The JWT Handbook

Sebastián E. Peyrott, Auth0 Inc.

Version 0.9.1, 2016

Abstract

An introduction to the wonders of JSON Web Tokens and associated technologies.

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

[6] https://tools.ietf.org/html/rfc7515
[7] https://tools.ietf.org/html/rfc7516
[8] https://en.wikipedia.org/wiki/Security_Assertion_Markup_Language

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 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].

[9] https://datatracker.ietf.org/wg/jose/history/
[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.

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

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 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.

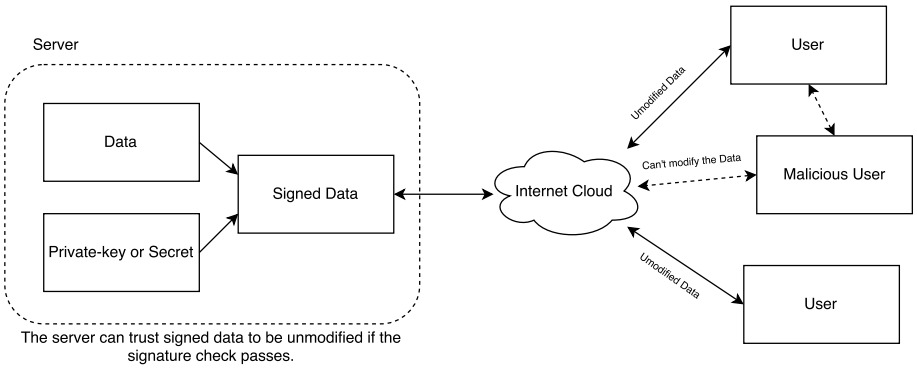

Figure 2.1: Client-side Signed Data

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.

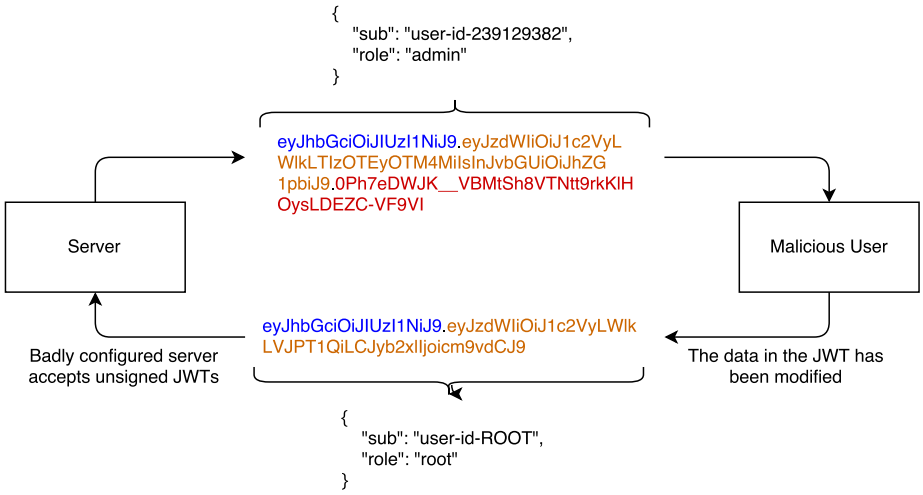

Figure 2.2: Signature Stripping

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.

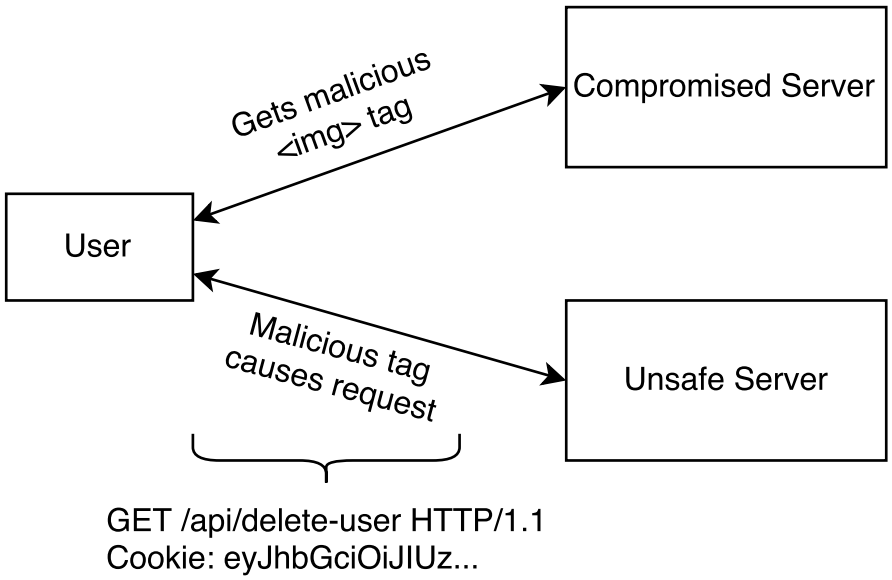

Figure 2.3: Cross-Site Request Forgery

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.

