
}

function i2osp(intRepr, expectedLength) {
    if(intRepr.greaterOrEquals(bigInt(256).pow(expectedLength))) {
        throw new Error('integer too large');
    }

    const result = new Uint8Array(expectedLength);
    let remainder = bigInt(intRepr);
    for(let i = expectedLength - 1; i >= 0; --i) {
        const position = bigInt(256).pow(i);
        const quotrem = remainder.divmod(position);
        remainder = quotrem.remainder;
        result[result.length - 1 - i] = quotrem.quotient.valueOf();
    }

    return result;
}
```

The `I2OSP` primitive essentially decomposes a number into its base 256[17] components.

The `RSASP1` primitive is essentially a single operation, and forms the basis of the algorithm:

[16] https://www.npmjs.com/package/big-integer
[17] https://en.wikipedia.org/wiki/Positional_notation

```
function rsasp1(privateKey, intMessage) {
    if(intMessage.isNegative() ||
       intMessage.greaterOrEquals(privateKey.n)) {
        throw new Error("message representative out of range");
    }

    // result = intMessage ^ d  (mod n)
    return intMessage.modPow(privateKey.d, privateKey.n);
}
```

For verifications, the `RSAVP1` primitive is used instead:

```
export function rsavp1(publicKey, intSignature) {
    if(intSignature.isNegative() ||
       intSignature.greaterOrEquals(publicKey.n)) {
        throw new Error("message representative out of range");
    }

    // result = intSignature ^ e (mod n)
    return intSignature.modPow(publicKey.e, publicKey.n);
}
```

Finally, the `EMSA-PKCS1-v1_5` primitive performs most of the hard work by transforming the message into its encoded and padded representation.

```
function emsaPkcs1v1_5(hashFn, hashType, expectedLength, message) {
    if(hashType !== hashTypes.sha256) {
        throw new Error("Unsupported hash type");
    }

    const digest = hashFn(message, true);

    // DER is a stricter set of BER, this (fortunately) works:
    const berWriter = new Ber.Writer();
    berWriter.startSequence();
        berWriter.startSequence();
        // SHA-256 OID
        berWriter.writeOID("2.16.840.1.101.3.4.2.1");
        berWriter.writeNull();
        berWriter.endSequence();
    berWriter.writeBuffer(Buffer.from(digest), ASN1.OctetString);
    berWriter.endSequence();

    // T is the name of this element in RFC 3447
    const t = berWriter.buffer;

    if(expectedLength < (t.length + 11)) {
        throw new Error('intended encoded message length too short');
```

```
    }

    const ps = new Uint8Array(expectedLength - t.length - 3);
    ps.fill(0xff);
    assert.ok(ps.length >= 8);

    return Uint8Array.of(0x00, 0x01, ...ps, 0x00, ...t);
}
```

For simplicity, only SHA-256 is supported. Adding other hash functions is as simple as adding the right OIDs.

The `signPkcs1v1_5` function puts all primitives together to perform the signature:

```
/**
 * Produces a signature for a message using the RSA algorithm as defined
 * in PKCS#1.
 * @param {privateKey} RSA private key, an object with
 *                     three members: size (size in bits), n (the modulus) and
 *                     d (the private exponent), both bigInts
 *                     (big-integer library).
 * @param {hashFn} the hash function as required by PKCS#1,
 *                 it should take a Uint8Array and return a Uint8Array
 * @param {hashType} A symbol identifying the type of hash function passed.
 *                   For now, only "SHA-256" is supported. See the "hashTypes"
 *                   object for possible values.
 * @param {message} A String or Uint8Array with arbitrary data to sign
 * @return {Uint8Array} The signature as a Uint8Array
 */
export function signPkcs1v1_5(privateKey, hashFn, hashType, message) {
    const encodedMessage =
        emsaPkcs1v1_5(hashFn, hashType, privateKey.size / 8, message);
    const intMessage = os2ip(encodedMessage);
    const intSignature = rsasp1(privateKey, intMessage);
    const signature = i2osp(intSignature, privateKey.size / 8);
    return signature;
}
```

To use this to sign JWTs, a simple wrapper is necessary:

```
export default function jwtEncode(header, payload, privateKey) {
    if(typeof header !== 'object' || typeof payload !== 'object') {
        throw new Error('header and payload must be objects');
    }

    header.alg = 'RS256';

    const encHeader = b64(JSON.stringify(header));
    const encPayload = b64(JSON.stringify(payload));
```

```
    const jwtUnprotected = `${encHeader}.${encPayload}`;
    const signature = b64(
        pkcs1v1_5.sign(privateKey,
                       msg => sha256(msg, true),
                       hashTypes.sha256, stringToUtf8(jwtUnprotected)));

    return `${jwtUnprotected}.${signature}`;
}
```

This function is very similar to the `jwtEncode` function for `HS256` shown in the HMAC section.

Verification is just as simple:

```
/**
 * Verifies a signature for a message using the RSASSA algorithm as defined
 * in PKCS#1.
 * @param {publicKey} RSA private key, an object with
 *                    three members: size (size in bits), n (the modulus) and
 *                    e (the public exponent), both bigInts
 *                    (big-integer library).
 * @param {hashFn} the hash function as required by PKCS#1,
 *                 it should take a Uint8Array and return a Uint8Array
 * @param {hashType} A symbol identifying the type of hash function passed.
 *                   For now, only "SHA-256" is supported. See the "hashTypes"
 *                   object for possible values.
 * @param {message} A String or Uint8Array with arbitrary data to verify
 * @param {signature} A Uint8Array with the signature
 * @return {Boolean} true if the signature is valid, false otherwise.
 */
export function verifyPkcs1v1_5(publicKey,
                                hashFn,
                                hashType,
                                message,
                                signature) {
    if(signature.length !== publicKey.size / 8) {
        throw new Error('invalid signature length');
    }

    const intSignature = os2ip(signature);
    const intVerification = rsavp1(publicKey, intSignature);
    const verificationMessage = i2osp(intVerification, publicKey.size / 8);

    const encodedMessage =
        emsaPkcs1v1_5(hashFn, hashType, publicKey.size / 8, message);

    return uint8ArrayEquals(encodedMessage, verificationMessage);
}
```

To use this to verify JWTs, a simple wrapper is necessary:

```
export function jwtVerifyAndDecode(jwt, publicKey) {
    if(!isString(jwt)) {
        throw new TypeError('jwt must be a string');
    }

    const split = jwt.split('.');
    if(split.length !== 3) {
        throw new Error('Invalid JWT format');
    }

    const header = JSON.parse(unb64(split[0]));
    if(header.alg !== 'RS256') {
        throw new Error(`Wrong algorithm: ${header.alg}`);
    }

    const jwtUnprotected = stringToUtf8(`${split[0]}.${split[1]}`);
    const valid = verifyPkcs1v1_5(publicKey,
                                  msg => sha256(msg, true),
                                  hashTypes.sha256,
                                  jwtUnprotected,
                                  base64.decode(split[2]));

    return {
        header: header,
        payload: JSON.parse(unb64(split[1])),
        valid: valid
    };
}
```

For simplicity, the private and public keys must be passed as JavaScript objects with two separate numbers: the modulus (`n`) and the private exponent (`d`) for the private key, and the modulus (`n`) and the public exponent (`e`) for the public key. This is in contrast to the usual PEM Encoded[18] format. See the `rs256.js` file for more details.

It is possible to use OpenSSL to export these numbers from a PEM key.

```
openssl rsa -text -noout -in testkey.pem
```

OpenSSL can also be used to generate a RSA key from scratch:

```
openssl genrsa -out testkey.pem 2048
```

You can then export the numbers from PEM format using the command shown above.

The private-key numbers embedded in the `testkey.js` file are from the `testkey.pem` file in the samples directory accompanying this handbook. The corresponding public key is in the `pubtestkey.pem` file.

[18] https://en.wikipedia.org/wiki/Privacy-enhanced_Electronic_Mail

Copy the output of running the rs256.js sample[19] into the JWT area at JWT.io[20]. Then copy the contents of `pubtestkey.pem` to the public-key area in the same page and the JWT will be successfully validated.

7.2.2.4 PS256: RSASSA-PSS using SHA-256 and MGF1 with SHA-256

RSASSA-PSS is another signature scheme with appendix based on RSA. "PSS" stands for Probabilistic Signature Scheme, in contrast with the usual *deterministic* approach. This scheme makes use of a cryptographically secure random number generator. If a secure RNG is not available, the resulting signature and verification operations provide a level of security comparable to deterministic approaches. This way RSASSA-PSS results in a net improvement over PKCS v1.5 signatures for best case scenarios. In the wild, however, both PSS and PKCS v1.5 schemes remain unbroken.

RSASSA-PSS is defined in Public Key Cryptography Standard #1 (PKCS #1)[21] and is not available in earlier versions of the standard.

7.2.2.4.1 Algorithm

To produce a signature:

1. Apply the **EMSA-PSS-ENCODE** primitive to the message. The primitive takes a parameter that should be the number of bits in the modulus of the key minus 1. The result is the **encoded message**.
2. Apply the **OS2IP** primitive to the encoded message. The result is the **integer message representative**. OS2IP is the acronym for "Octet-String to Integer Primitive".
3. Apply the **RSASP1** primitive to the integer message representative using the private key. The result is the **integer signature representative**.
4. Apply the **I2OSP** primitive to convert the integer signature representative to an array of octets (the **signature**). I2OSP is the acronym for "Integer to Octet-String Primitive".

A possible implementation in JavaScript, given the primitives mentioned above, could look like:

