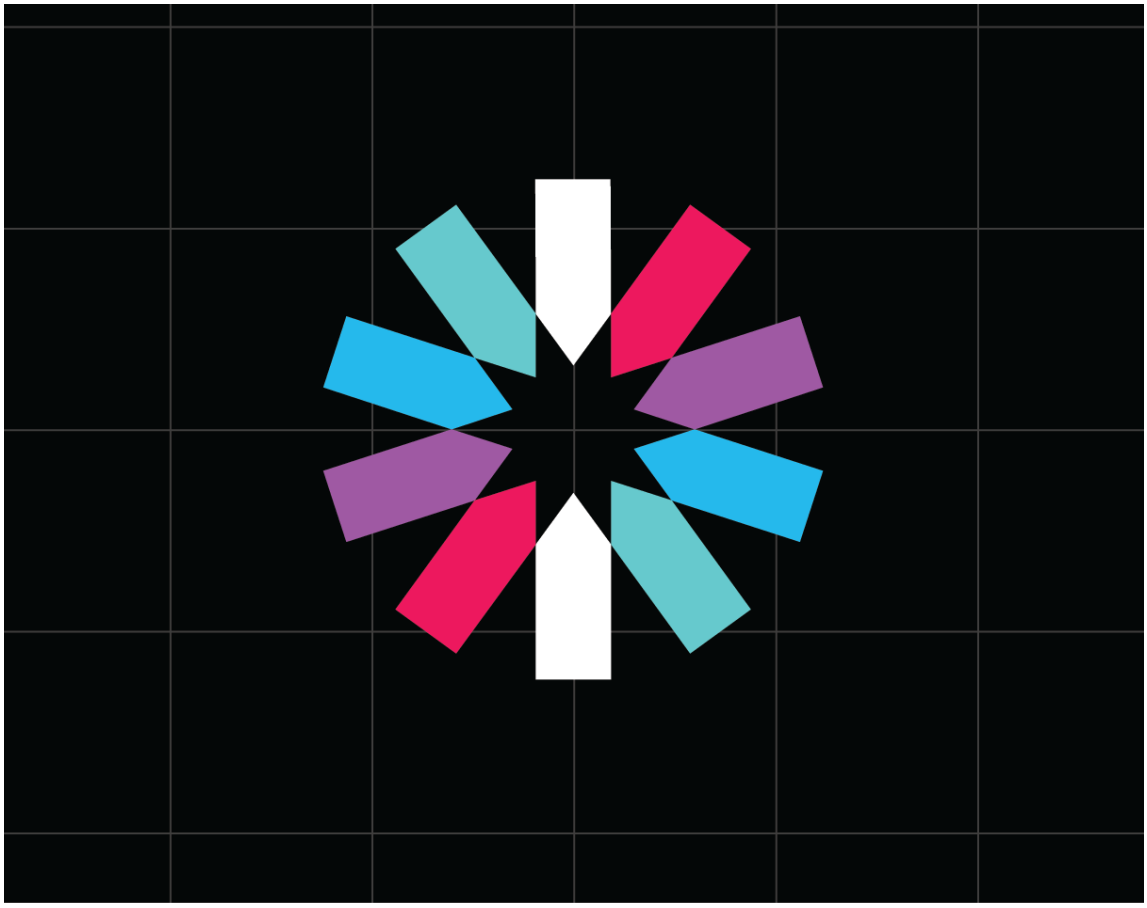

JWT ハンドブック

著者: Sebastian Peyrott

Auth0

JWT ハンドブック

Sebastián E. Peyrott、Auth0 Inc.

バージョン 0.14.1、2016〜2018

目次

謝辞

第 2 章のフェデレーション アイデンティティの例を提供してくれた **Prosper Otemuyiwa** 氏、本書のレビューを担当し、私の執筆時間を確保してくれた **Diego Poza** 氏、本書の難しい箇所をレビューしてくれた **Matías Woloski** 氏、私のリクエストに応じてくれた **Martín Gontovnikas** 氏、全体の体裁を整えてくれた **Bárbara Mercedes Muñoz Cruzado** 氏、本書の配布に関する前工程と後工程の業務を担当してくれた **Alejo Fernández** 氏と **Víctor Fernández** 氏、全力でチームメイトを支援してくれた **Sergio Fruto** 氏、プロジェクトを進行させながらも各関係者にヒアリングする時間を確保してくれた **Federico Jack** 氏に、心から感謝を申し上げます (順不同)。

第 1 章

はじめに

JSON Web Token (略称 JWT、「ジョット」と発音) は、スペースに制約のある環境でクレームを安全に渡すために利用される標準仕様です。JWT は多くの[1] 主要[2] Web[3] フレームワーク[4]で利用されています。このアーキテクチャの主な特徴は、シンプルかつコンパクトで使いやすいことです。JWT よりもはるかに複雑な方式[5] もまだ使用されていますが、JWT はさまざまな形で利用されています。本書では、JWT のバイナリ表現や JWT の構成に使用されるアルゴリズムなど、JWT のアーキテクチャの重要な特徴について説明するほか、JWT が業界で一般にどのように使用されているかを紹介します。

1.1 JSON Web Token とは

JSON Web Token (JWT) は次のようなものです (見やすいように改行を挿入しています)。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

この状態では何が書かれているのかさっぱりわかりませんが、JWT は非常に**コンパクト**かつ**表示可能**な形で一連のクレームを表現し、信頼性を検証するための**署名**も使用できるという特徴があります。

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

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

[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://ja.wikipedia.org/wiki/Security_Assertion_Markup_Language

}

クレームとは、特定の当事者またはオブジェクトに関する定義または表明です。クレームとその意味は、一部が JWT の仕様として定義されています。その他のクレームはユーザーにより定義されます。JWT が優れている点は、一般的な処理でよく使用されるクレームが標準化されていることです。一般的な処理とは、たとえば、特定の当事者のアイデンティティを確立することです。そのため JWT には、標準クレームの 1 つとして *sub* (「subject (主題)」を表す) クレームが用意されています。標準クレームについては、第 3 章で詳しく説明します。

JWT のもう 1 つの重要な特徴は、JSON Web Signatures (JWS、RFC7515[6]) を使用して署名したり、JSON Web Encryption (JWE、RFC7516[7]) を使用して暗号化したりできることです。JWT を JWS や JWE と組み合わせることで、さまざまな問題を解決する強力でセキュアなソリューションを実現できます。

1.2 JWT が解決する問題

JWT の主な用途は二者間でクレームをやり取りすることですが、この用途における JWT の最も重要な特徴は、検証や暗号化にも対応できるシンプルなコンテナー形式を利用した標準化の取り組みであることだと言えるでしょう。従来は、個人や組織がこの問題に対して個別に解決策を実施してきました。特定の当事者に関するクレームの設定には、従来型の標準[8] を使用することもできますが、JWT の登場により、シンプルかつ便利で標準的なコンテナー形式を利用できるようになりました。

ここまでの説明は少し抽象的でしたが、JWT の用途は、ログイン システムなど、容易に想像がつくものです。実際の利用例については、第 2 章で詳しく説明します。それ以外の利用例としては次のものが挙げられます。

- 認証
- 認可
- フェデレーション アイデンティティ
- クライアント側セッション ("ステートレス" セッション)
- クライアント側シークレット

1.3 JWT の簡単な歴史

2011 年、JSON Object Signing and Encryption (JOSE) グループが結成されました[9]。このグループの目的は、JSON を使用するプロトコルのセキュリティ サービスの相互運用性を実現するために、整合性保護 (署名および MAC) と暗号化のメカニズム、および鍵とアルゴリズム識別子の形式を標準化することでした。2013 年までに、グループが考案したアイデアのさまざまな使用例を示したクックブックなどを含め、一連の草稿が公開されました。これらの草稿は、後に JWT、JWS、JWE、JWK、JWA の RFC となりました。2016 年の時点では、これらの RFC はインターネット標準化過程にあります。また、正誤表は発行されていません。現在、グループは活動休止中です。

主な仕様の著者は、Mike Jones 氏[10]、Nat Sakimura 氏[11]、John Bradley 氏[12] Joe Hildebrand 氏[13]です。

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

第 2 章

実際の利用例

JWT の構造や構築について詳しく説明する前に、実際の利用例を見てみましょう。この章では、今日業界で使用されている一般的な JWT ベースのソリューションについて、その複雑さ (簡潔さ) に焦点を当てながら説明します。すべてのコードが公開リポジトリ[1] で提供されているため、自由に参照できます。以下の例は、本番環境を想定したものではありませんのでご注意ください。本番環境で使用するには、テスト ケース、ロギング、セキュリティ ベスト プラクティスが必要です。以下の例は説明用のものであり、要点だけを簡潔に示しています。

2.1 クライアント側セッション/ステートレス セッション

いわゆるステートレス セッションは、実際にはクライアント側のデータにすぎません。この利用例で重要なポイントは、セッションの内容を認証および保護するために署名を使用し、さらに多くの場合は暗号化も使用する点です。クライアント側のデータは改ざんされるおそれがあるため、バックエンドで慎重に処理する必要があります。

JWT では、JWS と JWE を利用することで、さまざまな種類の署名と暗号化を使用できます。署名を使用すると、データが改ざんされているかどうかを検証できます。暗号化を使用すると、第三者に読み取られないようにデータを保護することができます。

ほとんどの場合、セッションに必要なものは署名だけです。セッションに格納されているデータが第三者に読み取られても、セキュリティやプライバシーの問題が発生しないためです。たとえば *sub* クレーム (サブジェクト クレーム) は、通常は第三者に読み取られても安全なクレームです。サブジェクト クレームは一般的に、一方の当事者が他方の当事者を識別するために使用するものです (ユーザー ID やメール アドレスなどを思い浮かべてください)。このクレームは一意にする必要はありません。言い換えると、ユーザーを一意に識別するために別のクレームが必要になる場合があります。これは利用者が決めることができます。

ショッピング カートを表す items (アイテム) クレームなどは、第三者のアクセスを許すべきではないクレームです。ショッピング カートには、ユーザーが購入しようとしている、つまりユーザーのセッションに関連付けられているアイテムが含まれる可能性があります。暗号化されていない JWT にアイテムが保存されている場合、第三者 (クライアント側スクリプト) によってアイテムが収集される可能性があるため、プライバシーの問題が発生するおそれがあります。

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

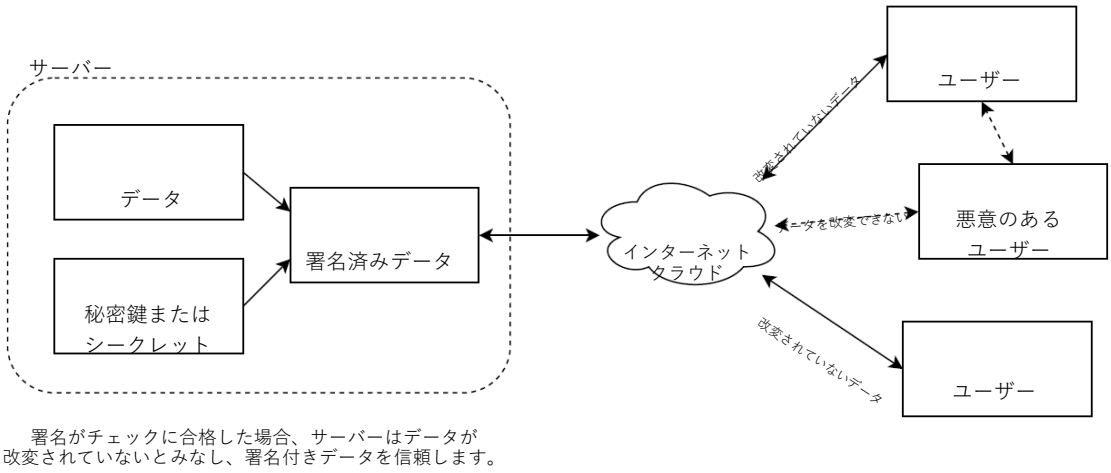

図 2.1: クライアント側の署名付きのデータ

2.1.1 セキュリティに関する考慮事項

2.1.1.1 署名の削除

署名付きの JWT に対する攻撃手法としてよく見られるのは、署名を単純に削除するというものです。署名付きの JWT は、ヘッダー、ペイロード、署名という 3 つの異なる要素で構成されます。3 つの部分は別々にエンコードされます。そのため、署名を削除してからヘッダーを改変することで、署名付きでない JWT であると宣言することが可能です。特定の JWT 検証ライブラリを不用意に使用すると、署名付きでないトークンが有効なトークンとみなされ、攻撃者の都合のいいようにペイロードが改変されるおそれがあります。この問題は、検証を実行するアプリケーションで、署名付きでない JWT を無効とみなすことで簡単に解決できます。

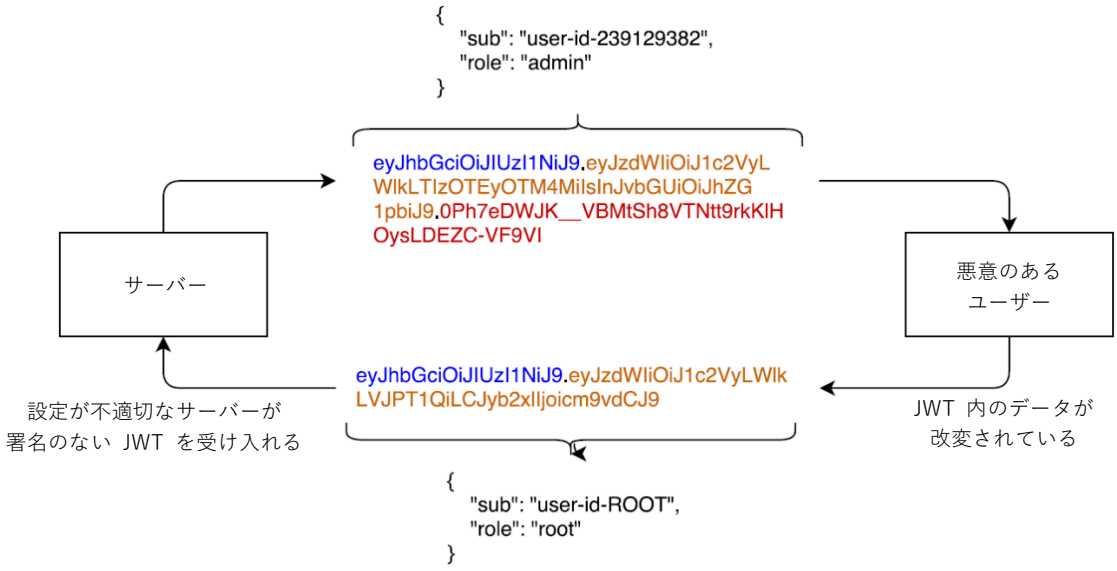

図 2.2: 署名の削除

2.1.1.2 クロスサイト リクエスト フォージェリ (CSRF)

クロスサイト リクエスト フォージェリとは、ユーザーのブラウザーを欺いて別のサイトから要求を送信させることに

より、ユーザーがログインしているサイトに対して要求を実行しようとする攻撃手法です。この攻撃を行うには、特別に細工したサイト (またはアイテム) に攻撃ターゲットへの URL を埋め込む必要があります。よく見られる例は、src 属性で攻撃ターゲットを指定した <img> タグを不正なページに埋め込む手法です。以下はその例です。

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

上記の <img> タグにより、このタグを含むページがロードされるたびに target.site.com に要求が送信されます。ユーザーが以前に target.site.com にログインしていて、このサイトが Cookie を使用してセッションをアクティブ状態に保っている場合は、その Cookie も送信されます。ターゲット サイトに CSRF 対策が実装されていない場合、この要求はユーザーからの有効な要求として処理されます。JWT は、他のクライアント側データと同様に、Cookie として保管することができます。

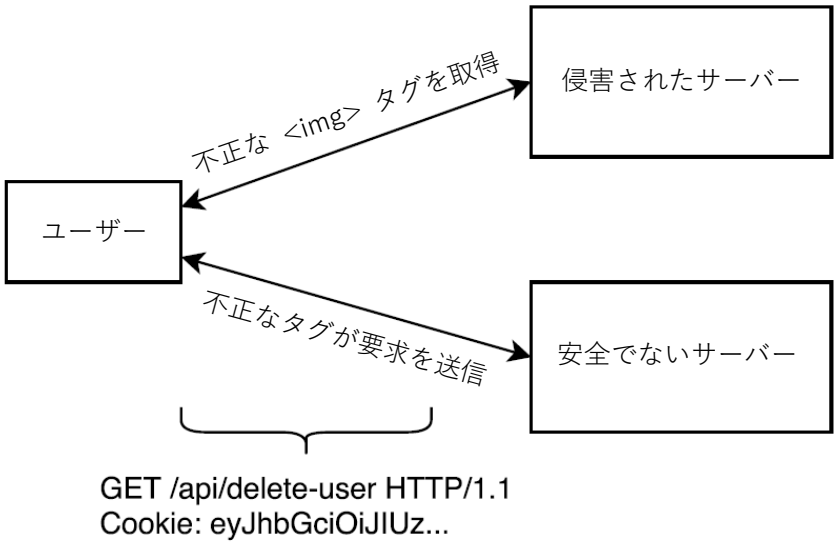

図 2.3: クロスサイト リクエスト フォージェリ

このケースでは、JWT の有効期間を短く設定することが効果的です。一般的な CSRF 対策としては、セッション Cookie ごと、およびリクエスト トークンごとに適切な要求元から要求が実行された場合にのみ、特殊なヘッダーを要求に追加する方法があります。JWT (およびセッション データ) が Cookie として保存されていなければ、CSRF 攻撃は不可能です。ただしそれでも、クロスサイト スクリプティング攻撃は可能です。

2.1.1.3 クロスサイト スクリプティング (XSS)

クロスサイト スクリプティング (XSS) とは、信頼できるサイトに JavaScript を挿入しようとする攻撃手法です。挿入された JavaScript は Cookie やローカル ストレージからトークンを盗み取ります。有効期限が切れていないアクセス トークンが漏えいすると、悪意のあるユーザーがそれを使用して保護されたリソースにアクセスするおそれがあります。XSS 攻撃は多くの場合、バックエンドに渡されたデータの検証の不備が原因で発生します (SQL インジェクション攻撃と似た手法です)。

XSS 攻撃の例としては、一般向けサイトのコメント セクションを利用したものが挙げられます。ユーザーがコメントを追加するたびに、そのコメントがバックエンドに保存され、コメント セクションをロードしたユーザーに表示されます。バックエンドでコメントをサニタイズしていない場合、悪意のあるユーザーが、ブラウザーに <script> タグとして解釈させることを狙ったコメントを書き込むことができます。悪意のあるユーザーが任意の JavaScript コードを挿入して各ユーザーのブラウザーで実行させることができるため、Cookie として保存されている資格情報やローカル ストレージに保存されている資格情報が盗まれるおそれがあります。

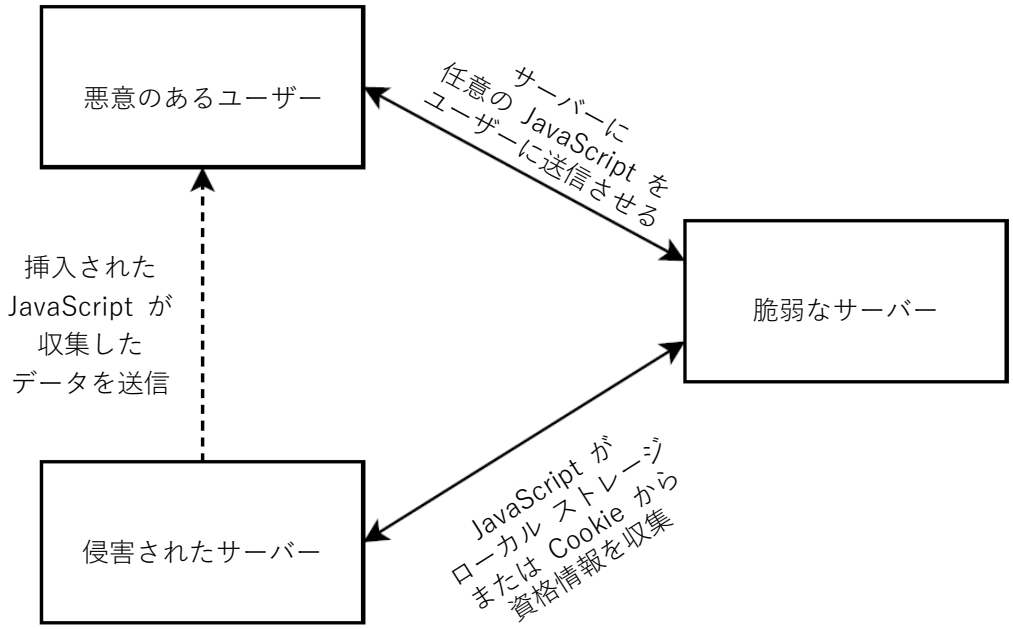

図 2.4: 永続的なクロスサイト スクリプティング

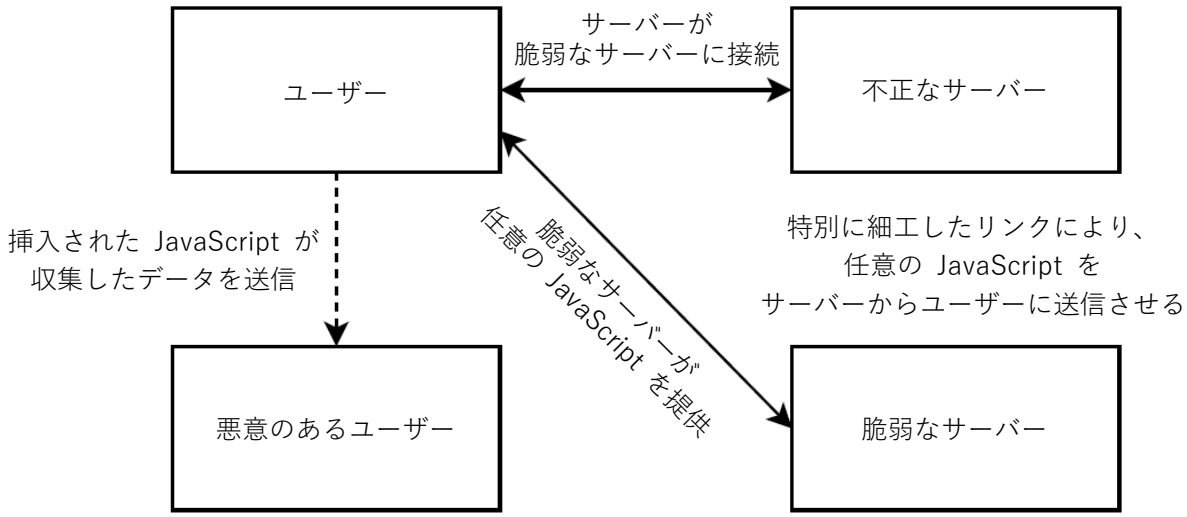

図 2.5: 反射型クロスサイト スクリプティング

対策は、バックエンドに渡されるすべてのデータを適切に検証することです。特に、クライアントから受信したデータは常にサニタイズする必要があります。Cookie を使用する場合は、HttpOnly フラグ[2] を設定することで、JavaScript によるアクセスから保護できます。HttpOnly フラグは便利ですが、CSRF 攻撃から Cookie を保護するためには役立ちません。

[2] https://www.owasp.org/index.php/HttpOnly

2.1.2 クライアント側セッションの有用性

どのようなアプローチにも長所と短所がありますが、クライアント側セッションも例外ではありません[3]。アプリケーションによっては、大きなセッションが必要になる場合があります。要求 (または要求のグループ) ごとにセッションのステート情報をやり取りするため、ほとんどの場合、バックエンドでの通信が削減されるというメリットが失われます。そのため、バックエンドでのクライアント側データ検索とデータベース検索の間である程度のバランスを取る必要があります。その方法は、アプリケーションのデータ モデルによって異なります。アプリケーションによっては、クライアント側セッションに適切にマッピングされないものもあります。また、クライアント側データに完全に依存しているアプリケーションもあります。この問題に関しては、皆様自身が最終的な判断を行ってください。ベンチマークを実行し、特定のステート情報をクライアント側に保持するメリットを検討しましょう。JWT のサイズは大きすぎないか? データ転送量に影響が及ばないか? データ転送量の増加により、バックエンドでの遅延の減少効果が相殺されないか? 複数の小さな要求を 1 つの大きな要求にまとめることはできないか? 要求を 1 つにまとめても、大規模なデータベースの検索が必要になるか? これらの問題を検討することで、適切なアプローチを決定できるでしょう。

2.1.3 例

この例では、簡単なショッピング アプリケーションを作成します。ユーザーのショッピング カートはクライアント側に保存されます。この例では、複数の JWT を使用しています。ショッピング カートもそのうちの 1 つです。

- ID トークン用の JWT。このトークンにユーザーのプロファイル情報が保持され、UI に使用されます。
- API のバックエンドとやり取りするための JWT (アクセス トークン)。
- クライアント側のステート用の JWT (ショッピング カート)。

デコード状態のショッピング カートは次のとおりです。

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

各アイテムは数値 ID で識別されます。エンコードして署名した JWT は次のようになります。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJpdGVtcyI6WzAsMiw0XSwiaWF0IjoxNDkzMTM5NjU5LCJleHAiOjE0OTMxND
MyNTl9. 932ZxtZzy1qhLXs932hd04J58Ihbg5_g_rIrj-Z16Js
```

カート内のアイテムを表示するには、フロントエンドで Cookie からアイテムを取得するだけです。

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

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

    const cart = jwt_decode(cartToken).items;
```

[3] https://auth0.com/blog/stateless-auth-for-stateful-minds/

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

フロントエンドでは署名がチェックされないことにご注意ください。JWT の内容を表示できるように、JWT がデコードされるだけです。チェック自体はバックエンドで実行されます。すべての JWT が検証されます。

次に、Express ミドルウェアとして実装したカート用 JWT の有効性をバックエンドでチェックする方法を示します。

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

    next();
}
```

アイテムが追加されると、新しいアイテムと署名が含まれた新しい JWT がバックエンドで作成されます。

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


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

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

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

    res.end();


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

接頭辞 `/protected` が付いている場所は、API アクセス トークンによっても保護されます。これは `express-jwt` を使用して設定します。

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

```
        return req.cookies['access_token'];
    }
}));
```

つまり、/protected/add_item エンドポイントがアクセス トークン検証ステップを通過した後でカートを検証する必要があります。一方のトークンは API へのアクセス (認可) を検証し、もう一方のトークンはクライアント側データ (カート) の整合性を検証します。

アクセス トークンと ID トークンは、Auth0 によってアプリケーションに割り当てられます。そのためには、Auth0 ダッシュボードを使用してクライアント[4] と API エンドポイント[5] を設定する必要があります[6]。これらはフロントエンドから呼び出される Auth0 JavaScript ライブラリを使って取得されます。

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

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

//(...)

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

audience クレームは、Auth0 ダッシュボードを使用して API エンドポイントに対して設定したクレームと一致している必要があります。

Auth0 認証/認可サーバーは、前述の設定を使用してログイン画面を表示した後、要求したトークンを使用して特定のパスにあるアプリケーションにリダイレクトします。これはバックエンドによって処理され、Cookie として設定されます。

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

});
```