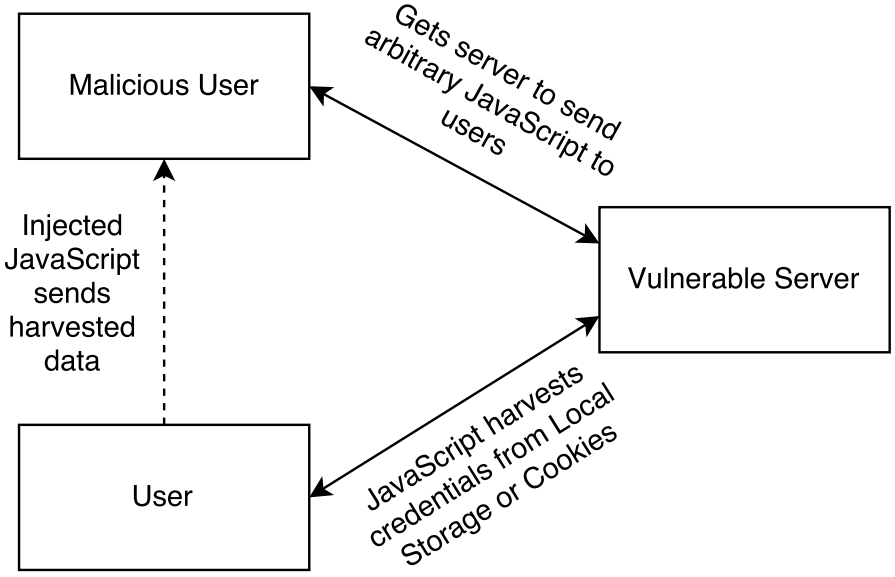

Figure 2.4: Persistent Cross Site Scripting

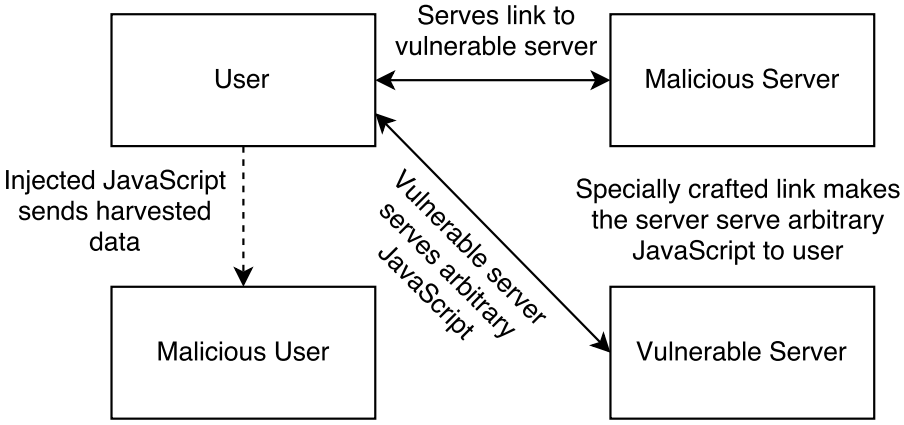

Figure 2.5: Reflective Cross Site Scripting

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. The cart will be stored inside the JWT used for authentication. This JWT, in turn, will be provided by the Auth0 authorization server.

The application carries all shopping cart items in the session, which decoded looks as follows:

```
{
  "name": "Sebastian Peyrott",
  "email": "sebastian.peyrott@auth0.com",
  "email_verified": true,
  "iss": "https://speyrott.auth0.com/",
  "sub": "google-oauth2|11111111111111111",
  "aud": "t42WY87weXzepAdUlwMiHYRBQj9qWVAT",
  "exp": 1474953988,
  "iat": 1474917988,
  "items": [
    "iphone7",
    "macbook pro",
    "airbud"
```

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

```
  ]
}
```

To render the items in the cart, the frontend only needs to retrieve it from local storage:

```
var token = localStorage.getItem('id_token');
if(token) {
  var decoded = jwt_decode(token);
  var body = $('#items');
  decoded.items.forEach(function(item) {
    body.append('<li>' + item + '</li>');
  });
}
```

Whenever an item is added to the cart, the JWT is checked for validity in the backend (which means the user is authenticated):

```
var authenticate = jwt({
  secret: new Buffer(process.env.AUTH0_CLIENT_SECRET, 'base64'),
  audience: process.env.AUTH0_CLIENT_ID
});

// (...)

app.use('/secured', authenticate);
```

