Manual do JWT

Por Sebastian Peyrott

Versão 0.14.1, 2016-2018

EBOOK

auth0.com

Contents

Agradecimentos especiais

Sem uma ordem especial: **Prosper Otemuyiwa** (por fornecer o exemplo de identidade federada do Capítulo 2), **Diego Poza** (por revisar este trabalho e manter minhas mãos livres enquanto eu trabalhava nele), **Matías Woloski** (por revisar as partes difíceis deste trabalho), **Martín Gontovnikas** (por aturar meus pedidos e fazer tudo para tornar o trabalho agradável), **Bárbara Mercedes Muñoz Cruzado** (por deixar tudo mais bonito), **Alejo Fernández e Víctor Fernández** (por fazer o trabalho de front-end e back-end para distribuir este manual), **Sergio Fruto** (por se esforçar para ajudar os companheiros de equipe), **Federico Jack** (por manter tudo funcionando e ainda encontrar tempo para ouvir todo mundo).

Capítulo 1

Introdução

O JSON Web Token, ou JWT (“jot”) para abreviar, é um padrão para passar *claims com segurança* em ambientes com restrição de espaço. Ele encontrou seu caminho em todos os principais frameworks da web. Simplicidade, comprimibilidade e usabilidade são os principais recursos de sua arquitetura. Embora sistemas muito mais complexos ainda estejam em uso, os JWTs ainda têm uma ampla variedade de aplicações. Neste pequeno manual, abordaremos os aspectos mais importantes da arquitetura de JWTs, incluindo sua representação binária e os algoritmos usados para construí-los, enquanto também examinamos como eles são comumente usados na indústria.

1.1 O que é um JSON Web Token?

Um JSON Web Token parece com isto (novas linhas inseridas para legibilidade):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Embora pareça uma bobagem, na verdade é uma representação muito **compacta** e **imprimível** de uma série de *claims*, juntamente com uma **assinatura** para verificar sua autenticidade.

```
{
   "alg": "HS256",
   "typ": "JWT"
}
{
   "sub": "1234567890",
   "name": "John Doe",
   "admin": true
}
```

Claims são *definições* ou *afirmações* feitas sobre uma determinada parte ou objeto. Alguns desses claims e seu significado são definidos como parte da especificação do JWT. Outros são definidos pelo usuário. A magia por trás dos JWTs é que eles padronizam certos claims que são úteis no contexto de algumas operações comuns. Por exemplo, uma dessas operações comuns é estabelecer a identidade de determinada parte. Portanto, um dos claims padrão encontrados nos JWTs é *sub* (de "subject", ou assunto). Vamos dar uma olhada mais profunda em cada um dos claims padrão no Capítulo 3.

Outro aspecto chave dos JWTs é a possibilidade de assiná-los com JSON Web Signatures (JWS, RFC 7515), e/ou criptografá-los, usando JSON Web Encryption (JWE, RFC 7516). Juntamente com JWS e JWE, os JWTs fornecem uma solução poderosa e segura para muitos problemas diferentes.

1.2 Que problema ele resolve?

Embora o principal objetivo dos JWTs seja transferir claims entre duas partes, sem dúvida o aspecto mais importante disso é o *esforço de padronização* com um *formato de contêiner simples, opcionalmente validado e/ou criptografado*. No passado, foram implementadas soluções ad hoc para este mesmo problema, tanto a nível privado como público. Normas mais antigas para estabelecer claims sobre certas partes também estão disponíveis. O que o JWT proporciona é um formato de contêiner *simples*, útil e padrão.

Embora a definição dada seja um pouco abstrata até agora, não é difícil imaginar como eles podem ser usados: sistemas de login (embora outros usos sejam possíveis). Vamos dar uma olhada mais de perto nas aplicações práticas no Capítulo 2. Algumas dessas aplicações incluem:

- Autenticação
- Autorização
- Identidade federada
- Sessões do lado do cliente (sessões "sem estado")
- Segredos do lado do cliente

1.3 Um pouco de história

O grupo JSON Object Signing and Encryption (JOSE) foi formado no ano de 2011. O objetivo do grupo era *"padronizar o mecanismo de proteção de integridade (assinatura e MAC) e criptografia, bem como o formato para chaves e identificadores de algoritmo para suportar a interoperabilidade dos serviços de segurança para protocolos que usam JSON"*. Até o ano de 2013, uma série de rascunhos, incluindo um livro de aplicação prática com diferentes exemplos de uso das ideias produzidas pelo grupo, estavam disponíveis. Esses rascunhos mais tarde se tornariam os RFCs JWT, JWS, JWE, JWK e JWA. No ano de 2016, esses RFCs estão no processo de rastreamento de padrões e erratas não foram encontradas neles. O grupo está atualmente inativo.

Os principais autores por trás das especificações são Mike Jones, Nat Sakimura, John Bradley e Joe Hildebrand.

Capítulo 2

Aplicações práticas

Antes de mergulhar profundamente na estrutura e construção de um JWT, vamos analisar várias aplicações práticas. Este capítulo lhe dará uma noção da complexidade (ou simplicidade) das soluções comuns baseadas em JWT usadas no setor, hoje. Todo o código está disponível em repositórios públicos para a sua conveniência. Esteja ciente de que as demonstrações a seguir *não se* destinam ao uso na produção. Casos de teste, registro e práticas recomendadas de segurança são essenciais para o código pronto para a produção. Essas amostras são apenas para fins educacionais e, portanto, permanecem simples e diretas ao ponto.

2.1 Sessões do lado do cliente/sem estado

As chamadas sessões *sem estado* são, de fato, nada mais do que dados do lado do cliente O aspecto chave desta aplicação está no uso de *assinatura* e, possivelmente, *criptografia* para autenticar e proteger o conteúdo da sessão. Os dados do lado do cliente estão sujeitos a *adulteração*. Como tal, devem ser tratados com muito cuidado pelo back-end.

Os JWTs, em razão do JWS e JWE, podem fornecer vários tipos de assinaturas e criptografia. As assinaturas são úteis para *validar* os dados contra adulteração. A criptografia é útil para *impedir* que os dados sejam lidos por terceiros.

Na maioria das vezes, as sessões só precisam ser assinadas. Em outras palavras, não há preocupação de segurança ou privacidade quando os dados armazenados nelas são lidos por terceiros. Um exemplo comum de um claim que geralmente pode ser lida com segurança por terceiros é *sub* ("assunto"). O claim de assunto geralmente identifica uma das partes para a outra (pense em IDs de usuário ou e-mails). Não é um requisito que este claim seja *único*. Em outras palavras, claims adicionais podem ser necessários para identificar exclusivamente um usuário. Isso fica para os usuários decidirem.

Um claim que não pode ser adequadamente deixado em aberto pode ser um de "itens", representando o carrinho de compras de um usuário. Este carrinho pode ser preenchido com itens que o usuário está prestes a comprar e, assim, estão associados à sua sessão. Um terceiro (um script do lado do cliente) pode ser capaz de coletar esses itens se tiverem sido

armazenados em um JWT não criptografado, o que pode fazer surgir problemas de privacidade.

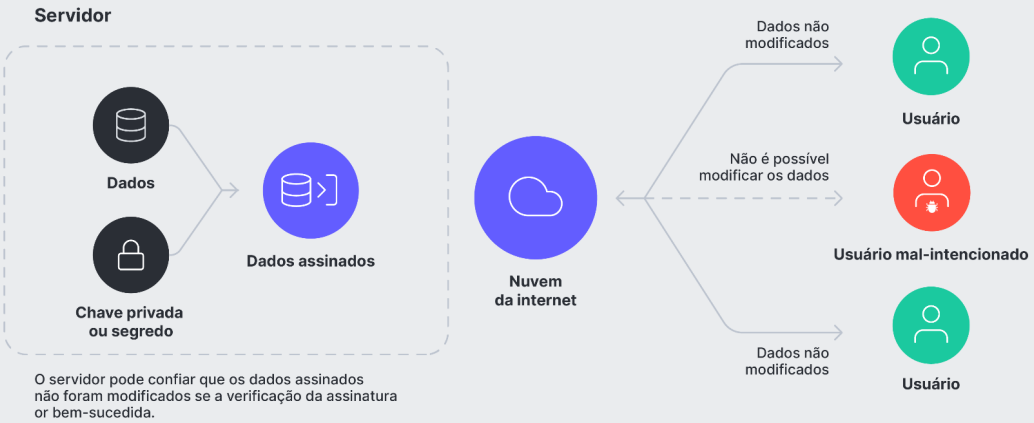


Figura 2.1: Dados assinados do lado do cliente

2.1.1 Considerações de segurança

2.1.1.1 Remoção de assinatura

Um método comum para atacar um JWT assinado é simplesmente remover a assinatura. Os JWTs assinados são construídos a partir de três partes diferentes: header, payload e signature. Essas três partes são codificadas separadamente. Assim, é possível remover a assinatura e, em seguida, *alterar* o header para alegar que o JWT *não está assinado*. O uso descuidado de certas bibliotecas de validação do JWT pode fazer com que tokens não assinados sejam vistos como tokens válidos, permitindo que um invasor modifique o payload a seu critério. Isso é facilmente resolvido garantindo que o aplicativo que executa a validação não considere válidos os JWTs não assinados.

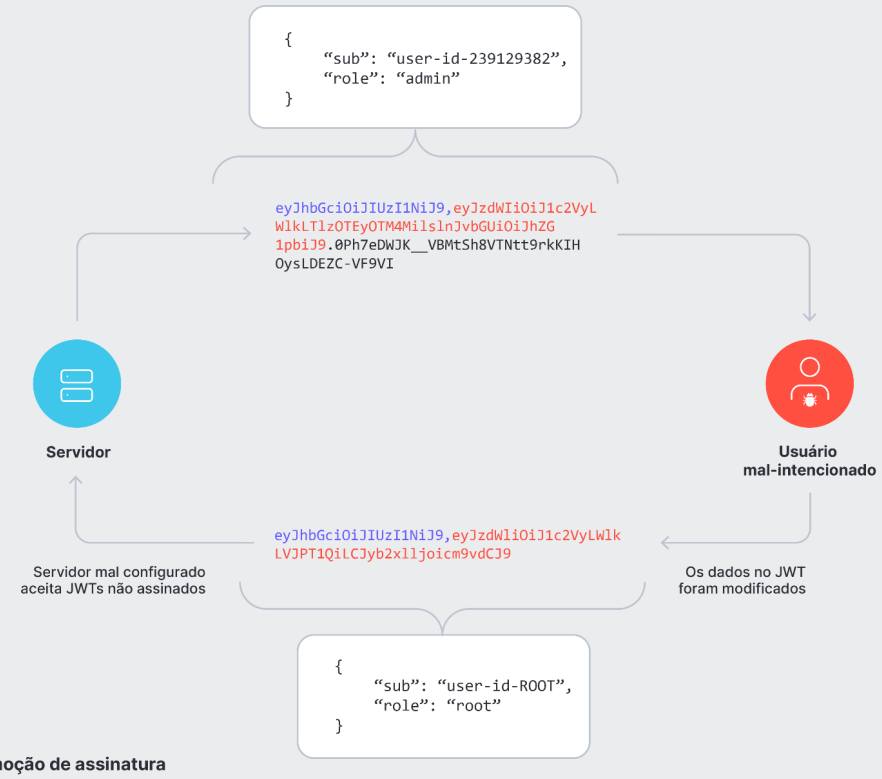


Figura 2.2: Remoção de assinatura

2.1.1.2 Falsificação de solicitação entre sites (CSRF)

Os ataques de falsificação de solicitação entre sites tentam executar solicitações em sites em que o usuário está conectado, enganando o navegador do usuário para que ele envie uma solicitação de um site diferente. Para conseguir isso, um site (ou item) especialmente criado deve conter o URL para o destino. Um exemplo comum é uma tag `<img>` embutida em uma página mal-intencionada com o `src` apontando para o alvo do ataque. Por exemplo:

```
<!-- Isso está incorporado ao site de outro domínio -->
<img src="http://target.site.com/add-user?user=name&grant=admin">
```

A tag `<img>` acima enviará uma solicitação para `target.site.com` sempre que a página que a contém for carregada. Se o usuário já tiver feito login em target.site.com e o site tiver usado um cookie para manter a sessão ativa, esse cookie também será enviado. Se o site de destino não implementar nenhuma técnica de mitigação de CSRF, a solicitação será tratada como uma

solicitação válida em nome do usuário. Os JWTs, como quaisquer outros dados do lado do cliente, podem ser armazenados como cookies.

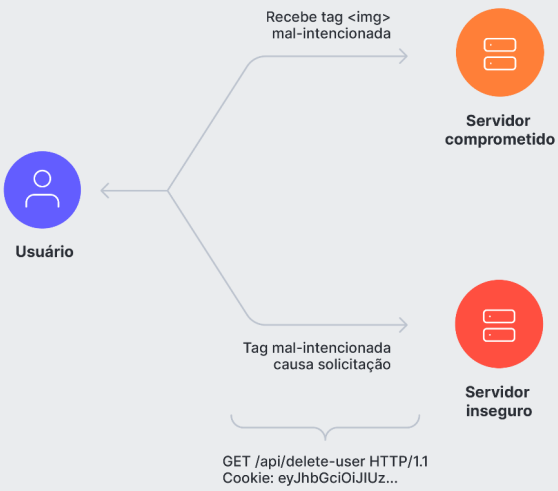


Figura 2.3: Falsificação de solicitação entre sites

JWTs de curta duração podem ajudar nesse caso. Técnicas comuns de mitigação de CSRF incluem cabeçalhos especiais que são adicionados às solicitações apenas quando são realizadas a partir da origem certa, segundo os cookies de sessão e os tokens de solicitação. Se os JWTs (e os dados da sessão) não forem armazenados como cookies, os ataques de CSRF não serão possíveis. Porém, os ataques de scripts entre sites ainda são possíveis.

2.1.1.3 Cross-site Scripting (XSS)

Os ataques de Cross-site Scripting (XSS) tentam injetar JavaScript em sites confiáveis. O JavaScript injetado pode então roubar tokens de cookies e armazenamento local. Se um token de acesso vazar antes de expirar, um usuário mal-intencionado poderá usá-lo para acessar recursos protegidos. Os ataques XSS comuns são geralmente causados pela validação imprópria de dados passados para o back-end (de maneira semelhante aos ataques de injeção SQL).