CSRF 対策の実装は皆様の課題とし、本書では省略します。完全なコード サンプルは、samples/stateless-sessions ディレクトリに納められています。

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

2.2 フェデレーション アイデンティティ

フェデレーション アイデンティティ[7] システムを使用すると、異なる (さらに無関係の) 当事者間で認証および認可サービスを共有することができます。言い換えるならば、ユーザーのアイデンティティを一元化できるということです。フェデレーション アイデンティティの管理ソリューションはいくつかありますが、その中でも SAML[8] と OpenID Connect[9] は最も一般的なものです。認証と認可を一元化する特殊な製品も販売されています。前述の標準を実装している製品もあれば、まったく異なるものを使用している製品もあります。中には一元化のために JWT を使用している製品もあります。

認証と認可の一元化のために JWT を使用するかどうかは製品によって異なりますが、認可プロセスの基本的な流れは次のようになります。

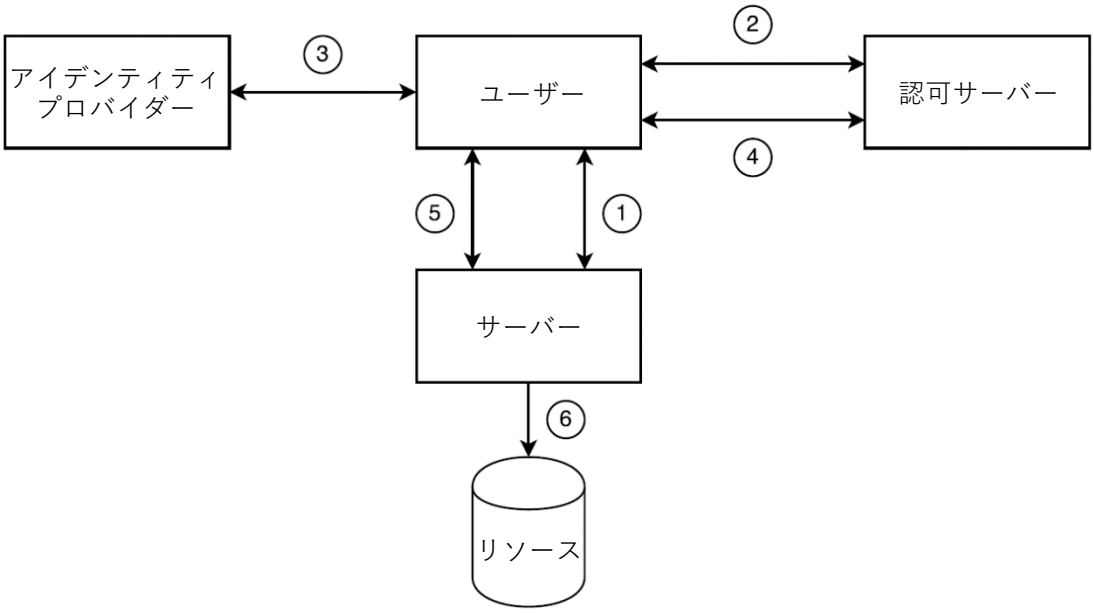

図 2.6: 一般的なフェデレーション アイデンティティの処理の流れ

1. ユーザーがサーバーによって制御されているリソースへのアクセスを試みる。
2. ユーザーがリソースにアクセスするための適切な資格情報を持っていないため、サーバーがユーザーを認可サーバーにリダイレクトする。認可サーバーは、アイデンティティ プロバイダーによって管理されている資格情報を使用してユーザーがログインできるように構成されている。
3. ユーザーが認可サーバーによってアイデンティティ プロバイダーのログイン画面にリダイレクトされる。
4. ユーザーが正常にログインし、認可サーバーにリダイレクトされる。認可サーバーは、アイデンティティ プロバイダーが提供する資格情報を使用して、リソース サーバーが要求する資格情報にアクセスする。
5. ユーザーが認可サーバーによってリソース サーバーにリダイレクトされる。この要求にはリソースへのアクセスに必要な正しい資格情報が含まれている。
6. ユーザーがリソースに正常にアクセスする。

サーバー間で渡されるすべてのデータは、リダイレクト要求に (通常は URL の一部として) 埋め込まれてユーザーを通過します。そのため、トランスポート セキュリティ (TLS) とデータ セキュリティが必要になります。

認可サーバーからユーザーに返される資格情報は、JWT としてエンコードできます。この例のように、認可サーバーがアイデンティティ プロバイダーを介してログインを許可している場合、認可サーバーは、統一されたインターフェイスと統一されたデータ (JWT) をユーザーに提供していると言えるでしょう。

本節で後ほど取り上げる例では、Auth0 を認可サーバーとして使用し、Twitter、Facebook、および一般的なユーザー データベースからのログインを処理します。

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

2.2.1 アクセス トークンとリフレッシュ トークン

アクセス トークンとリフレッシュ トークンは、各種のフェデレーション アイデンティティ ソリューションを解析するとよく見られるトークンです。ここでは、それぞれがどのようなものであり、認証プロセスや認可プロセスとの関連でどのように役立つかを簡単に説明します。

通常、どちらのトークンも OAuth2 仕様に基づいて実装されます[10]。OAuth2 仕様では、所有権からアクセス権を分離する形で、リソースへのアクセス権を提供するために必要な一連のステップを定義しています (つまり、異なるアクセス レベルを持つ複数の当事者が同じリソースにアクセスできます)。これらのステップの一部は実装で定義されます。そのため、複数の OAuth2 実装が競合している場合、相互運用できなくなる可能性があります。たとえば、トークンの実際のバイナリ形式は指定されません。トークンの目的と機能は指定されます。

アクセス トークンは、そのトークンを持つユーザーに保護リソースへのアクセス権を与えるトークンです。通常、この種のトークンは一時的なものであり、有効期限が埋め込まれている場合があります。また、追加情報 (要求が許可される IP アドレスなど) を保持していたり、関連付けられたりしている場合もあります。この追加情報は実装で定義されます。

一方、**リフレッシュ トークン**は、新しいアクセス トークンの要求をクライアントに許可するトークンです。たとえば、アクセス トークンの有効期限が切れた後に、クライアントが認可サーバーに対して新しいアクセス トークンを要求する場合があります。この要求を実行するには、リフレッシュ トークンが必要です。アクセス トークンとは対照的に、通常、リフレッシュ トークンは長期間有効です。

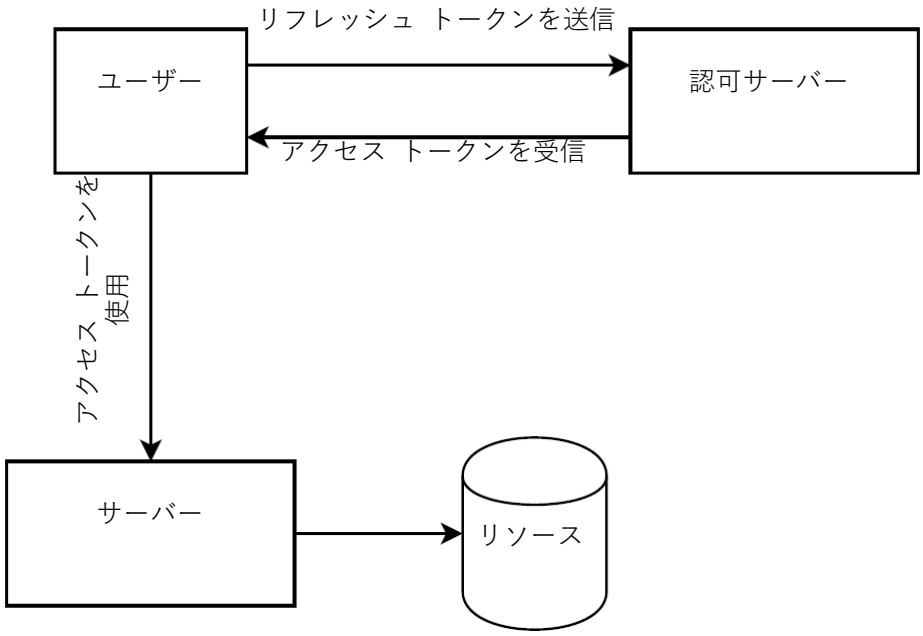

図 2.7: リフレッシュ トークンとアクセス トークン

アクセス トークンとリフレッシュ トークンを分離する主な目的は、アクセス トークンの検証を容易にすることにあります。署名を含むアクセス トークン (署名付き JWT など) は、リソース サーバーが単独で検証できます。検証のために認可サーバーにアクセスする必要はありません。

一方、リフレッシュ トークンについては認可サーバーにアクセスする必要があります。検証を認可サーバーへのクエリから分離しておくことで、遅延を改善し、アクセス パターンの複雑さを軽減することが可能になります。トークンの漏えいに備えて適切なセキュリティ対策を講じる場合は、アクセス トークンの有効期間をできる限り短くして、追加のチェック (クライアント チェックなど) を組み込みます。

リフレッシュ トークンは有効期間が長いため、漏えいから保護する必要があります。漏えいが発生した場合は、サーバーのブラックリストに登録する必要があります (アクセス トークンの有効期間が短いと最終的にリフレッシュ トークン

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

が強制的に使用されるため、リフレッシュ トークンがブラックリストに登録されて、すべてのアクセス トークンが期限切れになった後でリソースが保護されます)。

> 注: アクセス トークンとリフレッシュ トークンの概念は OAuth2 で導入されました。OAuth 1.0 および 1.0a では、トークンという言葉は違う意味で使用されています。

2.2.2 JWT と OAuth2

OAuth2 ではトークンの形式は特に定められていませんが、JWT は OAuth2 の要件に適しています。署名付きの JWT は、リソースへのアクセス レベルを区別するのに必要なすべてのデータをエンコードしたり、有効期限を保持したり、署名により認可サーバーに対する検証クエリを回避したりできるため、アクセス トークンとして優れています。一部のフェデレーション アイデンティティ プロバイダーは、JWT 形式のアクセス トークンを発行しています。

JWT はリフレッシュ トークンにも使用されますが、リフレッシュ トークンに JWT を使用する必要性はあまりありません。リフレッシュ トークンに必要なのは認可サーバーへのアクセスであり、ほとんどの場合は単純な UUID を使用するだけで十分であるため、リフレッシュ トークンにペイロードを含める必要はありません (ただし、署名をすることはあります)。

2.2.3 JWT と OpenID Connect

OpenID Connect[11] は、OAuth2 の一般的なユース ケースを、明確に定義された共通の仕様に準拠させることを目的とした標準化の取り組みです。OAuth2 を構成する多くの要素は実装者が自由に選択できるため、OpenID Connect では、不足している部分に対して適切な定義を提供することを目指しています。具体的には、OpenID Connect は OAuth2 認可フローを実行するための API とデータ形式を定義します。また、このフローに基づいて構築できる認証レイヤーも提供します。その一部のデータ形式として選ばれているのが、JSON Web Token です。特に、ID トークン[12] は、認証済みユーザーに関する情報を伝える特殊な種類のトークンです。

2.2.3.1 OpenID Connect のフローと JWT

OpenID Connect では、さまざまな方法でデータを返すフローがいくつか定義されています。このデータの中には、JWT 形式のものもあります。

- **認可フロー**: クライアントがエンドポイント (`/authorize`) に認可コードを要求します。このコードをトークン エンドポイント (`/token`) に対して使用すると、ID トークン (JWT 形式)、アクセス トークン、またはリフレッシュ トークンを要求することができます。
- **暗黙的フロー**: クライアントが認可エンドポイント (`/authorize`) から直接トークンを要求します。このトークンは要求で指定されます。ID トークンが要求されると、トークンは JWT 形式で返されます。
- **ハイブリッド フロー**: クライアントが認可コードと特定のトークンの両方を認可エンドポイント (`/authorize`) から要求します。ID トークンが要求されると、トークンは JWT 形式で返されます。ID トークンがこのステップで要求されなかった場合、後でトークン エンドポイント (`/token`) から直接要求されることがあります。

2.2.4 例

この例では、認可サーバーとして Auth0[13] を使用します。Auth0 では、さまざまなアイデンティティ プロバイダーを動的に設定できます。つまり、ユーザーがログインを試みる際に、認可サーバーで変更を行うことで、ユーザーがさまざまなアイデンティティ プロバイダー (Twitter や Facebook など) を使用してログインできるようにすることが可能です。デプロイ後にアプリケーションを特定のプロバイダーに委ねる必要はありません。そのため、この例は非常にシンプルです。すべてのサンプル サーバーで、Auth0.js[14] ライブラリを使用して Auth0 ログイン画面を設定しました。ユーザーは 1 つのサーバーにログインすれば、相互接続されていない他のサーバーにもアクセスできるようになります。

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

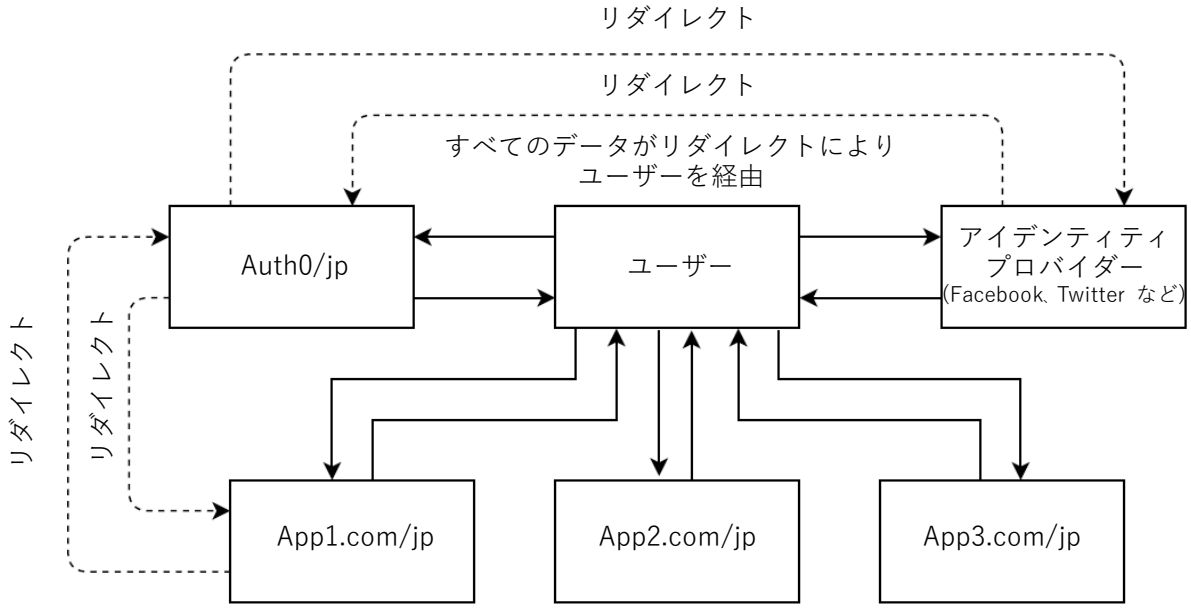

図 2.8: Auth0 を認可サーバーとして使用

2.2.4.1 Node.js アプリのための Auth0 Lock のセットアップ

Auth0 ライブラリ[15] の設定は次のように行います。ここでもステートレス セッションの例で使用したサンプルと同じものを使用します。

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

// (...)

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

authorize 呼び出しで prompt:'none' パラメーターを使用していることに注意してください。authorize 呼び出しによって、ユーザーを認可サーバーにリダイレクトします。ユーザーが保護リソースへのアクセスに自分の資格情報を使用する認可をアプリケーションに既に与えている場合、none パラメーターを指定すると、認可サーバーは単純にユーザーをアプリケーションにリダイレクトします。ユーザーからは、既にアプリケーションにログインしているように見えます。

この例では、app1.com と app2.com という 2 つのアプリケーションがあります。ユーザーが両方のアプリケーションを認可してから (認可は初回ログイン時に一度だけ行います)、どちらかのアプリケーションにログインすると、もう一方のアプリケーションでもログイン画面が表示されないままログインできるようになります。

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

これをテストするには、samples/single-sign-on-federated-identity ディレクトリにあるサンプルの README ファイルを参照して、両方のアプリケーションをセットアップして実行してください。両方のアプリケーションを実行した後で、app1.com:3000[16] と app2.com:3001[17] に移動してログインします。両方のアプリケーションからログアウトしたら、どちらかのアプリケーションにログインします。続いてもう一方のアプリケーションに戻り、ログインします。すると、どちらのアプリケーションでもログイン画面が表示されなくなります。認可サーバーは以前のログインを記憶しており、どちらかのアプリケーションによって要求された場合に新しいアクセス トークンを発行できます。したがって、ユーザーが認可サーバーのセッションを保持している限り、そのユーザーは両方のアプリケーションにログイン済みということになります。

CSRF 対策の実装は皆様の課題とし、本書では省略します。

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

第 3 章

JSON Web Token の詳細

第 1 章で説明したように、どの JWT も、ヘッダー、ペイロード、署名/暗号化データという 3 つの異なる要素で構成されます。ヘッダーとペイロードは、一定の構造を持つ JSON オブジェクトです。署名/暗号化データは、署名または暗号化に使用されるアルゴリズムに依存します。また、暗号化されない JWT の場合は省略されます。JWT は、JWS/JWE コンパクト シリアライゼーションと呼ばれるコンパクトな表現にエンコードできます。

> JWS および JWE の仕様では、JSON シリアライゼーションという 3 つ目のシリアライゼーション形式も定義されています。これは、1 つの JWT 内で複数の署名または受信者を使用できるコンパクトでない表現形式です。詳細は第 4 章と第 5 章で説明します。

コンパクト シリアライゼーションとは、最初の 2 つの JSON 要素 (ヘッダーとペイロード) の UTF-8[1] バイト配列と、必要に応じて署名または暗号化用のデータ (この要素は JSON オブジェクトではありません) を、URL セーフな Base64[2] でエンコードする方式です。署名/暗号化データも Base64-URL でエンコードされます。この 3 つの要素はドット (.) で区切られます。

> JWT では、URL が含まれていても問題のない Base64 エンコード方式を使用します。このエンコード方式では、"+" と "/" をそれぞれ "-" と "_" に置き換えます。パディングも削除されます。この種のエンコード方式は base64url[3] と呼ばれています。本書における Base64 エンコード方式とは、すべてこの方式を指すとお考えください。

結果は次のような表示可能な文字列になります (見やすいように改行を挿入しています)。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

ご覧のとおり、JWT の 3 つの要素がドットで区切られています (ヘッダー、ペイロード、署名の順)。

この例では、デコードされたヘッダーは次のようになります。

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

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

```
}
```

デコードされたペイロードは次のようになります。

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

署名の検証に必要なシークレットは secret です。

> JWT.io[4] では、JWT についてインタラクティブ形式で詳しく学ぶことができます。上記のトークンをコピーし、編集するとどのような結果になるか試してみてください。

3.1 ヘッダー

すべての JWT のヘッダー (JOSE ヘッダー) には、その JWT 自体に関するクレームが含まれています。このクレームによって、その JWT が署名付きか暗号化されているかにかかわらず、使用するアルゴリズムが設定されます。また一般的には、その JWT の残りの部分を解析する方法も設定されています。

JWT の種類によっては、その他の必須クレームがヘッダーに含まれている場合があります。たとえば、暗号化された JWT は、鍵の暗号化とコンテンツの暗号化に使用する暗号アルゴリズムに関する情報を保持しています。これらのフィールドは、暗号化されていない JWT にはありません。

暗号化されない JWT ヘッダーの必須クレームは、`alg` クレームだけです。

- **alg**: この JWT の署名や復号化に使用される主なアルゴリズム。

暗号化されない JWT の場合、このクレームの値を `none` に設定する必要があります。
オプションのヘッダー クレームとして、`typ` クレームと `cty` クレームがあります。

- **typ**: その JWT のメディア タイプ[5]。このパラメーターは、JWT が JOSE ヘッダーを含む他のオブジェクトと混在する可能性がある場合に、参照用としてのみ使用されます。そのような状況は、実際にはほとんど発生しません。そのような状況が生じた場合には、このクレームの値を `JWT` に設定する必要があります。
- **cty**: コンテンツ タイプ。ほとんどの JWT はペイロードに特定のクレームと任意のデータを保持しています。その場合、cty クレームを設定しないでください。ペイロードも JWT である場合 (ネストされた JWT)、このクレームを追加し、値を `JWT` に設定する必要があります。これにより、ネストされた JWT の処理が必要であることが通知されます。JWT がネストされることは非常に少ないため、cty クレームがヘッダーに含められることはまずありません。

したがって、暗号化されていない JWT の場合、ヘッダーは次のようになります。

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

これをエンコードすると、次のようになります。

```
eyJhbGciOiJub25lIn0
```

> ヘッダーにはさらにユーザー定義のクレームを追加することもできます。ただし、暗号化された JWT で復号化前に特定のユーザー固有のメタデータが必要になる場合を除き、ユーザー定義のクレームが追加されることは一般的ではありません。

3.2 ペイロード

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

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

ペイロードには、通常、すべての有用なユーザー データが追加されます。また、仕様で定義されている一部のクレームも含まれることがあります。ヘッダーと同様に、ペイロードは JSON オブジェクトです。必須クレームはありませんが、明確な意味を有する特定のクレームが存在します。JWT の仕様では、実装によって認識されないクレームは無視すべきであると規定されています。特定の意味が付与されているクレームは、登録済みクレームと呼ばれています。

3.2.1 登録済みクレーム

- **iss**: *issuer* (発行者) を表します。JWT の発行者を一意に識別する、大文字小文字の区別がある文字列または URI です。このクレームの解釈はアプリケーション固有です (発行者を管理する中央機関はありません)。
- **sub**: *subject* (主題) を表します。この JWT に情報が含まれている当事者を一意に識別する、大文字小文字の区別がある文字列または URI です。つまり、この JWT に含まれるクレームは、この当事者に関する記述ということになります。JWT の仕様では、このクレームは発行者のコンテキストにおいて一意でなければならず、それが不可能な場合にはグローバルに一意でなければならないと規定されています。このクレームの処理はアプリケーション固有です。
- **aud**: *audience* (受信者) を表します。大文字小文字の区別がある単一の文字列/URI、またはこの JWT が対象とする受信者を一意に識別する値の配列のどちらかです。つまり、このクレームがある場合、この JWT 内のデータを読み取る当事者は、*aud* クレームに自身に関する情報があることを認識するか、JWT 内に含まれるデータを無視しなければなりません。*iss* クレームや *sub* クレームの場合と同様に、このクレームの処理はアプリケーション固有です。
- **exp**: *expiration* (有効期限) を表します。POSIX[6] で定義されている形式である "エポックからの経過秒数" によって特定の日時を表す数値です。このクレームにより、この JWT が無効とみなされる時点を正確に設定します。実装によっては、クロック間における一定のスキューを許容する場合があります (この JWT が有効期限後も数分間有効であるとみなします)。
- **nbf** : *not before* (開始日時) を表します。これは *exp* クレームの反対です。POSIX[7] で定義されている形式である "エポックからの経過秒数" によって特定の日時を表す数値です。このクレームにより、この JWT が有効とみなされる時点を正確に設定します。この日時には、現在以降の日時を指定する必要があります。実装によっては、一定のスキューが許容されます。
- **iat**: *issued at* (発行日時) を表します。この JWT が発行された特定の日時 (*exp* および *nbf* と同じ形式) を表す数値です。
- **jti**: *JWT ID* を表します。この JWT の一意の ID を表す文字列です。このクレームは、(リプレイを防止する目的などで) JWT を他の類似するコンテンツと区別するために使用できます。一意性を保証するのは、実装側の責任です。

ご覧のように、どのクレームも短い名前が使用されています。これは JWT をできる限りコンパクトにするという JWT の設計要件に沿っています。

> 文字列または URI: JWT の仕様によると、URI は : を含む任意の文字列として解釈されます。有効な値を提供するのは、実装側の責任です。

3.2.2 パブリック クレームとプライベート クレーム

「登録済みクレーム」の節で挙げられていないクレームは、**プライベート** プレームまたは**パブリック** クレームのどちらかに該当します。

[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

- **プライベート** クレームとは、JWT のユーザー (利用者および作成者) によって定義されたクレームです。これらは特定の場合に使用される一時的なクレームです。そのため、名前の衝突を防ぐために注意を払う必要があります。
- **パブリック** クレームとは、IANA JSON Web Token Claims レジストリ[8] (ユーザーが衝突を防ぐために独自のクレームを登録できるレジストリ) に登録されたクレーム、または衝突しにくい名前 (たとえば先頭に名前空間を付加した名前) を使用して命名されたクレームを指します。

実際のところ、ほとんどのクレームは登録済みクレームかパブリック クレームのどちらかです。一般的に、ほとんどの JWT は特定の目的と明確な一連の潜在的ユーザーを想定して発行されます。その場合、衝突しにくい名前を選択しやすくなります。

JSON 解析ルールの場合と同様に、重複するクレーム (重複する JSON 鍵) は、最後のクレームだけを有効なものとして保持する形で処理されます。また、JWT の仕様では、実装において重複するクレームを含む JWT を無効とみなすことが可能です。JWT を処理する実装についてよく知らない場合は、重複するクレームを避けてください。

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

3.3 セキュリティ保護のない JWT

これまでに学習したことで、セキュリティ保護のない JWT を構築することができます。次に示すのは最もシンプルな JWT です。単純な (通常は静的) ヘッダーは次のようなものです。

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

ユーザー定義のペイロードは、次のようになります。

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

この JWT には署名も暗号化もないため、2 つの要素だけがエンコードされます (見やすいように改行を挿入しています)。

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

このようなセキュリティ保護のない JWT は、クライアント側での使用に適しています。たとえば、セッション ID が推測しにくい数値であり、その他のデータがクライアントによってビューの構築のためだけに使用される場合、署名を使用する必要はありません。このようなデータは、単一ページの Web アプリケーションで、ユーザーが最後に訪れたページにリダイレクトされている間に、バックエンドにアクセスすることなくユーザー名を含むビューを構築するために使用されることがあります。悪意のあるユーザーがこのデータを改変しても、何も得るものはないでしょう。

> エンコード後の末尾のドット (.) に注目してください。署名が付いていないため、文字列は空になっていますが、ドットは残ります。

しかし実際には、セキュリティ保護のない JWT はほとんど使用されません。

3.4 セキュリティ保護のない JWT の作成

JSON 形式のヘッダーとペイロードをコンパクトな表現に変換するには、次の手順を実行します。

1. ヘッダーを、UTF-8 表現のバイト配列として取得します。JWT の仕様では、エンコードの前に JSON を圧縮したり、無意味な文字 (空白など) を取り除いたりする必要はありません。
2. Base64-URL アルゴリズムを使用してバイト配列をエンコードし、末尾の等号 (=) を削除します。
3. ペイロードの UTF-8 表現のバイト配列を取得します。JWT の仕様では、エンコードの前に JSON を圧縮したり、無意味な文字 (空白など) を取り除いたりする必要はありません。
4. Base64-URL アルゴリズムを使用してバイト配列をエンコードし、末尾の等号 (=) を削除します。
5. 生成された 2 つの文字列を連結します。ヘッダー、"."、ペイロードの順につなげます。