The `/secured/add-item` route handles adding items to the cart. Since the session is stored client side, simply adding it to the JWT is enough (backend code):

```
router.post('/secured/add-item', function(req, res, next) {
  var token = getToken(req);
  if(!token) {
    res.sendStatus(500);
    return;
  }

  var decoded = jwt.decode(token, { complete: true });
  if(!decoded.payload.items) {
    decoded.payload.items = [];
  }
  decoded.payload.items.push(req.body.item);
  var encoded = jwt.sign(
    decoded.payload,
    new Buffer(process.env.AUTH0_CLIENT_SECRET, 'base64'),
    { header: decoded.header });

  res.json({
```

```
    'id_token': encoded
  });
});
```

The frontend must, in turn, update the token stored in local storage (if the token were stored as a cookie, this would not be necessary):

```
$('form').submit(function(event) {
  $.ajax({
    type: 'POST',
    url: '/secured/add-item',
    data: $('form').serialize(),
    success: function(data) {
      localStorage.setItem('id_token', data.id_token);
    }
  });
  event.preventDefault();
});
```

Implementing XSS mitigation techniques and server side validation of the items added 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[4] 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[5] and OpenID[6] 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:

[4] https://auth0.com/blog/2015/09/23/what-is-and-how-does-single-sign-on-work/

[5]

[6]

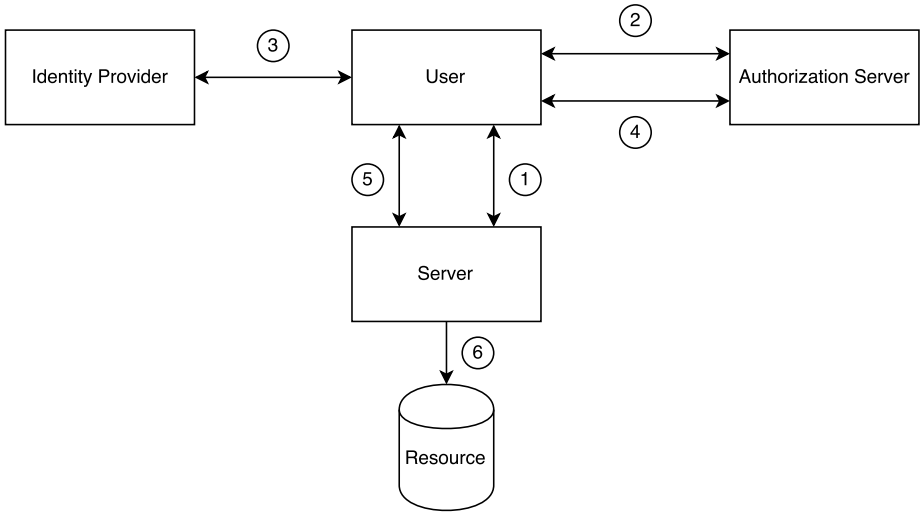

Figure 2.6: Common Federated Identity Flow

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[7]. 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.

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

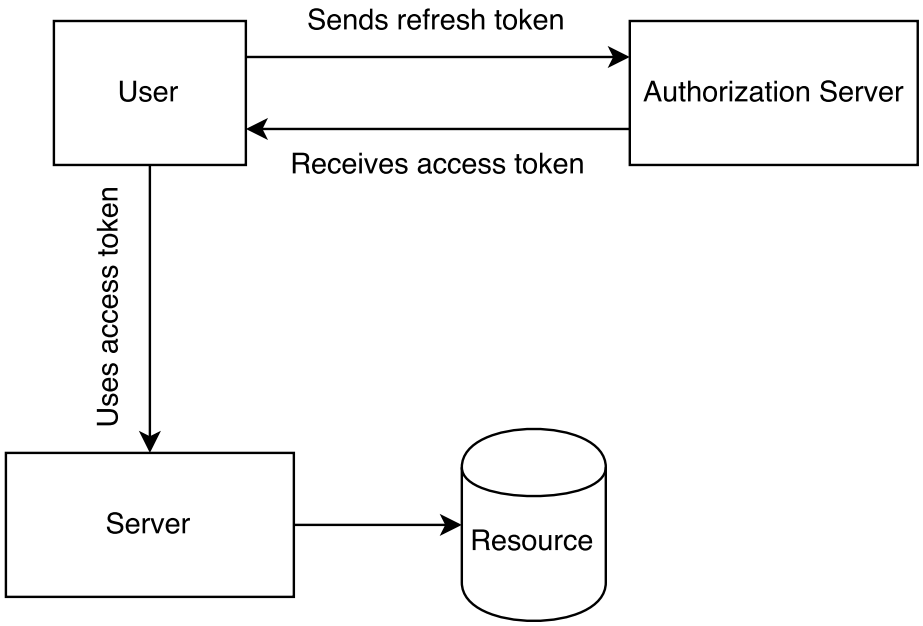

Figure 2.7: Refresh and access tokens

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 an ideal match for its requirements. Signed JWTs make an ideal match for 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).