Um exemplo de um ataque XSS pode estar relacionado à seção de comentários de um site público. Toda vez que um usuário adiciona um comentário, ele é armazenado pelo back-end e exibido aos usuários que carregam a seção de comentários. Se o back-end não sanitizar os

comentários, um usuário mal-intencionado poderá escrever um comentário de tal maneira que ele possa ser interpretado pelo navegador como uma tag <script>. Assim, um usuário mal-intencionado poderia inserir código JavaScript arbitrário e executá-lo no navegador de todos os usuários, roubando credenciais armazenadas como cookies e no armazenamento local.

Figura 2.4: Cross-site Scripting persistente

Figura 2.5: Cross-site Scripting refletido

As técnicas de mitigação dependem da validação adequada de todos os dados passados para o back-end. Em particular, todos os dados recebidos dos clientes devem ser sempre sanitizados. Se cookies forem usados, é possível protegê-los de serem acessados pelo JavaScript definindo o sinalizador HttpOnly. O sinalizador HttpOnly, embora útil, não protegerá o cookie de ataques CSRF.

2.1.2 As sessões do lado do cliente são úteis?

Há prós e contras para qualquer abordagem, e sessões do lado do cliente não são uma exceção. Alguns aplicativos podem exigir grandes sessões. Enviar e receber este estado para cada solicitação (ou grupo de solicitações) pode facilmente superar os benefícios da conversação reduzida no back-end. É necessário certo equilíbrio entre dados do lado do cliente e pesquisas de banco de dados no back-end. Isso depende do modelo de dados do seu aplicativo. Alguns aplicativos não são mapeados bem com as sessões do lado do cliente. Outros podem depender inteiramente de dados do lado do cliente. A palavra final sobre este assunto é sua! Execute benchmarks e estude os benefícios de manter determinado estado do lado do cliente. Os JWTs são muito grandes? Isso tem impacto sobre a largura de banda? Essa largura de banda adicional derruba a latência reduzida no back-end? Pequenas solicitações podem ser agregadas em uma única solicitação maior? Essas solicitações ainda exigem grandes pesquisas de banco de dados? Responder a essas perguntas ajudará você a decidir a abordagem correta.

2.1.3 Exemplo

Para nosso exemplo, faremos um aplicativo de compras simples. O carrinho de compras do usuário será armazenado do lado do cliente. Neste exemplo, existem vários JWTs presentes. Nosso carrinho de compras será um deles.

- Um JWT para o token de ID, um token que carrega as informações de perfil do usuário, útil para a IU.
- Um JWT para interagir com o back-end da API (o token de acesso).
- Um JWT para o nosso estado do lado do cliente: o carrinho de compras.

Eis como o carrinho de compras fica quando decodificado:

```
{
   "items": [
```

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

Cada item é identificado por uma ID numérica. O JWT codificado e assinado fica assim:

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

Para renderizar os itens no carrinho, o front-end só precisa recuperá-lo de seu cookie:

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

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

     const cart = jwt_decode(cartToken).items;

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

Observe que o front-end não verifica a assinatura, ele simplesmente decodifica o JWT para que ele possa exibir seu conteúdo. As verificações reais são executadas pelo back-end. Todos os JWTs são verificados.

Aqui está a verificação do back-end para a validade do carrinho que o JWT implementou como um middleware Express:

```
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: [] };
       }
}
```

Quando os itens são adicionados, o back-end constrói um novo JWT com o novo item e uma nova assinatura:

```
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,
                             cartVerifyJwtOptions);
```

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

    res.end();

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

Observe que os locais prefixados por `/protected` também são protegidos pelo token de acesso da API. Esta é a configuração usando `express-jwt`:

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

Em outras palavras, o endpoint `/protected/add_item` deve primeiro passar pela etapa de validação do token de acesso antes de validar o carrinho. Um token valida o acesso (autorização) à API e o outro token valida a integridade dos dados do lado do cliente (o carrinho).

O token de acesso e o token de ID são atribuídos pela Auth0 ao nosso aplicativo. Isso requer configurar um cliente e um endpoint da API usando o painel Auth0. Estes são então recuperados usando a biblioteca JavaScript da Auth0, chamada pelo nosso front-end:

```
//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();
});
```

O claim de público deve corresponder àquele configurado para seu endpoint de API usando o painel Auth0.

O servidor de autenticação e autorização da Auth0 exibe uma tela de login com nossas configurações e, em seguida, redireciona de volta ao nosso aplicativo em um caminho específico com os tokens solicitados. Estes são tratados pelo nosso back-end, que simplesmente os define como cookies:

```
app.post('/auth', (req, res) => {
    res.cookie('access_token', req.body.access_token, {
```

```
        httpOnly: true,
        maxAge: req.body.expires_in * 1000
    });
    res.cookie('id_token', req.body.id_token, {
        maxAge: req.body.expires_in * 1000
    });
    res.redirect('/');
});
```

A implementação de técnicas de mitigação de CSRF é deixada como um exercício para o leitor. O exemplo completo para este código pode ser encontrado no diretório `samples/stateless-sessions`.

2.2 Identidade federada

Os sistemas de identidade federada permitem que diferentes partes, possivelmente não relacionadas, compartilhem serviços de autenticação e autorização com outras partes. Em outras palavras, a identidade de um usuário é centralizada. Existem várias soluções para a gestão de identidades federada: SAML e OpenID Connect são duas das mais comuns. Certas empresas fornecem produtos especializados que centralizam a autenticação e a autorização. Eles podem implementar um dos padrões mencionados acima ou usar algo completamente diferente. Algumas dessas empresas usam JWTs para esse fim.

O uso de JWTs para autenticação e autorização centralizadas varia de empresa para empresa, mas o fluxo essencial do processo de autorização é:

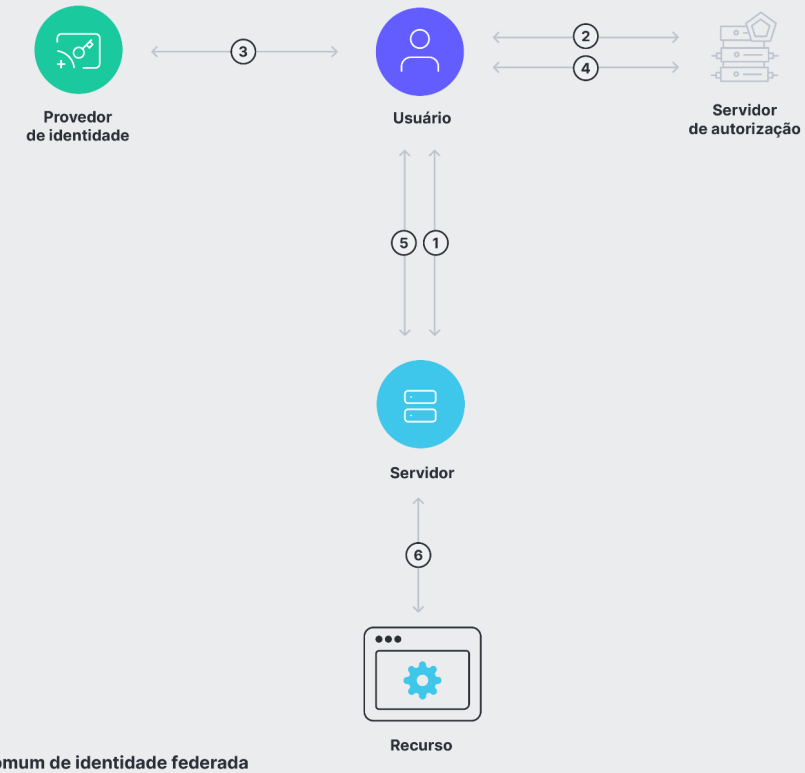


Figura 2.6: Fluxo comum de identidade federada

1. O usuário tenta acessar um recurso controlado por um servidor.
2. O usuário não tem as credenciais adequadas para acessar o recurso; portanto, o servidor redireciona o usuário para o servidor de autorização. O servidor de autorização está configurado para permitir que os usuários façam login usando as credenciais gerenciadas por um provedor de identidade.
3. O usuário é redirecionado pelo servidor de autorização para a tela de login do provedor de identidade.
4. O usuário faz login com sucesso e é redirecionado para o servidor de autorização. O servidor de autorização usa as credenciais fornecidas pelo provedor de identidade para acessar as credenciais exigidas pelo servidor de recursos.
5. O usuário é redirecionado para o servidor de recursos pelo servidor de autorização. A solicitação agora tem as credenciais corretas necessárias para acessar o recurso.
6. O usuário obtém acesso ao recurso com sucesso.

Todos os dados passados de servidor a servidor fluem através do usuário, sendo incorporados às solicitações de redirecionamento (geralmente como parte do URL). Isso torna a segurança de transporte (TLS) e a segurança de dados essenciais.

As credenciais retornadas do servidor de autorização para o usuário podem ser codificadas como um JWT. Se o servidor de autorização permitir logins através de um provedor de identidade (como é o caso neste exemplo), pode-se dizer que o servidor de autorização está fornecendo uma interface unificada e dados unificados (o JWT) ao usuário.

Para nosso exemplo mais adiante nesta seção, usaremos Auth0 como o servidor de autorização e lidaremos com logins através do Twitter, Facebook e um banco de dados de usuários comum.

2.2.1 Tokens de acesso e atualização

Os tokens de acesso e atualização são dois tipos de tokens que você verá muito ao analisar diferentes soluções de identidade federadas. Explicaremos brevemente quais são e como ajudam no contexto de autenticação e autorização.

Ambos os conceitos são geralmente implementados no contexto da especificação OAuth2. A especificação OAuth2 define uma série de etapas necessárias para fornecer acesso a recursos, separando o acesso da propriedade (em outras palavras, permite que várias partes com diferentes níveis de acesso acessem o mesmo recurso). Várias partes dessas etapas são definidas *na implementação*. Ou seja, implementações de OAuth2 concorrentes podem não ser interoperáveis. Por exemplo, o formato binário real dos tokens *não é especificado*. Seu propósito e funcionalidade são.

Tokens de acesso são tokens que dão acesso a recursos protegidos. Esses tokens geralmente têm vida curta e podem ter uma data de validade incorporada. Eles também podem transportar ou estar associados a informações adicionais (por exemplo, um token de acesso pode transportar o endereço IP a partir do qual as solicitações são permitidas). Esses dados adicionais são definidos pela implementação.

Os **tokens de atualização**, por outro lado, permitem que os clientes solicitem novos tokens de acesso. Por exemplo, depois que um token de acesso expirou, um cliente pode executar uma solicitação para um novo token de acesso ao servidor de autorização. Para que esta

solicitação seja atendida, um token de atualização é necessário. Em contraste com os tokens de acesso, os tokens de atualização geralmente são de longa duração.

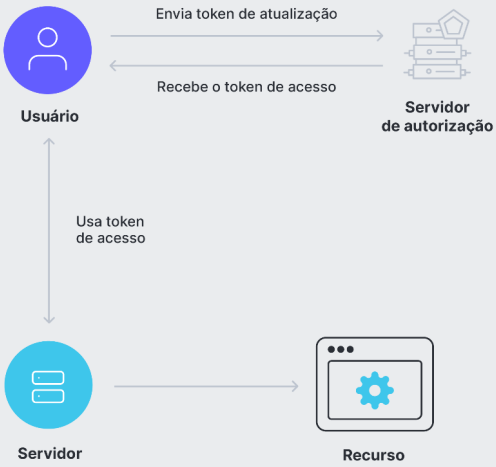


Figura 2.7: Tokens de atualização e de acesso

O aspecto fundamental da separação entre tokens de acesso e de atualização reside na possibilidade de tornar os tokens de acesso fáceis de validar. Um token de acesso que carrega uma assinatura (como um JWT assinado) pode ser validado pelo próprio servidor de recursos. Não há necessidade de entrar em contato com o servidor de autorização para esse fim.

Os tokens de atualização, por outro lado, exigem acesso ao servidor de autorização. Ao manter a validação separada das consultas ao servidor de autorização, é possível ter melhor latência e padrões de acesso menos complexos. A segurança apropriada em caso de vazamentos de token é alcançada reduzindo ao máximo a duração dos tokens de acesso e incorporando verificações adicionais (como verificações de cliente) a eles.

Os tokens de atualização, em virtude de serem de longa duração, precisam ser protegidos contra vazamentos. No caso de um vazamento, uma lista de bloqueio pode ser necessária no servidor (tokens de acesso de curta duração forçam o uso de tokens de atualização, protegendo o recurso depois que ele for colocado na lista de bloqueio e todos os tokens de acesso estiverem expirados).

> **Observação: os conceitos de token de acesso e token de atualização foram introduzidos no OAuth2. OAuth 1.0 e 1.0a usam a palavra token de forma diferente.**

2.2.2 JWTs e OAuth2

Embora o OAuth2 não mencione o formato de seus tokens, os JWTs são uma boa combinação para seus requisitos. JWTs assinados são bons tokens de acesso, pois podem codificar todos os dados necessários para diferenciar os níveis de acesso a um recurso, podem conter uma data de validade e são assinados para evitar consultas de validação em relação ao servidor de autorização. Vários provedores de serviços de identificação federada emitem tokens de acesso no formato JWT.

JWTs também podem ser usados para tokens de atualização. No entanto, há pouca razão para usá-los para esse fim. Como os tokens de atualização requerem acesso ao servidor de autorização, na maioria das vezes um UUID simples será suficiente, pois não há necessidade do token carregar um payload (ele pode ser assinado, no entanto).

2.2.3 JWTs e OpenID Connect

O OpenID Connect é um esforço de padronização para reunir casos de uso típicos do OAuth2 sob uma especificação comum e bem definida. Como muitos detalhes por trás do OAuth2 são deixados à escolha dos implementadores, o OpenID Connect tenta fornecer definições adequadas para as partes ausentes. Especificamente, o OpenID Connect define uma API e um formato de dados para executar fluxos de autorização OAuth2. Além disso, ele fornece uma camada de autenticação construída sobre esse fluxo. O formato de dados escolhido para algumas de suas partes é JSON Web Token. Em particular, o token de ID é um tipo especial de token que transporta informações sobre o usuário autenticado.