エンコードの前に、ヘッダーとペイロードを検証 (必須クレームが存在するか、各クレームが正しく使用されているか) する必要があります。

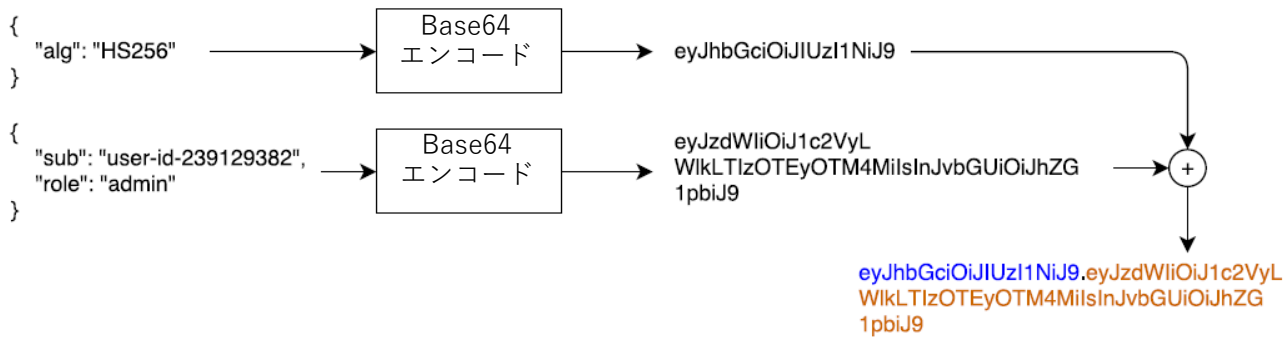

図 3.1: セキュリティ保護のない JWT のコンパクトな表現の作成

3.4.1 サンプル コード

```
// 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}`;
}
```

完全なサンプルは、付属のサンプル コードの `coding.js` ファイルにあります。

3.5 セキュリティ保護のない JWT の解析

コンパクト シリアライゼーション形式から JSON 表現に変換するには、次の手順を実行します。

1. 最初のピリオド "." を検索します。その前の文字列を取得します (ピリオドを含まない)。
2. Base64-URL アルゴリズムを使用して文字列をデコードします。その結果が JWT ヘッダーです。
3. 手順 1 のピリオドよりも後の文字列を取得します。
4. Base64-URL アルゴリズムを使用して文字列をデコードします。その結果が JWT ペイロードです。

生成された JSON 文字列に必要に応じて空白を追加することで、体裁を整えることができます。

3.5.1 サンプル コード

```
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)
```

```
    };
}
```

完全なサンプルは、付属のサンプル コードの coding.js ファイルにあります。

第 4 章

JSON Web Signatures

JSON Web Signature は、おそらく JWT の最も有用な機能の 1 つです。シンプルなデータ形式と、明確に定義された一連の署名アルゴリズムを組み合わせることで、JWT は急速に発展し、クライアントと仲介者の間でデータを安全に共有するのに最適なフォーマットとなりつつあります。

署名の目的は、1 つ以上の当事者が JWT の信頼性を確立できるようにすることです。ここで言う信頼性とは、JWT に含まれるデータが改ざんされていないことを指します。つまり、当事者が署名チェックを実行できれば、JWT によって提供されるコンテンツを信頼できるということになります。ただし、署名を行っても、他の当事者が JWT 内のコンテンツを読み取ることを防ぐことはできません。暗号化はそのために存在します。暗号化の目的については第 5 章で説明します。

JWT の署名をチェックするプロセスは、トークンの検証と呼ばれます。トークンは、そのヘッダーおよびペイロードで指定された制約がすべて満たされた場合に有効とみなされます。これは JWT の非常に重要な特徴です。JWT のヘッダーとペイロード (およびユーザーが求める要件) で指定された範囲まで、実装で JWT をチェックする必要があるからです。そのため、署名がなくても JWT は有効であるとみなされる場合があります (ヘッダーの *alg* クレームが `none` に設定されている場合)。さらに、JWT に有効な署名が含まれていても、他の理由 (*exp* クレームによって有効期限切れになった場合など) で無効とみなされることもあります。署名付き JWT を対象とした攻撃では、署名を取り除き、ヘッダーを変更してセキュアでない JWT に改変する手法がよく取られます。JWT がユーザー独自の要件に従って検証されていることを確認するのは、ユーザーの責任です。

署名付き JWT は、JSON Web Signature の仕様、RFC 7515[1] で定義されています。

4.1 署名付き JWT の構造

JWT の構造については、第 3 章で説明しました。本節では署名の要素に注目しながら、その構造を再確認します。

署名付き JWT は、ヘッダー、ペイロード、署名の 3 つの要素で構成されます (見やすいように改行を挿入しています)。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

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

最初の 2 つの要素 (ヘッダーとペイロード) のデコード プロセスは、セキュリティ保護のない JWT の場合と同じです。アルゴリズムとサンプル コードは第 3 章の最後にあります。

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

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

しかし、署名付き JWT には署名というもう 1 つの要素があります。コンパクト シリアライゼーション形式において、この要素は最後のピリオド (.) の後に表記されます。

JWS の仕様に従って使用できる署名アルゴリズムにはいくつかの種類があるため、この要素はさまざまな方法で解釈されます。JWS の仕様では、仕様に準拠するすべての実装で次のアルゴリズムをサポートすることを要件としています。

- SHA-256 を使用した HMAC。JWA 仕様では HS256 と呼ばれています。

また、仕様では推奨アルゴリズムも明示しています。

- SHA-256 を使用した RSASSA PKCS1 v1.5。JWA 仕様では `RS256` と呼ばれています。
- P-256 および SHA-256 を使用した ECDSA。JWA 仕様では `ES256` と呼ばれています。

  JWA とは、JSON Web Algorithm 仕様、RFC 7518[2] を指します。

これらのアルゴリズムについては、第 7 章で詳しく説明します。本章では、アルゴリズムの使用の実際的な側面を取り上げます。

JWS では、次に示すアルゴリズムもオプションとして使用できます。

- HS384、HS512: HS256 アルゴリズムのバリエーションで、SHA-256 をそれぞれ SHA-384 と SHA-512 に変えたもの。
- RS384、RS512: RS256 アルゴリズムのバリエーションで、SHA-256 をそれぞれ SHA-384 と SHA-512 に変えたもの。
- ES384、ES512: ES256 アルゴリズムのバリエーションで、SHA-256 をそれぞれ SHA-384 と SHA-512 に変えたもの。
- PS256、PS384、PS512: RSASSA-PSS + MGF1 のバリエーションで、それぞれ SHA256/384/512 を使用したもの。

これらは基本的に、先に挙げた 3 つの必須アルゴリズムと推奨アルゴリズムのバリエーションです。これらの略称の意味は第 7 章で詳しく説明します。

4.1.1 コンパクト シリアライゼーションに使用されるアルゴリズムの概要

これらのアルゴリズムの概要を説明するにあたり、まず JavaScript 2015 環境のいくつかの関数の意味を説明します。

- **base64:** オクテット配列を受け取り、Base64-URL アルゴリズムを使用して新しいオクテット配列を返す関数。
- **utf8:** 任意のエンコーディングのテキストを受け取り、UTF-8 エンコーディングのオクテット配列を返す関数。
- **JSON.stringify:** JavaScript オブジェクトを受け取り、それを文字列形式 (JSON) にシリアライズする関数。
- **sha256:** オクテット配列を受け取り、SHA-256 アルゴリズムを使用して新しいオクテット配列を返す関数。
- **hmac:** SHA 関数、オクテット配列、およびシークレットを受け取り、HMAC アルゴリズムを使用して新しいオクテット配列を返す関数。
- **rsassa:** SHA 関数、オクテットの配列、および秘密鍵を受け取り、RSASSA アルゴリズムを使用して新しいオクテット配列を返す関数。

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

HMAC ベースの署名アルゴリズムの場合:

```
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}`;
```

公開鍵署名アルゴリズムの場合:

```
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}`;
```

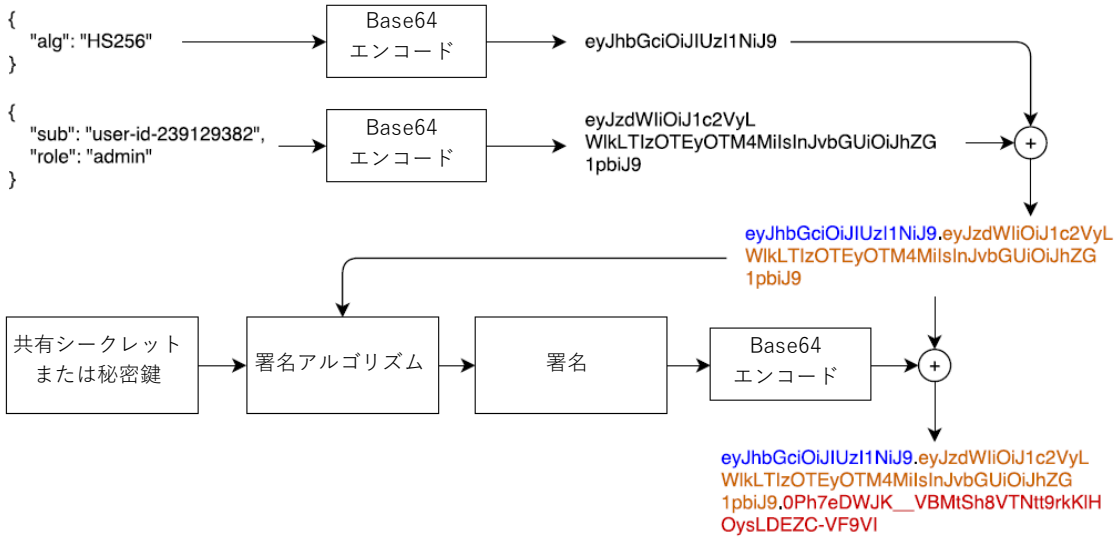

図 4.1: JWS コンパクト シリアライゼーション

これらのアルゴリズムの詳細については、第 7 章で説明します。

4.1.2 署名アルゴリズムの実際的な側面

どの署名アルゴリズムも目的は同じです。つまり、JWT に含まれるデータの信頼性を確立することです。しかし、その方法はそれぞれ異なります。

Keyed-Hash Message Authentication Code (HMAC) は、暗号学的ハッシュ関数[3] を使用して特定のペイロードとシークレットを組み合わせるアルゴリズムです。生成されるコードは、生成した側と検証する側の両方がシークレットを知っている場合にのみ、メッセージの検証に使用できます。つまり、**HMAC は共有シークレットによってメッセージの検証を可能にするアルゴリズム**であると言えます。

JWT の最も一般的な署名アルゴリズムである HS256 で使用される暗号学的ハッシュ関数は、SHA-256 です。SHA-256 については、第 7 章で詳しく説明します。暗号学的ハッシュ関数は、任意の長さのメッセージを受け取り、固定長の出力を生成します。同じメッセージは常に同じ出力を生成します。暗号学的な性質を持つハッシュ関数を使用した場合、関数の出力から元のメッセージを復元することは、数理的に不可能です。そのため、暗号学的ハッシュ関数は一方向性関数とされており、実際にメッセージを共有することなくメッセージを識別するために使用できます。メッセージを少し変え

[3] https://ja.wikipedia.org/wiki/%E6%9A%97%E5%8F%B7%E5%AD%A6%E7%9A%84%E3%83%8F%E3%83%83%E3%82%B7%E3%83%A5%E9%96%A2%E6%95%B0

ると (たとえば 1 バイト変えるだけでも)、出力がまったく違ったものになります。

RSASSA は、署名用に改良された RSA アルゴリズム[4] (第 7 章で説明) の一種です。RSA は公開鍵アルゴリズムです。公開鍵アルゴリズムは、分割鍵 (1 つの公開鍵と 1 つの秘密鍵) を生成します。この種のアルゴリズムでは、秘密鍵を署名付きのメッセージの作成とその信頼性の検証の両方に使用できます。一方、公開鍵はメッセージの信頼性の検証にのみ使用できます。そのため、このアルゴリズムを使用すると、**1 対多**のメッセージを安全に配信できます。受信者は、メッセージに関連付けられた公開鍵のコピーを持っていれば、メッセージの信頼性を検証することはできますが、それを使用しても新しいメッセージを作成することはできません。したがって、シークレットを共有する署名方式 (HMAC など) とは異なる使用シナリオが可能になります。HMAC と SHA-256 を使用する方式の場合、メッセージを検証できる当事者が、新しいメッセージを作成することもできます。たとえば、正規のユーザーが攻撃者になった場合、他の当事者に気づかれることなくメッセージを改変することができます。公開鍵方式では、ユーザーが攻撃者になったとしても公開鍵しか入手できないため、新しい署名付きメッセージを作成することはできません。

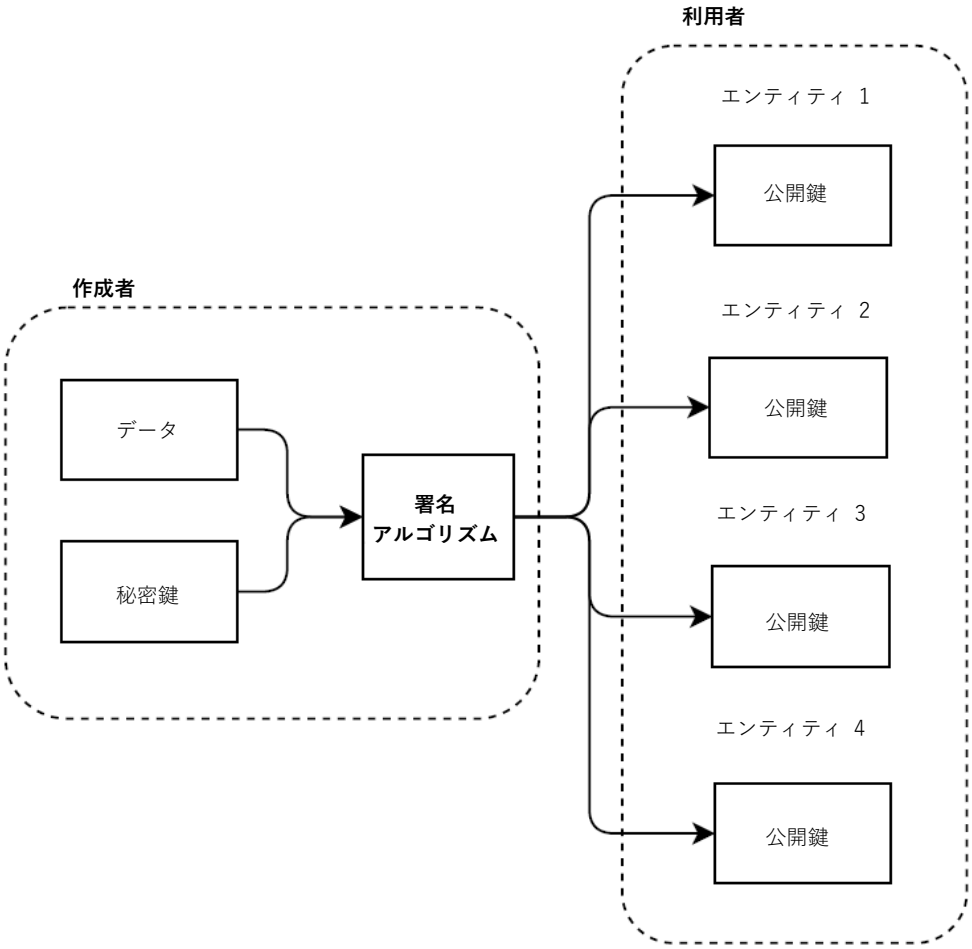

図 4.2: 1 対多の署名

公開鍵暗号[5] 方式を使用すると、別の利用シナリオが可能になります。たとえば、前述の RSA アルゴリズムの一種を使用し、公開鍵を使ってメッセージを暗号化することができます。暗号化したメッセージは、秘密鍵を使用しない限り復号化できません。そのため、**多対 1** のセキュアな通信チャネルを構築できます。この方式は、第 5 章で説明する暗号化された JWT に使用されます。

[4] https://ja.wikipedia.org/wiki/RSA%E6%9A%97%E5%8F%B7
[5] https://ja.wikipedia.org/wiki/%E5%85%AC%E9%96%8B%E9%8D%B5%E6%9A%97%E5%8F%B7

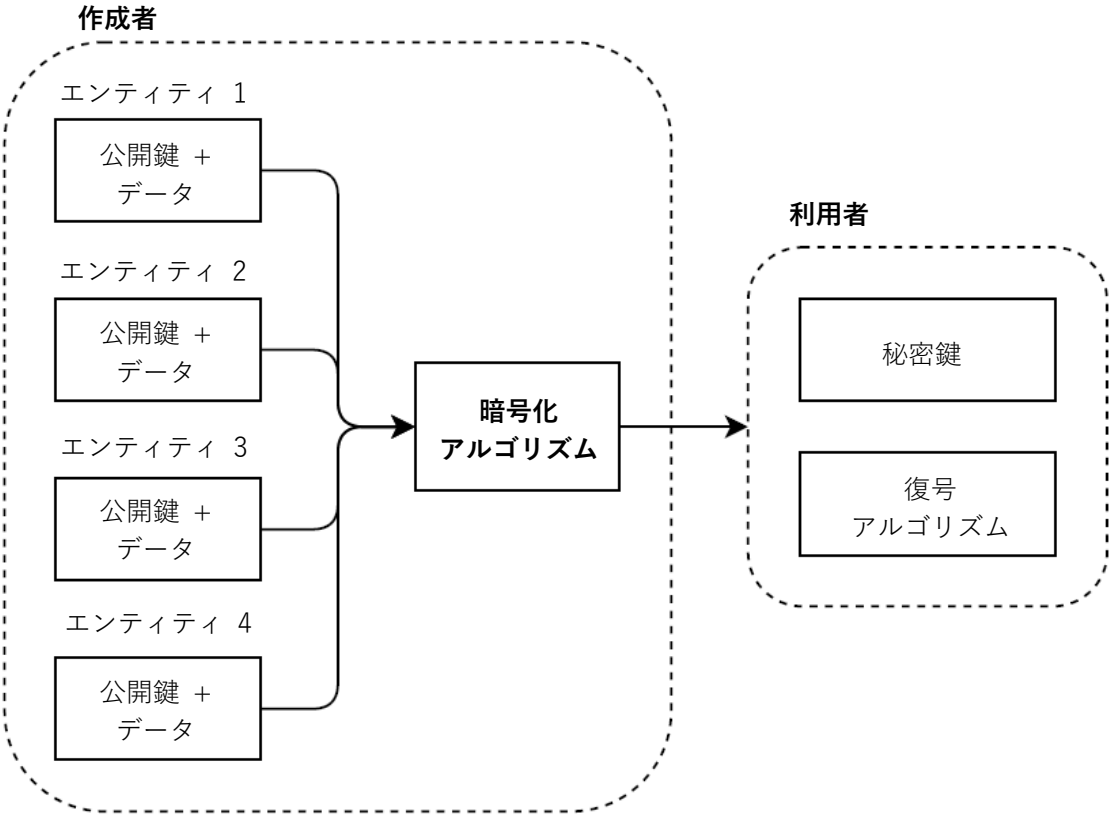

図 4.3: 多対 1 の暗号化

楕円曲線 DSA (ECDSA)[6] は、RSA に代わるアルゴリズムです。このアルゴリズムでも公開鍵と秘密鍵のペアが生成されますが、計算方法に違いがあります。この違いのおかげで、RSA を使用する場合よりも低いハードウェア要件で同様のセキュリティ保証を実現できます。

これらのアルゴリズムについては、第 7 章で詳しく説明します。

4.1.3 JWS のヘッダー クレーム

JWS では、多くのクレームをヘッダーに強制的に含める特殊な使用方法が認められています。たとえば、公開鍵署名アルゴリズムの場合、URL をクレームとして公開鍵に埋め込むことができます。以下に示したのは、JWS トークンで使用可能な登録済みヘッダー クレームです。これらのクレームはすべて、セキュリティ保護のない JWT で使用可能なクレームに加えて使用できるものであり、署名付きの JWT の使用方法に応じてオプションとして追加できます。

- **jku**: JSON Web Key (JWK) Set の URL。この JWT に署名するために使用する、JSON 形式にエンコードされた公開鍵のセットを指す URI です。この鍵を取得するには、トランスポート セキュリティ (HTTP の TLS など) を使用する必要があります。鍵の形式は JWK Set (第 6 章を参照) です。
- **jwk**: JSON Web Key。この JWT に JSON Web Key 形式で署名するために使用する鍵です (第 6 章を参照)。
- **kid**: 鍵 ID。この JWT に署名するために使用する単一の鍵を表すユーザー定義の文字列です。このクレームは、受信者に鍵の署名の変更を通知するために使用されます (複数の鍵を使用する場合)。
- **x5u**: X.509 URL。PEM 形式にエンコードされた X.509 (証明書形式の標準の 1 つ) 公開証明書のセットを指す URI。セット内の最初の証明書は、この JWT に署名するために使用する証明書である必要があります。後続の証明書はそれぞれ先行する証明書に署名します。これにより証明書チェーンが形成されます。X.509 は RFC 5280[7] で

[6] https://ja.wikipedia.org/wiki/%E6%A5%95%E5%86%86%E6%9B%B2%E7%B7%9ADSA
[7] https://tools.ietf.org/html/rfc5280

定義されています。証明書の転送には、トランスポート セキュリティが必要です。

- **x5c**: X.509 証明書チェーン。この JWS に署名するために使用する X.509 証明書の JSON 配列です。各証明書は、その DER PKIX 表現の Base64 エンコード値である必要があります。配列内の最初の証明書は、この JWT に署名するために使用される証明書である必要があります。以降に、証明書チェーン内のその他の証明書が続きます。
- **x5t**: X.509 証明書の SHA-1 フィンガープリント。この JWT に署名するために使用する、X.509 DER にエンコードされた証明書の SHA-1 フィンガープリントです。
- **x5t#S256**: **x5t** と同じですが、SHA-1 の代わりに SHA-256 を使用します。
- **typ**: 暗号化なしの JWT の **typ** の値と同じですが、"JOSE" と "JOSE+JSON" という値を追加で使用できます。それぞれ、コンパクト シリアライゼーション形式と JSON シリアライゼーション形式を示すために使用されます。このクレームは、1 つのコンテナー内に、JOSE ヘッダーを持つ類似したオブジェクトがこの JWT と混在する場合にのみ使用されます。
- **crit**: *critical* (必須) を表します。この JWT のパーサーで処理する必要がある実装定義の拡張として同じヘッダー内に示されているクレームの名前が含まれた文字列配列です。クレーム名が含まれていない文字列配列は指定できません (空の配列は無効値です)。

4.1.4 JWS JSON シリアライゼーション

JWS の仕様では、コンパクトではない別の種類のシリアライゼーション形式が定義されています。この形式では、同一の署名付き JWT で複数の署名を使用できます。これは JWS JSON シリアライゼーションと呼ばれています。

JWS JSON シリアライゼーション形式では、署名付き JWT は JSON 形式 (ブラウザーで `JSON.stringify` を呼び出すと得られる形式) の表示可能なテキストとして表現されます。最上位の JSON オブジェクトに、以下の鍵と値のペアを含める必要があります。

- **payload**: 実際の JWT ペイロード オブジェクトを Base64 エンコードした文字列。
- **signatures**: 署名を保持する JSON オブジェクトの配列。このオブジェクトは以下のように定義されます。

signatures 配列内の各 JSON オブジェクトには、次の鍵と値のペアが含まれている必要があります。

- **protected**: JWS ヘッダーを Base64 でエンコードした文字列。このヘッダーに含まれるクレームは、署名により保護されます。このヘッダーは、保護されていないヘッダーがない場合にのみ必要です。保護されていないヘッダーがある場合は、このヘッダーはあってもなくても構いません。
- **header**: ヘッダー クレームを含んでいる JSON オブジェクト。このヘッダーは署名によって保護されません。保護されないヘッダーが存在しない場合、この要素は必須です。保護されたヘッダーが存在する場合、この要素はオプションです。
- **signature**: JWS の署名を Base64 でエンコードした文字列。

コンパクト シリアライゼーション形式 (保護されたヘッダーのみが存在する場合) とは異なり、JSON シリアライゼーションでは、**protected** (保護されたヘッダー) と **unprotected** (保護されないヘッダー) の 2 種類のヘッダーを使用できます。保護されたヘッダーは署名によって検証されます。保護されないヘッダーは署名によって検証されません。どのクレームを含めるかは、実装側またはユーザーの自由に委ねられています。必ずどちらかのヘッダーが存在していなければなりません。両方とも存在していても構いません。

保護されたヘッダーと保護されないヘッダーの両方が存在する場合、実際の JOSE ヘッダーは、両方のヘッダーの要素を結合して作成されます。クレームの重複は認められません。

次の例は、JWS RFC[8] から引用したものです。

```
{
  "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogIm
              h0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
  "signatures": [
    {
```

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

```
      "protected": "eyJhbGciOiJSUzI1NiJ9",
      "header": { "kid": "2010-12-29" },
      "signature":
        "cC4hiUPoj9Eetdgtv3hF80EGrhuB_dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AA
        uHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAyn
        RFdiuB--f_nZLgrnbyTyWzO5vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB
        _eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqvhJ1phCnvWh6
        IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrBp0igcN_IoypGlU
        PQGe77Rw"
    },
    {
      "protected": "eyJhbGciOiJFUzI1NiJ9",
      "header": { "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
      "signature": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDx
                    w5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
    }
  ]
}
```

この例では、同一のペイロードに対して 2 つの署名 (RS256 署名と ES256 署名) をエンコードしています。

4.1.4.1 フラット化 JWS JSON シリアライゼーション

JWS JSON シリアライゼーションには、JWT の署名が 1 つしかない場合に使用できる簡略化された形式があります。この形式は、フラット化 JWS JSON シリアライゼーションと呼ばれています。フラット化シリアライゼーションでは、*signatures* 配列を削除し、*payload* 要素と同じレベルに 1 つの署名要素を置きます。

たとえば、上記の例から署名の 1 つを削除すると、次のようなフラット化 JSON シリアライゼーション オブジェクトになります。

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

4.2 トークンの署名と検証

トークンの署名と検証に使用されるアルゴリズムについては、第 7 章で詳しく説明します。署名付き JWT の使用方法は実際にはシンプルであり、これまでに説明した概念を適用するだけで効果的に使用できます。また、JWT の実装に便利な優れたライブラリも利用できます。ここでは、最も一般的な JavaScript ライブラリを使用する必須アルゴリズムと推奨アルゴリズムについて説明します。その他の一般的な言語やライブラリの例については、付属のコードを参照してください。