TODO: show which endpoints could return JWTs.

2.2.3 JWTs and OpenID Connect

OpenID Connect[8] 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[9] 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

[8] https://openid.net/connect/
[9] http://openid.net/specs/openid-connect-core-1_0.html#IDToken

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[10] as the authorization server. Auth0 allows for different identity providers to be set dinamically. 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 can be quite simple. We set up the Auth0 login screen (called Lock[11]) 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).

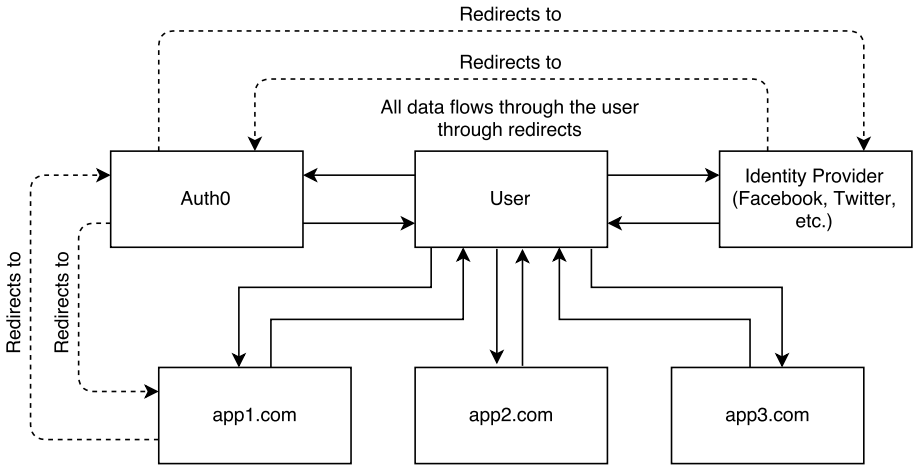

Figure 2.8: Auth0 as Authorization Server

2.2.4.1 Setting up Auth0 Lock for Node.js Applications

Setting up the Auth0 Lock[12] library for single page apps can be done as follows:

```
// instantiate Lock
var lock = new Auth0Lock('ye0F16vzCTBX5yTejqEfc18wEWIOwJWI',
                         'speyrott.auth0.com', {
    auth: {
        redirectUrl: 'http://app1.com:3000/',
```

[10] https://auth0.com
[11] https://auth0.com/docs/libraries/lock
[12] https://auth0.com/docs/sso/single-page-apps-sso

```
        responseType: 'token',
        sso: true,
        params: {
            // Learn about scopes: https://auth0.com/docs/scopes
            scope: 'openid name email'
        }
    }
});

// Listening for the authenticated event
lock.on("authenticated", function (authResult) {
    localStorage.setItem('idToken', authResult.idToken);
    goToHomepage(getQueryParameter('targetUrl'), authResult.idToken);
});
```

`goToHomepage` is a user defined function that performs the steps necessary after a successful login. The `idToken` returned by Auth0 Lock is in fact a signed JWT. This JWT carries the requested information according to the `scopes` parameter used to instantiate `Auth0Lock`. By validating this token, any server that knows the client secret can be sure the user is who he says he is.

In this case, JWTs are used as the transport format for secure user identification. Single-sign-on is handled by passing this information between the client, the authorization server and the identity provider in a secure manner.

Although this is enough to handle common logins, single-sign-on is a bit more complex. The following code is also necessary for single page apps:

```
// Get the user token if we've saved it in localStorage before
var idToken = localStorage.getItem('idToken');
if (idToken) {
    // This would go to a different route like
    // window.location.href = '#home';
    // But in this case, we just hide and show things
    goToHomepage(getQueryParameter('targetUrl'), idToken);
    return;
} else {
    client.getSSOData(function (err, data) {
        if (!err && data.sso) {
                // there is! redirect to Auth0 for SSO
                client.signin({
                responseType: 'token',
                scope: 'openid name email'
            }, function (err, profile, idToken) {
                if (!err) {
                    localStorage.setItem('idToken', idToken);
                    goToHomepage('', idToken);
                }
```

```
            });
        }
    });
}
```

This code checks with the Auth0 authorization server whether all conditions for single-sign-on are met. If they are, direct sign in is possible and performed. To enable single-sign-on additional condiguration steps are required for each identity provider (such as Google). Refer to the Auth0 docs[13] on how to perform this. You will also need to set the right client id and secret key[14] for each identity provider connection.

Regular multi-page apps require a series of different steps[15]. You can see these steps implemented for `app3.com` in the samples directory.

When you run this example, try to login to any of the apps, for example, `app1.com`. After a successful login, attempt to access `app2.com`. After the initial load, you will see the application login automatically.