2.2.3.1 Fluxos OpenID Connect e JWTs

O OpenID Connect define vários fluxos que retornam dados de maneiras diferentes. Alguns desses dados podem estar no formato JWT.

- **Fluxo de autorização:** o cliente solicita um código de autorização ao endpoint de autorização (`/authorize`). Este código pode ser usado novamente no endpoint do token

(`/token`) para solicitar um token de ID (no formato JWT), um token de acesso ou um token de atualização.

- **Fluxo implícito:** o cliente solicita tokens diretamente do endpoint de autorização (`/authorize`). Os tokens são especificados na solicitação. Se um token de ID for solicitado, ele será retornado no formato JWT.
- **Fluxo híbrido:** o cliente solicita um código de autorização e certos tokens do endpoint de autorização (`/authorize`). Se um token de ID for solicitado, ele é retornado no formato JWT. Se um token de ID não for solicitado nesta etapa, ele poderá ser solicitado diretamente do endpoint do token (`/token`).

2.2.4 Exemplo

Para este exemplo, usaremos Auth0 como servidor de autorização. O Auth0 permite que diferentes provedores de serviços de identificação sejam definidos dinamicamente. Em outras palavras, sempre que um usuário tenta fazer login, as alterações feitas no servidor de autorização podem permitir que os usuários façam login com diferentes provedores de serviços de identificação (como Twitter, Facebook, etc.). Os aplicativos não precisam se comprometer com provedores específicos uma vez implantados. Então, nosso exemplo pode ser bastante simples. Configuramos a tela de login Auth0 usando a biblioteca Auth0.js em todos os nossos servidores de amostra. Quando um usuário fizer login em um servidor, ele também terá acesso aos outros servidores (mesmo que não estejam interconectados).

Figura 2.8: Auth0 como servidor de autorização

2.2.4.1 Configuração de Auth0 Lock para aplicativos Node.js

A configuração da biblioteca Auth0 pode ser feita da seguinte forma. Usaremos o mesmo exemplo usado para o exemplo de sessões sem estado:

```
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();
        promt: 'none'
    });
});
```

Observe o uso do parâmetro `prompt: 'none'` para a chamada de `autorização`. A chamada de `autorização` redireciona o usuário para o servidor de autorização. Com o parâmetro none, se o usuário já tiver dado autorização para que um aplicativo use suas credenciais para acessar um recurso protegido, o servidor de autorização simplesmente redirecionará de volta ao aplicativo. Para o usuário, é como se ele já estivesse conectado no aplicativo.

Em nosso exemplo, há dois aplicativos: `app1.com` e `app2.com`. Depois que um usuário autorizar ambos os aplicativos (o que acontece apenas uma vez: na primeira vez que o usuário faz login), quaisquer logins subsequentes em qualquer um dos dois aplicativos também permitirão que o outro aplicativo faça login sem apresentar nenhuma tela de login.

Para testar isso, consulte o arquivo `README` para o exemplo localizado no diretório `samples/single-sign-on-federated-identity` para configurar os dois aplicativos e executá-los. Quando ambos estiverem em execução, vá para app1.com:3000 e app2.com:3001 e faça login. Em seguida, saia dos dois aplicativos. Agora tente fazer login em um deles. Em seguida, volte para o outro e faça login. Você perceberá que a tela de login estará ausente em ambos os aplicativos. O servidor de autorização se lembra de logins anteriores e pode emitir novos tokens de acesso quando solicitado por qualquer um desses aplicativos. Assim, desde que o usuário tenha uma sessão de servidor de autorização, já estará conectado a ambos os aplicativos.

A implementação de técnicas de mitigação de CSRF é deixada como um exercício para o leitor.

Capítulo 3

JSON Web Tokens em detalhes

Conforme descrito no Capítulo 1, todos os JWTs são construídos a partir de três elementos diferentes: header, payload e os dados de assinatura/criptografia. Os dois primeiros elementos são objetos JSON de uma determinada estrutura. O terceiro é dependente do algoritmo usado para assinatura ou criptografia e, no caso dos JWTs *não criptografados*, ele é omitido. Os JWTs podem ser codificados em uma *representação compacta* conhecida como *Serialização Compacta JWS/JWE*.

> As especificações de JWS e JWE definem um terceiro formato de serialização conhecido como *serialização JSON*, uma representação não compacta que permite várias assinaturas ou destinatários no mesmo JWT. Isso é explicado em detalhes nos Capítulos 4 e 5.

A serialização compacta é uma codificação segura para URL Base64 dos bytes UTF-8 dos dois primeiros elementos JSON (header e payload) e dos dados, conforme necessário, para assinatura ou criptografia (que não é um objeto JSON em si). Esses dados também são codificados por Base64-URL. Esses três elementos são separados por pontos (".").

> O JWT usa uma variante da codificação Base64 que é segura para URLs. Esta codificação basicamente substitui os caracteres "+" e "/" por caracteres "-" e "_", respectivamente. O padding também é removido. Esta variante é conhecida como base64url. Observe que todas as referências à codificação Base64 neste documento se referem a esta variante.

A sequência resultante é uma string imprimível como a seguinte (novas linhas inseridas para legibilidade):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Observe os pontos que separam os três elementos do JWT (em ordem: header, payload e assinatura).

Neste exemplo, o header decodificado é:

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

O payload decodificado é:

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

O segredo necessário para verificar a assinatura é secreto.

> JWT.io é uma brincadeira interativa para aprender mais sobre os JWTs. Copie o token acima e veja o que acontece quando você o edita.

3.1 O header

Cada JWT carrega um header (também conhecido como o *header JOSE*) com claims sobre si mesmo. Esses claims estabelecem os algoritmos usados, se o JWT está assinado ou criptografado e, em geral, como analisar o restante do JWT.

De acordo com o tipo de JWT em questão, mais campos podem ser obrigatórios no header. Por exemplo, os JWTs criptografados carregam informações sobre os algoritmos criptográficos usados para criptografia de chaves e criptografia de conteúdo. Esses campos não estão presentes para JWTs *não criptografados*.

O único claim obrigatório para um header JWT não criptografado é o claim alg:

- **alg**: o algoritmo principal em uso para assinar e/ou descriptografar este JWT.

Para JWTs não criptografados, este claim deve ser definido para o valor none.

Claims de header opcionais incluem os claims `typ` e `cty`:

- **typ**: o tipo de mídia do próprio JWT. Este parâmetro destina-se apenas a ser usado como uma ajuda para usos em que os JWTs podem ser combinados com outros objetos com um header JOSE. Na prática, isso raramente acontece. Quando presente, este claim deve ser definido para o `JWT` de valor.
- cty: o tipo de conteúdo. A maioria dos JWTs carrega claims específicos mais dados arbitrários como parte do seu payload. Para este caso, o claim de tipo de conteúdo não deve ser definido. Para casos em que o payload é um JWT em si (um JWT aninhado), este claim precisa estar presente e carregar o JWT de valor. Isso diz à implementação que o processamento adicional do JWT aninhado é necessário. Os JWTs aninhados são raros, portanto, o claim cty raramente está presente nos headers.

Assim, para JWTs não criptografados, o header é simplesmente:

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

que é codificado para:

`eyJhbGciOiJub25lIn0`

> É possível adicionar outros claims definidos pelo usuário ao header. Isso geralmente é de uso limitado, a menos que certos metadados específicos do usuário sejam necessários no caso de JWTs criptografados antes da descriptografia.

3.2 O payload

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

```
}
```

O payload é o elemento onde todos os dados interessantes do usuário são normalmente adicionados. Além disso, certos claims definidos na especificação também podem estar presentes. Assim como o header, o payload é um objeto JSON. Nenhum claim é obrigatório, embora claims específicos tenham um significado definido. A especificação do JWT especifica que os claims que não são compreendidos por uma implementação devem ser ignorados. Os claims com significados específicos anexados a eles são conhecidos como *claims registrados*.

3.2.1 Claims registrados

- **iss:** da palavra *issuer* (emissor, em inglês). Uma string ou URI sensível a maiúsculas e minúsculas que identifica exclusivamente a parte que emitiu o JWT. A sua interpretação é específica do aplicativo (não existe autoridade central que gerencie os emissores).
- **sub:** da palavra *subject* (assunto, em inglês). Uma string ou URI sensível a maiúsculas e minúsculas que identifica exclusivamente a parte sobre a qual este JWT transporta informações. Em outras palavras, os claims contidos neste JWT são declarações sobre esta parte. A especificação do JWT especifica que esse claim deve ser único no contexto do emissor ou, nos casos em que isso não for possível, globalmente único. O tratamento deste claim é específico do aplicativo.
- **aud:** da palavra *audience* (público, em inglês). Uma única string ou URI sensível a maiúsculas e minúsculas ou um conjunto de tais valores que identificam exclusivamente os destinatários pretendidos deste JWT. Em outras palavras, quando este claim estiver presente, a parte que lê os dados neste JWT precisa se encontrar no claim *aud* ou desconsiderar os dados contidos no JWT. Como no caso dos claims *iss* e *sub*, este claim é específico do aplicativo.
- **exp:** da palavra *expiration* (validade). Um número que representa uma data e hora específicas no formato "segundos desde a época", conforme definido pelo POSIX. Este claim define o momento exato do qual este JWT é considerado *inválido*. Algumas implementações podem permitir certo desvio entre os relógios (considerando que este JWT seja válido por alguns minutos após a data de expiração).
- **nbf:** de *not before* (não antes de) (hora). É o oposto do claim *exp*. Um número que representa uma data e hora específicas no formato "segundos desde a época",

conforme definido pelo POSIX. Este claim define o momento exato a partir do qual este JWT é considerado *válido*. A hora e a data atuais precisam ser iguais ou superiores a esta data e hora. Algumas implementações podem permitir um certo desvio.

- **iat:** de *issued at* (emitido às) (hora). Um número que representa uma data e hora específicas (no mesmo formato que *exp* e *nbf*) em que este JWT foi emitido.
- **jti:** de *JWT ID*. Uma string que representa um identificador exclusivo para este JWT. Este claim pode ser usado para diferenciar JWTs com outro conteúdo semelhante (evitando repetições, por exemplo). Cabe à implementação garantir a singularidade.

Como deve ter notado, todos os nomes são curtos. Isso está em conformidade com um dos requisitos de projeto: manter os JWTs os menores possível.

String ou URI: de acordo com a especificação do JWT, um URI é interpretado como qualquer string contendo um caractere :. Cabe à implementação fornecer valores válidos.

3.2.2 Claims públicos e privados

Todos os claims que não são parte da seção *claims registrados* são claims **públicos** ou **privados**.

- Claims **privados:** são aqueles definidos pelos *usuários* (consumidores e produtores) dos JWTs. Em outras palavras, são claims ad hoc usados para um caso específico. Como tal, deve-se tomar cuidado para evitar colisões.
- Claims **públicos:** são claims que são *registrados* no registro IANA JSON Web Token Claims (um registro onde os usuários podem registrar seus claims e, assim, evitar colisões) ou nomeados usando um nome resistente a colisões (por exemplo, anexando um namespace ao seu nome).

Na prática, a maioria dos claims é registrada ou privada. Em geral, a maioria dos JWTs é emitida com um propósito específico e um conjunto claro de usuários potenciais em mente. Isso torna a questão de escolher nomes resistentes a colisões simples.

Assim como nas regras de análise JSON, os claims duplicados (chaves JSON duplicadas) são tratados mantendo apenas a última ocorrência como válida. A especificação do JWT também torna possível para as implementações considerar JWTs com claims duplicados como

inválidos. Na prática, se você não tiver certeza sobre a implementação que lidará com seus JWTs, tome cuidado para evitar claims duplicados.

3.3 JWTs não protegidos

Com o que aprendemos até agora, é possível construir JWTs não protegidos. Estes são os JWTs mais simples, formados por um header simples (geralmente estático):

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

e um payload definido pelo usuário. Por exemplo:

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

Como não há assinatura ou criptografia, este JWT é codificado como simplesmente dois elementos (novas linhas inseridas para legibilidade):

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

Um JWT não protegido como o mostrado acima pode ser adequado para uso do lado do cliente. Por exemplo, se a ID da sessão é um número difícil de adivinhar, e o restante dos dados só é usado pelo cliente para construir uma visualização, o uso de uma assinatura é supérfluo. Esses dados podem ser usados por um aplicativo web de página única para construir uma visualização com o nome "pretty" para o usuário sem passar pelo back-end

enquanto ele é redirecionado para sua última página visitada. Mesmo que um usuário mal-intencionado modificasse esses dados, ele não ganharia nada.

> Observe o ponto final (.) na representação compacta. Como não há assinatura, é simplesmente uma string vazia. Porém, o ponto ainda é adicionado.

Na prática, entretanto, os JWTs não seguros são raros.

3.4 Criação de um JWT não protegido

Para chegar à representação compacta das versões JSON do header e do payload, execute as seguintes etapas:

1. Pegue o header como uma matriz de bytes de sua representação UTF-8. A especificação do JWT *não* requer que o JSON seja minimizado ou despojado de caracteres sem sentido (como espaço em branco) antes da codificação.
2. Codifique a matriz de bytes usando o algoritmo Base64-URL, removendo sinais de igual (=) posteriores.
3. Pegue o payload como uma matriz de bytes de sua representação UTF-8. A especificação do JWT *não* requer que o JSON seja minimizado ou despojado de caracteres sem sentido (como espaço em branco) antes da codificação.
4. Codifique a matriz de bytes usando o algoritmo Base64-URL, removendo sinais de igual (=) posteriores.
5. Concatene as strings resultantes, colocando primeiro o header, seguido por um caractere de “.”, seguido pelo payload.

A validação do header e do payload (com relação à presença dos claims necessários e ao uso correto de cada um) deve ser realizada antes da codificação.

Figura 3.1: Geração de JWT não protegido compacto

3.4.1 Código de exemplo

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

O exemplo completo está no arquivo coding.js do código de exemplo que acompanha.

3.5 Análise de um JWT não protegido

Para chegar à representação JSON do formulário de serialização compacta, execute as seguintes etapas:

1. Encontre o primeiro caractere de ponto ".". Tome a string antes dele (sem incluí-lo).
2. Decodifique a string usando o algoritmo Base64-URL. O resultado é o header do JWT.
3. Tome a string após o ponto da etapa 1.
4. Decodifique a string usando o algoritmo Base64-URL. O resultado é o payload do JWT.