以下の例では、いずれも一般的に使われている `jsonwebtoken` JavaScript ライブラリを利用しています。

```
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 署名には共有シークレットが必要です。任意の文字列を使用できます。

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

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

トークンの検証も同様に簡単です。

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

jsonwebtoken ライブラリは、署名と有効期限に基づいてトークンの有効性をチェックします。この例では、トークンが作成されてから 5 秒後にチェックした場合、トークンは無効と見なされ、例外がスローされます。

4.2.2 RS256: RSASSA + SHA256

RS256 で署名されたトークンの署名と検証も同様に簡単です。共有シークレットではなく、秘密鍵と公開鍵のペアを使う点だけ異なります。RSA 鍵を作成する方法は多数あります。OpenSSL は、鍵の作成と管理に使用される最も一般的なライブラリの 1 つです。

```
# 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
```

どちらの PEM ファイルも単純なテキスト ファイルです。これらのファイルの内容を JavaScript ソース ファイルにコピーして貼り付けて、jsonwebtoken ライブラリに渡すことができます。

```
// 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: P-256 および SHA-256 を使用した ECDSA

ECDSA アルゴリズムでも公開鍵を利用します。ただし、アルゴリズムの計算方法が異なるため、鍵を生成する手順も異

なります。このアルゴリズムの名前 "P-256" は、使用されるアルゴリズムのバージョンを示しています (詳細は第 7 章を参照)。OpenSSL を使用して鍵を生成することもできます。

```
# 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
```

これらのファイルを開くと、データが非常に少ないことがわかります。これは、RSA と比べた場合の ECDSA のメリットの 1 つです (詳細は第 7 章を参照)。生成されるファイルも PEM 形式であるため、ソースに貼り付けるだけで使用できます。

```
// 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'],
});
```

JWT でのこれらのアルゴリズムの実際の利用例については、第 2 章を参照してください。

第 5 章

JSON Web Encryption (JWE)

JSON Web Signature (JWS) はデータを検証する手段を提供しますが、JSON Web Encryption (JWE) はデータを第三者に対して不透明にする手段を提供します。この場合、不透明とは読み取れないことを意味します。暗号化されたトークンを第三者が調べることはできません。この機能は、さまざまな用途に活用できます。

暗号化によって検証と同じ保証が得られるように見えますが、暗号化にはデータを読み取り不可にする機能もあるため、必ずしも同じではありません。その理由を理解するにはまず、JWE では、JWS と同様に、2 つの方式 (共有シークレット方式と公開鍵/秘密鍵方式) を利用できることに注目しましょう。

共有シークレット方式では、すべての当事者に共有シークレットを提供します。共有シークレットを持つ各当事者は、情報の**暗号化**と**復号**の両方を実行できます。これは JWS で共有シークレットを使用する場合と似ています。シークレットを持っている当事者は、署名付きトークンの検証と生成の両方を行うことができます。

一方、公開鍵/秘密鍵方式の仕組みは異なります。JWS では、秘密鍵を持っている当事者はトークンに署名して検証できますが、公開鍵を持っている当事者はトークンを検証することしかできません。一方 JWE では、秘密鍵を持っている当事者のみがトークンを**復号**できます。つまり、公開鍵の所有者はデータを**暗号化**できますが、そのデータを**復号** (および暗号化) できるのは、秘密鍵の所有者だけです。具体的には、JWE の場合、公開鍵を持っている当事者が新しいデータを取り込んで交換することができます。一方 JWS の場合には、公開鍵を持っている当事者はデータを検証できるだけで、新しいデータを取り込むことができません。要するに、JWE は JWS と同じ保証を提供しないため、トークンの交換において JWS に代わる役割を果たすことはできません。公開鍵/秘密鍵方式を使用している場合、JWS と JWE は互いに補完する関係にあります。

このことを分かりやすくするために、作成者と利用者の観点から考えてみましょう。作成者がデータの署名または暗号化を行い、利用者がデータを検証または復号します。JWT 署名の場合、秘密鍵は JWT 署名のために使用され、公開鍵はそれを検証するために使用されます。作成者は秘密鍵を保有し、利用者は公開鍵を保有します。データは秘密鍵の保有者から公開鍵の保有者にのみ送られます。一方、JWT 暗号化の場合は、公開鍵を使用してデータを暗号化し、秘密鍵を使用してデータを復号します。この場合、データは公開鍵の保有者から秘密鍵の保有者にのみ送られます。公開鍵の保有者は作成者であり、秘密鍵の保有者は利用者です。

	JWS	JWE
作成者	秘密鍵	公開鍵
利用者	公開鍵	秘密鍵

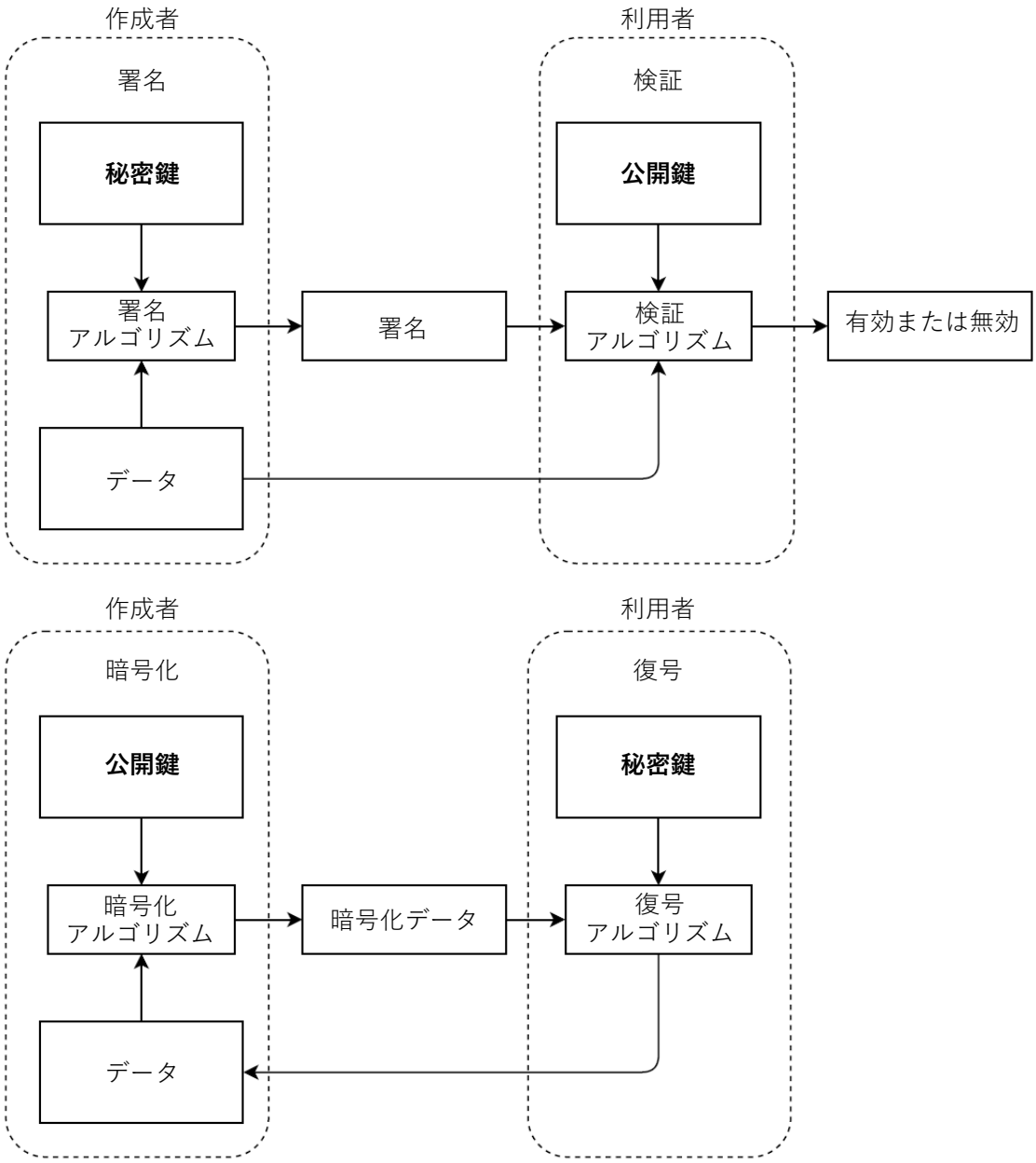

図 5.1: 公開鍵暗号方式を使用した署名と暗号化の比較

ここで、次のような疑問が浮かぶかもしれません。

JWE の場合、利用者にデータを送るすべての当事者に、秘密鍵を配布することはできないのでしょうか。利用者がデータを復号できれば、その利用者は同時にデータが有効であることも確認できます (復号できないデータは変更できないため)。

これは技術的には可能ですが、意味がありません。秘密鍵を共有することは、シークレットを共有することと同じです。したがって、秘密鍵を共有すると、公開鍵暗号方式は実質的に秘密鍵共有方式となり、公開鍵のメリットも失われます (公開鍵は秘密鍵から導出することができます)。

このような理由により、暗号化された JWT がネストされる、つまり暗号化された JWT が署名付き JWT のコンテナーとなる場合があります。そうすることで、両方の方式のメリットが得られるためです。

なお、以上の説明が当てはまるのは、利用者と作成者が別々である場合です。作成者がデータの利用者でもある場合、共有シークレットで暗号化された JWT は、暗号化および署名された JWT と同じ保証を提供します。

JWE で暗号化された JWT は、ネストされた署名付き JWT を含んでいるかどうかにかかわらず、認証タグを保持しています。このタグを使用すると、JWE 形式の JWT を検証できます。ただし、前述の問題により、この署名は JWS の署名と同じ用途では利用できません。このタグの目的は、パディング オラクル攻撃[1] や暗号文の操作を防ぐことです。

5.1 暗号化された JWT の構造

暗号化された JWT には、署名付きのセキュリティ保護のない JWT とは異なるコンパクトな表現があります (見やすいように改行を挿入しています)。

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-kFm1NJn8LE9XShH59_
i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ_deLKxGHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-
YxzZIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8Otv
zlV7elprCbuPhcCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-
cFPgwCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGljb3RoZQ.
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.
9hH0vgRfYgPnAHOd8stkvw
```

上記の例ではわかりにくいかもしれませんが、JWE コンパクト シリアライゼーションには 5 つの要素があります。JWS の場合と同様に、各要素がドットで区切られ、それぞれに含まれるデータが Base64 でエンコードされています。

コンパクト表現の 5 つの要素を順に説明します。

1. **保護されたヘッダー:** JWS ヘッダーと同様のヘッダー。
2. **暗号化鍵:** 暗号文やその他の暗号化されたデータを暗号化するために使用された対称鍵。この鍵は、ユーザーが指定した実際の暗号化用の鍵から導出されたものであるため、その鍵によって暗号化されます。
3. **初期化ベクトル:** 一部の暗号化アルゴリズムでは、追加の (通常、ランダムの) データが必要です。
4. **暗号化されたデータ (暗号文):** 暗号化されている実際のデータ。
5. **認証タグ:** アルゴリズムによって生成される追加データで、暗号文の内容が改ざんされていないかを検証するために使用できます。

署名が 1 つの JWS コンパクト シリアライゼーション形式の場合と同様に、JWE のコンパクト形式では、1 つの暗号化鍵を使用できます。

非対称暗号化 (公開鍵/秘密鍵暗号化) を使用する場合、一般的に、対称鍵を使用して実際の暗号化処理を行います。通常、非対称暗号化アルゴリズムは計算が非常に複雑であるため、長いデータ シーケンス (暗号文) の暗号化には最適ではありません。高速な対称暗号化と非対称暗号化の両方のメリットを活かす方法として、対称暗号化アルゴリズム用のランダム鍵を生成してから、その鍵を非対称暗号化アルゴリズムで暗号化する方法があります。これが上記の 2 番目の要素である、暗号化鍵です。

暗号化アルゴリズムの中には、渡されたすべてのデータを処理できるものもあります。そのようなアルゴリズムでは、暗号文が変更されていても (復号されていなくても)、暗号文を処理できます。認証タグを使用すると、これを防ぐことができます。そのため、認証タグは実質的に署名として機能します。とはいえ、前述のネストされた JWT の必要性がなくなるわけではありません。

5.1.1 鍵暗号化アルゴリズム

暗号化鍵が暗号化されるということは、同じ JWT に 2 つの暗号化アルゴリズムがあることを意味します。鍵の暗号化に使用できる暗号化アルゴリズムは次のとおりです。

- **RSA の変種**: RSAES PKCS #1 v1.5 (RSAES-PKCS1-v1_5)、RSAES OAEP、OAEP

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

- + MGF1 + SHA-256。
- **AES の変種**: 128 ビットから 256 ビットまでの AES 鍵ラップ、128 ビットから 256 ビットまでの AES Galois Counter Mode (GCM)。
- **楕円曲線暗号の変種**: Concat KDF を使用した Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES) 鍵共有、および上記のいずれかの非 GCM の AES 変種を使用して鍵を事前にラップする変種。
- **PKCS#5 の変種**: PBES2 (パスワードベースの暗号化) + HMAC (SHA-256 ～ 512) + 128 ～ 256 ビットの非 GCM の AES 変種。
- **直接**: 暗号化鍵の暗号化なし (CEK を直接使用)。

これらのアルゴリズムは、いずれも JWA の仕様では必須とされていません。以下は、仕様で (実装するよう) 推奨されているアルゴリズムです。

- **RSAES-PKCS1-v1_5** (今後推奨から除外される予定)
- 既定の設定の **RSAES-OAEP** (今後必須化される予定)
- **AES-128 鍵ラップ**
- **AES-256 鍵ラップ**
- Concat KDF を使用した **ECDH-ES** (今後必須化される予定)
- **ECDH-ES + AES-128 鍵ラップ**
- **ECDH-ES + AES-256 鍵ラップ**

これらのアルゴリズムの中には、追加のヘッダー パラメーターが必要なものもあります。

5.1.1.1 鍵管理モード

JWE の仕様では、さまざまな鍵管理モードが定義されています。鍵管理モードとは、ペイロードを暗号化するために使用する鍵を決定する方法です。JWE の仕様では、特に次の鍵管理モードについて説明されています。

- **鍵ラッピング**: 対称暗号化アルゴリズムを使用し、対象受信者用のコンテンツ暗号化鍵 (CEK) を暗号化します。

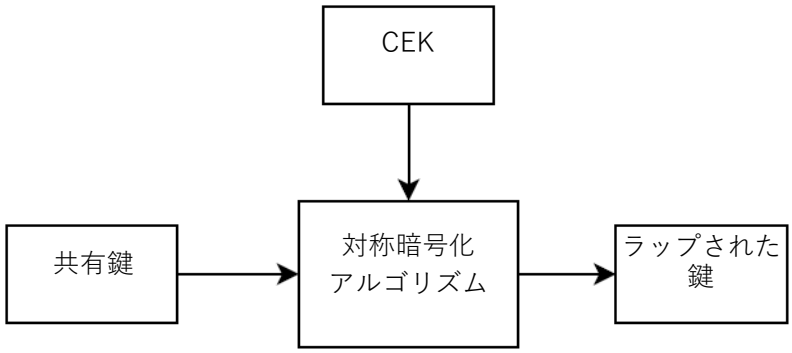

図 5.2: 鍵ラッピング

- **鍵暗号化**: 非対称暗号化アルゴリズムを使用し、対象受信者用の CEK を暗号化します。

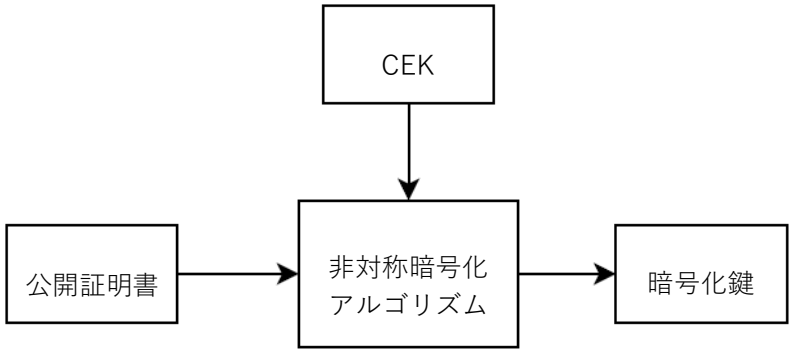

図 5.3: 鍵暗号化

- **直接鍵共有**: 鍵共有アルゴリズムを使用して CEK を選択します。

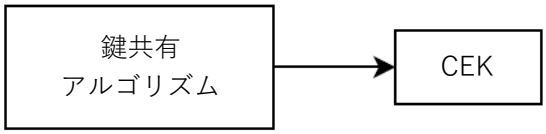

図 5.4: 直接鍵共有

- **鍵共有と鍵ラッピング**: 対称暗号化アルゴリズムを使用して暗号化された対称 CEK を、鍵共有アルゴリズムを使用して選択します。

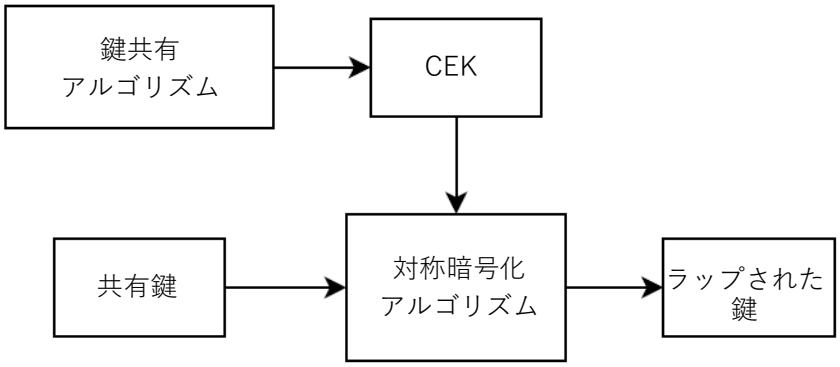

図 5.5: 鍵共有と鍵ラッピング

- **直接暗号化**: ユーザー定義の対称共有鍵を CEK として使用します (鍵の派生または生成なし)。

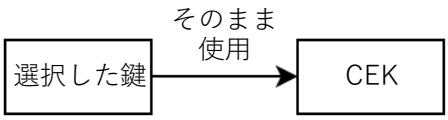

図 5.6: 直接暗号化

これは用語の問題になりますが、各管理モードの違いを理解し、それぞれに適切な名前を付けることが重要です。

5.1.1.2 コンテンツ暗号化鍵（CEK）と JWE 暗号化鍵

また、CEK と JWE 暗号化鍵の違いを理解することも重要です。CEK は、ペイロードの暗号化に使用される実際の鍵です。暗号化アルゴリズムは、CEK と平文を受け取り、暗号文を生成します。一方、JWE 暗号化鍵は、暗号化された形の CEK か、空のオクテット列のどちらかになります (選択したアルゴリズムによって異なります)。JWE 暗号化鍵が空の場合、アルゴリズムでは外部から提供された鍵を使用して、データを直接復号するか (直接暗号化)、実際の CEK を計算します (直接鍵共有)。

5.1.2 コンテンツ暗号化アルゴリズム

ペイロードを実際に暗号化するために使用されるコンテンツ暗号化アルゴリズムは次のとおりです。

- **AES CBC + HMAC SHA**: 暗号ブロック連鎖 (CBC) モードの AES 128 ～ 256 ビットと、検証用の HMAC + SHA-256 ～ 512。
- **AES GCM**: GCM を使用した AES 128 ～ 256。

これらのうち必須とされているのは、AES-128 CBC + HMAC SHA-256 と AES-256 CBC + HMAC SHA-512 の 2 つだけです。GCM を使用した AES-128 および AES-256 の変種は、推奨アルゴリズムです。

これらのアルゴリズムについては、第 7 章で詳しく説明します。

5.1.3 ヘッダー

JWS やセキュリティ保護のない JWT のヘッダーと同様に、ヘッダーには JWT をライブラリによって正しく処理するために必要な情報がすべて含まれています。JWE の仕様では、JWS で定義されている登録済みクレームの意味がその用途に合わせて変更されているほか、いくつかのクレームが追加されています。以下に新しいクレームと変更されているクレームを示します。

- **alg**: JWS と同じですが、CEK の暗号化と復号に使用されるアルゴリズムを定義するものである点が異なります。つまり、このアルゴリズムは、後でコンテンツの暗号化に使用される鍵を暗号化するために使用されます。
- **enc**: CEK を使用してコンテンツを暗号化するために使用されるアルゴリズムの名前。
- **zip**: 暗号化の前に、暗号化するデータに適用される圧縮アルゴリズム。このパラメーターはオプションです。指定しなかった場合、圧縮が実行されません。通常は `DEF` (一般的な deflate アルゴリズム[2]) がこの値に使用されます。
- **jku**: JWS と同じですが、CEK を暗号化するために使用する公開鍵を指す点が異なります。
- **jkw**: JWS と同じですが、CEK を暗号化するために使用する公開鍵を指す点が異なります。
- **kid**: JWS と同じですが、CEK を暗号化するために使用する公開鍵を指す点が異なります。
- **x5u**: JWS と同じですが、CEK を暗号化するために使用する公開鍵を指す点が異なります。
- **x5c**: JWS と同じですが、CEK を暗号化するために使用する公開鍵を指す点が異なります。
- **x5t**: JWS と同じですが、CEK を暗号化するために使用する公開鍵を指す点が異なります。
- **x5t#S256**: JWS と同じですが、CEK を暗号化するために使用する公開鍵を指す点が異なります。
- **typ**: JWS と同じです。
- **cty**: JWS と同じですが、暗号化されたコンテンツの種類である点が異なります。
- **crit**: JWS と同じですが、このヘッダーのパラメーターを参照する点が異なります。

使用している暗号化アルゴリズムによっては、追加のパラメーターが必要になる場合があります。これについては、各アルゴリズムについて取り上げる節で説明します。

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

5.1.4 コンパクト シリアライゼーションに使用されるアルゴリズムの概要

本章の冒頭で、JWE コンパクト シリアライゼーションについて簡単に触れました。これは基本的に、表示可能なテキスト形式にエンコードされ、ドット (.) で区切られた 5 つの要素で構成されます。JWE コンパクト シリアライゼーション形式の JWT を構築する基本的な計算手順を以下に示します。

1. 選択したアルゴリズムで必要な場合 (`alg` クレーム)、必要な長さの乱数を生成します。この乱数を生成する際には、ランダム性に関する一定の暗号要件に準拠する必要があります。RFC 4086[3] を参照するか、暗号論的に妥当性が確認されている乱数生成器を使用してください。
2. 鍵管理モード[4] に基づき、コンテンツ暗号化鍵を決定します。
   - **直接鍵共有**: 鍵共有アルゴリズムと乱数を使用して、CEK を計算します。
   - **鍵共有と鍵ラッピング**: 鍵共有アルゴリズムと乱数を使用して、CEK をラップするために使用する鍵を計算します。
   - **直接暗号化**: CEK が対称鍵です。
3. 鍵管理モードに基づき、JWE 暗号化鍵を決定します。
   - **直接鍵共有および直接暗号化**: JWE 暗号化鍵は空です。
   - **鍵ラッピング、鍵暗号化、鍵共有と鍵ラッピング**: 受信者用の CEK を暗号化します。これにより、JWE 暗号化鍵が生成されます。
4. 選択したアルゴリズムを使用して、必要とする長さの初期化ベクトル (IV) を計算します。不要な場合は、この手順をスキップします。
5. 必要に応じて、平文のコンテンツを圧縮します (`zip` ヘッダー クレームを使用)。
6. CEK、IV、追加認証データ (AAD) を使用してデータを暗号化します。これにより、暗号化されたコンテンツ (JWE 暗号文) と認証タグが生成されます。AAD はコンパクトでないシリアライゼーションの場合にのみ使用します。
7. 次のようにコンパクト表現を構築します。

```
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 シリアライゼーション

JWE では、コンパクトなシリアライゼーション形式に加えて、コンパクトでない JSON 表現も定義されています。この表現を使用すると、コンパクトでなくなる代わりに柔軟性が得られます。特に、同時に複数の公開鍵を使用し、複数の受信者用のコンテンツを暗号化できるという利点があります。これは、JWS JSON シリアライゼーションで使用できる複数の署名に似ています。

JWE JSON シリアライゼーションは、以下のメンバーを持つ JSON オブジェクトを、表示可能なテキスト形式にエンコードしたものです。

- **protected**: この JWE 形式の JWT によって保護される (検証はされるが暗号化はされない) ヘッダー クレームを Base64 でエンコードした JSON オブジェクト。これはオプションです。この要素か unprotected ヘッダーのどちらかが必要です。
- **unprotected**: JSON オブジェクトとして保護されない (検証されない) ヘッダー クレーム。この要素は Base64 でエンコードされません。これはオプションです。この要素か protected ヘッダーのどちらかが必要です。
- **iv**: 初期化ベクトルの Base64 文字列。これはオプションです (アルゴリズムが必要とする場合にのみ使用します)。
- **aad**: 追加認証データ。暗号化アルゴリズムによって保護される (検証される) 追加データの Base64 文字列。暗号化の手順で AAD を指定しない場合、このメンバーは不要です。
- **ciphertext**: 暗号化されたデータを Base64 でエンコードした文字列。
- **tag**: 暗号化アルゴリズムによって生成された認証タグの Base64 文字列。

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

- **recipients**: JSON オブジェクトの JSON 配列。各オブジェクトには、各受信者による復号に必要な情報が含まれています。

recipients 配列内のオブジェクトには、次のメンバーがあります。

- **header**: 保護されないヘッダー クレームの JSON オブジェクト。これはオプションです。
- **encrypted_key**: Base64 でエンコードされた JWE 暗号化鍵。JWE 暗号化鍵が使用される場合にのみ使用します。

受信者用の JWE 形式の JWT を復号するのに使用される実際のヘッダーは、存在する各ヘッダーを結合して作成されます。クレームの重複は認められません。

暗号化鍵の形式については、第 6 章 (JSON Web Key) で説明します。

次の例は、RFC 7516 (JWE) から引用したものです。

```
{
    "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"
}
```

この JWE JSON シリアライゼーション形式の JWT では、1 つのペイロードを 2 つの受信者に対して使用しています。暗号化アルゴリズムは AES-128 CBC + SHA-256 です。これは protected ヘッダーから取得できます。

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

各受信者用のすべてのクレームを結合することで、各受信者用の最終的なヘッダーが作成されます。

最初の受信者:

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

2 番目の受信者:

```
{
   "alg":"A128KW",
```

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

5.1.5.1 フラット化 JWE JSON シリアライゼーション

JWS と同様に、JWE ではフラットな JSON シリアライゼーション形式も定義されています。このシリアライゼーション形式は、受信者が 1 つの場合にのみ使用できます。この形式では、recipients 配列が header と encrypted_key のペアに置き換えられます (つまり、recipients 配列内の 1 つのオブジェクトの鍵が使用されます)。