App 3 requires the additional step of setting the secret Auth0 API key in the `.env` file in its own directory. Failure to do this will not allow this application to run. Create you own Auth0 account and then use your own Auth0 domain, client id and client secret in all of these examples to see them fully functional.

The full code for this example is located under `samples/single-sign-on-federated-identity`. See the `README` for instructions on how to build it. Implementing XSS mitigation techniques is left as en exercise for the reader.

[13] https://auth0.com/docs/sso
[14] https://auth0.com/docs/connections/social/google
[15] https://auth0.com/docs/sso/regular-web-apps-sso

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.
```

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

```
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:

```
{
  "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`.

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

- **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:

```
{
    "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 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.

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

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

- **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.

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

[8] https://tools.ietf.org/html/rfc7519#section-10.1

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.

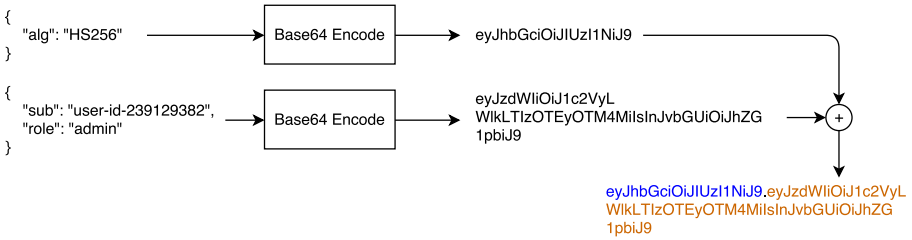

Figure 3.1: Compact Unsecured JWT Generation

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].

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

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.

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.

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

- 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.

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}`;
```

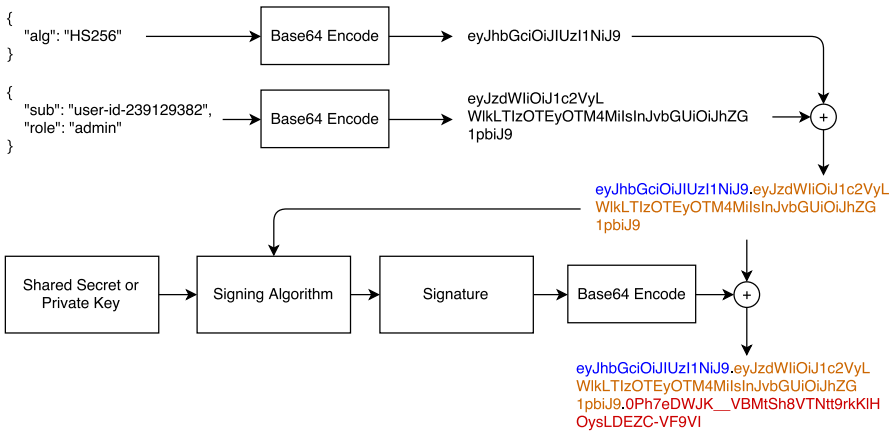

Figure 4.1: JWS Compact Serialization

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 key. In this specific variation of the

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

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.

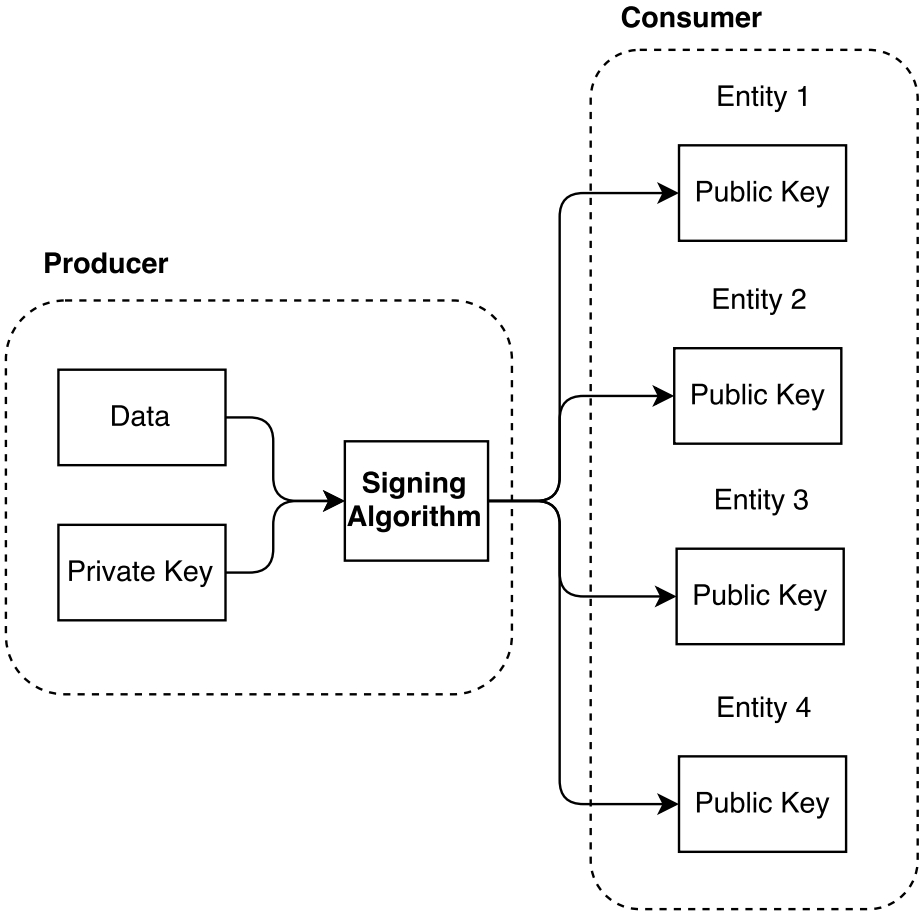

Figure 4.2: One-to-many signing

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>
```