```
export function signPss(privateKey, hashFn, hashType, message) {
    if(hashType !== hashTypes.sha256) {
        throw new Error('unsupported hash type');
    }

    const encodedMessage = emsaPssEncode(hashFn,
                                         hashType,
                                         mgf1.bind(null, hashFn),
                                         256 / 8, //size of hash
                                         privateKey.size - 1,
                                         message);
    const intMessage = os2ip(encodedMessage);
```

[19] https://github.com/auth0/jwt-handbook-samples/blob/master/rs256.js
[20] https://jwt.io
[21] https://www.ietf.org/rfc/rfc3447.txt

```
    const intSignature = rsasp1(privateKey, intMessage);
    const signature = i2osp(intSignature, privateKey.size / 8);
    return signature;
}
```

To verify a signature:

1. Apply the **OS2IP** primitive to the signature (an array of octets). This is the **integer signature representative**.
2. Apply the **RSAVP1** primitive to the previous result. This primitive also takes the public key as input. This is the **integer message representative**.
3. Apply the **I2OSP** primitive to the previous result. This primitive takes an expected size as input. This size should match the length of the key's modulus in number of octets. The result is the **encoded message**.
4. Apply the **EMSA-PSS-VERIFY** primitive to the message that is to be verified and the result of the previous step. This primitive outputs whether the signature is valid or not. This primitive makes use of a hash function (usually a SHA family hash function such as SHA-256). The primitive takes a parameter that should be the number of bits in the modulus of the key minus 1.

```
export function verifyPss(publicKey, hashFn, hashType, message, signature) {
    if(signature.length !== publicKey.size / 8) {
        throw new Error('invalid signature length');
    }

    const intSignature = os2ip(signature);
    const intVerification = rsavp1(publicKey, intSignature);
    const verificationMessage =
        i2osp(intVerification, Math.ceil( (publicKey.size - 1) / 8 ));

    return emsaPssVerify(hashFn,
                         hashType,
                         mgf1.bind(null, hashFn),
                         256 / 8,
                         publicKey.size - 1,
                         message,
                         verificationMessage);
}
```

7.2.2.4.1.1 MGF1: the mask generation function

Mask generation functions take input of any length and produce output of variable length. Like hash functions, they are deterministic: they produce the same output for the same input. In contrast to hash functions, though, the length of the output is variable. The Mask Generation Function 1 (`MGF1`) algorithm is defined in Public Key Cryptography Standard #1 (PKCS #1)[22].

[22] https://www.ietf.org/rfc/rfc3447.txt

`MGF1` takes a seed value and the intended length of the output as inputs. The maximum length of the output is defined as 2^{32}. `MGF1` uses internally a configurable hash function. `PS256` specifies this hash function as `SHA-256`.

1. If the intended length is bigger than 2^{32}, stop with error "mask too long".
2. Iterate from 0 to the ceiling of the intended length divided the length of the hash function output minus 1 (`ceiling(intendedLength / hashLength) - 1`) doing the following operations:
    1. Let `c = i2osp(counter, 4)` where `counter` is the current value of the iteration counter.
    2. Let `t = t.concat(hash(seed.concat(c)))` where `t` is preserved between iterations, `hash` is the selected hash function (`SHA-256`) and `seed` is the input seed value.
3. Output the leftmost intended length octets from the last value of `t` as the result of the function.

7.2.2.4.1.2 EMSA-PSS-ENCODE primitive

The primitive takes two elements:

- The message to be encoded as an octet sequence.
- The intended maximum length of the result in bits.

This primitive can be parameterized by the following elements:

- A hash function. In the case of `PS256` this `SHA-256`.
- A mask generation function. In the case of `PS256` this is `MGF1`.
- An intended length for the salt used internally.

These parameters are all specified by `PS256`, so they are not configurable and for the purposes of this description are considered constants.

Note that the intended length used as input is expressed in bits. For the following examples, consider:

```
const intendedLength = Math.ceil(intendedLengthBits / 8);
```

1. If the input is bigger than the maximum length of the hash function, stop. If not, apply the hash function to the message.

```
const hashed1 = sha256(inputMessage);
```

2. If the intended length of the message is less than the length of the hash plus the length of the salt plus 2, stop with an error.

```
if(intendedLength < (hashed1.length + intendedSaltLength + 2)) {
    throw new Error('Encoding error');
}
```

3. Generate a random octet sequence of the length of the salt.
4. Concatenate eight zero-valued octets with the hash of the message and the salt.

```
const m = [0,0,0,0,0,0,0,0, ...hashed1, ...salt];
```

5. Apply the hash function to the result of the previous step.

```
const hashed2 = sha256(m);
```

6. Generate a sequence of zero-valued octets of length: intended maximum length of result minus salt length minus hash length minus 2.

```
const ps = new Array(intendedLength - intendedSaltLength - 2).fill(0);
```

7. Concatenate the result of the previous step with the octet 0x01 and the salt.

```
const db = [...ps, 0x01, ...salt];
```

8. Apply the mask generation function to the result of step 5 and set the intended length of this function as the length of the result from step 7 (the mask generation function accepts an intended length parameter).

```
const dbMask = mgf1(hashed2, db.length);
```

9. Compute the result of applying the XOR operation to the results of steps 7 and 8.

```
const maskedDb = db.map((value, index) => {
    return value ^ dbMask[index];
});
```

10. If the length of the result of the previous operation is not a multiple of 8, find the difference in number of bits to make it a multiple of 8 by subtracting bits, then set this number of bits to 0 beginning from the left.

```
const zeroBits = 8 * intendedLength - intendedLengthBits;
const zeroBitsMask = 0xFF >>> zeroBits;
maskedDb[0] &= zeroBitsMask;
```

11. Concatenate the result of the previous step with the result of step 5 and the octet 0xBC. This is the result.

```
const result = [...maskedDb, ...hashed2, 0xBC];
```

7.2.2.4.1.3 EMSA-PSS-VERIFY primitive

The primitive takes three elements:

- The message to be verified.
- The signature as an encoded integer message.
- The intended maximum length of the encoded integer message.

This primitive can be parameterized by the following elements:

- A hash function. In the case of `PS256` this `SHA-256`.
- A mask generation function. In the case of `PS256` this is `MGF1`.
- An intended length for the salt used internally.

These parameters are all specified by `PS256`, so they are not configurable and for the purposes of this description are considered constants.

Note that the intended length used as input is expressed in bits. For the following examples, consider:

```
const expectedLength = Math.ceil(expectedLengthBits / 8);
```

1. Hash the message to be verified using the selected hash function.

```
const digest1 = hashFn(message, true);
```

2. If the expected length is smaller than the hash length plus the salt length plus 2, consider the signature invalid.

```
if(expectedLength < (digest1.length + saltLength + 2)) {
    return false;
}
```

3. Check that the last byte of the encoded message of the signature has the value of `0xBC`

```
if(verificationMessage[verificationMessage.length - 1] !== 0xBC) {
    return false;
}
```

4. Split the encoded message into two elements. The first element has a length of `expectedLength - hashLength - 1`. The second element starts at the end of the first and has a length of `hashLength`.

```
const maskedLength = expectedLength - digest1.length - 1;
const masked = verificationMessage.subarray(0, maskedLength);
const digest2 = verificationMessage.subarray(maskedLength,
                                             maskedLength + digest1.length);
```

5. Check that the leftmost `8 * expectedLength - expectedLengthBits` (the expected length in bits minus the requested length in bits) bits of `masked` are 0.

```
const zeroBits = 8 * expectedLength - expectedLengthBits;
const zeroBitsMask = 0xFF >>> zeroBits;
if((masked[0] & (~zeroBitsMask)) !== 0) {
    return false;
}
```

6. Pass the second element extracted from step 4 (the digest) to the selected MGF function. Request the result to have a length of `expectedLength - hashLength - 1`.

```
const dbMask = mgf(maskedLength, digest2);
```

7. For each byte from first element extracted in step 4 (`masked`) apply the XOR function using the corresponding byte from the element computed in the last step (`dbMask`).

```
const db = new Uint8Array(masked.length);
for(let i = 0; i < db.length; ++i) {
    db[i] = masked[i] ^ dbMask[i];
}
```

8. Set the leftmost `8 * expectedLength - expectedLengthBits` bits from the first byte in the element computed in the last step to 0.

```
const zeroBits = 8 * expectedLength - expectedLengthBits;
const zeroBitsMask = 0xFF >>> zeroBits;
db[0] &= zeroBitsMask;
```

9. Check that the leftmost `expectedLength - hashLength - saltLength - 2` bytes of the element computed in the last step are 0. Also check that the first element after the group of zeros is `0x01`.

```
const zeroCheckLength = expectedLength - (digest1.length + saltLength + 2);
if(!db.subarray(0, zeroCheckLength).every(v => v === 0) ||
    db[zeroCheckLength] !== 0x01) {
    return false;
}
```

10. Extract the salt from the last `saltLength` octets of the element computed in the last step (`db`).

```
const salt = db.subarray(db.length - saltLength);
```

11. Compute a new encoded message by concatenating eigth octets of value zero, the hash computed in step 1, and the salt extracted in the last step.