以下は、前節の例に最初の受信者だけを含めることでフラット化した表現です。

```
{
    "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 トークンの暗号化と復号

次の例は、広く使われている node-jose[5] ライブラリを使用して暗号化を実行する方法を示しています。このライブラリは、jsonwebtoken (JWS の例で使用) よりもはるかに広い範囲を対象としているため、少し複雑です。

5.2.1 導入: node-jose を使用した鍵の管理

次の例では、さまざまな形式の暗号化鍵を使用する必要があります。これは、keystore (鍵ストア) を介して node-jose により管理します。keystore は鍵を管理するオブジェクトです。ここでは、鍵をいくつか生成して鍵ストアに追加し、後の例で使用できるようにしておきます。JWS の例で見たように、jsonwebtoken ライブラリでは、このような抽象化は必要ありませんでした。keystore による抽象化は、node-jose の実装に含まれる要素の 1 つです。他の言語やライブラリでも、同様の抽象化が見られます。

空の鍵ストアを作成し、さまざまな種類の鍵をいくつか追加するには、次の手順を実行します。

```
// 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' }),
    keystore.generate('EC', 'P-256', { kid: 'example-3' }),
```

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

```
];
```

node-jose では、鍵の生成は簡単です。JWE および JWS で使用できるすべての鍵タイプがサポートされています。この例では、シンプルな AES 128 ビット鍵、RSA 2048 ビット鍵、曲線 P-256 を使用した楕円曲線鍵の 3 種類の鍵を作成します。これらの鍵は、暗号化と署名の両方に使用できます。公開鍵/秘密鍵ペアをサポートする鍵の場合、生成される鍵は秘密鍵です。公開鍵を取得するには、次の呼び出しを実行します。

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

公開鍵は JWK 形式で保存されます。既存の鍵を読み込むこともできます。

```
// 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 (鍵) + AES-128 GCM (コンテンツ)

AES-128 Key Wrap と AES-128 GCM は、対称鍵アルゴリズムです。つまり、暗号化と復号に同じ鍵が必要なアルゴリズムです。先ほど生成した "example-1" の鍵は、対称鍵です。AES-128 Key Wrap では、この鍵はランダムに生成された鍵をラップするために使用された後、AES-128 GCM アルゴリズムを使用してコンテンツを暗号化するために使用されます。この鍵は直接使用することもできます (直接暗号化モード)。

```
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));
}
```

node-jose ライブラリは、主に Promise[6] を使用して動作します。a128gcm が返すオブジェクトは Promise です。createEncrypt 関数は、渡された任意のコンテンツを暗号化できます。つまり、コンテンツが JWT である必要はありません (ほとんどの場合)。そのため、JSON.stringify は、この関数にデータを渡す前に呼び出す必要があります。

5.2.3 RSAES-OAEP (鍵) + AES-128 CBC + SHA-256 (コンテンツ)

createEncrypt 関数の呼び出し方法の違いは、この関数に渡されるオプションだけです。したがって、公開鍵/秘密鍵ペアも簡単に使用できます。createEncrypt に対称鍵を渡す代わりに、公開鍵または秘密鍵を渡すだけです (暗号化に必要なのは公開鍵だけですが、公開鍵は秘密鍵から導出することができます)。読みやすくするために、ここでは秘密鍵を使用しますが、実際には、このステップでは公開鍵を使用することがほとんどです。

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

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 では、実際の暗号化アルゴリズムを選択します。前述のように、使用できるのは 2 種類 (各鍵長の AES CBC + HMAC SHA と AES GCM) だけです。

5.2.4 ECDH-ES P-256 (鍵) + AES-128 GCM (コンテンツ)

楕円曲線の API は RSA の API と同じです。

```
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 ネストされた JWT: P-256 および SHA-256 を使用した ECDSA (署名) + RSAES-OAEP (暗号化鍵) + AES-128 CBC + SHA-256 (暗号化コンテンツ)

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

ネストされた JWT では、署名付き JWT を暗号化関数に渡すことになるため、多少の手間がかかります。署名と暗号化の手順を手動で実行する必要があるためです。ここでは、署名した後で暗号化する、という順序で行うことに注意してください。技術的には逆の順序で行うことも可能ですが、最初に JWT に署名しておけば、作成したトークンが署名削除攻撃を受ける危険を排除できます。

```
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;
}
```

上記の例からわかるように、node-jose は署名にも使用できます。署名にはその他のライブラリ (jsonwebtoken など) を使用することも可能です。しかし、node-jose の必要性を考えると、依存関係を増やし、別々の API を使用しても意味がありません。

> 最初に署名手順を実行できるのは、JWE では認証付き暗号を必須としているからです。つまり、暗号化アルゴリズムによって署名手順も行う必要があります。JWE では認証が必要であるにもかかわらず、JWS と JWE を効果的に組み合わせることができる理由については、第 5 章の冒頭で説明しました。他の方式 (一般的な暗号化 + 署名) では、通常、暗号化の後で署名を行います。これは、暗号化攻撃の原因となる暗号文の操作を防ぐためです。このことは、JWE で認証タグが必須とされている理由でもあります。

5.2.6 復号

復号も暗号化と同じくシンプルです。暗号化と同様に、異なるデータ形式のペイロードを明示的に変換する必要があります。

```
// 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);
```

```
});
```

RSA アルゴリズムと楕円曲線アルゴリズムの復号は類似しており、対称鍵ではなく秘密鍵を使用します。適切な `kid` クレームを持つ鍵ストアがある場合は、その鍵ストアを `createDecrypt` 関数に渡すだけで、適切な鍵を検索させることができます。したがって、上記の例では、まったく同じコードを使用して復号することができます。

```
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);
       });
```

第 6 章

JSON Web Key (JWK)

では、JWT、JWS、JWE の全体像を把握するために、JSON Web Key (JWK) の仕様を見ていきましょう。この仕様では、署名や暗号化に使用される鍵のさまざまな表現が取り上げられています。どの鍵にも確立された表現がありますが、JWK 仕様では、JSON Web Algorithms (JWA) 仕様でサポートされているすべての鍵に対して統一された表現を提供することを目的としています。鍵の表現形式を統一することで、共有を容易化するとともに、他の鍵交換形式の複雑さの影響を排除できます。

JWS および JWE は、X.509 証明書という別の種類の鍵形式をサポートしています。これは非常に一般的で、JWK よりも多くの情報を保持することができます。X.509 証明書は JWK に埋め込むことができ、X.509 証明書から JWK を構築することができます。

鍵はさまざまなヘッダー クレームで指定されます。JWK のリテラルは `jwk` クレームに記述されます。一方、`jku` クレームは、URL で指定される場所に保管されている鍵のセットを参照することができます。これらのクレームは両方とも JWK 形式です。

JWK の例を以下に示します。

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

6.1 JSON Web Key の構造

JSON Web Key は、鍵に必要なパラメーターを記述する一連の値を保持する JSON オブジェクトです。パラメーターは鍵の種類によって異なります。共通のパラメーターを以下に示します。

- **kty**: "key type" の略です。鍵の種類を識別するためのクレームです。サポートされている種類は、楕円曲線鍵を表す `EC`、RSA 鍵用の `RSA`、対称鍵を表す `oct` です。このクレームは必須です。
- **use**: 鍵の使用目的を指定します。指定できる使用目的は、`sig` (署名用) と`enc` (暗号化用) の 2 つです。このクレームはオプションです。同じ鍵を暗号化と署名に使用する場合、このメンバーを含めることはできません。
- **key_ops**: 鍵の詳細な使用方法を指定する文字列値の配列。指定できる値は、`sign`、`verify`、`encrypt`、`decrypt`、`wrapKey`、`unwrapKey`、`deriveKey`、`deriveBits` です。中には、組み合わせて使用できない値もあります。たとえば、`sign` と `verify` は同じ鍵に使用できますが、`sign` と `encrypt` は同時に使用できません。このクレームはオプションです。さらに、`use` クレームと同時に使用することはできません。両方のクレームを使用する場合、内容に一貫性があることが必要です。
- **alg**: "algorithm" の略です。この鍵とともに使用するアルゴリズムを表します。JWE または JWS の処理で使用できる任意のアルゴリズムを指定できます。このクレームはオプションです。
- **kid**: "key id" の略です。この鍵の一意の識別子です。これを使用して、JWE ヘッダーまたは JWS ヘッダー内の `kid` クレームと鍵を照合したり、アプリケーションのロジックに基づいて鍵のセットから鍵を選択したりできます。このクレームはオプションです。同じ鍵セット内の 2 つの鍵が同じ `kid` を持つことができるのは、それらの `kty` クレームが異なり、使用目的が同じである場合だけです。
- **x5u**: PEM エンコード形式の X.509 公開鍵証明書または証明書チェーンを指す URL。他にもオプションのクレームが含まれている場合、そのクレームの内容に証明書の内容との一貫性があることが必要です。このクレームはオプションです。
- **x5c**: Base64-URL でエンコードされた X.509 DER 証明書または証明書チェーン。証明書チェーンはこの証明書の配列として表されます。最初の証明書は、この JWK が参照する証明書である必要があります。この JWK 内に存在する他のすべてのクレームに、最初の証明書の値との一貫性があることが必要です。このクレームはオプションです。
- **x5t**: X.509 証明書の DER エンコードの SHA-1 サムプリント/フィンガープリントを Base64-URL エンコードしたもの。このサムプリントが参照する証明書に、この JWK のクレームの内容との一貫性があることが必要です。このクレームはオプションです。
- **x5t#S256**: `x5t` クレームと同じですが、X.509 証明書の SHA-256 サムプリントが含まれる点が異なります。

`x`、`y`、`d`(本章の冒頭の例を参照) など、その他のパラメーターは、鍵アルゴリズムに固有のものです。一方、RSA 鍵は、`n`、`e`、`dp` などのパラメーターを持ちます。これらのパラメーターの意味については、各鍵アルゴリズムについて詳しく説明している第 7 章で解説します。

6.1.1 JSON Web Key Set

JWK の仕様では、鍵のグループが認められています。これを "JWK Set" と呼びます。このセットに複数の鍵が含まれています。グループとしての鍵の意味と、鍵の順序の意味は、ユーザーが定義します。

JSON Web Key Set は、`keys` メンバーを持つ JSON オブジェクトです。keys メンバーは JWK の JSON 配列です。

JWK Set の例:

```
{
    "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"
       }
    ]
}
```

この例には、2 つの公開鍵があります。最初のものは楕円曲線鍵であり、`use` クレームによって暗号化処理での使用に制限されています。2 番目のものは RSA 鍵であり、`alg` クレームによって特定のアルゴリズム (`RS256`) に関連付けられています。つまり、この 2 番目の鍵は署名に使用されます。

第 7 章

JSON Web Algorithms

お気づきかもしれませんが、本書にはこの章への参照が数多くあります。その理由は、JWT を支える有用な機能の多くが、JWT で使用されるアルゴリズムを基盤としているからです。JWT の構造も重要ですが、これまで説明してきたさまざまな興味深い使用例は、アルゴリズムなしには不可能です。この章では、現在 JWT で使用されている最も重要なアルゴリズムについて説明します。JWT は深く理解していなくても十分効果的に使用できます。そのためこの章は、パズルの最後のピースを知りたいという好奇心のある方を対象としています。

7.1 一般的なアルゴリズム

以下のアルゴリズムは、JWT、JWS、JWE の仕様内で、さまざまな形で利用されます。Base64-URL などのアルゴリズムは、コンパクト シリアライゼーション形式とコンパクトでないシリアライゼーション形式で使用されます。SHA-256 などのその他のアルゴリズムは、署名、暗号化、鍵のフィンガープリントに使用されます。

7.1.1 Base64

Base64 はバイナリからテキストへのエンコードを行うアルゴリズムです。このアルゴリズムの主な目的は、オクテット列を表示可能な一連の文字列に変換することです。その代償として、データ サイズは増加します。数学用語で言えば、Base64 は基数 (Radix) 256 の数列を基数 64 の数列に変換します。Base64 という名前は、*Radix* の代わりに *Base* という用語を使用できることに由来します。

> 注: Base64 は JWT 仕様では使用されていません。JWT で使用されるのは、後ほど本章で説明する *Base64-URL* という変種です。

Base64 で一連の任意の数字をテキストに変換する方法を理解するには、まずテキスト エンコード方式について知る必要があります。テキスト エンコード方式では、数字を文字にマップします。このマッピングは任意であり、Base64 エンコーディングの場合は実装で定義できますが、Base64 の事実上の標準は RFC 4648[1] です。

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

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

Base64 のエンコード後の各文字は、元のデータの 6 ビット分に対応しています。エンコードは、4 文字のグループごとに行います。つまり、元のデータの 24 ビット分を 1 つにまとめて、4 つの Base64 文字としてエンコードします。元のデータは 8 ビット値の数列であるため、この 24 ビット分のデータは、左から順に 3 つの 8 ビット値を連結して作ります。

Base64 エンコーディング:

8 ビット値 x 3 -> 24 ビットの連結データ -> 6 ビット文字 x 4

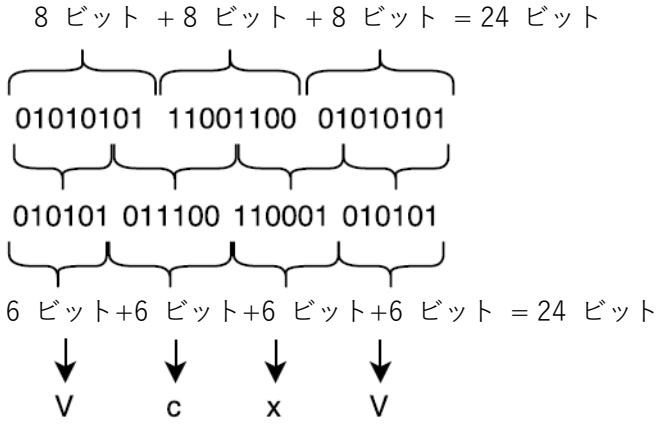

図 7.1: Base64 エンコーディング

入力データのオクテットの数が 3 で割り切れない場合、エンコード対象のデータの最後の部分が、24 ビットに満たないデータになります。この場合、連結した入力データにゼロを付加し、整数個の 6 ビット値のグループを作ります。このとき、次の 3 つの可能性が考えられます。

1. 24 ビットをすべて入力として使用できる場合は、特別な処理は行いません。
2. 16 ビットを入力として使用できる場合は、3 つの 6 ビット値を作り、最後の 6 ビット値の右にゼロを付加します。エンコード後の文字列には、8 ビットの入力がないことを明示するために、パディング文字の = を 1 つ追加します。
3. 8 ビットを入力として使用できる場合は、2 つの 6 ビット値を作り、最後の 6 ビット値の右にゼロを付加します。エンコード後の文字列には、16 ビットの入力がないことを明示するために、パディング文字の = を 2 つ追加します。

実装によっては、パディング文字 (=) は省略可能とされています。この手順を逆の順序で実行すると、パディング文字

の有無にかかわらず、元のデータが生成されます。

7.1.1.1 Base64-URL

標準の Base64 変換表に含まれている一部の文字は、URL セーフではありません。Base64 は、テキスト フィールドに任意のデータを渡すときに便利なエンコード方式です。URL に含めると問題が発生するのは 2 文字だけであるため、Base64 の URL セーフな変種は容易に実装できます。+と / を - と _ にそれぞれ置き換えるだけです。

7.1.1.2 サンプル コード

次のサンプルでは、ごく限られた機能を持つ Base64-URL エンコーダーを実装しています。このサンプルは速度よりも簡潔さを考慮して書かれています。

```
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

JWT の仕様で使用されている Secure Hash Algorithm (SHA) は、FIPS-180[2] で定義されています。SHA-1[3] 系のアルゴリズムと混同しないよう注意してください。SHA-1 系は 2010 年 に廃止されました。SHA-1 系と区別するために、これを *SHA-2* と呼ぶことがあります。

RFC 4634 で定義されているアルゴリズムは、SHA-224、SHA-256、SHA-384、SHA-512 です。JWT にとって重要なのは、SHA-256 と SHA-512 です。ここでは SHA-256 に焦点を当て、他の変種との違いを説明します。

多くのハッシュ アルゴリズムと同様に、SHA では、入力を固定サイズのチャンクで処理し、一連の算術演算を適用してから、直前の結果を使用して演算を繰り返し実行して結果を累積するという仕組みを採用しています。固定サイズの入力チャンクがすべて処理されると、ダイジェストが計算されることになっています。

SHA 系のアルゴリズムは、衝突を回避し、入力がわずかに違うだけでもまったく異なる出力を生成するように設計されました。この理由から、SHA はセキュアであるとみなされています。さまざまな入力の衝突を発見したり、生成されたダイジェストから元の入力を計算したりすることは、計算上不可能です。

このアルゴリズムには、事前に定義された一連の関数が必要です。

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

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

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

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

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

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

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

これらの関数は仕様で定義されてます。`rotr` 関数はビット ローテーション (右方向) を実行します。

また、このアルゴリズムでは、メッセージが所定の長さ (64 の倍数) である必要があります。したがって、パディング処理が必要です。パディング アルゴリズムでは次の処理が実行されます。

1. 元のメッセージの末尾にバイナリの `1` を 1 つ追加します。以下に例を示します。

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

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

```
    01011111 01010101 10101010 00111100 1
```

2. メッセージの長さが次の方程式の解と同じになるように、N 個のゼロを追加します。

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

3. 元のメッセージのビット数を 64 ビット整数に変換して追加します。

```
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
```

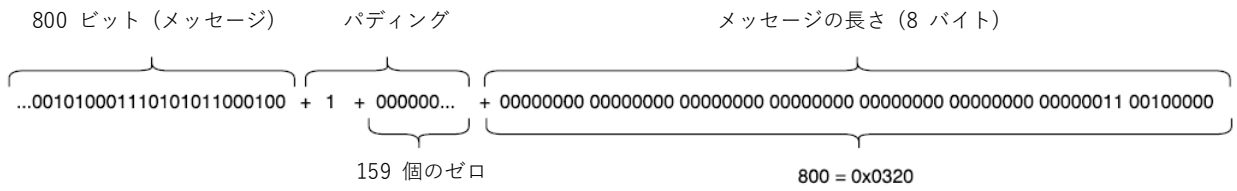

図 7.2: SHA のパディング

JavaScript での簡単な実装例を以下に示します。

```
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;
}
```

生成されるパディングされたメッセージは、512 ビットのブロックで処理されます。以下の実装例は、仕様で説明されている処理手順に基づいて作成されています。どの処理も 32 ビット整数に対して実行されます。

```
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;
        }

        //(...)
}
```

変数 `k` は、仕様で定義されている一連の定数を保持します。

最終的な結果は、変数 `h_[0..7]` に格納されます。あとはこれを読みやすい形式で表示するだけです。

```
if(returnBytes) {
```

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

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

これでも機能しますが、上記の実装例は最適なものではありません (2^{32} より長いメッセージはサポートされません)。

SHA-2 系の他の変種 (SHA-512 など) を使用する場合は、各反復で処理するブロックのサイズを変更し、定数とそのサイズを変えるだけです。特に、SHA-512 を利用するには、64 ビット演算が必要です。上記の実装例で SHA-512 を利用するには、64 ビット演算を行うための別のライブラリが必要です (JavaScript は 32 ビットのビット演算と 64 ビットの浮動小数点演算しかサポートしません)。

7.2 署名アルゴリズム

7.2.1 HMAC

ハッシュベース メッセージ認証コード (HMAC)[4] は、暗号学的ハッシュ関数 (前述の SHA 系などの関数) と鍵を使用して、特定のメッセージの認証コードを作成します。つまり HMAC ベースの認証方式では、入力としてハッシュ関数、メッセージ、秘密鍵を受け取り、出力として認証コードを生成します。暗号学的ハッシュ関数の特性により、秘密鍵がないとメッセージを変更できません。そのため、HMAC は認証とデータの完全性確認の両方の目的に使用できます。

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

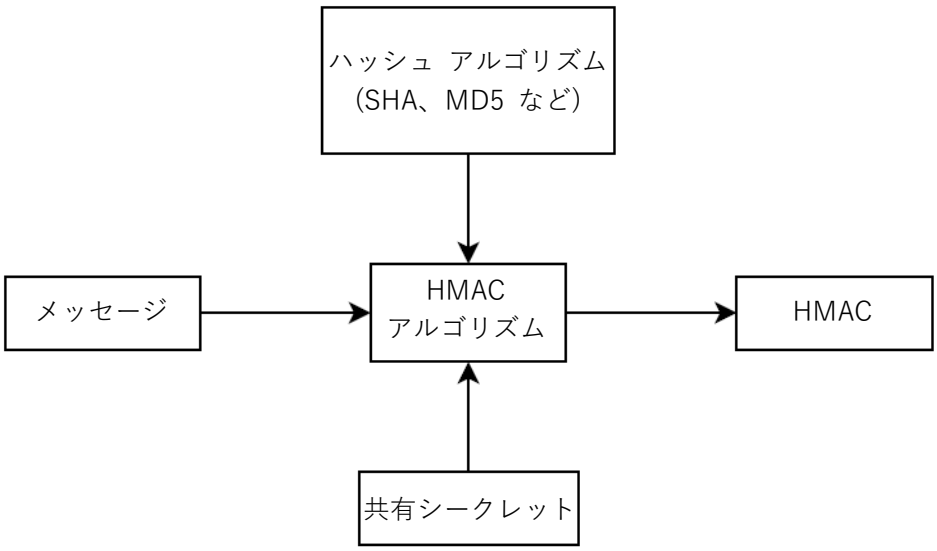

図 7.3: HMAC

ハッシュ関数が脆弱な場合、悪意のあるユーザーによって認証コードの有効性が危険にさらされるおそれがあります。したがって、HMAC を使用する場合には、強力なハッシュ関数を選択する必要があります。SHA-2 系の関数は、現在の標準としては十分に強力ですが、将来的には危うくなる可能性もあります。MD5 は、過去に広く使用されていた別の暗号ハッシュ関数であり、HMAC に使用できます。ただし、衝突攻撃やプレフィックス攻撃に対して脆弱である可能性があります。このような攻撃のおそれがあるからといって、MD5 は HMAC での使用に適していないわけではありません。しかし、より強力なアルゴリズムを容易に利用できるため、その利用を検討することをお勧めします。

HMAC アルゴリズムは、1 行に収まるほどシンプルなものです。

```
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' は、秘密鍵 K から次のように計算されます。

K が B より短い場合、K が B の長さになるまでゼロを追加します。その結果が K' です。K が B より長い場合、H を K に適用します、結果は K' になります。K がちょうど B バイトである場合、そのまま使用します (K は K' と同値)。

JavaScript での実装例を以下に示します。

```
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;
}
```

HMAC を使用してメッセージを検証する場合は、HMAC を計算し、その結果をメッセージに付属する HMAC と比較するだけです。これを行うには、すべての当事者 (メッセージの作成者とメッセージの検証のみを行う者) が秘密鍵を知っている必要があります。

7.2.1.1 HMAC + SHA256 (HS256)

JWS 仕様の HS256 署名アルゴリズムを実装するのに必要なものは、Base64-URL、SHA-256、HMAC の知識だけです。その知識をもとに、これまでに作成したすべてのサンプル コードを組み合わせることで、完全な署名付き JWT を作成できます。

```
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}`;
}
```

この関数では (オブジェクトであるかどうかの検証を除き)、ヘッダーやペイロードは検証されないことに注意してください。この関数は、次のようにして呼び出すことができます。

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

生成された JWT を JWT.io のデバッガー[5] に貼り付けて、デコードと検証の結果を確認します。

この関数は、署名アルゴリズムの例を示すために第 4 章で使用した関数と非常によく似ています。以下が第 4 章のものです。

```
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}`;
```

検証も簡単です。

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

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

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

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

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

署名を JWT から分離し、新しい署名を計算します。新しい署名が JWT に含まれている署名と一致する場合、その署名は有効です。

上記の関数は、次のように使用できます。

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

このコードは、本書に付属するサンプル[6] の `hs256.js` ファイルにあります。

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

7.2.2 RSA

RSA は今日最も広く使用されている暗号方式の 1 つです。1977 年に Ron Rivest 氏、Adi Shamir 氏、Leonard Adleman 氏によって開発され、3 氏のイニシャルを取って RSA と命名されました。RSA の重要な特徴は、その非対称性にあります。つまり、暗号化に使用される鍵が、復号に使用される鍵と異なるということです。この方式は公開鍵暗号方式 (PKI) と呼ばれます。この方式では、公開鍵が暗号化鍵、秘密鍵が復号鍵となります。

署名に関して言えば、秘密鍵は情報に署名するために使用され、公開鍵は特定の秘密鍵によってその情報が署名されたことを検証 (情報の内容はわかりません) するために使用されます。

RSA アルゴリズムには、署名用と暗号化用のバリエーションがあります。ここではまず一般的なアルゴリズムを取り上げてから、JWT で使用されるさまざまなバリエーションを確認します。

多くの暗号アルゴリズム、特に RSA は、特定の数学的演算を行うことが比較的困難であることを利用したものです。RSA では、主な数学的手法として素因数分解[7] を使用します。素因数分解とは、複数の数値が乗算された数値から元の数値を求める数学問題です。言い換えると、整数の因数とは、乗算するとその整数になる一連の整数のペアです。

```
整数 = 因数_1 x 因数_2
```

この問題は一見、簡単に見えます。数が小さい場合は確かに簡単です。たとえば、数が `35` の場合は、次の解になります。