As strings JSON resultantes podem ser "embelezadas" adicionando espaço em branco conforme a necessidade.

3.5.1 Código de exemplo

```
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),
    };
}
```

O exemplo completo está no arquivo `coding.js` do código de exemplo que acompanha.

Capítulo 4

JSON Web Signatures

JSON Web Signatures são provavelmente o recurso mais útil dos JWTs. Combinando um formato de dados simples com uma série bem definida de algoritmos de assinatura, os JWTs estão rapidamente se tornando o formato ideal para compartilhar dados com segurança entre clientes e intermediários.

O objetivo de uma assinatura é permitir que uma ou mais partes estabeleçam a *autenticidade* do JWT. A autenticidade neste contexto significa que os dados contidos no JWT não foram violados. Em outras palavras, qualquer parte que possa realizar uma *verificação de assinatura* pode confiar no conteúdo fornecido pelo JWT. É importante ressaltar que uma assinatura não impede que outras partes *leiam* o conteúdo dentro do JWT. Isso é o que a criptografia deve fazer, e falaremos sobre isso mais tarde no Capítulo 5.

O processo de verificação de assinatura de um JWT é conhecido como *validação* ou *validar* um token. Um token é considerado válido quando todas as restrições especificadas em seu header e payload são satisfeitas. Este é um aspecto *muito importante* dos JWTs: as implementações são necessárias para verificar um JWT até o ponto especificado por seu header e payload (e, adicionalmente, o que quer que o usuário exija). Assim, um JWT pode ser considerado válido *mesmo que não tenha uma assinatura* (se o header tiver o claim *alg* definido como none). Além disso, mesmo que um JWT tenha uma assinatura válida, pode ser considerado inválido por outros motivos (por exemplo, pode ter expirado, de acordo com o claim *exp*). Um ataque comum contra JWTs assinados remove a assinatura e, em seguida, altera o header para torná-lo um JWT não seguro. É responsabilidade do usuário garantir que os JWTs sejam validados de acordo com seus próprios requisitos.

Os JWTs assinados são definidos na especificação de JSON Web Signature, RFC 7515.

4.1 Estrutura de um JWT assinado

Abordamos a estrutura de um JWT no Capítulo 3. Vamos revisá-la aqui, com atenção especial ao componente de assinatura.

Um JWT assinado é composto por três elementos: header, payload e assinatura (novas linhas inseridas para legibilidade):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

O processo para decodificar os dois primeiros elementos (header e payload) é idêntico ao caso dos JWTs não protegidos. O algoritmo e o código de exemplo podem ser encontrados no final do Capítulo 3.

```
{
   "alg": "HS256";
}

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

Os JWTs assinados, no entanto, carregam um elemento adicional: a assinatura. Este elemento aparece após o último ponto (.) no formulário de serialização compacta.

Há vários tipos de algoritmos de assinatura disponíveis de acordo com a especificação do JWS, portanto, a maneira como esses octetos são interpretados varia. A especificação do JWS requer que um único algoritmo seja suportado por todas as implementações conformes:

- HMAC usando SHA-256, chamado `HS256` na especificação do JWA.

A especificação também define uma série de algoritmos *recomendados*:

- RSASSA PKCS1 v1.5 usando SHA-256, chamado `RS256` na especificação do JWA.
- ECDSA usando P-256 e SHA-256, chamado `ES256` na especificação do JWA.

JWA é a especificação de JSON Web Algorithms, RFC 7518.

Esses algoritmos serão explicados em detalhes no Capítulo 7. Neste capítulo, focaremos nos aspectos práticos de seu uso.

Os outros algoritmos suportados pela especificação, em capacidade opcional, são:

- HS384, HS512: variações SHA-384 e SHA-512 do algoritmo HS256.
- RS384, RS512: variações SHA-384 e SHA-512 do algoritmo RS256.
- ES384, ES512: variações SHA-384 e SHA-512 do algoritmo ES256.
- PS256, PS384, PS512: RSASSA-PSS + MGF1 com variantes SHA256/384/512.

São, essencialmente, variações dos três principais algoritmos exigidos e recomendados. O significado desses acrônimos se tornará mais claro no Capítulo 7.

4.1.1 Visão geral do algoritmo para serialização compacta

Para discutir esses algoritmos em geral, vamos primeiro definir algumas funções em um ambiente JavaScript 2015:

- **base64:** uma função que recebe um array de octetos e retorna um novo array de octetos usando o algoritmo Base64-URL.
- **utf8:** uma função que recebe texto em qualquer codificação e retorna um array de octetos com codificação UTF-8.
- **JSON.stringify:** uma função que pega um objeto JavaScript e o serializa na forma de string (JSON).
- **sha256:** uma função que pega um array de octetos e retorna um novo array de octetos usando o algoritmo SHA-256.
- **hmac:** uma função que pega uma função SHA, um array de octetos e um segredo e retorna um novo array de octetos usando o algoritmo HMAC.
- **rsassa:** uma função que pega uma função SHA, um array de octetos e a chave privada e retorna um novo array de octetos usando o algoritmo RSASSA.

Para algoritmos de assinatura baseados em HMAC:

```
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}`;
```

Para algoritmos de assinatura de chave pública:

```
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}`;
```

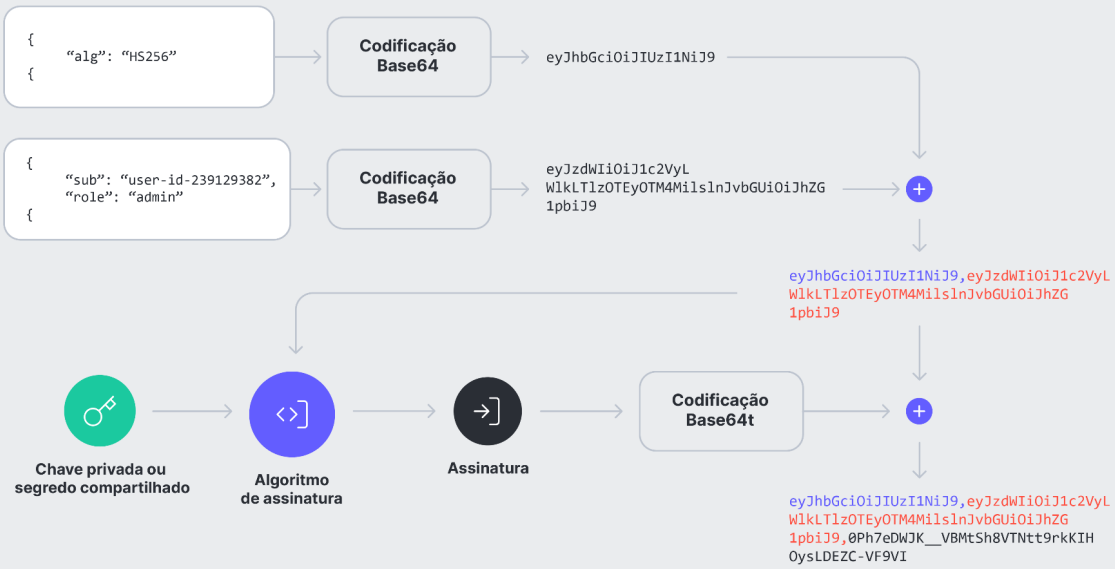


Figura 4.1: Serialização compacta JWS

Os detalhes completos desses algoritmos são mostrados no Capítulo 7.

4.1.2 Aspectos práticos de algoritmos de assinatura

Todos os algoritmos de assinatura permitem estabelecer a autenticidade dos dados contidos no JWT. A maneira de fazer isso pode variar.

HMAC (Keyed-Hash Message Authentication Code) é um algoritmo que combina um determinado payload com um *segredo* usando uma função hash criptográfica. O resultado é um código que pode ser usado para verificar uma mensagem *somente* se ambas as partes, geradoras e verificadoras, souberem o segredo. Em outras palavras, os **HMACs permitem que as mensagens sejam verificadas por meio de segredos compartilhados.**

A função hash criptográfica usada em HS256, o algoritmo de assinatura mais comum para JWTs, é SHA-256. O SHA-256 é explicado em detalhes no Capítulo 7. Funções de hash criptográficas pegam uma mensagem de comprimento arbitrário e produzem uma saída de comprimento fixo. A mesma mensagem sempre produzirá a mesma saída. A parte *criptográfica* de uma função hash garante que seja matematicamente inviável recuperar a mensagem original usando a saída da função. Desta forma, funções de hash criptográficas são funções *unidirecionais* que podem ser usadas para identificar mensagens sem realmente compartilhar a mensagem. Uma pequena variação na mensagem (um único byte, por exemplo) produzirá uma saída totalmente diferente.

RSASSA é uma variação do algoritmo RSA (explicado no Capítulo 7) adaptada para assinaturas. RSA é um algoritmo de chave pública. Algoritmos de chave pública geram chaves divididas: uma chave pública e uma chave privada. Nesta variação específica do algoritmo, a chave privada pode ser usada tanto para criar uma mensagem assinada quanto para verificar sua autenticidade. A chave pública, em contraste, só pode ser usada para verificar a autenticidade de uma mensagem. Assim, este esquema permite a distribuição segura de uma mensagem one-to-many. Os destinatários podem verificar a autenticidade de uma mensagem mantendo uma cópia da chave pública associada a ela, mas não podem criar novas mensagens com ela. Isso permite cenários de uso diferentes dos esquemas de assinatura de segredo compartilhado, como o HMAC. Com o HMAC + SHA-256, qualquer parte que possa verificar uma mensagem também pode *criar novas mensagens*. Por exemplo, se um usuário legítimo se tornasse mal-intencionado, poderia modificar as mensagens sem que as outras partes percebessem. Com um esquema de chave pública, um usuário que se tornasse mal-intencionado só teria a chave pública em sua posse e, portanto, não poderia criar novas mensagens assinadas com ela.

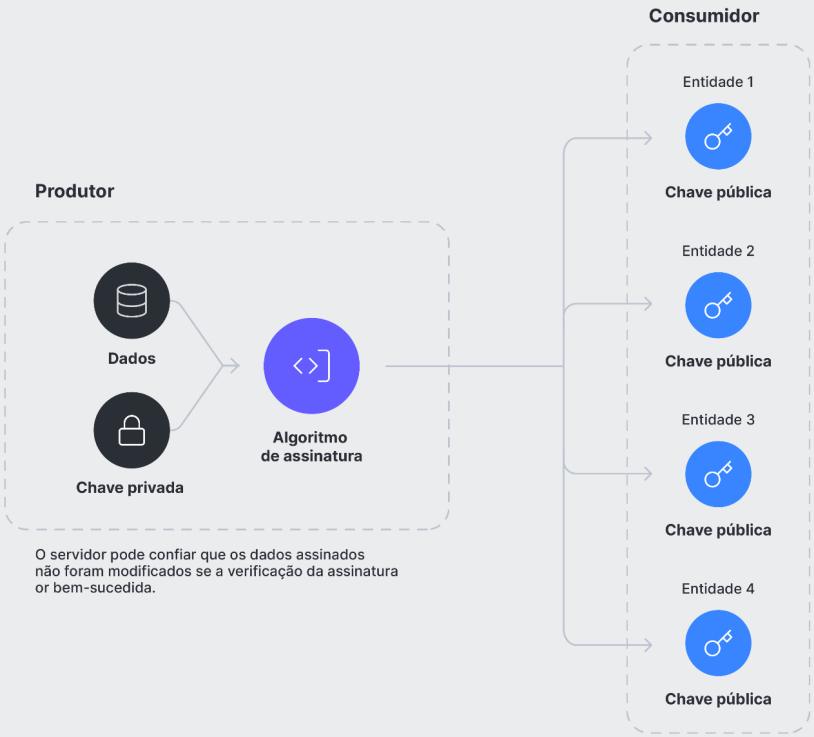


Figura 4.2: Assinatura one-to-many

A criptografia de chave pública permite outros cenários de uso. Por exemplo, usando uma variação do mesmo algoritmo RSA, é possível criptografar mensagens usando a chave pública. Essas mensagens só podem ser descriptografadas usando a chave privada. Isso permite que um canal de comunicação seguro **many-to-one** seja construído. Essa variação é usada para JWTs criptografados, que são discutidos no `<div id="Capítulo 5"></div>`

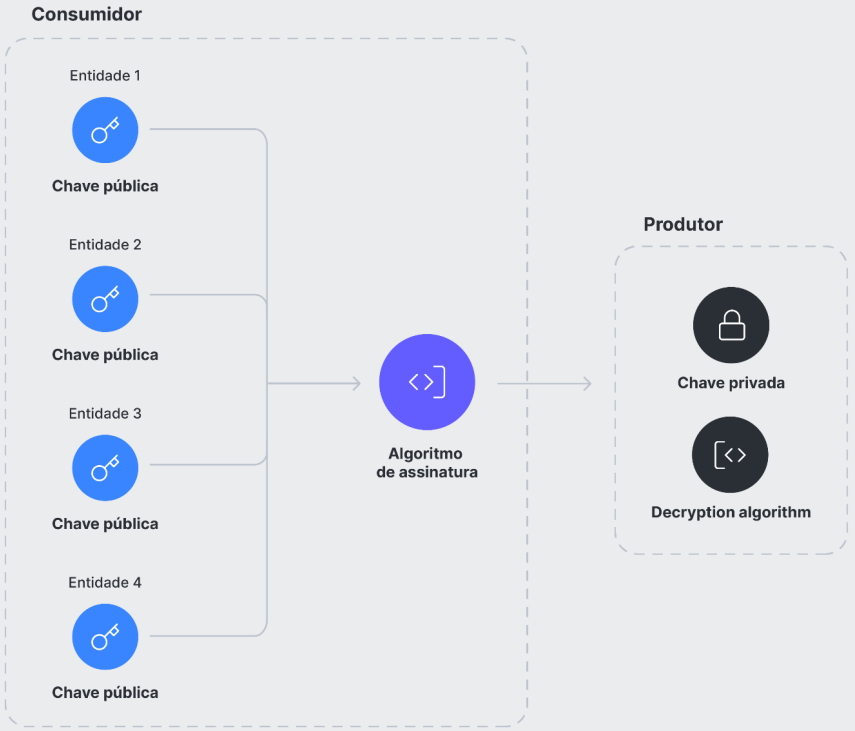


Figura 4.3: Criptografia many-to-one

O algoritmo ECDSA (Algoritmo de assinatura digital de curva elíptica) é uma alternativa ao RSA. Este algoritmo também gera um par de chaves pública e privada, mas a matemática por trás dele é diferente. Essa diferença reduz os requisitos de hardware em relação ao RSA, para garantias de segurança semelhantes.

Estudaremos esses algoritmos em mais detalhes no Capítulo 7.

4.1.3 Claims do header do JWS

O JWS permite casos de uso especiais que forçam o header a carregar mais claims. Por exemplo, para algoritmos de assinatura de chave pública, é possível incorporar o URL à chave pública como um claim. O que se segue é a lista de claims de header registrados disponíveis para tokens JWS. Todos esses claims são *adicionais* àqueles disponíveis para JWTs não protegidos e são opcionais, dependendo de como o JWT assinado deve ser usado.

- **jku:** URL definido por JSON Web Key (JWK). Um URI apontando para um conjunto de chaves públicas codificadas por JSON usadas para assinar este JWT. A segurança do transporte (como TLS para HTTP) deve ser usada para recuperar as chaves. O formato das chaves é um Conjunto JWK (consulte o Capítulo 6).
- **jwk:** JSON Web Key. A chave usada para assinar este JWT no formato de JSON Web Key (consulte o Capítulo 6).
- **kid:** ID da chave. Uma sequência de caracteres definida pelo usuário representando uma única chave usada para assinar este JWT. Este claim é usado para sinalizar alterações de assinatura de chave para os destinatários (quando várias chaves são usadas).
- **x5u:** X.509 URL. Um URI apontando para um conjunto de certificados públicos X.509 (um padrão de formato de certificado) codificados em formato PEM. O primeiro certificado no conjunto deve ser o usado para assinar este JWT. Cada certificado subsequente assina o anterior, completando assim a cadeia de certificados. X.509 é definido no RFC 5280. A segurança do transporte é necessária para a transferência dos certificados.
- **x5c:** cadeia de certificados X.509. Um array JSON de certificados X.509 usado para assinar este JWS. Cada certificado deve ser o valor codificado Base64 de sua representação DER PKIX. O primeiro certificado no array deve ser o usado para assinar este JWT, seguido pelo restante dos certificados na cadeia de certificados.
- **x5t:** impressão digital SHA-1 do certificado X.509. A impressão digital SHA-1 do certificado codificado por X.509 DER usado para assinar este JWT.
- **x5t#S256:** idêntico a **x5t**, mas usa SHA-256 em vez de SHA-1.
- **typ:** idêntico ao valor typ para JWTs não criptografados, com valores adicionais “JOSE” e “JOSE+JSON” usados para indicar serialização compacta e serialização JSON, respectivamente. Isso é usado apenas nos casos em que objetos com headers JOSE semelhantes são combinados com este JWT em um único contêiner.
- **crit:** de critical. Um array de strings com os nomes de claims que estão presentes neste mesmo header usados como extensões definidas pela implementação que devem ser tratadas por analisadores deste JWT. Ele deve conter os nomes dos claims ou não estar presente (o array vazio não é um valor válido).

4.1.4 Serialização JWS JSON

A especificação JWS define um tipo diferente de formato de serialização que não é compacto. Esta representação permite várias assinaturas no mesmo JWT assinado. É conhecido como *Serialização JWS JSON*.

No formulário de Serialização JWS JSON, os JWTs assinados são representados como texto imprimível com formato JSON (ou seja, o que você obteria chamando `JSON.stringify` em um navegador). Um objeto JSON superior que carrega os seguintes pares de chave-valor é necessário:

- **payload:** uma string codificada em Base64 do objeto de payload JWT real.
- **signatures:** um array de objetos JSON que carrega assinaturas. Esses objetos são definidos a seguir.

Por sua vez, cada objeto JSON dentro do array de **signatures** deve conter os seguintes pares chave-valor:

- **protected:** uma string codificada em Base64 do header do JWS. Claims contidos neste header são protegidos pela assinatura. Este header é necessário somente se não houver headers desprotegidos. Se os headers desprotegidos estiverem presentes, esse header pode ou não estar presente.
- **header:** um objeto JSON contendo claims de headers. Este header está desprotegido pela assinatura. Se nenhum header protegido estiver presente, esse elemento é obrigatório. Se um header protegido estiver presente, esse elemento é opcional.
- **signature:** uma string codificada em Base64 da assinatura do JWS.

Em contraste com o formato de serialização compacta (onde apenas um header protegido está presente), a serialização JSON admite dois tipos de headers: **protegidos** e **desprotegidos**. O header protegido é validado pela assinatura. O header desprotegido não é validado por ela. Cabe à implementação ou ao usuário escolher quais claims colocar em qualquer um deles. Pelo menos um desses headers precisa estar presente. Ambos podem estar presentes ao mesmo tempo também.

Quando os headers protegidos e desprotegidos estão presentes, o header JOSE real é construído a partir da união dos elementos em ambos os headers. Nenhum claim duplicado pode estar presente.

O exemplo a seguir é retirado do JWS RFC:

```
{
   "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogIm
               h0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
   "signatures": [
     {
       "protected": "eyJhbGciOiJSUzI1NiJ9",
       "header": { "kid": "2010-12-29" },
       "signature":
         "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AA
         uHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAyn
         RFdiuB--f_nZLgrnbyTyWzO5vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB
         _eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqvhJ1phCnvWh6
         IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrBp0igcN_IoypGlU
         PQGe77Rw"
     },
     {
       "protected": "eyJhbGciOiJFUzI1NiJ9",
       "header": { "kid": ""e9bc097a-ce51-4036-9562-d2ade882db0d" },
       "signature": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDx
                     w5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
     }
   ]
}
```

Este exemplo codifica duas assinaturas para o mesmo payload: uma assinatura RS256 e uma assinatura ES256.

4.1.4.1 Serialização JSON do JWS achatada

A serialização JSON do JWS define um formato simplificado para JWTs com apenas uma única assinatura. Este formato é conhecido como *serialização JSON do JWS achatada*. A serialização achatada remove o array de *signatures* e coloca os elementos de uma única assinatura no mesmo nível que o elemento *payload*.

Por exemplo, ao remover uma das assinaturas do exemplo anterior, um objeto de serialização JSON achatado seria:

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

4.2 Assinatura e validação de tokens

Os algoritmos usados para assinar e validar tokens são explicados em detalhes no Capítulo 7. Usar JWTs assinados é simples o suficiente na prática para que você possa aplicar os conceitos explicados até agora para usá-los de forma eficaz. Além disso, existem boas bibliotecas que você pode usar para implementá-las convenientemente. Vamos revisar os algoritmos necessários e recomendados usando a mais popular dessas bibliotecas para JavaScript. Exemplos de outras linguagens e bibliotecas populares podem ser encontrados no código que acompanha.

Os exemplos a seguir usam a conhecida biblioteca JavaScript `jsonwebtoken`.

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

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

4.2.1 HS256: HMAC + SHA-256

Assinaturas HMAC requerem um segredo compartilhado. Qualquer string servirá:

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

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

Verificar o token também é fácil:

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

A biblioteca `jsonwebtoken` verifica a validade do token com base na assinatura e na data de validade. Nesse caso, se o token fosse verificado após 5 segundos de ser criado, ele seria considerado inválido e uma exceção seria lançada.

4.2.2 RS256: RSASSA + SHA256

Assinar e verificar tokens assinados pelo RS256 também é fácil. A única diferença está no uso de um par de chaves privada/pública em vez de um segredo compartilhado. Existem várias maneiras de criar chaves RSA. OpenSSL é uma das bibliotecas mais populares para criação e gerenciamento de chaves:

```
# Gerar uma chave privada
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt
rsa_keygen_bits:2048
# Obter a chave pública a partir da chave privada
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Ambos os arquivos PEM são arquivos de texto simples. Seu conteúdo pode ser copiado e colado em seus arquivos fonte JavaScript e passado para a biblioteca `jsonwebtoken`.

```
// Você pode obter isso em private_key.pem acima.
const privateRsaKey = `<SUA-CHAVE-RSA-PRIVADA>`;

const signed = jwt.sign(payload, privateRsaKey, {
    algorithm: 'RS256',
    expiresIn: '5s'
});
// Você pode obter isso em public_key.pem acima.
const publicRsaKey = `<SUA-CHAVE-RSA-PÚBLICA>`;

const decoded = jwt.verify(signed, publicRsaKey, {
    // Nunca se esqueça de  tornar explícito para evitar
    // ataques de remoção de assinatura.
    algorithms: ['RS256'],
});
```

4.2.3ES256: ECDSA usando P-256 e SHA-256

Os algoritmos ECDSA também fazem uso de chaves públicas. A matemática por trás do algoritmo é diferente, então as etapas para gerar as chaves também são diferentes. O "P-256" no nome deste algoritmo nos diz exatamente qual versão do algoritmo usar (mais detalhes sobre isso no Capítulo 7). Podemos usar OpenSSL para gerar a chave também:

```
# Gerar uma chave privada (prime256v1 é o nome dos parâmetros usados
# para gerar a chave, isso é o mesmo que P-256 na especificação do JWA).
openssl ecparam -name prime256v1 -genkey -noout -out ecdsa_private_key.pem
# Obter a chave pública a partir da chave privada
openssl ec -in ecdsa_private_key.pem -pubout -out ecdsa_public_key.pem
```

Se você abrir esses arquivos, notará que há muito menos dados neles. Este é um dos benefícios do ECDSA em relação ao RSA (mais sobre isso no Capítulo 7). Os arquivos gerados também estão no formato PEM, então simplesmente colá-los em sua fonte será suficiente.

```
// Você pode obter isso em private_key.pem acima.
const privateEcdsaKey = `<SUA-CHAVE-ECDSA-PRIVADA>`;;

const signed = jwt.sign(payload, privateEcdsaKey, {
    algorithm: 'ES256',
    expiresIn: '5s'
});
// Você pode obter isso em public_key.pem acima.
const publicEcdsaKey = `<SUA-CHAVE-ECDSA-PÚBLICA>`;

const decoded = jwt.verify(signed, publicEcdsaKey, {
    // Nunca se esqueça de tornar explícito para evitar
    // ataques de remoção de assinatura.
```

```
    algorithms: ['ES256'],
});
```

Consulte o Capítulo 2 para aplicações práticas desses algoritmos no contexto de JWTs.

Capítulo 5

JSON Web Encryption (JWE)

Enquanto o JSON Web Signature (JWS) fornece um meio para *validar* dados, o JSON Web Encryption (JWE) fornece uma maneira de manter os dados *opacos* a terceiros. Opaco neste caso significa *ilegível*. Os tokens criptografados não podem ser inspecionados por terceiros. Isso permite casos de uso adicionais interessantes.

Embora pareça que a criptografia fornece as mesmas garantias que a validação, com o recurso adicional de tornar os dados ilegíveis, esse nem sempre é o caso. Para entender o porquê, primeiro é importante observar que, assim como no JWS, o JWE fornece essencialmente dois esquemas: um esquema de segredo compartilhado e um esquema de chave pública/privada.

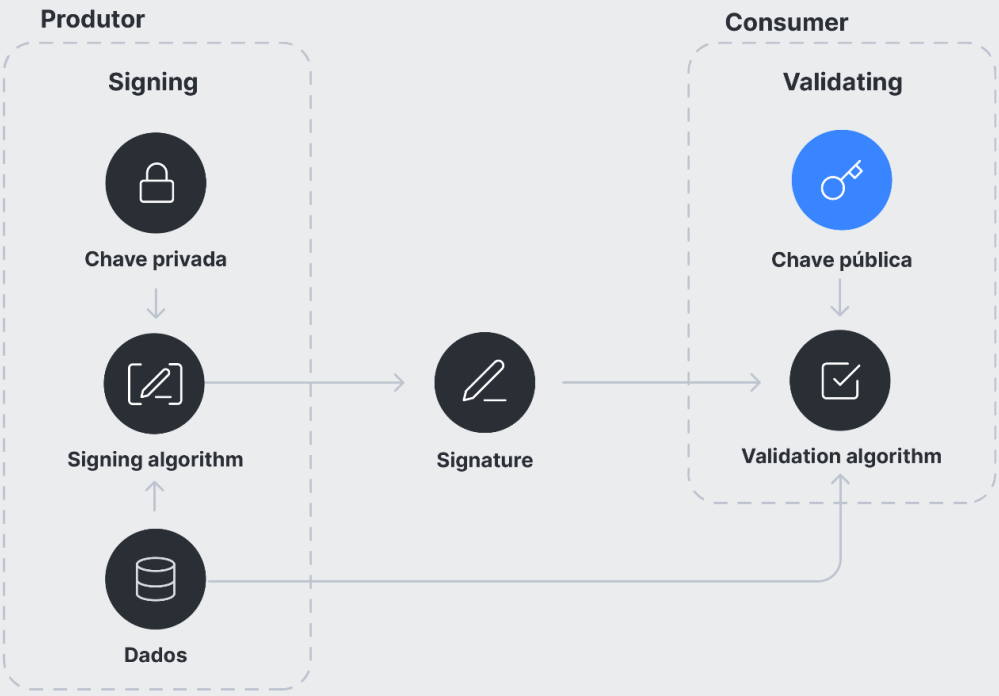
O esquema de segredo compartilhado funciona fazendo com que todas as partes saibam um segredo compartilhado. Cada parte que detém o segredo compartilhado pode **criptografar** e **descriptografar** informações. Isso é análogo ao caso de um segredo compartilhado no JWS: as partes que detêm o segredo podem verificar e gerar tokens assinados.