```
const m = Uint8Array.of(0, 0, 0, 0, 0, 0, 0, 0, ...digest1, ...salt);
```

12. Compute the hash of the element computed in the last step.

```
const expectedDigest = hashFn(m, true);
```

13. Compare the element computed in the last step to the second element extracted in step 4. If they match, the signature is valid, otherwise it is not.

```
return uint8ArrayEquals(digest2, expectedDigest);
```

7.2.2.4.2 Sample code

As expected of a variant of RSASSA, most of the code required for this algorithm is already present in the implementation of RS256. The only differences are the additions of the `EMSA-PSS-ENCODE`, `EMSA-PSS-VERIFY`, and `MGF1` primitives.

```
export function mgf1(hashFn, expectedLength, seed) {
    if(expectedLength > Math.pow(2, 32)) {
        throw new Error('mask too long');
    }

    const hashSize = hashFn(Uint8Array.of(0), true).byteLength;
    const count = Math.ceil(expectedLength / hashSize);
    const result = new Uint8Array(hashSize * count);
    for(let i = 0; i < count; ++i) {
        const c = i2osp(bigInt(i), 4);
```

```
        const value = hashFn(Uint8Array.of(...seed, ...c), true);
        result.set(value, i * hashSize);
    }
    return result.subarray(0, expectedLength);
}

export function emsaPssEncode(hashFn,
                              hashType,
                              mgf,
                              saltLength,
                              expectedLengthBits,
                              message) {
    const expectedLength = Math.ceil(expectedLengthBits / 8);

    const digest1 = hashFn(message, true);
    if(expectedLength < (digest1.length + saltLength + 2)) {
        throw new Error('encoding error');
    }

    const salt = crypto.randomBytes(saltLength);
    const m = Uint8Array.of(...(new Uint8Array(8)),
                            ...digest1,
                            ...salt);
    const digest2 = hashFn(m, true);
    const ps = new Uint8Array(expectedLength - saltLength - digest2.length - 2);
    const db = Uint8Array.of(...ps, 0x01, ...salt);
    const dbMask = mgf(db.length, digest2);
    const masked = db.map((value, index) => value ^ dbMask[index]);

    const zeroBits = 8 * expectedLength - expectedLengthBits;
    const zeroBitsMask = 0xFF >>> zeroBits;
    masked[0] &= zeroBitsMask;

    return Uint8Array.of(...masked, ...digest2, 0xbc);
}

export function emsaPssVerify(hashFn,
                              hashType,
                              mgf,
                              saltLength,
                              expectedLengthBits,
                              message,
                              verificationMessage) {
    const expectedLength = Math.ceil(expectedLengthBits / 8);

    const digest1 = hashFn(message, true);
    if(expectedLength < (digest1.length + saltLength + 2)) {
```

```
        return false;
    }

    if(verificationMessage.length === 0) {
        return false;
    }

    if(verificationMessage[verificationMessage.length - 1] !== 0xBC) {
        return false;
    }

    const maskedLength = expectedLength - digest1.length - 1;
    const masked = verificationMessage.subarray(0, maskedLength);
    const digest2 = verificationMessage.subarray(maskedLength,
                                                 maskedLength + digest1.length);

    const zeroBits = 8 * expectedLength - expectedLengthBits;
    const zeroBitsMask = 0xFF >>> zeroBits;
    if((masked[0] & (~zeroBitsMask)) !== 0) {
        return false;
    }

    const dbMask = mgf(maskedLength, digest2);
    const db = masked.map((value, index) => value ^ dbMask[index]);
    db[0] &= zeroBitsMask;

    const zeroCheckLength = expectedLength - (digest1.length + saltLength + 2);
    if(!db.subarray(0, zeroCheckLength).every(v => v === 0) ||
       db[zeroCheckLength] !== 0x01) {
        return false;
    }

    const salt = db.subarray(db.length - saltLength);
    const m = Uint8Array.of(0, 0, 0, 0, 0, 0, 0, 0, ...digest1, ...salt);
    const expectedDigest = hashFn(m, true);

    return uint8ArrayEquals(digest2, expectedDigest);
}
```

The complete example[23] is available in the files `ps256.js`, `rsassa.js`, and `pkcs.js`. The private-key numbers embedded in the `testkey.js` file are from the `testkey.pem` file in the samples directory accompanying this handbook. The corresponding public key is in the `pubtestkey.pem` file. For help in creating keys see the RS256 example.

[23] https://github.com/auth0/jwt-handbook-samples

7.2.3 Elliptic Curve

Elliptic Curve (EC) algorithms, just like RSA, rely on a class of mathematical problems that are intractable for certain conditions. Intractability refers to the possibility of finding a solution given enough resources, but that, in practice, is hard to achieve. While RSA relies on the intractability of the factoring problem[24] (finding the prime factors of a big coprime number), elliptic curve algorithms rely on the intractability of the elliptic curve discrete logarithm problem.

Elliptic curves are discribed by the following equation:

$$y^2 = x^3 + ax + b$$

Figure 7.11: Elliptic curve equation

By setting `a` and `b` to different values, we get the following sample curves:

[24] https://en.wikipedia.org/wiki/Integer_factorization

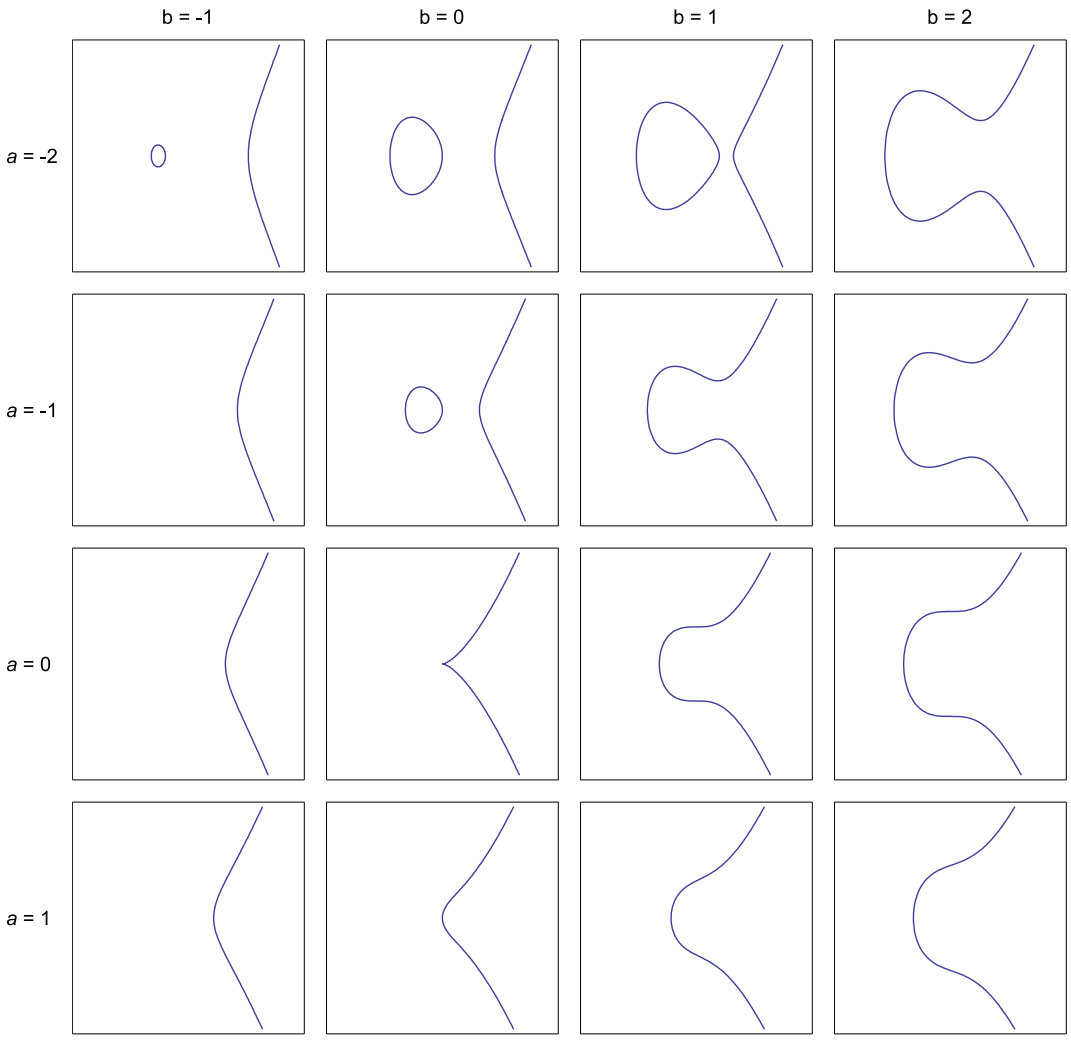

Figure 7.12: Some elliptic curves

Public domain image taken from Wikimedia.[25]

In contrast to RSA, elliptic-curve algorithms are defined for specific finite fields. Of interest for EC cryptography are binary and prime fields. The JWA specification uses only prime fields, so we will focus on these.

> A field, in mathematical terms, is a set of elements for which the four basic arithmetic operations are defined: subtraction, addition, multiplication, and division.

"Finite" means elliptic curve cryptography works on finite sets of numbers, rather than the infinite set of real numbers.

[25] https://commons.wikimedia.org/wiki/File:EllipticCurveCatalog.svg

A prime field is a field that contains a prime number `p` of elements. All elements and arithmetic operations are implemented modulo `p` (the prime number of elements).

By making the field finite, the algorithms used to perform mathematical operations change. In particular, the discrete logarithm[26] must be used instead of the ordinary logarithm. Logarithms find the value of `k` for expressions of the following form:

$$a^k = c$$
$$\log_a(c) = k$$

Figure 7.13: Exponential function

There is no known efficient, general purpose algorithm for computing the discrete logarithm. This limitation makes the discrete logarithm ideal for cryptography. Elliptic curves, as used by cryptography, exploit this limitation to provide secure asymmetric encryption and signature operations.

The intractability of the discrete logarithm problem depends on carefully choosing the parameters of the field on which it is to be used. This means that for elliptic curve cryptography to be effective, certain parameters must be chosen with great care. Elliptic curve algorithms have been the target in past attacks due to misuse[27].

An interesting aspect of Elliptic-Curve cryptography is that key sizes can be smaller while providing a similar level of security compared to bigger keys used in RSA. This allows cryptography even on memory limited devices. In general terms, a 256-bit elliptic-curve key is similar to a 3072-bit RSA key in cryptographic strength.

7.2.3.1 Elliptic-Curve Arithmetic

For the purposes of implementing elliptic-curve signatures, it is necessary to implement elliptic-curve arithmetic. The three basic operations are: point addition, point doubling, and point scalar multiplication. All three operations result in valid points on the same curve.

7.2.3.1.1 Point Addition

[26] https://en.wikipedia.org/wiki/Discrete_logarithm
[27] https://safecurves.cr.yp.to/

$$P+Q \equiv R \pmod{q} \quad (P \neq Q)$$
$$(x_p, y_p)+(x_q, y_q) \equiv (x_r, y_r) \pmod{q}$$

$$\lambda \equiv \frac{y_q - y_p}{x_q - x_p} \pmod{q}$$

$$x_r \equiv \lambda^2 - x_p - x_q \pmod{q}$$
$$y_r \equiv \lambda(x_p - x_r) - y_p \pmod{q}$$

Figure 7.14: Point addition

7.2.3.1.2 Point Doubling

$$P+Q \equiv R \pmod{q} \quad (P = Q)$$
$$2P \equiv R \pmod{q}$$
$$(x_p, y_p)+(x_p, y_p) \equiv (x_r, y_r) \pmod{q}$$

$$\lambda \equiv \frac{3x_p^2 + a}{2y_p} \pmod{q}$$

$$x_r \equiv \lambda^2 - 2x_p \pmod{q}$$
$$y_r \equiv \lambda(x_p - x_r) - y_p \pmod{q}$$

Figure 7.15: Point doubling

7.2.3.1.3 Scalar Multiplication

For scalar multiplication, the factor `k` is decomposed into its binary representation.

$$kP \equiv R \pmod{q}$$
$$k = k_0 + 2k_1 + 2^2 k_2 + \ldots + 2^m k_m \quad where \quad [k_0 \ldots k_m] \in \{0,1\}$$

Figure 7.16: Scalar multiplication

Then, the following algorithm is applied:

1. Let `N` be the point `P`.
2. Let `Q` be the point at infinity `(0, 0)`.
3. For `i` from `0` to `m` do:

    1. If `k~i~` is `1` then let `Q` be the result of adding `Q` to `N` (elliptic-curve addition).
    2. Let `N` be the result of doubling `N` (elliptic-curve doubling).
4. Return `Q`.

Sample implementation in JavaScript:

```
function ecMultiply(P, k, modulus) {
    let N = Object.assign({}, p);
    let Q = {
        x: bigInt(0),
        y: bigInt(0)
    };

    for(k = bigInt(k); !k.isZero(); k = k.shiftRight(1)) {
        if(k.isOdd()) {
            Q = ecAdd(Q, N, modulus);
        }
        N = ecDouble(N, modulus);
    }

    return Q;
}
```

One thing to note is that in modular arithmetic, division is implemented as the multiplication between the numerator and the inverse of the divisor.

JavaScript versions of these operations can be found in the samples repository[28] in the `ecdsa.js` file. These naive implementations, though functional, are vulnerable to timing attacks. Production-ready implementations use different algorithms that take these attacks into account.

7.2.3.2 Elliptic-Curve Digital Signature Algorithm (ECDSA)

The Elliptic-Curve Digital Signature Algorithm (ECDSA) was developed by a committee for the American National Standards Institute (ANSI)[29]. The standard is X9.63[30]. The standard specifies all the needed parameters for proper use of elliptic-curves for signatures in a secure way. The JWA specification relies on this specification (and FIPS 186-4[31]) for picking curve parameters and specifying the algorithm.

For use with JWTs, JWA specifies that the input to signing algorithm is the `Base64` encoded header and payload, just like any other signing algorithm, but the result is two integers `r` and `s` rather than one. These integers are to be converted to 32-byte sequences in big-endian order, which are then concatenated to form a single 64-byte signature.

```
export default function jwtEncode(header, payload, privateKey) {
    if(typeof header !== 'object' || typeof payload !== 'object') {
```

[28] https://github.com/auth0/jwt-handbook-samples/
[29] https://www.ansi.org/
[30] https://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.63-2011+(R2017)
[31] http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

```
        throw new Error('header and payload must be objects');
    }

    header.alg = 'ES256';

    const encHeader = b64(JSON.stringify(header));
    const encPayload = b64(JSON.stringify(payload));
    const jwtUnprotected = `${encHeader}.${encPayload}`;
    const ecSignature = sign(privateKey, sha256,
                             sha256.hashType, stringToUtf8(jwtUnprotected));
    const ecR = i2osp(ecSignature.r, 32);
    const ecS = i2osp(ecSignature.s, 32);
    const signature = b64(Uint8Array.of(...ecR, ...ecS));

    return `${jwtUnprotected}.${signature}`;
}
```

This code is similar to what was used for RSA and HMAC signatures. The main difference lies in converting the two signature numbers `r` and `s` into 32-byte octets. For this we can use the `i2osp` function from PKCS, which we also used for RSA.

Checking the signature requires retrieving parameters `r` and `s`:

```
export function jwtVerifyAndDecode(jwt, publicKey) {
    const header = JSON.parse(unb64(split[0]));
    if(header.alg !== 'ES256') {
        throw new Error(`Wrong algorithm: ${header.alg}`);
    }

    const jwtUnprotected = stringToUtf8(`${split[0]}.${split[1]}`);

    const signature = base64.decode(split[2]);
    const ecR = signature.slice(0, 32);
    const ecS = signature.slice(32);
    const ecSignature = {
        r: os2ip(ecR),
        s: os2ip(ecS)
    };

    const valid = verify(publicKey,
                         sha256,
                         sha256.hashType,
                         jwtUnprotected,
                         ecSignature);

    return {
        header: header,
        payload: JSON.parse(unb64(split[1])),
```

```
        valid: valid
    };
}
```

Again, the procedure for checking the validity of the signature is similar to RSA and HMAC. In this case, values `r` and `s` must be retrieved from the 64-byte JWT signature. The first 32 bytes are the element `r`, and the remaining 32 bytes are the element `s`. To convert these values into numbers we can use the `os2ip` primitive from PKCS.

7.2.3.2.1 Elliptic-Curve Domain Parameters

Elliptic-curve operations as used by ECDSA depend on a few key parameters:

- `p` or `q`: the prime used to define the prime field[32] on which arithmetic operations are performed. Prime field operations use modular arithmetic[33].
- `a`: coefficient of `x` in the curve equation.
- `b`: constant in the curve equation (y-intercept).
- `G`: a valid curve point used as `base point` for elliptic-curve operations. The base point is used in arithmetic operations to obtain other points on the curve.
- `n`: the order of base point `G`. This parameter is the number of valid points in the curve that can be constructed by using point `G` as base point.

For elliptic-curve operations to be secure, these parameters must be chosen carefully. In the context of `JWA`, there are only three curves that are considered valid: `P-256`, `P-384`, and `P-521`. These curves are defined in FIPS 186-4[34] and other associated standards.

For our code sample, we will be using curve P-256:

```
const p256 = {
    q: bigInt('00ffffffff00000001000000000000' +
              '000000000000ffffffffffffffffff' +
              'ffffff', 16),
    // order of base point
    n: bigInt('115792089210356248762697446949407573529996955224135760342' +
              '422259061068512044369'),
    // base point
    G: {
        x: bigInt('6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0' +
                  'f4a13945d898c296', 16),
        y: bigInt('4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ece' +
                  'cbb6406837bf51f5', 16)
    },
    //a: bigInt(-3)
    a: bigInt('00ffffffff00000001000000000000' +
              '000000000000ffffffffffffffffff' +
```

[32] https://en.wikipedia.org/wiki/Finite_field
[33] https://en.wikipedia.org/wiki/Modular_arithmetic
[34] http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

```
            'fffffc', 16),
    b: bigInt('5ac635d8aa3a93e7b3ebbd55769886' +
            'bc651d06b0cc53b0f63bce3c3e27d2' +
            '604b', 16)
};
```

7.2.3.2.2 Public and Private Keys

Constructing public and private keys using elliptic curves is really simple.

A private key can be constructed by picking a random number between 1 and the order `n` of the base point `G`. In other words:

```
const privateKey = bigInt.randBetween(1, p256.n);
```

That's it! As simple as that.

The public key can be computed from the private key by multiplying the base point `G` with the private key:

```
// ecMultiply is the elliptic-curve scalar multiplication operation
const publicKey = ecMultiply(G, privateKey);
```

In other words, the public key is a point on the elliptic-curve, while the private key is simply a scalar value.

7.2.3.2.2.1 The Discrete Logarithm Problem

Given that derivation of the public key from the private key is simple, doing the opposite appears to be simple as well. We want to find a number `d` such that `G` multiplied by it yields the public key `Q`.

$$dG \equiv Q \pmod{q}$$
$$\log_G(Q) \equiv d \pmod{q}$$

Figure 7.17: Private key as the logarithm of public key

In the context of an additive group[35] such as the prime field chosen for elliptic curves, computing `k` *is* the discrete logarithm problem. There is no known general purpose algorithm that can compute this efficiently. For 256-bit numbers like the ones used for curve P-256 the complexity is well beyond current computational capabilities. This is where the strength of elliptic-curve cryptography lies.

7.2.3.2.3 ES256: ECDSA using P-256 and SHA-256

The signing algorithm itself is simple, and requires modular arithmetic and elliptic curve operations:

[35] https://crypto.stackexchange.com/questions/15075/is-the-term-elliptic-curve-discrete-logarithm-problem-a-misnomer

1. Compute the digest of the message to sign using a cryptographically secure hash function. Let this number be `e`.
2. Use a cryptographically secure random number generator to pick a number `k` in the range `1` to `n - 1`.
3. Mutiply the base point `G` with `k` (`mod q`).
4. Let `r` be the result of taking the `x` coordinate of the point from the previous step modulo the order of `G` (`n`).
5. If `r` is zero repeat steps 2 to 5 until it is not zero.
6. Let `d` be the private key and `s` the result of:

$$s \equiv \frac{dr+e}{k} \pmod{n}$$

Figure 7.18: s

7. If `s` is zero, repeat steps 2 to 7 until it is not zero.

The signature is the tuple `r` and `s`. For the purposes of JWA, `r` and `s` are represented as two 32-byte octet sequences concatenated (first `r` and then `s`).

Sample implementation:

```
export function sign(privateKey, hashFn, hashType, message) {
    if(hashType !== hashTypes.sha256) {
        throw new Error('unsupported hash type');
    }

    // Algorithm as described in ANS X9.62-1998, 5.3

    const e = bigInt(hashFn(message), 16);

    let r;
    let s;
    do {
        let k;
        do {
            // Warning: use a secure RNG here
            k = bigInt.randBetween(1, p256.nMin1);
            const point = ecMultiply(p256.G, k, p256.q);
            r = point.x.fixedMod(p256.n);
        } while(r.isZero());

        const dr = r.multiply(privateKey.d);
        const edr = dr.add(e);
        s = edr.multiply(k.modInv(p256.n)).fixedMod(p256.n);
    } while(s.isZero());

    return {
```

```
        r: r,
        s: s
    };
}
```

Verification is just as simple. For a given signature `(r,s)`:

1. Compute the digest of the message to sign using a cryptographically secure hash function. Let this number be `e`.
2. Let `c` be multiplicative inverse of `s` modulo the order `n`.
3. Let `u1` be `e` multiplied by `c` modulo `n`.
4. Let `u2` be `r` multiplied by `c` modulo `n`.
5. Let point `A` be the base point `G` multiplied by `u1` modulo `q`.
6. Let point `B` be the public key `Q` multiplied by `u2` modulo `q`.
7. Let point `C` be the elliptic-curve addition of points `A` and `B` (modulo `q`).
8. Let `v` be the `x` coordinate of point `C` modulo `n`.
9. If `v` is equal to `r` the signature is valid, otherwise it is not.

Sample implementation:

```
export function verify(publicKey, hashFn, hashType, message, signature) {
    if(hashType !== hashTypes.sha256) {
        throw new Error('unsupported hash type');
    }

    if(signature.r.compare(1) === -1 || signature.r.compare(p256.nMin1) === 1 ||
       signature.s.compare(1) === -1 || signature.s.compare(p256.nMin1) === 1) {
        return false;
    }

    // Check whether the public key is a valid curve point
    if(!isValidPoint(publicKey.Q)) {
        return false;
    }

    // Algorithm as described in ANS X9.62-1998, 5.4

    const e = bigInt(hashFn(message), 16);

    const c = signature.s.modInv(p256.n);
    const u1 = e.multiply(c).fixedMod(p256.n);
    const u2 = signature.r.multiply(c).fixedMod(p256.n);

    const pointA = ecMultiply(p256.G, u1, p256.q);
    const pointB = ecMultiply(publicKey.Q, u2, p256.q);
    const point = ecAdd(pointA, pointB, p256.q);

    const v = point.x.fixedMod(p256.n);
```

```
    return v.compare(signature.r) === 0;
}
```

An important part of the algorithm that is often overlooked is the check of validity of the public key. This has been a source of attacks in the past[36]. If an attacker controls the public key that a verifying party uses for validation of the message signature, and the public key is not validated as a point on the curve, the attacker can craft a special public key that can be used to leak information to the attacker. This is particularly important in the light of key agreement protocols, some of which are used for encrypting JWTs.

This sample implementation can be found in the samples repository[37] in the `ecdsa.js` file.

7.3 Future Updates

The JWA specification has many more algorithms. In future versions of this handbook we will go over the remaining algorithms.

[36] http://blogs.adobe.com/security/2017/03/critical-vulnerability-uncovered-in-json-encryption.html
[37] https://github.com/auth0/jwt-handbook-samples/

Chapter 8

Annex A. Best Current Practices

Since their release, JWTs have been used in many different places. This has exposed JWTs, and library implementations, to a number of attacks. We have mentioned some of them in the preceding chapters. In this section we will take a look at the current best practices for working with JWTs.

This section is based on the draft for JWT Best Current Practices[1] from the IETF OAuth Working Group[2]. The version of the draft used in this section is 00, dated July 19, 2017[3].

8.1 Pitfalls and Common Attacks

Before taking a look at the first attack, it is important to note that many of these attacks are related to the implementation, rather than the design, of JSON Web Tokens. This does not make them less critical. It is arguable whether some of these attacks could be mitigated, or removed, by changing the underlying design. For the moment, the JWT specification and format is set in stone, so most changes happen in the implementation space (changes to libraries, APIs, programming practices and conventions).

It is also important to have a basic idea of the most common representation for JWTs: the JWS Compact Serialization format. Unserialized JWTs have two main JSON objects in them: the `header` and the `payload`.

The `header` object contains information about the JWT itself: the type of token, the signature or encryption algorithm used, the key id, etc.

The `payload` object contains all the relevant information carried by the token. There are several standard claims, like `sub` (subject) or `iat` (issued at), but any custom claims can be included as part of the payload.

[1] https://tools.ietf.org/wg/oauth/draft-ietf-oauth-jwt-bcp/
[2] https://tools.ietf.org/wg/oauth/
[3] https://tools.ietf.org/html/draft-ietf-oauth-jwt-bcp-00

These objects are encoded using the JWS Compact Serialization format to produce something like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

This is a signed JWT. Signed JWTs in compact format are simply the header and payload objects encoded using Base64-URL encoding and separated by a dot (`.`). The last part of the compact representation is the signature. In other words, the format is:

```
[Base64-URL encoded header].[Base64-URL encoded payload].[Signature]
```

This only applies to signed tokens. Encrypted tokens have a different serialized compact format that also relies on Base64-URL encoding and dot-separated fields.

If you want to play with JWTs and see how they are encoded/decoded, check JWT.io[4].

8.1.1 "alg: none" Attack

As we mentioned before, JWTs carry two JSON objects with important information, the `header` and the `payload`. The header includes information about the algorithm used by the JWT to sign or encrypt the data contained in it. Signed JWTs sign both the header and the payload, while encrypted JWTs only encrypt the payload (the header must always be readable).

In the case of signed tokens, although the signature does protect the header and payload against tampering, it is possible to *rewrite* the JWT without using the signature and changing the data contained in it. How does this work?

Take for instance a JWT with a certain header and payload. Something like this:

```
header: {
  alg: "HS256",
  typ: "JWT"
},
payload: {
  sub: "joe"
  role: "user"
}
```

Now, let's say you encode this token into its compact serialized form with a signature and a signing key of value "secret". We can use JWT.io[5] for that. The result is (newlines inserted for readability):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiJqb2UiLCJyb2xlIjoidXNlciJ9.
vqf3WzGLAxHW-X7UP-co3bU_lSUdVjF2MKtLtSU1kzU
```

Go ahead and paste that on JWT.io[6].

[4] https://jwt.io
[5] https://jwt.io
[6] https://jwt.io

Now, since this is a signed token, we are free to read it. This also means we could construct a similar token with slightly changed data in it, although in that case we would not be able to sign it unless we knew the signing key. Let's say an attacker does not know the signing key, what could he or she do? In this type of attack, the malicious user could attempt to use a token with no signature! How does that work?