```
35 = 7 x 5
```

7 または 5 の乗算表を知っていれば、掛け合わせたときに `35` になる数を簡単に見つけることができます。ある整数の因数を見つける単純なアルゴリズムとして、次のようなものを考えることができます。

1. `n` を因数分解する数とします。
2. `x` を `2` 以上 `n / 2` 以下の数とします。
3. `n` を `x` で除算し、余りが `0` かどうかを確認します。余りが 0 である場合、因数は `x` と商の値になります。
4. `x` が上限の `n / 2` に達するまで、`x` を `1` ずつ増やしながら、手順 3 を繰り返し実行します。こうすることで、`n` のすべての因数が見つかります。

これは基本的に総当たりで因数を見つける方法です。ご想像のとおり、このアルゴリズムは非常に非効率的です。

> このアルゴリズムを改良した試行除算と呼ばれる手法では、`x` の条件をより厳密に設定します。具体的には、`x` の上限を `sqrt(n)` と定義し、`x` を `1` ずつ増加させるのではなく、`x` にもっと大きな素数を代入します。これらの条件を使用することで、なぜこのアルゴリズムを正しく効率化できるかを示すことは難しくありませんが、それは本書の範囲外です。

さらに効率的なアルゴリズムも存在しますが、そのような効率的なアルゴリズムや今日のコンピューターを使用しても、計算上、因数分解することが難しい数があります。そのような数を `n` として選択すると、因数分解問題は複雑になります。たとえば、`n` が 2 つの素数[8] を掛け合わせた結果である場合 (その 2 つの素数だけが唯一可能な因数であるとします)、その因数を見つけることははるかに困難になります。

> `n` が 2 つの非素数を掛け合わせた結果であるならば、その因数ははるかに簡単に見つかります。なぜなら、非素数は (定義上) その非素数以外の約数を持ち、その約数もその約数を乗算して算出された数値の約数であるからです。つまり、ある数の因数の約数は、その数の因数でもあります。別の言い方をすれば、`n` に素数以外の因数がある場合、n は 2 つ以上の因数を持つことになります。そのため、`n` が 2 つの素数を乗算した結果であれば、その因数は 2 つだけということになります (この場合、n は最小数の因数を持つ、それ自体素数ではない数です)。因数の数が少ないほど、因数を見つけることは困難です。

異なる大きな素数を 2 つ選んで乗算すると、さらに大きな別の数になります (これは半素数と呼ばれます)。この数には、因数分解が難しいという特殊な性質があります。一般数体ふるい法[9] など、最も効率的な因数分解アルゴリズムを使用しても、大きな素数を掛け合わせてできる大きな数を現実的な時間内に因数分解することはできません。どれだけの規模かと言えば、たとえば 2009 年に完了した 768 ビットの数値 (232 桁の十進数) の因数分解[10] では、コンピューター クラスターを使用して 2 年間にわたり計算が行われました。RSA の一般的な利用例では、2048 ビット以上の数値が使用されています。

---

[7] https://ja.wikipedia.org/wiki/%E7%B4%A0%E5%9B%A0%E6%95%B0%E5%88%86%E8%A7%A3
[8] https://ja.wikipedia.org/wiki/%E7%B4%A0%E6%95%B0
[9] https://en.wikipedia.org/wiki/General_number_field_sieve
[10] http://eprint.iacr.org/2010/006

ショアのアルゴリズム[11] は特殊な因数分解アルゴリズムであり、将来大きな変化をもたらす可能性があります。ほとんどの因数分解アルゴリズムは、古典的な性質を持ち、標準的なコンピューターで処理されますが、ショアのアルゴリズムでは量子コンピューターを利用します[12]。量子コンピューターは、ある種の量子現象の性質を利用して、いくつかの古典的な演算を高速化できます。特に、ショアのアルゴリズムは因数分解を高速化し、その複雑性を (指数関数時間ではなく) 多項式時間の複雑性の問題に変換することができます。これは、現在のどの古典的アルゴリズムよりもはるかに効率的です。このようなアルゴリズムが量子コンピューター上で実行可能になった場合、現在の RSA 鍵は役に立たなくなることが予想されます。ショアのアルゴリズムに必要な実用的な量子コンピューターはまだ開発されていませんが、この分野は現在活発に研究されています。

現在のところ、大きな半素数の素因数分解は実質的に計算不能とされていますが、それが数学的に証明されているわけではありません。つまり、将来、現実的な時間内に素因数分解を解くアルゴリズムが発見される可能性があります。RSA についても同じことが言えます。

以上を念頭に、RSA アルゴリズムの説明に入ります。基本原理は次の式で表されます。

$$(m^e)^d \equiv m (mod\ n)$$

図 7.4: RSA の基本式

上記の式を満たす 3 つの非常に大きな整数 e、d、n を計算で求めることは不可能です。RSA アルゴリズムでは、他の 2 つの数がわかっていても d を見つけることが困難であることを利用しています。つまり、この式は一方向性関数に変換することができます。その際、d を秘密鍵、n と e を公開鍵と考えることができます。

7.2.2.1 e、d、n の選択

1. 2 つの異なる素数 p と q を選択します。
   - p と q の候補を選択する際には、暗号的に安全な乱数生成器を使用する必要があります。安全な乱数生成器でないと、攻撃者にこれらの数を発見されるおそれがあります。
   - 素数をランダムに生成する方法はないため、2 つの乱数を選択してから、その乱数に対して素数判定を行う必要があります。決定的素数判定法はコストがかかることがあるため、一部の実装例では確率的判定法を利用しています。この判定法では、擬素数が見つかる確率を考える必要があります。
   - p と q は、大きさは近いが同一ではなく、長さが数桁違うものである必要があります。
2. n は p × q の結果です。これは上記の式のモジュラス値 (mod) です。n のビット数がアルゴリズムの鍵長になります。
3. n に対するオイラーのトーシェント関数[13] を計算します。n は半素数であるため、これは `n - (p + q - 1)` という式で簡単に計算できます。この値を`phi (n)` とします。
4. 次の条件を満たす e を選択します。
   - `1 < e < phi(n)`
   - e と `phi (n)` は互いに素である
5. 次の式を満たす d を選択します。

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

図 7.5: RSA の基本式

公開鍵は値 n と e で構成されます。秘密鍵は値 n と d で構成されます。値 p、q、`phi(n)` は、d を見つける手がかりになるため、破棄するか非公開にする必要があります。

上記の式から、e と d が数学的に対称であることは明らかです。手順 5 の式は次のように書き換えることができます。

$$d\ .\ e \equiv 1 (mod\ \ \phi(n))$$

図 7.6: e と d の対称性

[11] https://en.wikipedia.org/wiki/Shor%27s_algorithm
[12] https://ja.wikipedia.org/wiki/%E9%87%8F%E5%AD%90%E3%82%B3%E3%83%B3%E3%83%94%E3%83%A5%E3%83%BC%E3%82%BF
[13] https://ja.wikipedia.org/wiki/%E3%82%AA%E3%82%A4%E3%83%A9%E3%83%BC%E3%81%AE%CF%86%E9%96%A2%E6%95%B0

ここで、値 e と n を公開しても、なぜ RSA が安全なのかと疑問に思うかもしれません。これらの値を使用して、d を発見することはできないのでしょうか。ここで重要なのは、合同算術には可能な解が複数あるということです。選択した d が上記の式を満たす限り、どの値も有効です。値が大きいほど、発見が困難になります。そのため、値 e または d のどちらか一方のみを公開すれば、RSA は機能します。また、どちらか一方の値を選択できるのであれば、できる限り小さい値を公開する値として選ぶことができます。そうすることで、アルゴリズムの安全性を損なうことなく計算が高速化されます。

7.2.2.2 基本的な署名方法

RSA での署名は、次の手順で実行します。

1. ハッシュ関数によって署名するメッセージからメッセージ ダイジェストを生成します。
2. このダイジェストを d 乗して n (秘密鍵の一部) で除算した余りを求めます。
3. 結果を署名としてメッセージに添付します。

公開鍵を持っている受信者がメッセージの信頼性を検証する際には、次のように逆順に処理を行います。

1. 署名を e 乗し、n で除算した余りを求めます。結果の値は参照ダイジェスト値です。
2. 署名手順と同じハッシュ関数を使用して、メッセージからメッセージ ダイジェストを生成します。
3. 手順 1 と 2 の結果を比較します。一致する場合、署名者は秘密鍵を所有しているはずです。

この署名/検証方式は "Signature Scheme with Appendix: SSA (付加物を使用した署名方式)" と呼ばれています。この方式では、メッセージを検証するために元のメッセージを使用する必要があります。つまり、署名からメッセージを復元することはできません (メッセージと署名が分離されています)。

7.2.2.3 RS256: SHA-256 を使用した RSASSA PKCS1 v1.5

RSA の仕組みについて基本的な概念を説明したので、次はその変種の 1 つ、SHA-256 を使用した PKCS#1 RSASSA v1.5 を取り上げます。これは JWA 仕様では RS256 と呼ばれています。

Public Key Cryptography Standard #1 (PKCS #1)[14] 仕様では、RSA アルゴリズムに基づく一連のプリミティブ、フォーマット、暗号化方式が定義されています。これらの要素を組み合わせることで、最新のコンピューティング プラットフォームで利用できる詳細な RSA 実装を実現できます。RSASSA はこの仕様で定義されている方式の 1 つです。この方式を利用することで、署名で RSA を使用することができます。

7.2.2.3.1 アルゴリズム

署名を作成するには、次の手順を実行します。

1. **EMSA-PKCS1-V1_5-ENCODE** プリミティブをメッセージ (オクテット配列) に適用します。出力結果は**エンコードされたメッセージ**です。このプリミティブはハッシュ関数 (通常、SHA-256 などの SHA 系のハッシュ関数) を使用します。このプリミティブは、期待されるエンコード後のメッセージの長さを受け取ります。ここでは、これはオクテット単位の RSA 値 n の長さ (鍵の長さ) です。
2. エンコードされたメッセージに **OS2IP** プリミティブを適用します。出力結果は**整数メッセージ表現**です。OS2IP は、Octet-String to Integer Primitive (オクテット列から整数に変換するプリミティブ) の頭字語です。
3. 秘密鍵を使用して、**RSASP1** プリミティブを整数メッセージ表現に適用します。出力は**整数署名表現**です。
4. **I2OSP** プリミティブを適用して、整数署名表現をオクテット配列 (**署名**) に変換します。I2OSP は、Integer to Octet-String Primitive (整数からオクテット列に変換するプリミティブ) の頭字語です。

以上のプリミティブに基づく JavaScript での実装例を以下に紹介します。

```
/**
 * Produces a signature for a message using the RSA algorithm as defined
 * in PKCS#1.
 * @param {privateKey} RSA private key, an object with
 *                     three members: size (size in bits), n (the modulus) and
 *                     d (the private exponent), both bigInts
 *                     (big-integer library).
 * @param {hashFn} the hash function as required by PKCS#1,
```

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

```
 *                it should take a Uint8Array and return a Uint8Array
 * @param {hashType} A symbol identifying the type of hash function passed.
 *                   For now, only "SHA-256" is supported. See the "hashTypes"
 *                   object for possible values.
 * @param {message} A String or Uint8Array with arbitrary data to sign
 * @return {Uint8Array} The signature as a Uint8Array
 */
export function sign(privateKey, hashFn, hashType, message) {
    const encodedMessage =
        emsaPkcs1v1_5(hashFn, hashType, privateKey.size / 8, message);
    const intMessage = os2ip(encodedMessage);
    const intSignature = rsasp1(privateKey, intMessage);
    const signature = i2osp(intSignature, privateKey.size / 8);
    return signature;
}
```

署名を検証するには、次の手順を実行します。

1. **OS2IP** プリミティブを署名 (オクテット配列) に適用します。出力結果は**整数署名表現**です。
2. **RSAVP1** プリミティブを前の結果に適用します。このプリミティブは、公開鍵も入力として受け取ります。出力結果は**整数メッセージ表現**です。
3. **I2OSP** プリミティブを前の結果に適用します。このプリミティブは、期待される長さを入力として受け取ります。この長さは、鍵のモジュラス値の長さ (オクテット数) と一致している必要があります。出力結果は**エンコードされたメッセージ**です。
4. **EMSA-PKCS1-V1_5-ENCODE** プリミティブを、検証するメッセージに適用します。出力結果は別の**エンコードされたメッセージ**です。このプリミティブはハッシュ関数 (通常、SHA-256 などの SHA 系のハッシュ関数) を使用します。このプリミティブは、期待されるエンコード後のメッセージの長さを受け取ります。ここでは、これはオクテット単位の RSA 値 `n` の長さ (鍵の長さ) です。
5. 2 つのエンコードされたメッセージ (手順 3 と 4 のメッセージ) を比較します。一致する場合、署名は有効ですが、一致しない場合は無効です。

JavaScript での実装例は次のようになります。

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

    const intSignature = os2ip(signature);
```

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

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

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

7.2.2.3.1.1 EMSA-PKCS1-v1_5 プリミティブ

このプリミティブは次の 3 つの要素を受け取ります。

- メッセージ
- 目的とする出力結果の長さ
- 使用するハッシュ関数 (手順 2 で挙げられているいずれかの関数)

1. 選択したハッシュ関数をメッセージに適用します。
2. 次の ASN.1 構造の DER エンコードを生成します。

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

`digest` は手順 1 の出力結果であり、`DigestAlgorithm` は次のいずれかです。

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

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

3. 求められている出力結果の長さが手順 3 の結果に `11` を加えたものよりも短い場合 (reqLength < step2Length + 11)、プリミティブは結果の生成に失敗し、エラー メッセージ "intended encoded message length too short" (目的とするエンコード後のメッセージ長が短すぎます) を出力します。
4. オクテット `0xFF` を `requested length + step2Length - 3` 回繰り返して、オクテット配列を生成します。このオクテット配列は `PS` と呼ばれます。
5. エンコードされた最終メッセージ (EM) を生成します (`||` は連結演算子)。

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

通常、ASN.1 OID は独自の仕様で定義されます。そのため、PKCS#1 の仕様に SHA-256 OID はありません。SHA-1 および SHA-2 OID は RFC 3560[15] で定義されています。

7.2.2.3.1.2 OS2IP プリミティブ

OS2IP プリミティブはオクテット配列を受け取り、整数表現を出力します。

- X_1、X_2、...、X_n を、最初から最後までの入力オクテットとします。
- 結果を次のように計算します。

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

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

図 7.7: OS2IP の出力結果

7.2.2.3.1.3 RSASP1 プリミティブ

RSASP1 プリミティブは、秘密鍵とメッセージ表現を受け取り、署名表現を生成します。

- `n` と `d` を秘密鍵の RSA 値とします。
- `m` をメッセージ表現とします。

1. メッセージ表現が 0 から `n - 1` の範囲にあることを確認します。
2. 結果を次のように計算します。

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

図 7.8: RSASP1 の出力結果

> PKCS#1 では代替手段として、計算上便利な秘密鍵の格納方法が定義されています。この方法では、秘密鍵を `n` と `d` として格納するのではなく、事前計算された異なる値の組み合わせを格納し、特定の処理に使用します。これらの値を特定の処理で直接使用することで、計算を大幅に高速化できます。ほとんどの秘密鍵は、この方法を使用して格納されています。ただし、秘密鍵を `n` と `d` として格納する方法も適正な方法です。

7.2.2.3.1.4 RSAVP1 プリミティブ

RSAVP1 プリミティブは、公開鍵と整数署名表現を受け取り、整数メッセージ表現を生成します。

- `n` と `e` を公開鍵の RSA 値とします。
- `s` を整数署名表現とします。

1. メッセージ表現が 0 から `n - 1` の範囲にあることを確認します。
2. 結果を次のように計算します。

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

図 7.9: RSAVP1 の出力結果

7.2.2.3.1.5 I2OSP プリミティブ

I2OSP プリミティブは整数表現を受け取り、オクテット配列を生成します。

- `len` をオクテット配列の期待される長さとします。
- `x` を整数表現とします。

1. $x > 256^{len}$ である場合、整数が大きすぎ、引数が正しくありません。
2. 整数の 256 進数表現を計算します。

$$\chi \ = \chi_1 \ 256^{len-1} + \chi_2 \ 256^{len-2} + \cdots + \chi_{len-1} \ 256 + \chi_{len}$$

図 7.10: I2OSP による分解

3. それぞれの項の係数 x_{len-i} を順に取得します。これが結果のオクテット配列です。

7.2.2.3.2 サンプル コード

RSA は任意精度の演算が必要であるため、ここでは big-integer[16] JavaScript ライブラリを使用します。

`OS2IP` プリミティブおよび `I2OSP` プリミティブは非常にシンプルです。

```
function os2ip(bytes) {
```

[16] https://www.npmjs.com/package/big-integer

```
    let result = bigInt();

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

    return result;
}

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

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

    return result;
}
```

I2OSP プリミティブは基本的に、数を 256 進法[17] の要素に分解します。

RSASP1 プリミティブは基本的に単一の演算であり、アルゴリズムの基礎を成しています。

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

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

検証目的には、代わりに RSAVP1 プリミティブを使用します。

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

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

最後に、EMSA-PKCS1-v1_5 プリミティブでメッセージをエンコードされたパディング付き表現に変換します。演算の

[17] https://ja.wikipedia.org/wiki/%E4%BD%8D%E5%8F%96%E3%82%8A%E8%A8%98%E6%95%B0%E6%B3%95

大部分はここで実行されます。

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

    const digest = hashFn(message, true);

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

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

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


    }

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

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

ここでは、簡潔にするために SHA-256 のみをサポートしています。他のハッシュ関数は、適切な OID を追加するだけで追加されます。

`signPkcs1v1_5` 関数は、すべてのプリミティブをまとめ、署名を実行します。

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

```
    const intMessage = os2ip(encodedMessage);
    const intSignature = rsasp1(privateKey, intMessage);
    const signature = i2osp(intSignature, privateKey.size / 8);
    return signature;
}
```

これを使用して JWT に署名するには、シンプルなラッパーが必要です。

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

    header.alg = 'RS256';

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


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

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

この関数は、HMAC の節で示した `HS256` 用の `jwtEncode` 関数とよく似ています。検証も簡単です。

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

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

    const encodedMessage =
```

```
        emsaPkcs1v1_5(hashFn, hashType, publicKey.size / 8, message);

    return uint8ArrayEquals(encodedMessage, verificationMessage);
```

これを使用して JWT を検証するには、シンプルなラッパーが必要です。

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

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

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

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

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

簡潔にするために、ここでは秘密鍵と公開鍵を、2 つの別々の数値を持つ JavaScript オブジェクトとして渡しています。その 2 つの数値は、秘密鍵ではモジュラス値 (n) と秘密指数 (d)、公開鍵ではモジュラス値 (n) と公開指数 (e) となります。これは通常の PEM エンコード[18] 形式とは異なります。詳細については、`rs256.js` ファイルを参照してください。

OpenSSL を使用すると、PEM 鍵からこれらの数値を書き出すことができます。

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

OpenSSL を使用すると、RSA 鍵をゼロから生成することもできます。

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

生成したら、上記のコマンドを使用して、PEM 形式から数値を書き出すことができます。

`testkey.js` ファイルに埋め込まれている秘密鍵の値は、本書に付属する samples ディレクトリ内にある `testkey.pem` ファイルに含まれているものです。対応する公開鍵は、`pubtestkey.pem` ファイルにあります。

rs256.js のサンプル[19] を実行した出力を JWT.io[20] の JWT の領域にコピーします。続いて `pubtestkey.pem` の内容をコピーして同じページの公開鍵の領域に貼り付けると、JWT が問題なく検証されます。

[18] https://en.wikipedia.org/wiki/Privacy-enhanced_Electronic_Mail
[19] https://github.com/auth0/jwt-handbook-samples/blob/master/rs256.js
[20] https://jwt.io

7.2.2.4 PS256: SHA-256 および MGF1 と SHA-256 を使用した RSASSA-PSS

RSASSA-PSS は、もう 1 つの RSA ベースの SSA です。"PSS" は Probabilistic Signature Scheme (確率的署名方式) を表し、通常の決定的手法とはまったく異なります。この方式では、暗号論的に安全な乱数発生器を使用します。安全な乱数発生器が利用できない場合も、署名処理および検証処理によって決定的手法と同等のレベルのセキュリティを実現できます。そのため、RSASSA-PSS は、理想的な条件のもとでは、PKCS v1.5 署名よりも優れたセキュリティを実現します。ただし、PSS 方式も PKCS v 1.5 方式も、実際に破られたことはありません。

RSASSA-PSS は Public Key Cryptography Standard #1 (PKCS #1)[21] で定義されており、以前のバージョンの規格では使用できません。

7.2.2.4.1 アルゴリズム

署名を作成するには、次の手順を実行します。

1. **EMSA-PSS-ENCODE** プリミティブをメッセージに適用します。このプリミティブは、鍵のモジュラス値のビット数から 1 を引いた数のパラメーターを受け取ります。出力結果は**エンコードされたメッセージ**です。
2. エンコードされたメッセージに **OS2IP** プリミティブを適用します。出力結果は**整数メッセージ表現**です。OS2IP は、Octet-String to Integer Primitive (オクテット列から整数に変換するプリミティブ) の頭字語です。
3. 秘密鍵を使用して、**RSASP1** プリミティブを整数メッセージ表現に適用します。出力は**整数署名表現**です。
4. **I2OSP** プリミティブを適用して、整数署名表現をオクテット配列 (**署名**) に変換します。I2OSP は、Integer to Octet-String Primitive (整数からオクテット列に変換するプリミティブ) の頭字語です。

以上のプリミティブに基づく JavaScript での実装例を以下に紹介します。

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

    const encodedMessage = emsaPssEncode(hashFn,
                                          hashType,
                                          mgf1.bind(null, hashFn),
                                          256 / 8, //size of hash
                                          privateKey.size - 1,
                                          message);
    const intMessage = os2ip(encodedMessage);
    const intSignature = rsasp1(privateKey, intMessage);
    const signature = i2osp(intSignature, privateKey.size / 8);
    return signature;
}
```

署名を検証するには、次の手順を実行します。

1. **OS2IP** プリミティブを署名 (オクテット配列) に適用します。出力結果は**整数署名表現**です。
2. **RSAVP1** プリミティブを前の結果に適用します。このプリミティブは、公開鍵も入力として受け取ります。出力結果は**整数メッセージ表現**です。
3. **I2OSP** プリミティブを前の結果に適用します。このプリミティブは、期待される長さを入力として受け取ります。この長さは、鍵のモジュラス値の長さ (オクテット数) と一致している必要があります。出力結果は**エンコードされたメッセージ**です。
4. 検証対象のメッセージと直前の手順の結果に **EMSA-PSS-VERIFY** プリミティブを適用します。このプリミティブは、署名が有効か無効かを出力します。このプリミティブはハッシュ関数 (通常、SHA-256 などの SHA 系のハッシュ関数) を使用します。このプリミティブは、鍵のモジュラス値のビット数から 1 を引いた数のパラメーターを受け取ります。

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

    const intSignature = os2ip(signature);
```

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

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

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

7.2.2.4.1.1 MGF1: マスク生成関数

マスク生成関数は、任意の長さの入力を受け取り、可変長の出力を生成します。ハッシュ関数と同様に、これは決定的関数であり、同じ入力に対して同じ出力を生成します。ただし、ハッシュ関数とは異なり、出力の長さは可変です。Mask Generation Function 1 (`MGF1`) アルゴリズムは、Public Key Cryptography Standard #1 (PKCS #1)[22] で定義されています。

`MGF1` は入力として、シード値と、出力結果の目標長さを受け取ります。出力の最大長は 2^{32} と定義されます。`MGF1` では内部的に、設定可能なハッシュ関数を使用します。`PS256` では、このハッシュ関数が `SHA-256` と指定されています。

1. 目標長さが 2^{32} より大きい場合は、"mask too long (マスクが長すぎます)" というエラーが発生して停止します。
2. 0 から `ceiling(intendedLength / hashLength) - 1` (目標長さをハッシュ関数の出力長で割った値を最小の整数に切り上げてから 1 を引いた値) までの範囲で、次の演算を反復します。
    1. `c = i2osp(counter, 4)` とします。`counter` は反復カウンターの現在の値です。
    2. `t = t.concat(hash(seed.concat(c)))` とします。`t` は反復間で保持されます。`hash` は選択したハッシュ関数 (`SHA-256`)、`seed` は入力シード値です。
3. 関数の出力結果として、`t` の最後の値から目標長さ左端までのオクテットを出力します。

7.2.2.4.1.2 EMSA-PSS-ENCODE プリミティブ

このプリミティブは次の 2 つの要素を受け取ります。

- オクテット列として符号化されるメッセージ。
- 目的とする出力結果の最大長 (ビット単位)。

このプリミティブは、次の要素でパラメーター表現できます。

- ハッシュ関数。`PS256` の場合、これは `SHA-256` です。
- マスク生成関数。`PS256` の場合、これは `MGF1` です。
- 内部で使用するソルトの目標長さ。

これらのパラメーターはすべて `PS256` で指定されているため、設定可能ではありません。ここでは説明上、定数とみなします。

入力として使用される目標長さは、ビット単位で表現されます。以下の例では、次の定義を使用します。

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

1. 入力がハッシュ関数の最大長より大きい場合は、処理を停止します。そうでない場合は、メッセージにハッシュ関数を適用します。

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

2. メッセージの目標長さが、ハッシュの長さにソルトの長さを加えたものに 2 を加えた値よりも短い場合は、停止してエラーを発生します。

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

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

3. ソルトの長さと同じランダムなオクテット列を生成します。
4. 8 個の値ゼロのオクテットを、メッセージのハッシュとソルトに連結します。

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

5. ハッシュ関数を直前の手順の結果に適用します。

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

6. 目標とする出力結果最大長さからソルト長、ハッシュ長、および 2 を引いた値と同じ長さの値ゼロのオクテット列を生成します。

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

7. 直前の手順の結果をオクテット 0x01 とソルトに連結します。

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

8. 手順 5 の結果にマスク生成関数を適用し、この関数の目標長さを手順 7 の結果の長さに設定します (マスク生成関数は指定された長さのパラメーターを受け取ります)。

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

9. 手順 7 と 8 の結果に XOR 演算を適用した結果を計算します。

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

10. 直前の演算結果の長さが 8 の倍数でない場合は、減算するとその長さが 8 の倍数になるビット数の差を求め、このビットを左から 0 に設定します。

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

11. 直前の手順の結果を手順 5 の結果とオクテット 0xBC に連結します。これが出力結果です。

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

7.2.2.4.1.3 EMSA-PSS-VERIFY プリミティブ

このプリミティブは次の 3 つの要素を受け取ります。

- 検証対象のメッセージ。
- エンコードされた整数メッセージ形式の署名。
- エンコード後の整数メッセージの目標最大長。

このプリミティブは、次の要素でパラメーター表現できます。

- ハッシュ関数。PS256 の場合、これは SHA-256 です。
- マスク生成関数。PS256 の場合、これは MGF1 です。
- 内部で使用するソルトの目標長さ。

これらのパラメーターはすべて PS256 で指定されているため、設定可能ではありません。ここでは説明上、定数とみなします。

入力として使用される目標長さは、ビット単位で表現されます。以下の例では、次の定義を使用します。

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

1. 選択したハッシュ関数を使用して、検証対象のメッセージをハッシュ化します。

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

2. 目標長さがハッシュ長 + ソルト長 +2 より小さい場合、署名を無効とみなします。