O esquema de chave pública/privada, no entanto, funciona de maneira diferente. Enquanto no JWS a parte que detém a chave privada pode assinar e verificar tokens, e as partes que detêm a chave pública só podem verificar esses tokens, no JWE a parte que detém a chave privada é a única parte que pode **descriptografar** o token. Em outras palavras, os detentores de chaves públicas podem criptografar dados, mas apenas a parte que detém a chave privada pode **descriptografar** (e criptografar) esses dados. Na prática, isso significa que, no JWE, as partes detentoras da chave *pública* podem introduzir novos dados em uma troca. Em contraste, no JWS, as partes que detêm a chave pública só podem *verificar* dados, mas não introduzir novos dados. Em termos simples, o JWE não fornece as mesmas garantias que o JWS e, portanto, não substitui o papel do JWS em uma troca de tokens. JWS e JWE são complementares quando esquemas de chave pública/privada estão sendo usados.

Uma maneira mais simples de entender isso é pensar em termos de produtores e consumidores. O produtor assina ou criptografa os dados, para que os consumidores possam validá-los ou descriptografá-los. No caso de assinaturas JWT, a chave privada é usada para assinar JWTs, enquanto a chave pública pode ser usada para validá-la. O produtor detém a chave privada e os consumidores a chave pública. Os dados *só* podem fluir de detentores de

chaves privadas para detentores de chaves públicas. Em contraste, para criptografia JWT, a chave pública é usada para criptografar os dados e a chave privada para descriptografá-los. Neste caso, os dados só podem fluir de detentores de chave pública para detentores de chave privada; os detentores de chave pública são os produtores, e os detentores de chave privada são os consumidores:

	JWS	JWE
Produtor	Chave privada	Chave pública
Consumidor	Chave pública	Chave privada



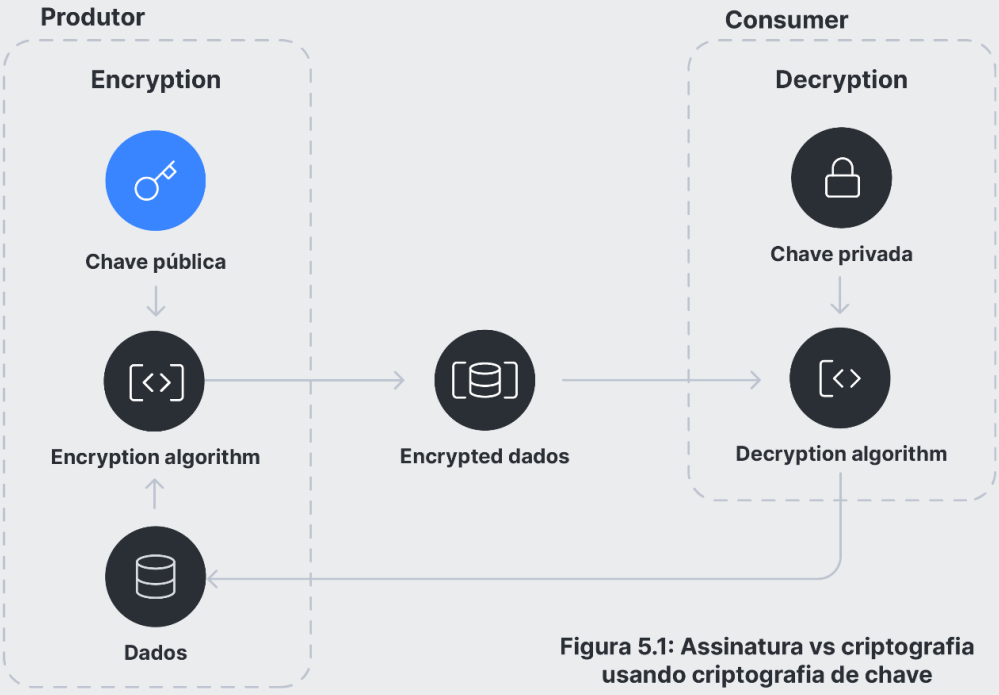


Figura 5.1: Assinatura vs criptografia usando criptografia de chave

Nesse ponto, algumas pessoas podem perguntar:

> No caso do JWE, não poderíamos distribuir a chave privada para todas as partes que querem enviar dados para um consumidor? Assim, se um consumidor pode descriptografar os dados, pode ter certeza também de que são válidos (porque não se pode alterar dados que não podem ser descriptografados).

Tecnicamente, seria possível, mas não faria sentido. Compartilhar a chave privada é *equivalente* a compartilhar o segredo. Portanto, compartilhar a chave privada, em essência, transforma o esquema em um esquema secreto compartilhado, sem os benefícios reais das chaves públicas (lembre-se de que as chaves públicas podem ser derivadas de chaves privadas).

Por esse motivo, os JWTs criptografados às vezes são *aninhados*: um JWT criptografado serve como contêiner para um JWT assinado. Dessa forma, você obtém os benefícios de ambos.

> Note-se que tudo isto se aplica em situações em que os consumidores são entidades diferentes dos produtores. Se o produtor for a mesma entidade que consome os dados, então um JWT criptografado com segredo compartilhado fornece as mesmas garantias que um JWT criptografado e assinado.
>
> Os JWTs criptografados pelo JWE, independentemente de terem um JWT assinado aninhado neles ou não, carregam uma tag de autenticação. Esta tag permite que os JWTs JWE sejam validados. No entanto, devido aos problemas mencionados acima, essa assinatura não se aplica aos mesmos casos de uso que as assinaturas JWS. O propósito desta tag é evitar os ataques Padding Oracle ou manipulação de texto cifrado.

5.1 Estrutura de um JWT criptografado

Ao contrário dos JWTs assinados e não protegidos, os JWTs criptografados têm uma representação compacta diferente (novas linhas inseridas para legibilidade):

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-kFm1NJn8LE9XShH59_
i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ deLKxGHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-
YxzZIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8Otv
zlV7elprCbuPhcCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-
```

```
cFPgwCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGljb3RoZQ. KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.
9hH0vgRfYgPnAHOd8stkvw
```

Embora possa ser difícil de ver no exemplo acima, a Serialização Compacta JWE tem cinco elementos. Como no caso de JWS, esses elementos são separados por pontos, e os dados contidos neles são codificados por Base64.

Os cinco elementos da representação compacta são, em ordem:

1. **O header protegido:** um header análogo ao header do JWS.
2. **A chave criptografada:** uma chave simétrica usada para criptografar o texto cifrado e outros dados criptografados. Esta chave é derivada da chave de criptografia real especificada pelo usuário e, portanto, é criptografada por ela.
3. **O vetor de inicialização:** alguns algoritmos de criptografia requerem dados adicionais (geralmente aleatórios).
4. **Os dados criptografados (texto cifrado):** os dados reais que estão sendo criptografados.
5. **A tag de autenticação:** dados adicionais produzidos pelos algoritmos que podem ser usados para *validar* o conteúdo do texto cifrado contra adulteração.

Como no caso do JWS e assinaturas únicas na serialização compacta, o JWE suporta uma única chave de criptografia em sua forma compacta.

> Usar uma chave simétrica para executar o processo de criptografia real é uma prática comum ao usar criptografia assimétrica (criptografia de chave pública/privada). Algoritmos de criptografia assimétrica são geralmente de alta complexidade computacional e, portanto, criptografar longas sequências de dados (o texto cifrado) não é o ideal. Uma maneira de explorar os benefícios da criptografia simétrica (mais rápida) e assimétrica é gerar uma chave aleatória para um algoritmo de criptografia simétrica e criptografar essa chave com o algoritmo assimétrico. Este é o segundo elemento mostrado acima, a chave criptografada.
>
> Alguns algoritmos de criptografia podem processar quaisquer dados passados para eles. Se o texto cifrado for modificado (mesmo sem ser descriptografado), os algoritmos podem processá-lo mesmo assim. A tag de autenticação pode ser usada

para evitar isso, atuando essencialmente como uma assinatura. Isso, no entanto, não elimina a necessidade dos JWTs aninhados explicados acima.

5.1.1 Algoritmos de criptografia de chave

Ter uma chave de criptografia criptografada significa que há dois algoritmos de criptografia em jogo no mesmo JWT. A seguir estão os algoritmos de criptografia disponíveis para criptografia de chaves:

- **Variantes RSA:** RSAES PKCS #1 v1.5 (RSAES-PKCS1-v1_5), RSAES OAEP e OAEP + MGF1 + SHA-256.
- **Variantes AES:** AES Key Wrap de 128 a 256 bits, AES Galois Counter Mode (GCM) de 128 a 256 bits.
- **Variantes de curva elíptica:** O acordo de chave estática efêmera ECDH (Curva Elíptica Diffie-Hellman) usando Concat KDF e variantes pré-encapsulando a chave com qualquer uma das variantes AES não-GCM acima.
- **Variantes PKCS #5:** PBES2 (criptografia baseada em senha) + HMAC (SHA-256 a 512) + variantes AES não GCM de 128 a 256 bits.
- **Direto:** sem criptografia para a chave de criptografia (uso direto de CEK).

Nenhum desses algoritmos é realmente exigido pela especificação do JWA. A seguir estão os algoritmos recomendados (a serem implementados) pela especificação:

- **RSAES-PKCS1-v1_5** (marcado para remoção da recomendação no futuro)
- **RSAES-OAEP com padrões** (marcado para se tornar obrigatório no futuro)
- **AES-128 Key Wrap**
- **AES-256 Key Wrap**
- **ECDH-ES (Curva Elíptica Diffie-Hellman Estática Efêmera (ECDH-ES)** usando Concat KDF (marcado para se tornar obrigatório no futuro)
- **ECDH-ES + AES-128 Key Wrap**
- **ECDH-ES + AES-256 Key Wrap**

Alguns desses algoritmos exigem parâmetros de header adicionais.

5.1.1.1 Modos de gerenciamento de chaves

A especificação JWE define diferentes modos de gerenciamento de chaves. Essas são, em essência, maneiras pelas quais a chave usada para criptografar o payload é determinada. Em particular, a especificação da JWE descreve esses modos de gerenciamento de chaves:

- **Encapsulamento de chave:** a chave de criptografia de conteúdo (CEK, Content Encryption Key) é criptografada para o destinatário pretendido usando um algoritmo de criptografia *simétrica*.

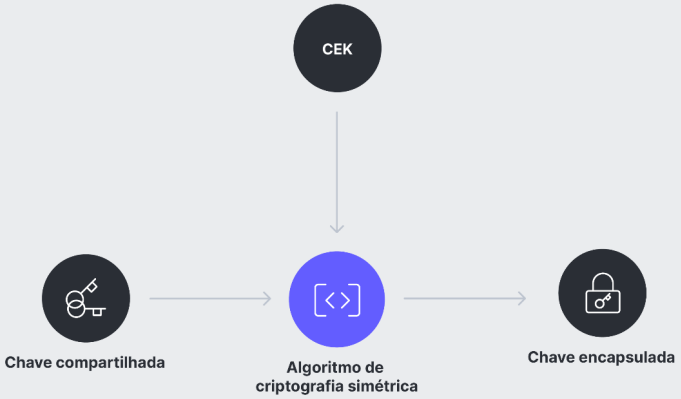


Figura 5.2: Encapsulamento de chave

- **Criptografia de chave:** a CEK é criptografada para o destinatário pretendido usando um algoritmo de criptografia assimétrica

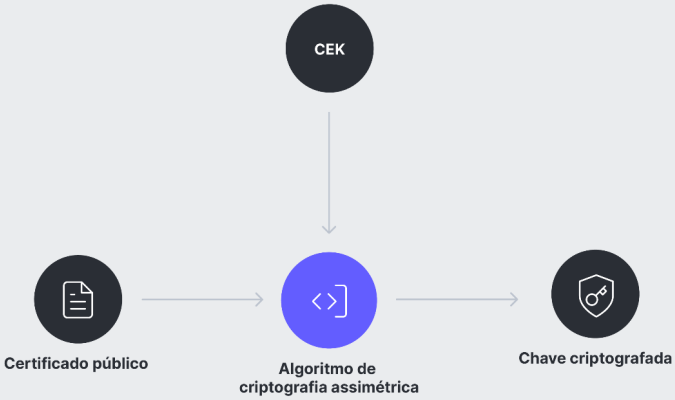


Figura 5.3: Criptografia de chave

- **Acordo de chave direto:** um algoritmo de acordo de chave é usado para escolher a CEK.

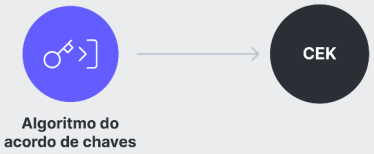


Figura 5.4: Acordo de chave direto

- **Acordo de chave com encapsulamento de chave:** um algoritmo de acordo de chave é usado para escolher um CEK simétrico usando um algoritmo de criptografia simétrica.

Figura 5.5: Acordo de chave direto

- **Criptografia direta:** uma chave compartilhada simétrica definida pelo usuário é usada como CEK (sem derivação ou geração de chave).

Figura 5.6: Acordo de chave direto

Embora isso constitua uma questão de terminologia, é importante entender as diferenças entre cada modo de gerenciamento e dar a cada um deles um nome conveniente.

5.1.1.2 Chave de criptografia de conteúdo (CEK) e chave de criptografia JWE

Também é importante entender a diferença entre CEK e chave de criptografia JWE. CEK é a chave real usada para criptografar o payload: um algoritmo de criptografia usa CEK e o texto simples para produzir o texto cifrado. Em contraste, a chave de criptografia JWE é a forma criptografada da CEK ou uma sequência de octeto vazia (conforme exigido pelo algoritmo

escolhido). Uma chave de criptografia JWE vazia significa que o algoritmo faz uso de uma chave fornecida externamente para descriptografar diretamente os dados (criptografia direta) ou calcular a CEK real (acordo de chave direto).

5.1.2 Algoritmos de criptografia de conteúdo

A seguir estão os algoritmos de criptografia de conteúdo, ou seja, aqueles usados para criptografar o payload:

- **AES CBC + HMAC SHA:** AES 128 a 256 bits com encadeamento de blocos de criptografia e HMAC + SHA-256 a 512 para validação.
- **AES GCM:** AES 128 a 256 usando Galois Counter Mode.

Destes, apenas dois são necessários: AES-128 CBC + HMAC SHA-256 e AES-256 CBC + HMAC SHA-512. As variantes AES-128 e AES-256 usando GCM são recomendadas.

Esses algoritmos são explicados em detalhes no Capítulo 7.

5.1.3 O header

Assim como o header para JWS e JWTs não protegidos, o header carrega todas as informações necessárias para que o JWT seja processado corretamente pelas bibliotecas. A especificação JWE adapta os significados dos claims registrados definidos no JWS ao seu próprio uso e adiciona alguns claims por conta própria. Estes são os claims novos e modificados:

- **alg:** idêntico ao JWS, exceto que define o algoritmo a ser usado para criptografar e descriptografar a chave de criptografia de conteúdo (CEK). Em outras palavras, esse algoritmo é usado para criptografar a chave real que é usada posteriormente para criptografar o conteúdo.
- **enc:** o nome do algoritmo usado para criptografar o conteúdo usando a CEK.
- **zip:** um algoritmo de compressão a ser aplicado aos dados criptografados antes da criptografia. Este parâmetro é opcional. Quando está ausente, nenhuma compressão é realizada. Um valor usual para isso é `DEF`, o algoritmo de deflação comum.
- **jku:** idêntico ao JWS, exceto que, neste caso, o claim aponta para a chave pública usada para criptografar a CEK.

- **jkw:** idêntico ao JWS, exceto que, neste caso, o claim aponta para a chave pública usada para criptografar a CEK.
- **kid:** idêntico ao JWS, exceto que, neste caso, o claim aponta para a chave pública usada para criptografar a CEK.
- **x5u:** idêntico ao JWS, exceto que, neste caso, o claim aponta para a chave pública usada para criptografar a CEK.
- **x5c:** idêntico ao JWS, exceto que, neste caso, o claim aponta para a chave pública usada para criptografar a CEK.
- **x5t:** idêntico ao JWS, exceto que, neste caso, o claim aponta para a chave pública usada para criptografar a CEK.
- **x5t#S256:** idêntico ao JWS, exceto que, neste caso, o claim aponta para a chave pública usada para criptografar a CEK.
- **typ:** idêntico ao JWS.
- **cty:** idêntico ao JWS, exceto que este é o tipo de conteúdo criptografado.
- **crit:** idêntico ao JWS, exceto que se refere aos parâmetros deste header.

Parâmetros adicionais podem ser necessários, dependendo dos algoritmos de criptografia em uso. Você os encontrará explicados na seção que discute cada algoritmo.

5.1.4 Visão geral do algoritmo para serialização compacta

No início deste capítulo, a Serialização Compacta de JWE foi mencionada brevemente. Ela é basicamente composta por cinco elementos codificados em formato de texto imprimível e separado por pontos (.). O algoritmo básico para construir um JWT JWE de serialização compacta é:

1. Se exigido pelo algoritmo escolhido (claim alg), gere um número aleatório do tamanho necessário. É essencial cumprir certos requisitos criptográficos para aleatoriedade ao gerar esse valor. Consulte a RFC 4086 ou use um gerador de números aleatórios validados criptograficamente.
2. Determine a chave de criptografia de conteúdo de acordo com o modo de gerenciamento de chave:
    - Para acordo de chave direto: use o algoritmo de acordo de chave e o número aleatório para calcular a chave de criptografia de conteúdo (CEK).

- Para acordo de chave com encapsulamento de chave: use o algoritmo de acordo de chave com o número aleatório para calcular a chave que será usada para encapsular a CEK.
- Para criptografia direta: a CEK é a chave simétrica.

3. Determine a chave de criptografia de JWE de acordo com o modo de gerenciamento de chave:
    - Para acordo de chave direto e criptografia direta: a chave criptografada de JWE está vazia.
    - Para encapsulamento de chave, criptografia de chave e acordo de chave com encapsulamento de chave: criptografe a CEK para o destinatário. O resultado é a chave de criptografia de JWE.
4. Calcule um vetor de inicialização (IV) do tamanho exigido pelo algoritmo escolhido. Se não for necessário, ignore esta etapa.
5. Compacte o texto simples do conteúdo, se necessário (claim de header zip).
6. Criptografe os dados usando a CEK, o IV e os Dados Autenticados Adicionais (AAD). O resultado é o conteúdo criptografado (JWE Ciphertext) e a tag de autenticação. Os AAD são usados apenas para serializações não compactas.
7. Construa a representação compacta como:

```
base64(header) + '.' +
base64(encryptedKey) + '.' +          // Etapas 2 e 3
base64(initializationVector) + '.' + // Etapa 4
base64(ciphertext) + '.' +            // Etapa 6
base64(authenticationTag)             // Etapa 6
```

5.1.5 Serialização JSON do JWE

Além da serialização compacta, o JWE também define uma representação JSON não compacta. Essa representação troca o tamanho por flexibilidade, permitindo, entre outras coisas, a criptografia do conteúdo para vários destinatários usando várias chaves públicas ao mesmo tempo. Isso é análogo às múltiplas assinaturas permitidas pela serialização JWS JSON.

A Serialização JWE JSON é a codificação de texto imprimível de um objeto JSON com os seguintes membros:

- **protected:** O objeto JSON codificado em Base64 dos claims do header a serem protegidos (validado, não criptografado) por este JWT JWE. Opcional. Pelo menos este elemento ou o header não protegido deve estar presente.
- **unprotected:** claims de header que não são protegidos (validados) como um objeto JSON (não codificados em Base64). Opcional. Pelo menos este elemento ou o header protegido deve estar presente.
- **iv:** String Base64 do vetor de inicialização. Opcional (apenas presente quando exigido pelo algoritmo).
- **aad:** dados autenticados adicionais. String Base64 dos dados adicionais que são protegidos (validados) pelo algoritmo de criptografia. Se nenhum AAD for fornecido na etapa de criptografia, este membro deve estar ausente.
- **ciphertext:** string codificada em Base64 dos dados criptografados.
- **tag:** string Base64 da tag de autenticação gerada pelo algoritmo de criptografia.
- **recipient:** um array JSON de objetos JSON, cada um contendo as informações necessárias para a descriptografia por cada destinatário.

A seguir estão os membros dos objetos no array de destinatários:

- **header:** um objeto JSON de claims de headers não protegidos. Opcional.
- **encrypted_key:** chave de criptografia de JWE codificada em Base64. Somente presente quando uma chave criptografada JWE é usada.

O header real usado para descriptografar um JWT JWE para um destinatário é construído a partir da união de cada header presente. Nenhum claim repetido é permitido.

O formato das chaves criptografadas é descrito no Capítulo 6 (JSON Web Keys). O exemplo a seguir é retirado do RFC 7516 (JWE):

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

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

Este JWT de JWE serializado com JSON carrega um único payload para dois destinatários. O algoritmo de criptografia é AES-128 CBC + SHA-256, que você pode obter no header protegido:

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

Ao realizar a união de todos os claims para cada destinatário, o header final para cada destinatário é construído:

Primeiro destinatário:

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

Segundo destinatário:

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

5.1.5.1 Serialização JSON do JWE achatada

Como ocorre com o JWS, o JWE define uma serialização JSON *plana*. Este formato de serialização só pode ser usado para um único destinatário. Neste formato, o array de `destinatários` é substituído por um `header` e um par ou elementos `encrypted_key` (ou seja, as chaves de um único objeto do array de destinatários tomam seu lugar).

Esta é a representação achatada do exemplo da seção anterior resultante da inclusão apenas do primeiro destinatário:

```
{
    "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 Criptografia e descriptografia de tokens

Os exemplos a seguir mostram como executar criptografia usando a popular biblioteca node-jose. Esta biblioteca é um pouco mais complexa do que a `jsonwebtoken` (usada pelos exemplos do JWS), pois cobre muito mais terreno.

5.2.1 Introdução: gerenciamento de chaves com node-jose

Para os propósitos dos exemplos a seguir, precisaremos usar chaves de criptografia em várias formas. Isso é gerenciado por `node-jose` através de um `keystore`. Um `keystore` é um objeto que gerencia chaves. Geraremos e adicionaremos algumas chaves ao nosso keystore para que possamos usá-las posteriormente nos exemplos. Você pode se lembrar dos exemplos do JWS que essa abstração não era necessária para a biblioteca `jsonwebtoken`. A abstração do `keystore` é um detalhe de implementação de `node-jose`. Você pode encontrar outras abstrações semelhantes em outras linguagens e bibliotecas.

Para criar uma keystore vazia e adicionar algumas chaves de diferentes tipos:

```
// Criar uma keystore vazia
```

```
const keystore = jose.JWK.createKeyStore();
// Gere algumas chaves. Você também pode importar chaves geradas a partir de
// fontes externas.
const promises = [
    keystore.generate('oct', 128, { kid: 'example-1' }),
    keystore.generate('RSA', 2048, { kid: 'example-2' }),
    keystore.generate('EC', 'P-256', { kid: 'example-3' }),
];
```

Com node-jose, a geração de chaves é uma questão bastante simples. Todos os tipos de chaves utilizáveis com JWE e JWS são suportados. Neste exemplo, criamos três chaves diferentes: uma chave simples AES de 128 bits, uma chave RSA de 2048 bits e uma chave de curva elíptica usando a curva P-256. Essas chaves podem ser usadas tanto para criptografia quanto para assinaturas. No caso de chaves que suportam pares de chaves pública/privada, a chave gerada é a chave privada. Para obter as chaves públicas, basta chamar:

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

A chave pública será armazenada em formato JWK.

Também é possível importar chaves pré-existentes:

```
// em que a entrada é  :
// *  instância jose.JWK.Key
// *  Representação de objeto JSON de um JWK
jose.JWK.asKey(input).
        then(function(result) {
          // {result} é um jose.JWK.Key
          // {result.keystore} é um jose.JWK.KeyStore único
        });
```

```
// em que a entrada é  :
```

```
// *  Serialização de string de um JWK JSON/(codificado em base64)
//    PEM/(binary-encoded) DER
// *  Buffer de um JSON JWK/(codificação base64) PEM/(codificação binária) DER
// formato é:
// * "json" para um JSON stringified JWK
// * "pkcs8" para um DER codificado (não criptografado!) Chave privada PKCS8
// * "spki" para uma chave pública SPKI codificada por DER
// * "pkix" para um certificado PKIX X.509  codificado por DER
// * "x509" para um certificado PKIX X.509 codificado por DER
// * "pem" para uma codificação PEM de PKCS8 / SPKI / PKIX
jose.JWK.asKey(input, form).
         then(function(result) {
           // {result} é um jose.JWK.Key
           // {result.keystore} é um jose.JWK.KeyStore único
         });
```

5.2.2 AES-128 Key Wrap (Chave) + AES-128 GCM (Conteúdo)

AES-128 Key Wrap e AES-128 GCM são algoritmos de chave simétrica. Isso significa que a mesma chave é necessária para criptografia e descriptografia. A chave para o "example-1" que geramos antes é uma dessas chaves. No AES-128 Key Wrap, essa chave é usada para encapsular uma chave gerada aleatoriamente, que é usada para criptografar o conteúdo usando o algoritmo AES-128 GCM. Também seria possível usar essa chave diretamente (modo de criptografia direta).

```
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));
}
```

A biblioteca `node-jose` funciona principalmente com promessas6. O objeto retornado por `a128gcm` é uma promessa. A função `createEncrypt` pode criptografar qualquer conteúdo que seja passado para ela. Em outras palavras, não é necessário que o conteúdo seja um JWT (embora na maioria das vezes seja). É por essa razão que `JSON.stringify` deve ser chamado antes de passar os dados para essa função.

5.2.3 RSAES-OAEP (Chave) + AES-128 CBC + SHA-256 (Conteúdo)

A única coisa que muda entre as invocações da função `createEncrypt` são as opções passadas para ela. Portanto, é igualmente fácil usar um par de chaves pública/privada. Em vez de passar a chave simétrica para `createEncrypt`, basta passar a chave pública ou a chave privada (para criptografia, apenas a chave pública é necessária, embora esta possa ser derivada da chave privada). Para fins de legibilidade, simplesmente usamos a chave privada, mas, na prática, a chave pública provavelmente será usada nesta etapa.

```
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` seleciona o algoritmo de criptografia real. Lembre-se de que existem apenas duas variantes (com tamanhos de chave diferentes): AES CBC + HMAC SHA e AES GCM.

5.2.4 ECDH-ES P-256 (Chave) + AES-128 GCM (Conteúdo)

A API para curvas elípticas é idêntica àquela da RSA:

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

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

5.2.5 JWT aninhado: ECDSA usando P-256 e SHA-256 (assinatura) + RSAES-OAEP (chave criptografada) + AES-128 CBC + SHA-256 (conteúdo criptografado)

JWTs aninhados requerem um pouco de malabarismo para passar o JWT assinado para a função de criptografia. Especificamente, as etapas de assinatura + criptografia devem ser realizadas manualmente. Lembre-se de que essas etapas são executadas nessa ordem: primeiro a assinatura e, em seguida, a criptografia. Embora tecnicamente nada impeça que a ordem seja invertida, assinar o JWT primeiro impede que o token resultante seja vulnerável a ataques de remoção de assinatura.

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

```
}
```

Como pode ser visto no exemplo acima, `node-jose` também pode ser usado para assinatura. Não há nada que impeça o uso de outras bibliotecas (como a `jsonwebtoken`) para esse fim. No entanto, dada a necessidade de `node-jose`, não há sentido em adicionar dependências e usar APIs inconsistentes.

> Executar a etapa de assinatura primeiro só é possível porque a JWE exige criptografia autenticada. Em outras palavras, o algoritmo de criptografia também deve executar a etapa de assinatura. As razões pelas quais JWS e JWE podem ser combinados de maneira útil, apesar da autenticação do JWE, foram descritas no início do Capítulo 5. Para outros esquemas (ou seja, para criptografia geral + assinatura), a norma é primeiro criptografar e depois assinar. Isso evita a manipulação do texto cifrado que pode resultar em ataques de criptografia. Também é a razão pela qual o JWE exige a presença de uma tag de autenticação.

5.2.6 Descriptografia

A descriptografia é tão simples quanto a criptografia. Tal como acontece com a criptografia, o payload deve ser convertido entre diferentes formatos de dados explicitamente.