First, the attacker modifies the token. For example:

```
header: {
  alg: "none",
  typ: "JWT"
},
payload: {
  sub: "joe"
  role: "admin"
}
```

Encoded (newlines inserted for readability):

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOiJqb2UiLCJyb2xlIjoiYWRtaW4ifQ.
```

Note that this token does not include a signature (`"alg": "none"`) and that the `role` claim of the payload has been changed. If the attacker manages to use this token successfully, he or she may achieve an escalation of privilege attack! Why would an attack like this work? Let's take a look at how some hypothetical JWT library could work. Let's say we have decoding function that looks like this:

```
function jwtDecode(token, secret) {
  // (...)
}
```

This function takes an encoded token and a secret and attempts to verify the token and then return the decoded data in it. If verification fails, it throws an exception. To pick the right algorithm for verification, the function relies on the `alg` claim from the header. This is where the attack succeeds. In the past[7], many libraries relied on this claim to pick the verification algorithm, and, as you may have guessed, in our malicious token the `alg` claim is `none`. That means that there's no verification algorithm, and the verification step always succeeds.

As you can see, this is a classic example of an attack that relies on a certain ambiguity of the API of a specific library, rather than a vulnerability in the specification itself. Even so, this is a real attack that was possible in several different implementations in the past. For this reason, many libraries today report `"alg": "none"` tokens as invalid, even though there's no signature in place. There are other possible mitigations for this type of attack, the most important one being to always check the algorithm specified in the header before attempting to verify a token. Another one is to use libraries that require the verification algorithm as an input to the verification function, rather than rely on the `alg` claim.

[7] https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

8.1.2 RS256 Public-Key as HS256 Secret Attack

This attack is similar to the `"alg": "none"` attack and also relies on ambiguity in the API of certain JWT libraries. Our sample token will be similar to the one for that attack. In this case, however, rather than removing the signature, we will construct a valid signature that the verification library will also consider valid by relying on a loophole in many APIs. First, consider the typical function signature of some JWT libraries for the verification function:

```
function jwtDecode(token, secretOrPublicKey) {
  // (...)
}
```

As you can see here, this function is essentially identical to the one from the `"alg": "none"` attack. If verification is successful, the token is decoded and returned, otherwise an exception is thrown. In this case, however, the function also accepts a public key as the second parameter. In a way, this makes sense: both the public key and the shared secret are usually strings or byte arrays, so from the point of view of the necessary types for that function argument, a single argument can represent both a public key (for RS, ES, or PS algorithms) and a shared secret (for HS algorithms). This type of function signature is common for many JWT libraries.

Now, suppose the attacker gets an encoded token signed with an RSA key pair. It looks like this (newlines inserted for readability):

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJqb2UiLCJyb2xlIjoidXNlciJ9.
QDjcv11Kcb69THVLKMErYqzy9htWlCDtBdonVR5SX4geZa_R8StjwUuuskveUsdJVgjgXwMso7p
uAJZzoE9LEr9XCxau7SF1ddws4ONiqxSVXZbO0pSgbKm3FpkVz4Jyy4oNTs-
bIYyE0xf8snFlT1MbBWcG5psnuG04IEle4s
```

Decoded:

```
header: {
  "alg": "RS256",
  "typ": "JWT"
},
payload: {
  "sub": "joe",
  "role": "user"
}
```

This token is signed with an RSA key-pair. RSA signatures are produced with the private key, while verification is done with the public key. Whoever verifies the token in the future could make a call to our hypothetical `jwtDecode` function from before like so:

```
const publicKey = '...';
const decoded = jwtDecode(token, publicKey);
```

But here's the problem: the public key is, like the name implies, usually public. The attacker may get his or her hands on it, and that should be OK. But what if the attacker were to create a new token using the following scheme. First, the attacker modifies the header and chooses `HS256` as the signing algorithm:

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

Then he or she escalates permissions by changing the `role` claim in the payload:

```
payload: {
  "sub": "joe",
  "role": "admin"
}
```

Now, here's the attack: the attacker proceeds to create a newly encoded JWT by using the public key, which is a simple string, as the HS256 shared secret! In other words, since the shared secret for HS256 can be any string, even a string like the public key for the RS256 algorithm can be used for that.

Now if we go back to our hypothetical use of the `jwtDecode` function from before:

```
const publicKey = '...';
const decoded = jwtDecode(token, publicKey);
```

We can now clearly see the problem, the token will be considered valid! The public key will get passed to the `jwtDecode` function as the second argument, but rather than being used as a public key for the RS256 algorithm, it will be used as a shared secret for the HS256 algorithm. This is caused by the `jwtDecode` function relying on the `alg` claim from the header to pick the verification algorithm for the JWT. And the attacker changed that:

```
header: {
  "alg": "HS256", // <-- changed by the attacker from RS256
  "typ": "JWT"
}
```

Just like in the `"alg": "none"` case, relying on the `alg` claim combined with a bad or confusing API can result in a successful attack by a malicious user.

Mitigations against this attack include passing an explicit algorithm to the `jwtDecode` function, checking the `alg` claim, or using APIs that separate public-key algorithms from shared secret algorithms.

8.1.3 Weak HMAC Keys

HMAC algorithms rely on a shared secret to produce and verify signatures. Some people assume that shared secrets are similar to passwords, and in a sense, they are: they should be kept secret. However, that is where the similarities end. For passwords, although the length is an important property, the minimum required length is relatively small compared to other types of secrets. This is a consequence of the hashing algorithms that are used to store passwords (along with a salt) that prevent brute force attacks in reasonable timeframes.

On the other hand, HMAC shared secrets, as used by JWTs, are optimized for speed. This allows many sign/verify operations to be performed efficiently but make brute force attacks easier[8]. So, the length of the shared secret for HS256/384/512 is of the utmost importance. In fact, JSON Web Algorithms[9] defines the minimum key length to be equal to the size in bits of the hash function used along with the HMAC algorithm:

> "A key of the same size as the hash output (for instance, 256 bits for"HS256") or larger MUST be used with this algorithm." - JSON Web Algorithms (RFC 7518), 3.2 HMAC with SHA-2 Functions[10]

In other words, many passwords that could be used in other contexts are simply not good enough for use with HMAC-signed JWTs. 256-bits equals 32 ASCII characters, so if you are using something human readable, consider that number to be the minimum number of characters to include in the secret. Another good option is to switch to RS256 or other public-key algorithms, which are much more robust and flexible. This is not simply a hypothetical attack, it has been shown that brute force attacks for HS256 are simple enough to perform[11] if the shared secret is too short.

8.1.4 Wrong Stacked Encryption + Signature Verification Assumptions

Signatures provide protection against tampering. That is, although they don't protect data from being readable, they make it immutable: any changes to the data result in an invalid signature. Encryption, on the other hand, makes data unreadable unless you know the shared key or private key.

For many applications, signatures are all that is necessary. However, for sensitive data, encryption may be required. JWTs support both: signatures and encryption.

It is very common to wrongly assume that encryption also provides protection against tampering in all cases. The rationale for this assumption is usually something like this: "if the data can't be read, how would an attacker be able to modify it for their benefit?". Unfortunately, this underestimates attackers and their knowledge of the algorithms involved in the process.

Some encryption/decryption algorithms produce output regardless of the validity of the data passed to them. In other words, even if the encrypted data was modified, something will come out of the decryption process. Blindly modifying data usually results in garbage as output, but to a malicious attacker this may be enough to get access to a system. For example, consider a JWT payload that looks like this:

```
{
  "sub": "joe",
  "admin": false
}
```

As we can see here, the `admin` claim is simply a boolean. If an attacker can manage to produce a change in the decrypted data that results in that boolean value being flipped, he or she may

[8] https://auth0.com/blog/brute-forcing-hs256-is-possible-the-importance-of-using-strong-keys-to-sign-jwts/
[9] https://tools.ietf.org/html/rfc7518
[10] https://tools.ietf.org/html/rfc7518#section-3.2
[11] https://auth0.com/blog/brute-forcing-hs256-is-possible-the-importance-of-using-strong-keys-to-sign-jwts/

successfully execute an escalation of privileges attack. In particular, attackers that have ample time windows to perform attacks can try as many changes to encrypted data as they like, without having the system discard the token as invalid before processing it. Other attacks may involve feeding invalid data to subsystems that expect data to be already sanitized at that point, triggering bugs, failures, or serving as the entry point for other types of attacks.

For this reason, JSON Web Algorithms[12] only defines encryption algorithms that also include data integrity verification. In other words, as long as the encryption algorithm is one of the algorithms sanctioned by JWA[13], it may not be necessary for your application to stack an encrypted JWT on top of a signed JWT. However, if you encrypt a JWT using a non-standard algorithm, you must either make sure that data integrity is provided by that algorithm, or you will need to nest JWTs, using a signed JWT as the innermost JWT to ensure data integrity.

Nested JWTs[14] are explicitly defined and supported by the specification. Although unusual, they may also appear in other scenarios, like sending a token issued by some party through a third party system that also uses JWTs.

A common mistake in these scenarios is related to the validation of the nested JWT. To make sure that data integrity is preserved, and that data is properly decoded, all layers of JWTs must pass all validations related to the algorithms defined in their headers. In other words, even if the outermost JWT can be decrypted and validated, it is also necessary to validate (or decrypt) all the innermost JWTs. Failing to do so, especially in the case of an outermost encrypted JWT carrying an innermost signed JWT, can result in the use of unverified data, with all the associated security issues related to that.