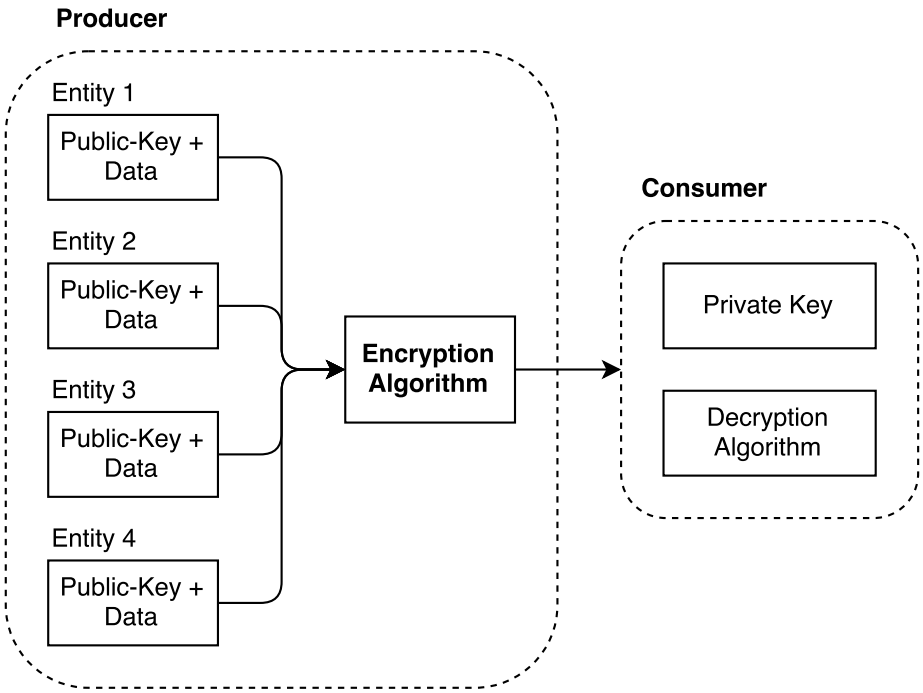

Figure 4.3: Many-to-one encryption

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).

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

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*.

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
              h0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
  "signatures": [
    {
      "protected": "eyJhbGciOiJSUzI1NiJ9",
```

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

```
      "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" },
      "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