```
if(expectedLength < (digest1.length + saltLength + 2)) {
```

```
    return false;
}
```

3. 署名のエンコードされたメッセージの最後のバイトの値が `0xBC` であることを確認します。

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

4. エンコードされたメッセージを 2 つの要素に分割します。最初の要素の長さは `expectedLength - hashLength - 1` です。2 番目の要素は最初の要素の終わりから始まり、長さは `hashLength` です。

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

5. `masked` の左端の `8 * expectedLength - expectedLengthBits` (期待されるビット長から要求されるビット長を引いたもの) ビットが 0 であることを確認します。

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

6. 手順 4 で抽出した 2 番目の要素 (ダイジェスト) を、選択した MGF 関数に渡します。`expectedLength - hashLength - 1` の長さを持つ結果を要求します。

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

7. 手順 4 (`masked`) で抽出した最初の要素の各バイトに対して、直前の手順 (`dbMask`) で計算した要素の対応するバイトを使用して、XOR 関数を適用します。

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

8. 直前の手順で計算した要素内の最初のバイトから左端の `8 * expectedLength - expectedLengthBits` ビットを 0 に設定します。

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

9. 直前の手順で計算した要素の左端の `expectedLength - hashLength - saltLength - 2` バイトが 0 であることを確認します。また、ゼロのグループの後に続く最初の要素が `0x01` であることを確認します。

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

10. 直前の手順 (`db`) で計算した要素の最後の `saltLength` オクテットからソルトを抽出します。

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

11. 8 個の値ゼロのオクテット、手順 1 で計算したハッシュ、および直前の手順で抽出したソルトを連結して、新しいエンコードされたメッセージを算出します。

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

12. 直前の手順で計算した要素のハッシュを計算します。

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

13. 直前の手順で計算した要素を、手順 4 で抽出した 2 番目の要素と比較します。一致する場合、署名は有効です

が、一致しない場合は無効です。

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

7.2.2.4.2 サンプル コード

このアルゴリズムは RSASSA の変種であるため、必要なコードの大部分は、RS256 の実装例で既に示したものです。唯一の違いは、EMSA-PSS-ENCODE、EMSA-PSS-VERIFY、および MGF1 プリミティブが追加される点です。

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

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

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

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

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

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

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

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

export function emsaPssVerify(hashFn,
                              hashType,
                              mgf,
                              saltLength,
```

```
                                expectedLengthBits,
                                message,
                                verificationMessage) {
    const expectedLength = Math.ceil(expectedLengthBits / 8);

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

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

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

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

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

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

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

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

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

完全なサンプル[23] は、ps256.js、rsassa.js、pkcs.js にあります。testkey.js ファイルに埋め込まれている秘密鍵の値は、本書に付属する samples ディレクトリ内にある testkey.pem ファイルに含まれているものです。対応する公開鍵は、pubtestkey.pem ファイルにあります。鍵の作成については、RS256 の例を参照してください。

7.2.3 楕円曲線

楕円曲線 (EC) アルゴリズムは、RSA アルゴリズムと同様に、一定の条件下では解くことが難しい数学問題の一種を利用したものです。ここで言う難しさとは、十分なリソースがあれば解を発見できる可能性はあるものの、実際には解を発

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

見するのが困難であるという意味です。RSA は因数分解問題[24] の難しさ (大きな複素数の素因数を見つけること) を利用したものですが、楕円曲線アルゴリズムは楕円曲線上の離散対数問題の難しさを利用したものです。

楕円曲線は次の方程式で表されます。

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

図 7.11: 楕円曲線の式

a と b に異なる値を設定すると、次のような曲線が得られます。

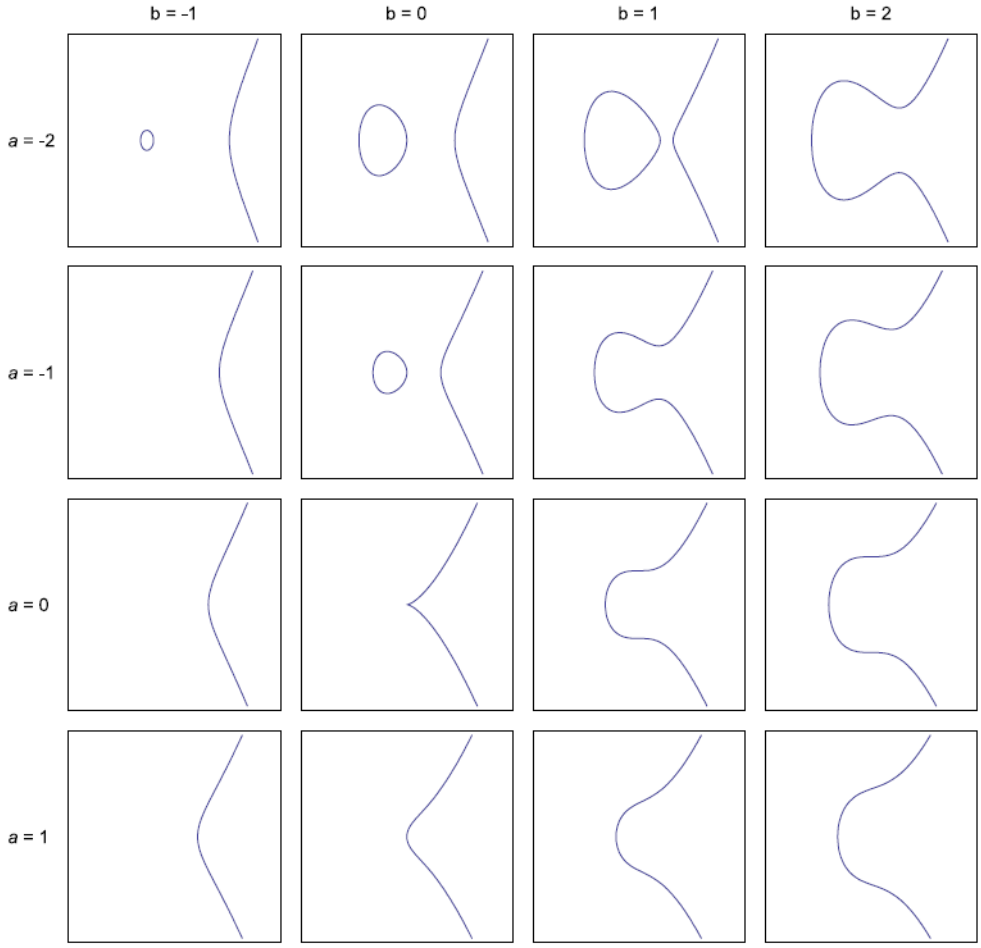

図 7.12: 楕円曲線の例

Wikimedia から引用したパブリック ドメインの画像です[25]。

RSA とは異なり、楕円曲線アルゴリズムは特定の有限体に対して定義されます。EC 暗号で重要なのは、標数 2 の体と素体です。JWA 仕様では素体のみを使用しているため、本書では素体を取り上げます。

> 数学で言う「体」とは、4 つの基本的な算術演算 (減算、加算、乗算、除算) が定義されている要素の集合を指します。

"有限" とは、楕円曲線暗号で無限の実数の集合に対して演算を行うのではなく、有限の数の集合に対して演算を行うことを意味します。

素体とは、素数個 p の要素を含む体です。すべての要素と算術演算は、p (素数個の要素) を法として実行されます。

[24] https://ja.wikipedia.org/wiki/%E7%B4%A0%E5%9B%A0%E6%95%B0%E5%88%86%E8%A7%A3
[25] https://commons.wikimedia.org/wiki/File:EllipticCurveCatalog.svg

有限体を扱う場合、数学的演算の実行に使用する算法が変化します。特に、通常の対数の代わりに離散対数[26] を使用する必要があります。対数は次の形の式で表されます (k の値)。

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

図 7.13: 指数関数

離散対数を計算できる効率的かつ汎用的なアルゴリズムは発見されていません。離散対数が暗号に最適なのは、このような限界があるためです。この限界を利用し、楕円曲線を暗号に使用することで、セキュアな非対称暗号化/署名処理を実現できます。

離散対数問題の求解困難性は、問題で使用する体のパラメーターに依存します。つまり、効果的な楕円曲線暗号を実現するには、特定のパラメーターを慎重に選ぶ必要があるということです。過去には楕円曲線アルゴリズムの不適切な使用が攻撃の標的とされたこともあります[27]。

楕円曲線暗号には、RSA で使用される大きな鍵と同等のセキュリティ レベルをより小さな鍵で実現できるという興味深い特徴があります。そのため、メモリが限られたデバイスでも暗号化を実行できます。一般に、256 ビットの楕円曲線鍵は、3072 ビットの RSA 鍵と同等の暗号強度を持つと言われています。

7.2.3.1 楕円曲線の演算

楕円曲線署名を実装するには、楕円曲線演算の実装が必要です。基本的な演算は、点の加算、点の 2 倍算、点のスカラー倍算の 3 つです。3 つの演算をすべて実行することで、同一曲線上に有効な点が生成されます。

7.2.3.1.1 点の加算

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

$$\lambda \equiv \frac{y_q - y_p}{x_q - x_p} \ (mod\ q)$$

$$x_r \equiv \lambda^2 - x_p - x_p \ (mod\ q)$$
$$y_r \equiv \lambda(x_p - x_r) - y_p \ (mod\ q)$$

図 7.14: 点の加算

7.2.3.1.2 点の 2 倍算

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

$$\lambda \equiv \frac{3x_p^2 + a}{2y_p} \ (mod\ q)$$

$$x_r \equiv \lambda^2 - 2x_p \ (mod\ q)$$

[26] https://ja.wikipedia.org/wiki/%E9%9B%A2%E6%95%A3%E5%AF%BE%E6%95%B0
[27] https://safecurves.cr.yp.to/

$$y_r \equiv \lambda(x_p - x_r) - y_p \ (mod\ q)$$

図 7.15: 点の 2 倍算

7.2.3.1.3 スカラー倍算

スカラー倍算では、係数 k を 2 進数表現に分解します。

$$kPR \equiv (\text{mod } q)$$

$$k = k_0 + 2k_1 + 2^2k_2 + \cdots + 2^mk_m \text{ where } [k_0 \cdots k_m] \in \{0,1\}$$

図 7.16: スカラー倍算

続いて次の処理手順を適用します。

1. N を点 P とします。
2. Q を無限遠の点 (0, 0) とします。
3. i が 0 から m までの範囲内にある場合、以下を実行します。
    1. k~i~ が 1 の場合、Q を、N に Q を加えた結果とします (楕円曲線加算)。
    2. N を 2 倍算した結果を N とします (楕円曲線 2 倍算)。
4. Q を返します。

JavaScript での実装例を以下に示します。

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

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

    return Q;
}
```

合同算術では、除算は分子と除数の逆数との乗算として行うことに注意してください。

これらの演算の JavaScript での実装例は、サンプル リポジトリ[28] にある ecdsa.js ファイルにあります。このような単純な実装でも使用できますが、タイミング攻撃に対しては脆弱です。本番環境用の実装では、こうした攻撃を考慮して、さまざまなアルゴリズムを使用します。

7.2.3.2 楕円曲線デジタル署名アルゴリズム (ECDSA)

楕円曲線デジタル署名アルゴリズム (ECDSA) は、米国国家規格協会 (ANSI)[29] の委員会によって開発されました。規格は X9.63[30] です。この規格では、安全かつ適切に楕円曲線を署名に使用するために必要なすべてのパラメーターを規定しています。JWA 仕様では、この仕様 (および FIPS 186-4[31]) に基づき、曲線のパラメーターの選択とアルゴリズムの指定を行っています。

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

JWT での使用に関して、JWA 仕様では、他の署名アルゴリズムと同様に、署名アルゴリズムへの入力を Base64 でエンコードされたヘッダーとペイロードとするよう定めていますが、結果は 1 つの整数ではなく 2 つの整数 r と s になります。これらの整数を、ビッグエンディアン方式で 32 バイト列に変換してから連結し、1 つの 64 バイトの署名を作成します。

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

    header.alg = 'ES256';

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

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

このコードは、RSA および HMAC 署名で使用したものに似ています。主な違いは、2 つの署名値 r と s を 32 バイトのオクテットに変換する点です。これには、RSA でも使用した PKCS の i2osp 関数を使用できます。

署名を検証するには、パラメーター r と s を取得する必要があります。

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

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

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

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

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

ここでも、署名の有効性を検証する手順は RSA や HMAC の場合と似ています。この場合、値 r と s は 64 バイトの

JWT 署名から取得する必要があります。最初の 32 バイトが要素 r で、残りの 32 バイトが要素 s です。これらの値を数値に変換するには、PKCS の os2ip プリミティブを使用します。

7.2.3.2.1 楕円曲線のドメイン パラメーター

ECDSA で使用される楕円曲線演算は、いくつかの主要パラメーターに依存しています。

- p または q: 算術演算の実行対象の素体[32] を定義するために使用する素数。素体の演算では合同算術を使用します[33]。
- a: 曲線方程式の x の係数。
- b: 曲線方程式の定数 (y 切片)。
- G: 楕円曲線演算のベース ポイントとして使用される有効な曲線上の点。ベース ポイントは、曲線上の他の点を得るために算術演算で使用されます。
- n: ベース ポイント G の位数。このパラメーターは、点 G をベース ポイントとして使用して構築できる曲線内の有効な点の数を表します。

楕円曲線演算を安全に実行するには、これらのパラメーターを慎重に選択する必要があります。JWA において、有効と見なされる曲線は P-256、P-384、P-521 の 3 つだけです。これらの曲線は、FIPS 186-4[34] やその他の関連する規格で定義されています。

コード サンプルでは、P-256 曲線を使用します。

```
const p256 = {
    q: bigInt('00ffffffff00000001000000000000' +
              '000000000000ffffffffffffffffff'  +
              'ffffff', 16),
    // order of base point
    n: bigInt('1157920892103562487626974469494075735299969552241357603 42' +
               '4222590610685120443 69'),
    // base point
    G: {
        x: bigInt('6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0' +
                  'f4a13945d898c296', 16),
        y: bigInt('4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ece' +
                  'cbb6406837bf51f5', 16)
    },
    //a: bigInt(-3)
    a: bigInt('00ffffffff00000001000000000000' +
              '000000000000ffffffffffffffffff'  +
              'fffffc', 16),
    b: bigInt('5ac635d8aa3a93e7b3ebbd55769886' +
              'bc651d06b0cc53b0f63bce3c3e27d2'  +
              '604b', 16)
};
```

7.2.3.2.2 公開鍵と秘密鍵

楕円曲線を使用した公開鍵と秘密鍵の作成は、非常に簡単です。

秘密鍵は、1 からベース ポイント G の位数 n までの範囲から乱数を選択すれば、作成できます。

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

たったこれだけであり、きわめて簡単です。

公開鍵は秘密鍵から計算でき、ベース ポイント G に秘密鍵を乗算して算出します。

---

[32] https://ja.wikipedia.org/wiki/%E6%9C%89%E9%99%90%E4%BD%93
[33] https://ja.wikipedia.org/wiki/%E5%90%88%E5%90%8C%E7%AE%97%E8%A1%93
[34] http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

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

つまり、公開鍵は楕円曲線上の点であり、秘密鍵は単純にスカラー値です。

7.2.3.2.2.1 離散対数問題

秘密鍵からの公開鍵の導出が簡単であるならば、その逆も簡単なのではないかと思うかもしれません。ここで探す必要があるのは、`G` に乗算すると公開鍵 `Q` が得られるような数 `d` です。

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

図 7.17: 公開鍵の対数として表される秘密鍵

楕円曲線に対して選択した加法群[35] (素体など) においては、`k` の計算は離散対数問題になります。これを効率的に計算できる汎用的なアルゴリズムは発見されていません。曲線 P-256 に使用されるような 256 ビット数を使用した場合、複雑すぎるため、現在の計算能力では計算不可能です。楕円曲線暗号の強みはそこにあります。

7.2.3.2.3 ES256: P-256 および SHA-256 を使用した ECDSA

署名アルゴリズム自体はシンプルであり、必要なものは合同算術と楕円曲線演算だけです。

1. 暗号論的に安全なハッシュ関数を使用して、署名対象のメッセージのダイジェストを計算します。この数値を `e` とします。
2. 暗号論的に安全な乱数発生器を使用して、`1` から `n − 1` までの範囲内にある数 `k` を選択します。
3. ベース ポイント `G`に `k`(mod q) を乗算します。
4. 前の手順の点の `x` 座標を `G`(n) の位数で除算した余りを `r` とします。
5. `r` がゼロの場合は、ゼロ以外になるまで手順 2 ～ 5 を繰り返します。
6. 秘密鍵を `d` とし、`s` を次の式の結果とします。

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

図 7.18: s

7. `s` がゼロの場合は、ゼロ以外になるまで手順 2 ～ 7 を繰り返します。

署名は `r` と `s` のタプルです。JWA では、`r` と `s` は連結された 2 つの 32 バイトのオクテット列として表現します (最初に `r`、次に `s`)。

実装例を以下に示します。

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

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

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

    let   r;
    let   s;
    do    {
        let k;
        do {
```

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

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

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

    return {


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

検証も簡単です。署名を `(r,s)` とした場合、次の手順を実行します。

1. 暗号論的に安全なハッシュ関数を使用して、署名対象のメッセージのダイジェストを計算します。この数値を `e` とします。
2. `s` を位数 `n` で除算した余りの逆数を `c` とします。
3. `c` を `n` で除算した余りと `e` を乗算した値を `u1` とします。
4. `c` を `n` で除算した余りと `r` を乗算した値を `u2` とします。
5. `u1` を `q` で除算した余りをベース ポイント `G` に乗算した値を点 `A` とします。
6. `u2` を `q` で除算した余りを公開鍵 `Q` に乗算した値を点 `B` とします。
7. 点 `A` および点 `B`(`q` を法とする) の楕円曲線加算の和を点 `C` とします。
8. 点 `C` の `x` 座標を `n` で除算した余りを `v` とします。
9. `v` が `r` と等しい場合、署名は有効ですが、等しくない場合は無効です。

実装例を以下に示します。

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

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

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

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

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

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

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

```
    const v = point.x.fixedMod(p256.n);
    return v.compare(signature.r) === 0;
}
```

このアルゴリズムの重要な部分でよく見落とされがちなのは、公開鍵の有効性の検証です。これは過去に攻撃の原因となっています[36]。メッセージ署名の検証に使用される検証者用の公開鍵を攻撃者が自由に操作でき、かつ公開鍵が曲線上の点として検証されていない場合、攻撃者は特別な公開鍵を作成し、それを使用して情報を盗み取ることができます。JWT の暗号化に使用される鍵共有プロトコルを考慮すると、これは特に重要です。

実装例はサンプル リポジトリ[37] 内の `ecdsa.js` ファイルにあります。

7.3 今後の更新

JWA 仕様では、他にも多くのアルゴリズムを定義しています。残りのアルゴリズムについては、本書の更新版で取り上げる予定です。

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

第 8 章

付録 A: 現在のベスト プラクティス

公開以来、JWT はさまざまな場所で使用されてきました。その結果、JWT やライブラリの実装は数々の攻撃にさらされてきました。この種の攻撃については、以前の章でいくつか紹介しました。本章では、JWT の利用に関する現在のベスト プラクティスについて説明します。

本章は、IETF OAuth Working Group[1] の JWT Best Current Practices[2] の草稿に基づいています。使用した草案のバージョンは 00 (2017 年 7 月 19 日付け) です[3]。

8.1 隠れた危険とよく見られる攻撃

攻撃の説明に進む前に、以下の攻撃の多くが JSON Web Token の設計ではなく実装に関連していることに注意することが重要です。実装に関連するといっても、危険度が下がるわけではありません。以下の攻撃が基礎的な設計を変更することによって軽減または排除できるかどうかは、確かに議論の余地があります。しかし当分の間、JWT の仕様と形式は変わらないため、大半の変更は実装の領域で行うことになります (ライブラリ、API、プログラミング手法、規約の変更)。

また、JWT の最も一般的な表現である JWS コンパクト シリアライゼーション形式の基本的な概念を理解しておくことも重要です。シリアライズ前の JWT には、`header` と `payload` という 2 つの主な JSON オブジェクトがあります。

`header` オブジェクトには、トークンの種類、使用される署名または暗号化アルゴリズム、鍵 ID など、JWT 自体に関する情報が含まれています。

`payload` オブジェクトには、トークンによって伝えられるすべての重要な情報が含まれています。`sub` (主題) や `iat` (発行日時) のような標準クレームがいくつかありますが、任意のカスタム クレームをペイロードの一部として含めることもできます。

これらのオブジェクトは、JWS コンパクト シリアライゼーション形式を使用して次のようにエンコードされます。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

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

これは署名付き JWT です。コンパクト形式の署名付き JWT は単純にヘッダーとペイロードであり、Base64-URL エンコーディングを使用してエンコードされ、ドット (.) で区切られます。コンパクト表現の最後の部分に、署名があります。つまり、次のような形式になっています。

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

これは署名付きトークン固有の形式です。暗号化されたトークンには別のコンパクト形式があります。その形式でも、Base64-URL エンコーディングとドット区切りのフィールドを使用します。

> JWT の使い方を試したり、JWT がどのようにエンコード/デコードされるかを確認したい場合は、JWT.io をご覧ください[4]。

8.1.1 "alg: none" 攻撃

前述したように、JWT には重要な情報を保持する 2 つの JSON オブジェクトである `header` および `payload` があります。ヘッダーには、JWT がオブジェクトに含まれるデータを署名または暗号化するために使用するアルゴリズムに関する情報が含まれています。署名付き JWT では、ヘッダーとペイロードの両方が署名の対象ですが、暗号化された JWT では、ペイロードだけが暗号化の対象です (ヘッダーは常に読み取り可能である必要があります)。

署名付きトークンの場合、ヘッダーとペイロードは署名によって改ざんから保護されますが、署名を使用したり、JWT に含まれるデータを変更したりすることなく、JWT を書き換えることができます。なぜこのようなことが可能なのでしょうか。

次のようなヘッダーとペイロードを持つ JWT を例に考えてみましょう。

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

ここで、このトークンを、署名と "secret" という値の署名鍵を含むコンパクト シリアライゼーション形式にエンコードするとします。これには JWT.io[5] を使用するとよいでしょう。結果は次のようになります (読みやいように改行を挿入しています)。

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

これを JWT.io[6] に貼り付けてみてください。

これは署名付きトークンであるため、自由に読み取ることができます。データを少しだけ変えた同様のトークンを作成することもできますが、その場合、署名鍵を知らなければ署名することはできません。署名鍵を知らなくても、攻撃者は何かできるのでしょうか。この種の攻撃では、攻撃者は署名のないトークンを利用します。その方法を説明しましょう。

まず、攻撃者はトークンを改変します。たとえば、次のように改変します。

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

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

```
}
```

エンコードすると次のようになります (見やすいように改行を挿入しています)。

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.
eyJzdWIiOiJqb2UiLCJyb2xlIjoiYWRtaW4ifQ.
```

ご覧のとおり、このトークンには署名が含まれておらず (`"alg": "none"`)、またペイロードの `role` クレームが変更されています。攻撃者がこのトークンの利用に成功すると、特権エスカレーション攻撃を実行するおそれがあります。このような攻撃はどのように行われるのでしょうか。架空の JWT ライブラリを例に、この攻撃の仕組みを見てみましょう。次のようなデコード関数があるとします。

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

この関数は、エンコードされたトークンとシークレットを受け取り、トークンを検証して、デコードされたデータを返します。検証に失敗すると、例外をスローします。この関数は、適切な検証アルゴリズムを選択するために、ヘッダーの `alg` クレームを利用します。攻撃者が狙うはこの箇所です。以前は多くのライブラリが検証アルゴリズムを選択するために、このクレームを利用していました[7]。後はご想像どおりです。先ほど挙げた不正なトークンの `alg` クレームの値は `none` (なし) です。検証アルゴリズムが指定されていないため、検証が常に成功することになります。

これは、仕様自体の脆弱性ではなく、特定のライブラリの API の曖昧さを利用した古典的な攻撃の例です。それにもかかわらず、これは実際に行われた攻撃であり、さまざまな実装で可能なものでした。このため、今日の多くのライブラリでは、署名がない場合も `"alg": "none"` トークンは無効であると表明しています。この種の攻撃を回避する方法は他にもあります。最も重要なのは、トークンの検証を試みる前に、ヘッダーで指定されているアルゴリズムを必ず確認することです。また、`alg` クレームを利用せずに、検証関数への入力として検証アルゴリズムを要求するライブラリを使用する方法もあります。

8.1.2 RS256 公開鍵を HS256 共有シークレットとして使用する攻撃

この攻撃は `"alg": "none"` 攻撃に似ており、特定の JWT ライブラリの API の曖昧さを利用した攻撃です。今回の例も、先ほどの攻撃で使用されるトークンと似ています。ただしこの場合は、署名を削除するのではなく、多くの API にある抜け穴を利用して、検証ライブラリによって有効とみなされる署名を作成します。まず、検証関数に使用される JWT ライブラリの一般的な関数シグネチャを考えてみます。

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

ご覧のように、この関数は基本的に `"alg": "none"` 攻撃のものと同じです。検証が成功した場合、トークンはデコードされて返されます。失敗した場合は例外がスローされます。ただし、この関数は 2 番目の引数として公開鍵も受け取ります。これはある観点から見れば理にかなっていることです。公開鍵と共有シークレットは通常どちらも文字列かバイト配列であるため、関数の引数に必要な型の観点から見れば、1 つの引数が公開鍵 (RS、ES、または PS アルゴリズムの場合) と共有シークレット (HS アルゴリズムの場合) の両方を表すことは考えられます。この種の関数シグニチャは、多くの JWT ライブラリで見られます。

ここで、攻撃者が RSA 鍵ペアで署名されたエンコード済みトークンを入手したとします。トークンは次のようなものです (見やすいように改行を挿入しています)。

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

デコードすると次のようになります。

```
header: {
  "alg": "RS256",
```

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

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

このトークンは RSA 鍵ペアで署名されています。RSA 署名は秘密鍵で生成され、検証は公開鍵で行います。トークンの検証者はだれでも、例に挙げた `jwtDecode` 関数を呼び出すことができます。

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

しかし、ここで問題が発生します。公開鍵は名前のとおり、通常は公開されるものです。攻撃者が公開鍵を入手する可能性がありますが、それは問題ありません。しかし、攻撃者が次の手法で新しいトークンを作成したらどうでしょうか。まず、攻撃者はヘッダーを変更し、署名アルゴリズムとして `HS256` を選択します。

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

さらに、ペイロード内の `role` クレームを変更して、権限をエスカレーションします。

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

ここがこの攻撃の核心部分です。攻撃者は、単純な文字列である公開鍵を HS256 共有シークレットとして使用して、新たにエンコードされた JWT を作成します。HS256 の共有シークレットは任意の文字列でよいため、RS256 アルゴリズムの公開鍵のような文字列でも、共有シークレットとして使用できます。

ここで、先ほどの `jwtDecode` 関数の使用に戻ります。

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

これで問題がはっきりとわかるでしょう。つまり、トークンが有効とみなされてしまうのです。公開鍵は 2 番目の引数として `jwtDecode` 関数に渡されますが、この公開鍵は、RS256 アルゴリズムの公開鍵としてではなく、HS256 アルゴリズムの共有シークレットとして使用されます。これは、`jwtDecode` 関数がヘッダーの `alg` クレームに基づいて JWT の検証アルゴリズムを選択するためです。攻撃者はこのクレームを変更したのは、そのためです。

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

`"alg": "none"` 攻撃の場合と同じように、`alg` クレームに加えて、不適切な API や仕組みがわかりにくい API を利用すると、悪意のあるユーザーによる攻撃を受けるおそれがあります。

この攻撃の対策としては、明示的なアルゴリズムを `jwtDecode` 関数に渡す、`alg` クレームを確認する、公開鍵アルゴリズムと共有シークレット アルゴリズムを分離する API を使用することなどが挙げられます。

8.1.3 脆弱な HMAC 鍵

HMAC アルゴリズムは、共有シークレットを利用して署名の生成と検証を行います。共有シークレットはパスワードに似たものだと考えている人がいます。確かに、秘密に保管する必要があるという点では似ています。しかし、似ているのはその点だけです。長さは重要な特性ですが、パスワードの場合、必要最小限の長さが他の種類のシークレットと比べて比較的短くなります。これは、適切な期間、総当たり攻撃を防げるパスワードを (ソルトとともに) 格納するために使用されるハッシュ アルゴリズムが原因です。

一方、JWT で使用される HMAC 共有シークレットは、速度を優先して最適化されています。そのため、多くの署名/検証処理を効率的に実行できますが、総当たり攻撃には脆弱です[8]。そのため、HS256/384/512 の共有シークレットは、長さがきわめて重要です。実際に、JSON Web Algorithms[9] では、鍵の最小長が、HMAC アルゴリズムとともに使用されるハッシュ関数のビット長さと同じに定められています。