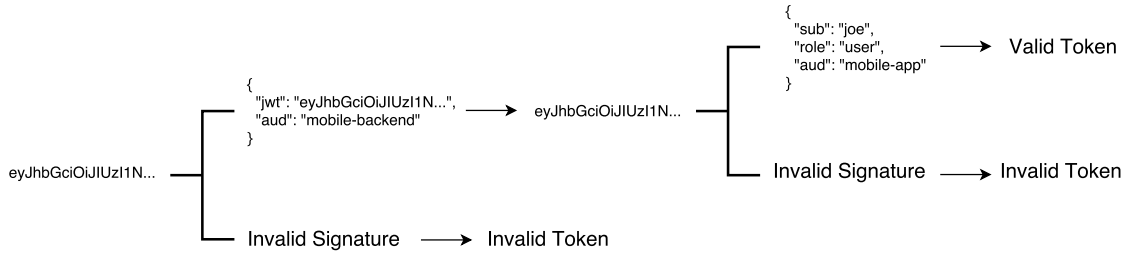

Figure 8.1: Validation of Nested JWT

8.1.5 Invalid Elliptic-Curve Attacks

Elliptic-curve cryptography is one of the public-key algorithm families supported by JSON Web Algorithms[15]. Elliptic-curve cryptography relies on the intractability of the elliptic-curve discrete logarithm problem, a mathematical problem that cannot be solved in reasonable times for big

[12] https://tools.ietf.org/html/rfc7518
[13] https://tools.ietf.org/html/rfc7518
[14] https://tools.ietf.org/html/rfc7519#section-2
[15] https://tools.ietf.org/html/rfc7518

enough numbers. This problem prevents the recovery of the private key from a public key, an encrypted message, and its plaintext. When compared to RSA, another public-key algorithm which is also supported by JSON Web Algorithms, elliptic-curves provide a similar level of strength while requiring smaller keys.

Elliptic-curves, as required for cryptographic operations, are defined over finite fields. In other words, they operate on sets of discrete numbers (rather than all real numbers). This means that all numbers involved in cryptographic elliptic-curve operations are integers.

All mathematical operations of elliptic-curves result in valid points over the curve. In other words, the math for elliptic-curves is defined in such a way that invalid points are simply not possible. If an invalid point is produced, then there is an error in the inputs to the operations. The main arithmetic operations on elliptic curves are:

- **Point addition**: adding two points on the same curve resulting in a third point on the same curve.
- **Point doubling**: adding a point to itself, resulting in a new point on the same curve.
- **Scalar multiplication**: multiplying a single point on the curve by a scalar number, defined as repeatedly adding that number to itself `k` times (where `k` is the scalar value).

All cryptographic operations on elliptic-curves rely on these arithmetic operations. Some implementations, however, fail to validate the inputs to them. In elliptic-curve cryptography, the public key is a point on the elliptic curve, while the private key is simply a number that sits within a special, but very big, range. If inputs to these operations are not validated, the arithmetic operations may produce seemingly valid results even when they are not. These results, when used in the context of cryptographic operations such as decryption, can be used to recover the private key. This attack has been demonstrated in the past[16]. This class of attacks are known as invalid curve attacks[17]. Good-quality implementations always check that all inputs passed to any public function are valid. This includes verifying that public-keys are a valid elliptic-curve point for the chosen curve and that private keys sit inside the valid range of values.

8.1.6 Substitution Attacks

Substitution attacks are a class of attacks where an attacker manages to intercept at least two different tokens. The attacker then manages to use one or both of these tokens for purposes other than the one they were intended for.

There are two types of substitution attacks: same recipient (called cross JWT in the draft), and different recipient.

8.1.6.1 Different Recipient

Different recipient attacks work by sending a token intended for one recipient to a different recipient. Let's say there is an authorization server that issues tokens for a third party service. The authorization token is a signed JWT with the following payload:

[16] http://blogs.adobe.com/security/2017/03/critical-vulnerability-uncovered-in-json-encryption.html
[17] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.3920&rep=rep1&type=pdf

```
{
  "sub": "joe",
  "role": "admin"
}
```

This token can be used against an API to perform authenticated operations. Furthermore, at least when it comes to this service, the user `joe` has administrator level privileges. However, there is a problem with this token: there is no intended recipient or even an issuer in it. What would happen if a different API, different from the intended recipient this token was issued for, used the signature as the only check for validity? Let's say there's also a user `joe` in the database for that service or API. The attacker could send this same token to that other service and instantly gain administrator privileges!

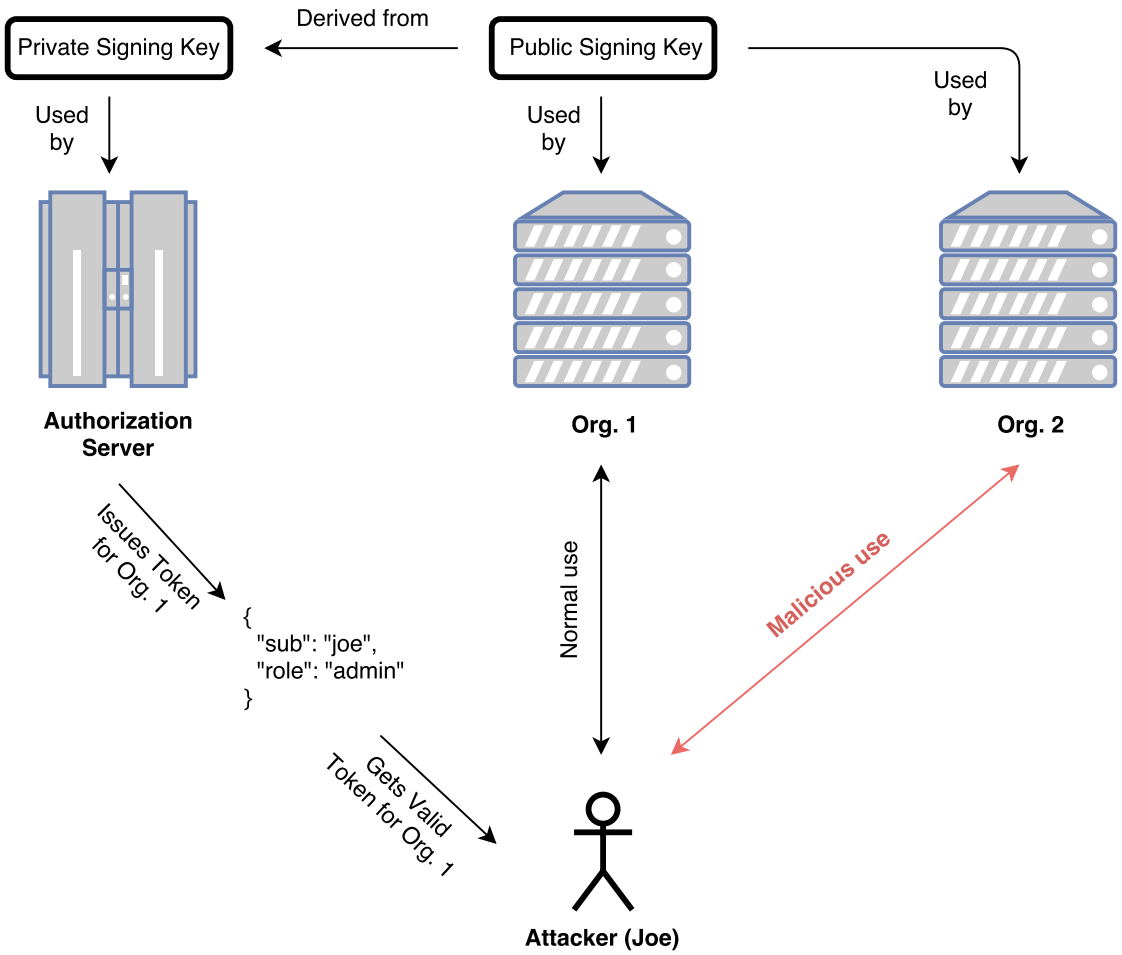

Figure 8.2: Different Recipient JWT Substitution Attack

To prevent these attacks, token validation must rely on either unique, per-service keys or secrets, or specific claims. For instance, this token could include an `aud` claim specifying the intended audience. This way, even if the signature is valid, the token cannot be used on other services that share the same secret or signing key.

8.1.6.2 Same Recipient/Cross JWT

This attack is similar to the previous one, but rather than relying on a token issued for a different recipient, in this case, the recipient is the same. What changes in this case is that the attacker sends the token to a different service rather the one intended for (inside the same company or service provider).

Let's imagine a token with the following payload:

```
{
  "sub": "joe",
  "perms": "write",
  "aud": "cool-company/user-database",
  "iss": "cool-company"
}
```

This token looks much more secure. We have an issuer (`iss`) claim, an audience (`aud`) claim, and a permissions (`perm`) claim. The API for which this token was issued checks all of these claims even if the signature of the token is valid. This way, even if the attacker manages to get his or her hands on a token signed with the same private key or secret, he or she cannot use it to operate on this service if it's not intended for it.

However, `cool-company` has other public services. One of these services, the `cool-company/item-database` service, has recently been upgraded to check claims along with the token signature. However, during the upgrades, the team in charge of selecting the claims that would be validated made a mistake: they did not validate the `aud` claim correctly. Rather than checking for an exact match, they decided to check for the presence of the `cool-company` string. It turns out that the other service, the hypothetical `cool-company/user-database` service, emits tokens that also pass this check. In other words, an attacker could use the token intended for the `user-database` service in place for the token for the `item-database` service. This would grant the attacker write permissions to the item database when he or she should only have write permissions for the user database!