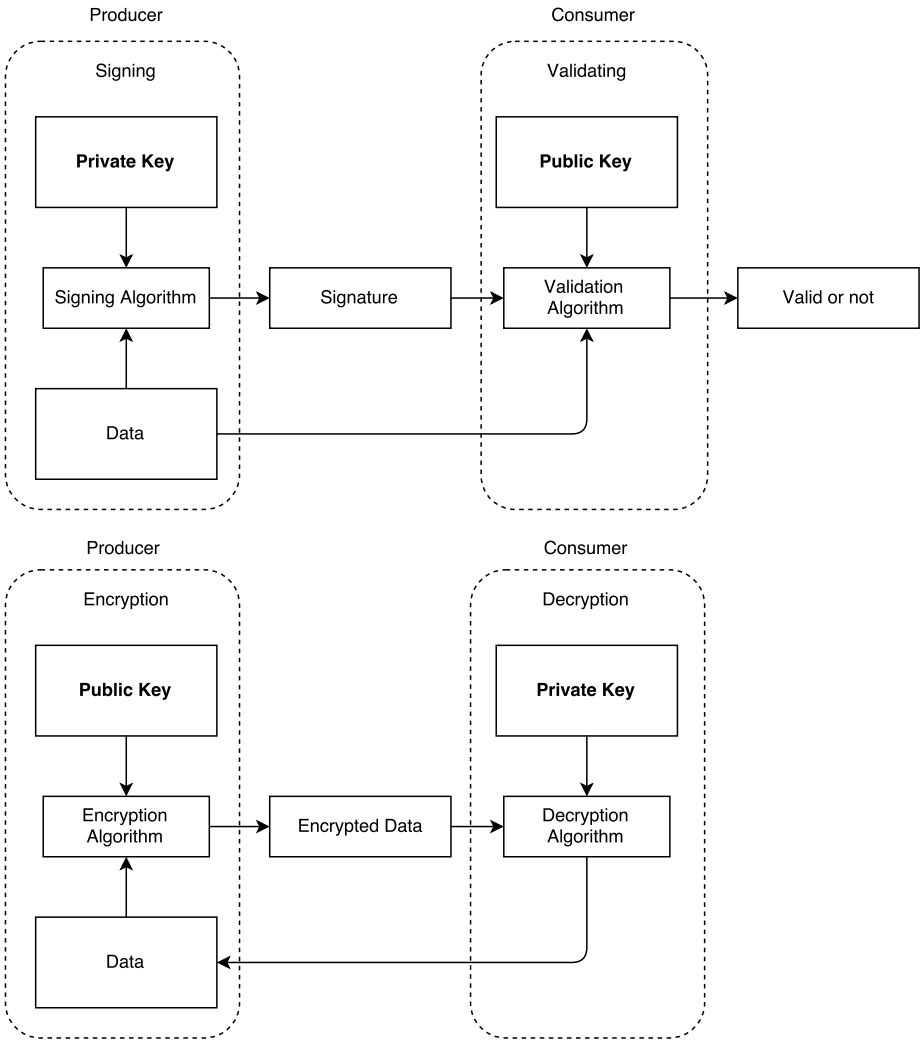

Figure 5.1: Signing vs encryption using public-key cryptography

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.
3. **The initialization vector:** some encryption algorithms require additional (usually random) data.

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

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.