> "このアルゴリズムでは、ハッシュ出力と同じかそれ以上の長さ (HS256 の場合は 256 ビット) の鍵を使用しなければならない。" - JSON Web Algorithms (RFC 7518)、3.2 HMAC with SHA-2 Functions[10]

要するに、他の環境で使用できる多くのパスワードも、HMAC 署名付き JWT で使用するには不十分です。256 ビットは 32 個の ASCII 文字に相当します。そのため、人が読めるものを使用している場合は、この長さをシークレットの最小文字数とすることをお勧めします。また、より堅牢で柔軟な RS256 やその他の公開鍵アルゴリズムに切り替えるのもよいでしょう。この攻撃は現実の脅威です。共有シークレットが短すぎる場合、HS256 に対する総当たり攻撃は容易に実行できることが実証されています[11]。

8.1.4 暗号化と署名検証の併用に関する誤解

署名はデータを改ざんから保護します。つまり、署名はデータを読み取り不可能にするわけではなく、データを変更できないようにします。データを変更すると、署名は無効になります。一方、暗号化を行うと、共有鍵または秘密鍵を知らない限り、データは読み取れなくなります。

多くの用途では、署名だけで十分です。しかし、機密データを扱う場合は暗号化が必要になることがあります。JWT は署名と暗号化の両方をサポートします。

暗号化はあらゆるケースで改ざんに対する保護を実現する、というのは、よくある誤解です。多くの場合、この誤解の根底にあるのは、"データが読み取り不可能なら攻撃者が自分の都合のいいように改変できるはずがない" という認識です。残念ながらこれは、暗号化/復号アルゴリズムに関する攻撃者の知識を過小評価しています。

一部の暗号化/復号アルゴリズムは、渡されたデータの有効性に関係なく、出力を生成します。つまり、暗号化されたデータが改変されている場合でも、何らかの出力が復号処理によって生じます。何の考えもなくデータを変えただけでは、無意味なデータしか出力されませんが、悪意のある攻撃者はそれすらも利用してシステムにアクセスできます。たとえば、次のような JWT ペイロードを考えてみます。

```
{
  "sub": "joe",
  "admin": false
}
```

ご覧のように、`admin` クレームはブール型です。攻撃者が復号されたデータに変更を加え、ブール値を反転させることができれば、特権エスカレーション攻撃を実行できます。特に、攻撃者に十分な時間的余裕がある場合は、暗号化されたデータに対して必要なだけ変更を試みることができ、トークンが処理前に無効としてシステムに廃棄されることを避けられます。他にも、サニタイズされたデータが必要なサブシステムに無効なデータを送信してバグや障害を引き起こす攻撃や、他の種類の攻撃を準備する足がかりとなる攻撃などがあります。

このような理由から、JSON Web Algorithms[12] では、データ完全性の検証も行う暗号化アルゴリズムのみを定義しています。つまり、使用する暗号化アルゴリズムが JWA で認められているアルゴリズムであれば[13]、署名付き JWT の上に暗号化された JWT を積み重ねる必要はありません。ただし、標準以外のアルゴリズムを使用して JWT を暗号化する場合は、そのアルゴリズムによってデータ完全性を確保できることを確認する必要があります。または、データ完全性を保証するために、署名付き JWT を最も内側の JWT として使用して JWT をネストする必要があります。

ネストされた JWT[14] は、仕様で明示的に定義およびサポートされています。あまり多くは見られないものの、ネストさ

[8] https://auth0.com/blog/brute-forcing-hs256-is-possible-the-importance-of-using-strong-keys-to-sign-jwts/
[9] https://tools.ietf.org/html/rfc7518
[10] https://tools.ietf.org/html/rfc7518#section-3.2
[11] https://auth0.com/blog/brute-forcing-hs256-is-possible-the-importance-of-using-strong-keys-to-sign-jwts/
[12] https://tools.ietf.org/html/rfc7518
[13] https://tools.ietf.org/html/rfc7518
[14] https://tools.ietf.org/html/rfc7519#section-2

れた JWT は他のシナリオでも使用されることがあります。たとえば、JWT を使用する第三者のシステムを介して当事者が発行したトークンを送信する場合などです。

このようなシナリオでは、ネストされた JWT の検証に関する間違いがよく見られます。データ完全性を維持し、データを適切にデコードするには、JWT のすべての層が、ヘッダーで定義されているアルゴリズムに関連するすべての検証をパスする必要があります。つまり、最も外側の JWT を復号および検証できる場合でも、最も内側の JWT まですべて検証 (または復号) する必要があります。これを行わないと、特に最も内側の署名付き JWT が最も外側の暗号化された JWT で伝達される場合に、未検証のデータが使用され、そのデータに関連するセキュリティ問題が発生するおそれがあります。

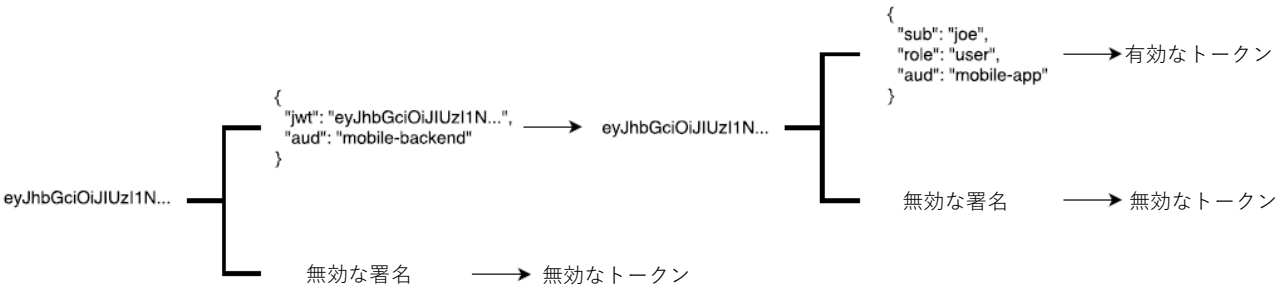

図 8.1: ネストされた JWT の検証

8.1.5 無効な楕円曲線による攻撃

楕円曲線暗号は、JSON Web Algorithms[15] でサポートされている公開鍵アルゴリズムの一種です。楕円曲線暗号は、楕円曲線離散対数問題の難しさを利用した暗号です。これは使用する数値が大きいと、現実的な時間内に解くことができない数学問題です。この問題を利用することで、公開鍵、暗号化されたメッセージ、メッセージの平文からの秘密鍵の復元を防止できます。JSON Web Algorithms でサポートされているもう 1 つの公開鍵アルゴリズムである RSA と比べると、楕円曲線暗号はより小さな鍵で同程度の強度を実現できます。

暗号処理に必要な楕円曲線は、有限体上で定義されます。言い換えると、楕円曲線は (すべての実数ではなく) 不連続な数の集合をもとに演算されます。そのため、楕円曲線に基づく暗号処理で扱われる数は、すべて整数です。

楕円曲線の数学的演算の結果は、すべて曲線上の有効な点を表します。つまり、楕円曲線の計算は無効な点が発生しないように定義されます。無効な点が作成されると、演算への入力のエラーが生じます。楕円曲線の主な算術演算を以下に示します。

- **点の加算**: 同一曲線上の 2 つの点を加算し、その曲線上に 3 番目の点を作成します。
- **点の 2 倍算**: 点自身に対して点を加算し、同一曲線上に新しい点を作成します。
- **スカラー倍算**: 曲線上の 1 つの点にスカラー値を乗算します。つまり、その点を `k` 回繰り返し加算します (`k` はスカラー値)。

楕円曲線に基づく暗号処理は、すべてこれらの算術演算を利用して行います。しかし、一部の実装例では、演算への入力を検証できていません。楕円曲線暗号において、公開鍵は楕円曲線上の点を表します。一方、秘密鍵は特殊かつ非常に大きな範囲内にある単純な数値です。演算への入力が検証されない場合、実際は有効でなくても、有効に見える結果が生成される可能性があります。このような結果を復号などの暗号処理などで使用して、秘密鍵を回復させることができます。この種の攻撃は過去に実証されており[16]、無効曲線攻撃と呼ばれています[17]。優れた実装では、公開関数に渡されるすべての入力の有効性を常に確認します。その際には、公開鍵が選択した曲線の有効な楕円曲線上の点を表すことと、秘密鍵が有効な値の範囲内にあることも確認します。

[15] https://tools.ietf.org/html/rfc7518
[16] http://blogs.adobe.com/security/2017/03/critical-vulnerability-uncovered-in-json-encryption.html
[17] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.3920&rep=rep1&type=pdf

8.1.6 置換攻撃

置換攻撃は、攻撃者が 2 つ以上の異なるトークンを傍受しようと試みる攻撃の一種です。攻撃者は傍受したトークンの一方または両方を、所定の目的以外の目的で使用しようとします。

置換攻撃には 2 種類あり、受信者が同じ場合 (RFC の草案で Cross-JWT Confusion と呼ばれているもの) と受信者が異なる場合があります。

8.1.6.1 受信者が異なる場合

受信者が異なる置換攻撃では、ある受信者用のトークンを別の受信者に送信する手法を使います。第三者のサービス用のトークンを発行する認可サーバーがあるとします。認可トークンは次のペイロードを持つ署名付き JWT です。

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

このトークンは認証処理を実行する API に対して使用できます。また、ユーザー `joe` は、少なくともこのサービスに関しては管理者レベルの権限を持っています。しかし、このトークンには問題があります。対象の受信者だけでなく発行者も指定されていないからです。ここでこのトークンの対象受信者ではない別の API が、有効性の検証手段として署名のみを使用していたとしたらどうなるでしょうか。そのサービスまたは API のデータベースに `joe` というユーザーが存在する場合、攻撃者はこの別のサービスにこのトークンを送信することで、即座に管理者権限を得ることができます。

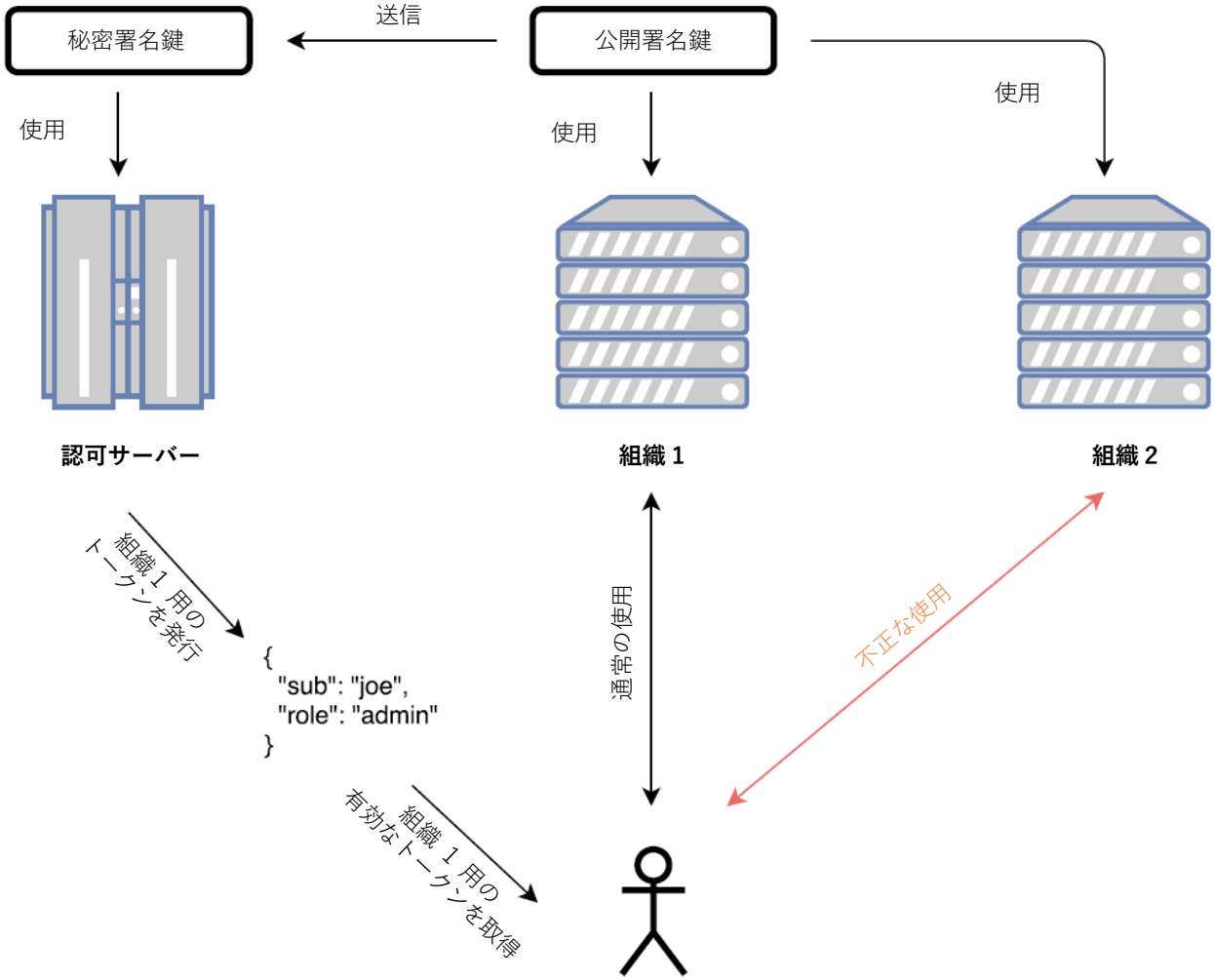

図 8.2: 受信者が異なる JWT 置換攻撃

この攻撃を防ぐには、サービスごとの一意の鍵またはシークレットか、特定のクレームのどちらかを利用してトークンを検証する必要があります。たとえば、対象受信者を指定する aud クレームをこのトークンに含めれば、署名が有効であっても、同じ秘密鍵または署名鍵を使用する他のサービスでこのトークンを使用できなくなります。

8.1.6.2 受信者が同じ場合/Cross JWT

この攻撃は前の攻撃と似ていますが、別の受信者に対して発行されたトークンは利用しません。この場合、受信者は同一です。この攻撃では、攻撃者は本来のサービスとは異なる (同じ会社やサービス プロバイダー内の) 別のサービスにトークンを送信します。

次のペイロードを持つトークンがあるとしましょう。

```
{
  "sub": "joe",
  "perms": "write",
  "aud": "cool-company/user-database",
  "iss": "cool-company"
}
```

このトークンは先ほどのものよりもはるかに安全に見えます。発行者 (iss) クレーム、受信者 (aud) クレーム、権限 (perms) クレームが含まれています。このトークンの発行対象の API は、トークンの署名が有効であっても、これらのクレームをすべて確認します。そのため、攻撃者が同じ秘密鍵またはシークレットで署名されたトークンを入手したとしても、そのトークンがそのサービスを対象としたものでなければ、それを使用してこのサービスを操作することはできません。

しかし、cool-company には他にも公開しているサービスがあります。そのうちの 1 つのサービスである cool-company/item-database では、最近のアップグレードで、トークンの署名とともにクレームを確認するようになりました。しかし、アップグレード時に検証対象のクレームを選択する担当者がミスを犯し、aud クレームが正しく検証されませんでした。担当者は完全に一致するかどうか確認せず、cool-company という文字列があるかどうかしか検証しませんでした。同社の別のサービスである cool-company/user-database では、この検証をパスするトークンを発行しています。つまり、攻撃者は item-database サービス用のトークンの代わりに、user-database サービス用のトークンを使用できます。そのため、攻撃者は user-database への書き込み権限しか持っていなくても、item-database への書き込み権限を得ることができます。

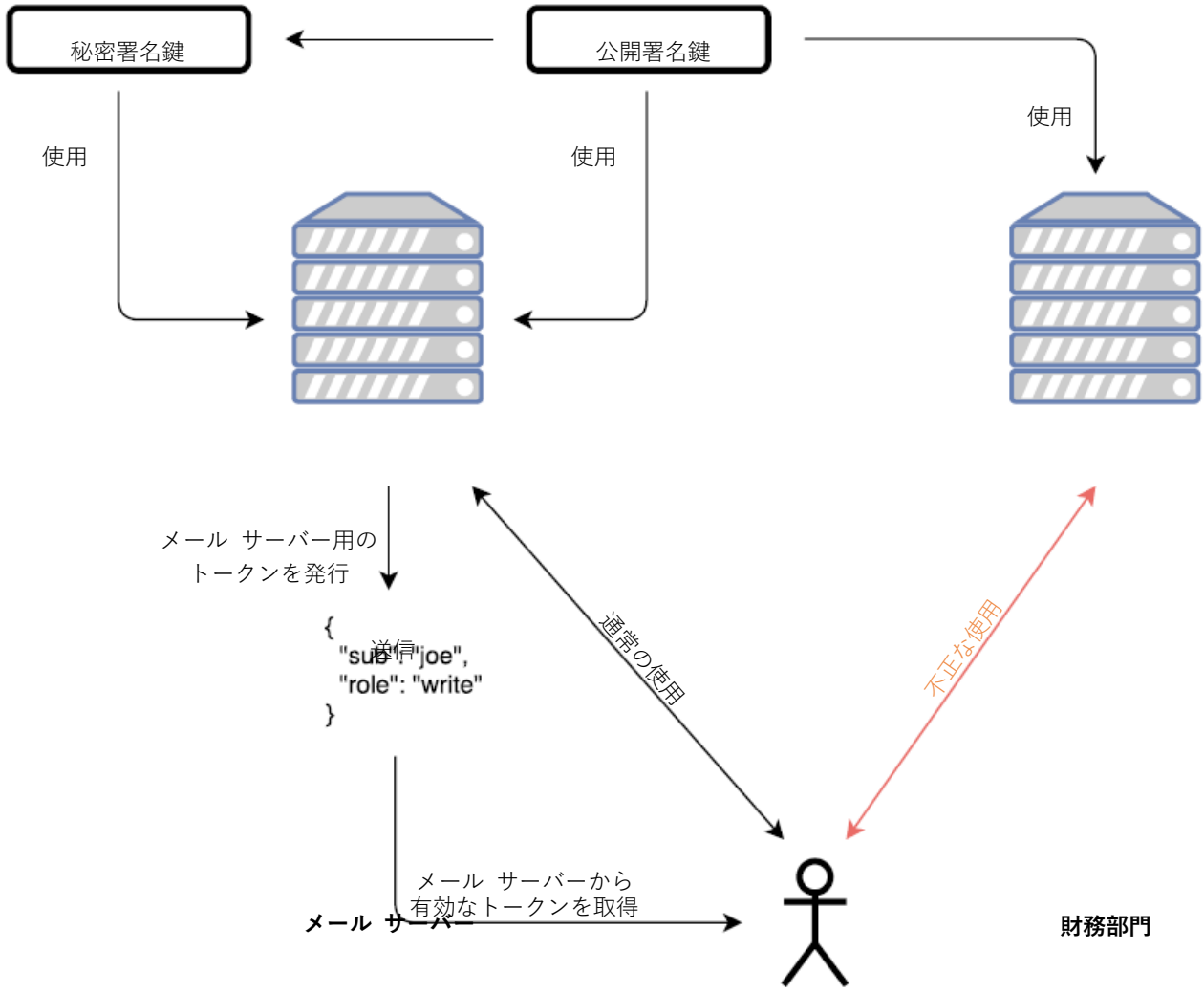

図 8.3: 受信者が同じ JWT 置換攻撃

-

8.2 対策とベストプラクティス

ここまでは JWT を使用した一般的な攻撃を見てきました。次は、最新のベスト プラクティスを見ていきましょう。前述の攻撃は、いずれも以下の推奨事項を実行することでうまく防ぐことができます。

8.2.1 アルゴリズムの検証を常に実行する

`"alg": "none"` 攻撃や、RS256 公開鍵を HS256 共有シークレットとして使用する攻撃は、この対策で防止できます。攻撃者に権限を奪取されないように、JWT を検証する際は、常にアルゴリズムを明示的に選択する必要があります。従来のライブラリでは検証用のアルゴリズムを選択する際に、ヘッダーの `alg` クレームを利用してきました。前述のような攻撃が実際に見られるようになってからは[18]、多くのライブラリで、ヘッダーで指定されている内容を無視し、検証用のアルゴリズムを明示的に指定する方法が提供されるようになりました。それでもやはり、一部のライブラリではヘッ

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

ダーで指定されたアルゴリズムを使用することができます。そのため、開発者は常に明示的にアルゴリズムを選択するように注意する必要があります。

8.2.2 適切なアルゴリズムを使用する

JSON Web Algorithms の仕様では、推奨アルゴリズムと必須アルゴリズムが示されていますが、個々のシナリオに合わせて適切なアルゴリズムを選択することは、依然としてユーザーの責任となっています。たとえば、1 台のサーバー上の単一ページの Web アプリケーションから発行された小さなトークンをユーザーのブラウザーに保存する場合は、HMAC 署名で署名された JWT を使うだけで十分でしょう。一方、共有シークレットのアルゴリズムは、フェデレーション アイデンティティを使用する場合は非常に不便です。

また、これは考え方を変えれば、JWT の検証アルゴリズムがアプリケーションで受け入れられるものでない限り、すべての JWT を無効とみなすということです。つまり、検証者がトークンの検証に必要な鍵と手段を持っていたとしても、検証アルゴリズムがアプリケーションにとって適切なものでない場合には、無効であるとみなすということです。これは、前述の推奨事項である "アルゴリズムの検証を常に実行する" ことと同じです。

8.2.3 常にすべての検証を実行する

ネストされたトークンの場合は、各トークンのヘッダーで宣言されているすべての検証手順を常に実行する必要があります。つまり、最も外側のトークンを復号または検証するだけでは不十分であり、内側のトークンの検証を省略すべきではありません。署名付き JWT のみ使用する場合も、すべての署名を検証する必要があります。不十分な検証に起因する誤りは、外部から発行された別の JWT を伝達するために JWT を使用するアプリケーションでよく見られます。

8.2.4 暗号の入力を必ず検証する

攻撃に関する節で説明したように、一部の暗号処理は、その処理の範囲外の入力に対して十分に定義されていない場合があります。無効な入力が悪用されると予期しない結果が生成されたり、完全な侵害の手がかりとなる機密情報 (攻撃者が秘密鍵を入手した場合) が抽出されたりすることがあります。

先に例として挙げた楕円曲線演算では、公開鍵を使用する前に、必ず公開鍵を検証する必要があります (つまり、公開鍵が選択した曲線上の有効な点を表していることを確認する必要があります)。通常、この種の検証は基礎的な暗号ライブラリによって処理します。開発者は、選択したライブラリでこの種の検証が実行されることを確認するか、アプリケーション レベルで検証を実行するために必要なコードを追加する必要があります。さもないと、秘密鍵が侵害されるおそれがあります。

8.2.5 強力な鍵を選択する

この推奨事項はすべての暗号鍵に該当するものであるにもかかわらず、多くの場合、無視されています。前述のように、HMAC 共有シークレットの最小長は見落とされがちです。しかし、共有シークレットは十分に長いだけでなく、完全にランダムなものでなければなりません。長い鍵でも、ランダム性のレベル (エントロピー) が低いと、総当たり攻撃や推測が可能になります。これを防ぐには、初期化の際に適切にシードが設定される暗号用途に適した品質の擬似乱数生成器 (PRNG) を鍵生成ライブラリで使用する必要があります。可能であればハードウェアの乱数生成器を使用することをお勧めします。

この推奨事項は、共有鍵アルゴリズムと公開鍵アルゴリズムの両方に適用されます。さらに、共有鍵アルゴリズムの場合、人が読めるパスワードでは安全性が低いと考えられ、辞書攻撃に対して脆弱です。

8.2.6 使用可能なすべてのクレームを検証する

先ほど説明した攻撃の中には、検証に関する誤った思い込みを突いたものもありました。そのような攻撃は、とりわけ検証の唯一の手段である署名検証や復号を利用します。また、正しく署名されたトークンや暗号化されたトークンにアクセスして、想定外の状況で不正な目的でトークンを使用する攻撃者も存在します。このような攻撃を防ぐには、署名と内容の両方が有効な場合にのみトークンを有効とみなすことが重要です。このため、sub(主題)、exp(有効期限)、iat(発行日時)、aud(受信者)、iss(発行者)、nbf(開始日時) などのクレームは最も重要であり、これらが存在する場合は必ず有効性を確認する必要があります。トークンを作成する場合は、別のコンテキストでの使用を防ぐために必要なだけの数のクレームを追加することをお勧めします。一般に、sub、iss、aud、exp は常に役に立つため、必ず含めることをお勧めします。

8.2.7 typ クレームを使用してトークンの種類を区別する

ほとんどの場合、typ クレームの値は 1 つ (JWT) ですが、このクレームはさまざまな種類のアプリケーション固有の JWT を区別するためにも使用できます。これは、システムでさまざまな種類のトークンを処理する必要がある場合に便利です。また、このクレームを使用して追加のクレーム検証を行うことで、別のコンテキストでのトークンの不正使用を防ぐこともできます。JWS 標準では、アプリケーション固有の typ クレームの値が明示的に許可されています。

8.2.8 トークンごとに異なる検証ルールを使用する

このプラクティスは、これまでに挙げたものの要点をまとめたものです。攻撃を防ぐためには、非常に明確で固有の検証ルールを、発行する各トークンに設定することが重要です。そのためには、適切な場合に typ クレームを使用したり、iss や aud などの使用可能なすべてのクレームを検証したりするだけでなく、可能な場合にはトークン間の鍵の再利用を避けたり、さまざまなカスタム クレームやクレーム形式を活用したりする必要があります。そうすることで、1 つの場所で使用するよう設計されたトークンを要件が非常によく似た別のトークンで置き換えることが不可能になります。

別の言い方をすれば、すべての種類のトークンに同じ秘密鍵を使用するのではなく、アーキテクチャのサブシステムごとに異なる秘密鍵を使用することが重要です。また、一定の内部形式をクレームに指定することでクレームを特徴付けることもできます。たとえば、iss クレームの値を企業名ではなくそのトークンを発行したサブシステムの URL にすれば、トークンの再利用が困難になります。

8.3 まとめ

JSON Web Token は暗号を利用したツールです。類似するツール、特に機密情報の処理に使用するツールと同様に、慎重に使用する必要があります。一見シンプルに見えるため、正しい共有シークレットや公開鍵アルゴリズムを選ぶだけで JWT を使用できると勘違いしがちですが、これまで説明したように、残念ながらそれは間違いです。どのようなツールを利用する場合も、それぞれのベスト プラクティスに従うことが最も重要であり、JWT も例外ではありません。特に、実地でテストされた高品質のライブラリを選択すること、ペイロードとヘッダーのクレームを検証すること、適切なアルゴリズムを選択すること、強力な鍵が生成されるか確認すること、各 API の細かな特徴に注意することが重要です。これらをすべて実行することが難しい場合は、外部のサービス プロバイダーを利用して負担を軽減することをお勧めします。Auth0[19] ならそのようなニーズにお応えすることができます。独力で実行するのであれば、本書で説明した推奨事項を十分にご検討ください。また、独自のセキュリティ方式を開発するのではなく[20]、テスト済みのコードを利用することをお勧めします。

[19] https://auth0.com
[20] https://security.stackexchange.com/questions/18197/why-shouldnt-we-roll-our-own