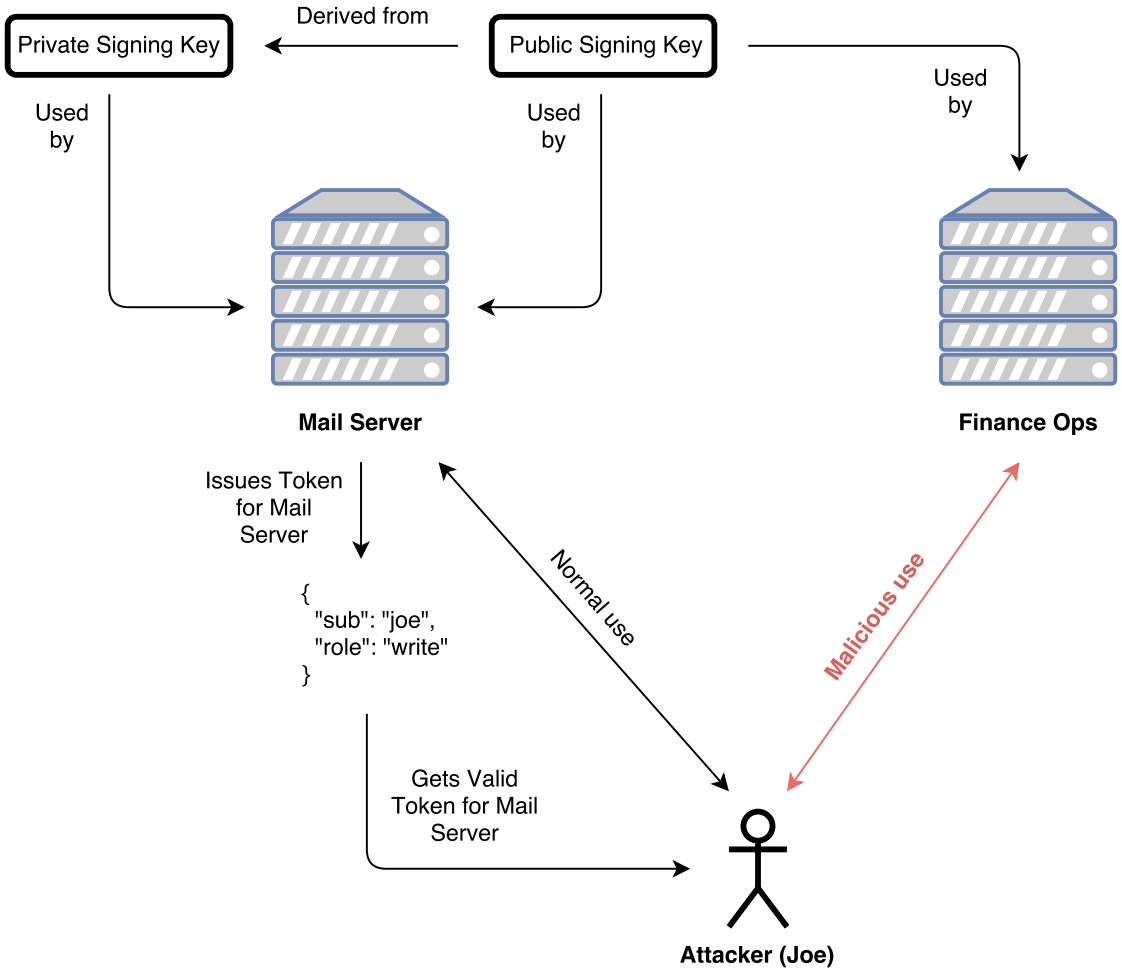

Figure 8.3: Same Recipient JWT Substitution Attack

8.2 Mitigations and Best Practices

We've had a look at common attacks using JWTs, now let's take a look at the current list of best practices. All of these attacks can be successfully prevented by following these recommendations.

8.2.1 Always Perform Algorithm Verification

The `"alg": "none"` attack and the "RS256 public-key as HS256 shared secret" attack can be prevented by this mitigation. Every time a JWT is to be validated, the algorithm must be explicitly

selected to prevent giving attackers control. Libraries used to rely on the header `alg` claim to select the algorithm for validation. From the moment attacks like these were seen in the wild[18], libraries have switched to at least providing the option of explicitly specifying the selected algorithms for validation, disregarding what is specified in the header. Still, some libraries provide the option of using whatever is specified in the header, so developers must take care to always use explicit algorithm selection.

8.2.2 Use Appropriate Algorithms

Although the JSON Web Algorithms spec declares a series of recommended and required algorithms, picking the right one for a specific scenario is still up to the users. For example, a JWT signed with an HMAC signature may be enough for storing a small token from your single-server, single-page web application in a user's browser. In contrast, a shared secret algorithm would be sorely inconvenient in a federated identity scenario.

Another way of thinking about this is to consider all JWTs invalid unless that validation algorithm is acceptable to the application. In other words, even if the validating party has the keys and the means necessary for validating a token, it should still be considered invalid if the validation algorithm is not the right one for the application. This is also another way of saying what we mentioned in our previous recommendation: always perform algorithm verification.

8.2.3 Always Perform All Validations

In the case of nested tokens, it is necessary to always perform all validation steps as declared in the headers of each token. In other words, it is not sufficient to decrypt or validate the outermost token and then skip validation for the inner ones. Even in the case of only having signed JWTs, it is necessary to validate all signatures. This is a source of common mistakes in applications that use JWTs to carry other JWTs issued by external parties.

8.2.4 Always Validate Cryptographic Inputs

As we have shown in the attacks section before, certain cryptographic operations are not well defined for inputs outside their range of operation. These invalid inputs can be exploited to produce unexpected results, or to extract sensitive information that may lead to a full compromise (i.e. the attackers getting hold of a private key).

In the case of elliptic-curve operations, our example from before, libraries must always validate public-keys before using them (i.e. confirming they represent a valid point on the selected curve). These types of checks are normally handled by the underlying cryptographic library. Developers must make sure that their library of choice performs these validations, or they must add the necessary code to perform them at the application level. Failing to do so can result in compromise of their private key(s).

[18] https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

8.2.5 Pick Strong Keys

Although this recommendation applies to any cryptographic key, it is still ignored many times. As we have shown above, the minimum necessary length for HMAC shared secrets is often overlooked. But even if the shared secret were long enough, it must also be fully random. A long key with a bad level of randomness (a.k.a. "entropy") can still be brute-forced or guessed. To ensure this is not the case, key generating libraries should rely on cryptographic-quality pseudo-random number generators (PRNGs) properly seeded during initialization. At best, a hardware number generator may be used.

This recommendation applies to both shared-key algorithms and public-key algorithms. Furthermore, in the case of shared key algorithms, human-readable passwords are not considered good enough and are vulnerable to dictionary attacks.

8.2.6 Validate All Possible Claims

Some of the attacks we have discussed rely on incorrect validation assumptions. In particular, they rely on signature validation or decryption as the only means of validation. Some attackers may get access to correctly signed or encrypted tokens that can be used for malicious purposes, usually by using them in unexpected contexts. The right way to prevent these attacks is to only consider a token valid when both the signature and the content are valid. For this reason, claims such as `sub` (subject), `exp` (expiration time), `iat` (issued at), `aud` (audience), `iss` (issuer), `nbf` (not valid before) are of the utmost importance and should always be validated when present. If you are creating tokens, consider adding as many claims as necessary to prevent its use in different contexts. In general, `sub`, `iss`, `aud`, and `exp` are always useful and should be present.

8.2.7 Use The `typ` Claim To Separate Types Of Tokens

Although most of the time the `typ` claim has a single value (`JWT`), it can also be used to separate different types of application specific JWTs. This can be useful in case your system must handle many different types of tokens. This claim can also prevent misuse of a token in a different context by means of an additional claim check. The JWS standard explicitly allows for application-specific values of the `typ` claim.

8.2.8 Use Different Validation Rules For Each Token

This practice sums up many of the ones that have been enumerated before. To prevent attacks it is of key importance to make sure each token that is issued has very clear and specific validation rules. This not only means using the `typ` claim when appropriate, or validating all possible claims such as `iss` or `aud`, but it also implies avoiding key reuse for different tokens where possible or using different custom claims or claim formats. This way, tokens that are meant to be used in a single place cannot be substituted by other tokens with very similar requirements.

In other words, rather than using the same private key for signing all kinds of tokens, consider using different private keys for each subsystem of your architecture. You can also make claims

more specific by specifying a certain internal format for them. The `iss` claim, for instance, could be an URL of the subsystem that issued that token, rather than the name of the company, making it harder to be reused.

8.3 Conclusion

JSON Web Tokens are a tool that makes use of cryptography. Like all tools that do so, and especially those that are used to handle sensitive information, they should be used with care. Their apparent simplicity may confuse some developers and make them think that using JWTs is just a matter of picking the right shared secret or public key algorithm. Unfortunately, as we have seen above, that is not the case. It is of the utmost importance to follow the best practices for each tool in your toolbox, and JWTs are no exception. This includes picking battle-tested, high-quality libraries; validating payload and header claims; choosing the right algorithms; making sure strong keys are generated; paying attention to the subtleties of each API; among other things. If all of this seems daunting, consider offloading some of the burdens to external providers. Auth0[19] is one such provider. If you cannot do this, consider these recommendations carefully, and remember: don't roll your own crypto[20], rely on tried and tested code.

[19] https://auth0.com
[20] https://security.stackexchange.com/questions/18197/why-shouldnt-we-roll-our-own