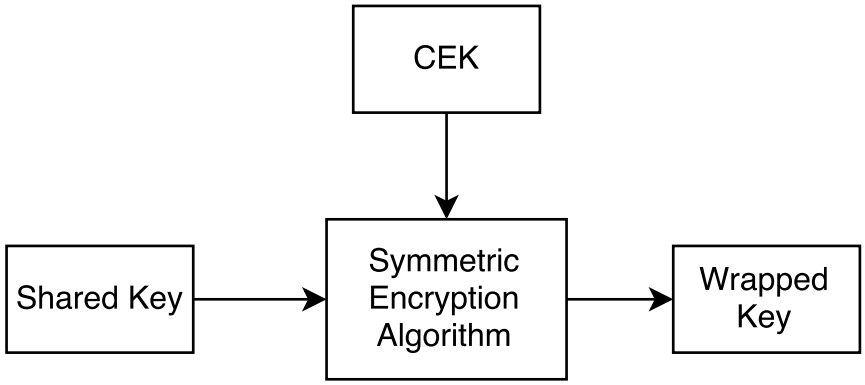

Figure 5.2: Key wrapping

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

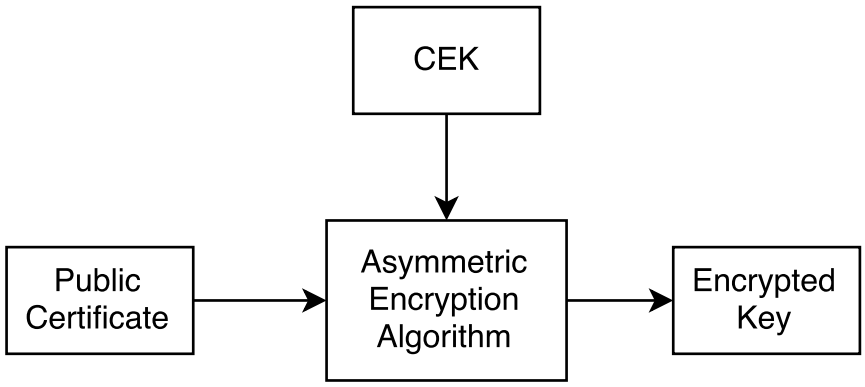

Figure 5.3: Key encryption

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

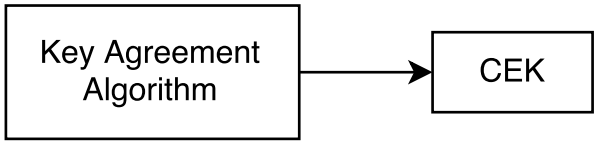

Figure 5.4: Direct key agreement

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

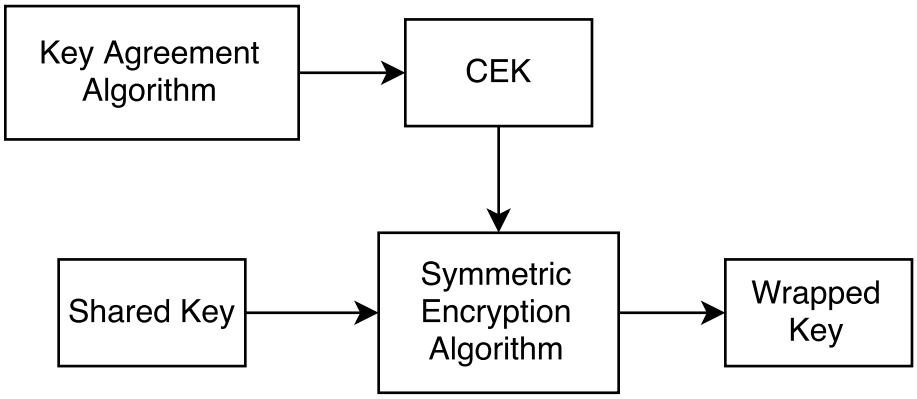

Figure 5.5: Direct key agreement

- **Direct Encryption**: a user-defined symmetric shared key is used as the

CEK (no key derivation or generation).

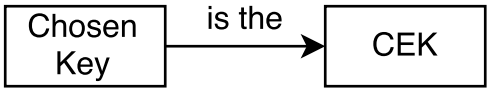

Figure 5.6: Direct key agreement

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.
- **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

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

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
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.

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

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();

// 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' }),
```

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

```
    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();
}
```

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

```
function rsa(compact) {
    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].

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
```

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

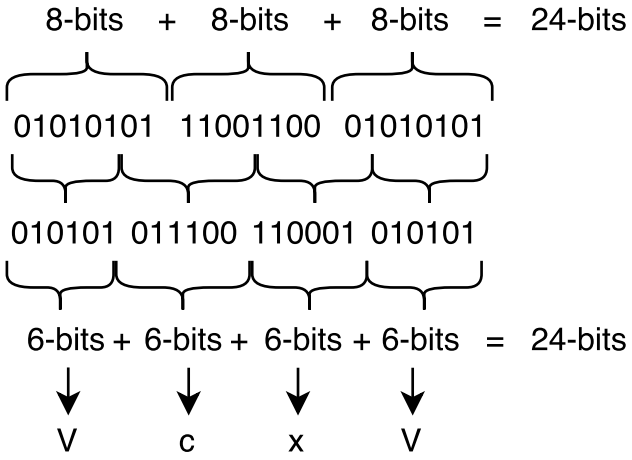

Figure 7.1: Base64 encoding

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:

```
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);
```

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

```
}

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
   ```

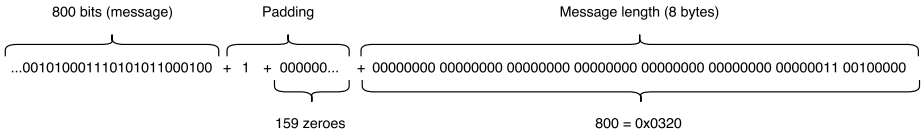

Figure 7.2: SHA padding

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) {
    return h_;
} 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*.

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

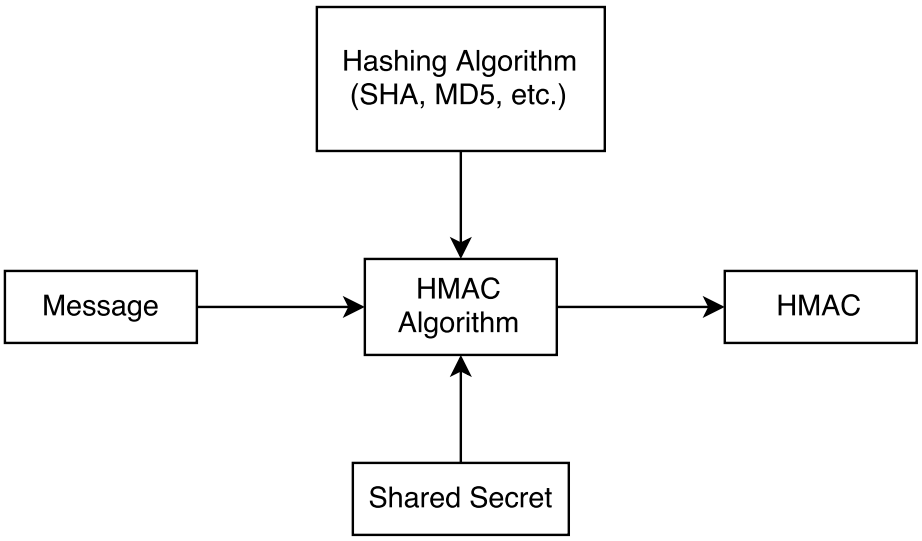

Figure 7.3: HMAC

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
      (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;
}
```

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:

```
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}`;
```

7.3 Future Updates

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

[5] https://jwt.io