```
// Teste de descriptografia
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);
});
```

A descriptografia dos algoritmos RSA e Curva Elíptica é análoga, usando a chave privada em vez da chave simétrica. Se você tiver uma keystore com os claims kid certos, é possível simplesmente passar o keystore para a função `createDecrypt` e fazer com que ela procure a chave certa. Portanto, qualquer um dos exemplos acima pode ser descriptografado usando exatamente o mesmo código:

```
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);
       });
```

Capítulo 6

JSON Web Keys (JWK)

Para completar a imagem do JWT, JWS e JWE, chegamos agora à especificação do JSON Web Key (JWK). Esta especificação lida com as diferentes representações para as chaves usadas para assinaturas e criptografia. Embora existam representações estabelecidas para todas as chaves, a especificação do JWK visa fornecer uma representação unificada para todas as chaves suportadas na especificação do JSON Web Algorithms (JWA). Um formato de representação unificado para chaves permite o compartilhamento fácil e mantém as chaves independentes das complexidades de outros formatos de troca de chaves.

JWS e JWE oferecem suporte a um tipo diferente de formato de chave: certificados X.509. Estes são bastante comuns e podem transportar mais informações do que um JWK. Os certificados X.509 podem ser incorporados aos JWKs, e os JWKs podem ser construídos a partir deles.

As chaves são especificadas em diferentes claims de cabeçalho. JWKs literais são colocados sob o claim do JWK. O claim jku, por outro lado, pode apontar para um conjunto de chaves armazenadas sob um URL. Ambos os claims estão em formato JWK.

Uma amostra do JWK:

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

6.1 Estrutura de um JSON Web Key

Os JSON Web Keys são simplesmente objetos JSON com uma série de valores que descrevem os parâmetros exigidos pela chave. Esses parâmetros variam de acordo com o tipo da chave. Parâmetros comuns são:

- **kty:** tipo de chave. Este claim diferencia tipos de chaves. Os tipos suportados são `EC`, para chaves de curva elíptica; `RSA` para chaves RSA e OCT para chaves simétricas. Este claim é necessário.
- **use:** este claim especifica o uso pretendido da chave. Existem dois usos possíveis: `sig` (para assinatura) e `enc` (para criptografia). Este claim é opcional. A mesma chave pode ser usada para criptografia e assinaturas, caso em que este membro não deve estar presente.
- **key_ops:** um array de valores de string que especifica usos detalhados para a chave. Os valores possíveis são: `sign`, `verify`, `encrypt`, `decrypt`, `wrapKey`, `unwrapKey`, `deriveKey`, `deriveBits`. Certas operações não devem ser usadas em conjunto. Por exemplo, `sign` e `verify` são apropriados para a mesma chave, enquanto `sign` e `encrypt` não são. Este claim é opcional e não deve ser usado ao mesmo tempo que o claim `use`. Nos casos em que ambos estão presentes, seu conteúdo deve ser consistente.
- **alg:** algoritmo. O algoritmo destinado a ser usado com esta chave. Pode ser qualquer um dos algoritmos admitidos para operações JWE ou JWS. Este claim é opcional.
- **kid:** id de chave. Um identificador exclusivo para esta chave. Ele pode ser usado para combinar uma chave com um claim `kid` no cabeçalho JWE ou JWS, ou para escolher uma chave de um conjunto de chaves de acordo com a lógica do aplicativo. Este claim é opcional. Duas chaves no mesmo conjunto de chaves podem transportar o mesmo `kid` somente se tiverem claims `kty` diferentes e forem destinadas ao mesmo uso.
- **x5u:** um URL que aponta para um certificado de chave pública X.509 ou cadeia de certificados em formato codificado PEM. Se outros claims opcionais estiverem presentes, devem ser consistentes com o conteúdo do certificado. Este claim é opcional.
- **x5c:** um certificado X.509 DER codificado com Base64-URL ou cadeia de certificados. Uma cadeia de certificados é representada como um array de tais certificados. O

primeiro certificado precisa ser o certificado mencionado neste JWK. Todos os outros claims presentes neste JWK devem ser consistentes com os valores do primeiro certificado. Este claim é opcional.

- **x5t:** uma impressão digital/miniatura de SHA-1 codificado por Base64-URL da codificação DER de um certificado X.509. O certificado para o qual esta impressão digital aponta precisa ser consistente com os claims neste JWK. Este claim é opcional.
- **x5t#S256:** idêntico ao claim `x5t`, mas com a impressão digital SHA-256 do certificado.

Outros parâmetros, como `x`, `y` ou `d` (do exemplo na abertura deste capítulo) são específicos para o algoritmo chave. As chaves RSA, por outro lado, carregam parâmetros como `n`, `e`, `dp`, etc. O significado desses parâmetros se tornará claro no Capítulo 7, onde cada algoritmo de chave é explicado em detalhes.

6.1.1 JSON Web Key Set

A especificação do JWK admite grupos de chaves. São conhecidos como "Conjuntos JWK". Esses conjuntos carregam mais de uma chave. O significado das chaves como um grupo e o significado da ordem dessas chaves é definido pelo usuário.

Um conjunto de chaves Web JSON é simplesmente um objeto JSON com um membro de `chaves`. Este membro é um array JSON de JWKs.

Amostra do conjunto JWK:

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

```
        {
            "kty": "RCA",
            "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"
        }
    ]
}
```

Neste exemplo, duas chaves públicas estão disponíveis. O primeiro é do tipo curva elíptica e está limitado a operações de *criptografia* pelo claim `use`. O segundo é do tipo RSA e está associado a um algoritmo específico (`RS256`) pelo claim `alg`. Isso significa que essa segunda chave deve ser usada para *assinaturas*.

Capítulo 7

Algoritmos JSON Web

Você provavelmente observou que há muitas referências a este capítulo ao longo deste manual. A razão é que uma grande parte da magia por trás dos JWTs está nos algoritmos empregados com ele. A estrutura é importante, mas os muitos usos interessantes descritos até agora só são possíveis devido aos algoritmos em jogo. Este capítulo abordará os algoritmos mais importantes em uso com JWTs hoje. Compreendê-los em profundidade não é necessário para usar os JWTs de forma eficaz, e por isso este capítulo é voltado para mentes curiosas que desejam entender a última peça do quebra-cabeça.

7.1 Algoritmos gerais

Os algoritmos a seguir têm muitas aplicações diferentes dentro das especificações de JWT, JWS e JWE. Alguns algoritmos, como o Base64-URL, são usados para formas de serialização compactas e não compactas. Outros, como o SHA-256, são usados para assinaturas, criptografia e impressões digitais de chaves.

7.1.1 Base64

Base64 é um algoritmo de codificação binário para texto. Seu objetivo principal é transformar uma sequência de octetos em uma sequência de caracteres imprimíveis, com o custo de um tamanho maior. Em termos matemáticos, o Base64 transforma uma sequência de números radix-256 em uma sequência de números radix-64. A palavra *base* pode ser usada no lugar de *radix*, daí o nome do algoritmo.

> Nota: Base64 não é realmente usado pela especificação de JWT. É a variante *Base64-URL* descrita mais adiante neste capítulo que é usada pelo JWT.

Para entender como Base64 pode transformar uma série de números arbitrários em texto, primeiro é necessário estar familiarizado com sistemas de codificação de texto. Sistemas de codificação de texto mapeiam números para caracteres. Embora esse mapeamento seja arbitrário e, no caso do Base64, possa ser definido pela implementação, o padrão de fato para a codificação Base64 é RFC 4648.

0 A	17 R	34 i	51 z
1B	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	22W	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	26a	43 r	60 8
10 K	27 b	44 s	61 9
11 L	28 c	45 t	62 +
12 M 13 N 14 O	29 d 30 e 31 f	46 u 48 w	63 / (pad) =
15 P	32 g	49 x	

16 Q	33 h	50 y	PAGINA 86 WORD DOC

Na codificação Base64, cada caractere representa 6 bits dos dados originais. A codificação é realizada em grupos de quatro caracteres codificados. Assim, 24 bits de dados originais são tomados em conjunto e codificados como quatro caracteres Base64. Como os dados originais devem ser uma sequência de valores de 8 bits, os 24 bits são formados pela concatenação de três valores de 8 bits da esquerda para a direita.

Codificação Base64:

```
3 x valores de 8 bits -> dados concatenados de 24 bits -> 4 x caracteres de 6 bits
```

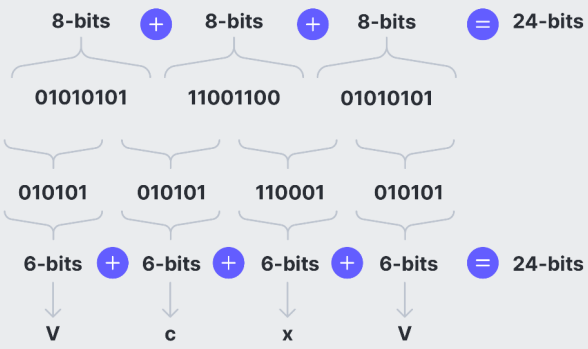


Figura 7.1: Codificação Base64

Se o número de octetos nos dados de entrada não for divisível por três, a última parte dos dados a codificar terá menos de 24 bits de dados. Quando este é o caso, zeros são adicionados aos dados de entrada concatenados para formar um número integral de grupos de 6 bits. Existem três possibilidades:

1. Os 24 bits completos estão disponíveis como entrada; nenhum processamento especial é executado.
2. 16 bits de entrada estão disponíveis, três valores de 6 bits são formados e o último valor de 6 bits recebe zeros extras adicionados à direita. A string codificada resultante é

preenchida com um caractere extra = para tornar explícito que 8 bits de entrada estavam faltando.

3. 8 bits de entrada estão disponíveis, dois valores de 6 bits são formados e o último valor de 6 bits recebe zeros extras adicionados à direita. A string codificada resultante é preenchida com dois caracteres extras = para tornar explícito que 16 bits de entrada estavam faltando.

O caractere de preenchimento (=) é considerado opcional por algumas implementações. Executar as etapas na ordem oposta renderá os dados originais, independentemente da presença dos caracteres de preenchimento.

7.1.1.1 Base64-URL

Certos caracteres da tabela de conversão Base64 padrão não são seguros para URLs. Base64 é uma codificação conveniente para passar dados arbitrários em campos de texto. Como apenas dois caracteres do Base64 são problemáticos como parte do URL, uma variante segura para URL é fácil de implementar. O caractere + e o caractere / são substituídos pelo caractere - e pelo caractere _.

7.1.1.2 Código de exemplo

O exemplo a seguir implementa um codificador inócuo Base64-URL. O exemplo é escrito pensando na simplicidade e não na velocidade.

```
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 insere um buffer, Uint8Array ou Int8Array, Array
 * @retorna uma string com os valores codificados
 */
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 - 18)) & 0x3F] + "==";
    }
 }
 return result;
}
```

7.1.2 SHA

O Algoritmo de Hash Seguro (SHA) usado nas especificações de JWT é definido em FIPS-180. Não deve ser confundido com a família de algoritmos SHA-1, que foram descontinuados em 2010. Para diferenciar essa família da anterior, ela às vezes é chamada de *SHA-2*.

Os algoritmos do RFC 4634 são SHA-224, SHA-256, SHA-384 e SHA-512. De importância para o JWT são SHA-256 e SHA-512. Vamos nos concentrar na variante SHA-256 e explicar suas diferenças com relação às outras variantes.

Como muitos algoritmos de hash, o SHA trabalha processando a entrada em blocos de tamanho fixo, aplicando uma série de operações matemáticas e depois acumulando o resultado por meio de uma operação com os resultados das iterações anteriores. Depois que todos os blocos de entrada de tamanho fixo são processados, o resumo é considerado como computado.

A família de algoritmos SHA foi projetada para evitar colisões e produzir saídas radicalmente diferentes, mesmo quando a entrada é apenas ligeiramente alterada. É por essa razão que eles são considerados seguros: é computacionalmente inviável encontrar colisões para diferentes entradas ou calcular a entrada original a partir do resumo produzido.

O algoritmo requer uma série de funções predefinidas:

```
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) ^ rotr(x >>> 3);
}

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

Essas funções são definidas na especificação. A função `rotr` executa a rotação bit a bit (para a direita).

Além disso, o algoritmo requer que a mensagem tenha um comprimento predefinido (um múltiplo de 64); portanto, o preenchimento é necessário. O algoritmo de preenchimento funciona da seguinte forma:

1. Um único binário 1 é anexado ao final da mensagem original. Por exemplo:

```
Mensagem original:
    01011111 01010101 10101010 00111100
Extra 1  no  final:
    01011111 01010101 10101010 00111100 1
```

2. Um número N de zeros é adicionado de modo que o comprimento resultante da mensagem seja a solução para esta equação:

```
L = Comprimento da mensagem  em bits
0 = (65 + N + L) mod 512
```

3. Em seguida, o número de bits na mensagem original é adicionado como um número inteiro de 64 bits:

```
Mensagem original:
    01011111 01010101 10101010 00111100
Extra 1  no  final:
    01011111 01010101 10101010 00111100 1
N zeros:
    01011111 01010101 10101010 00111100 10000000 ...0...
Mensagem com preenchimento:
    01011111 01010101 10101010 00111100 10000000 ...0... 00000000 00100000
```

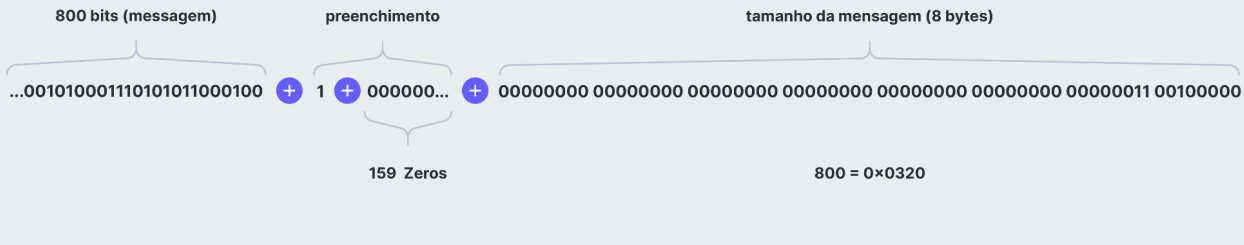


Figura 7.2: Preenchimento SHA

Uma implementação simples em JavaScript pode ser:

```
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: suporte para  mensagens com bitLength  maior do que 2^32
    padded[padded.length - 1] = bitLength;

    return padded;
}
```

A mensagem com preenchimento resultante é então processada em blocos de 512 bits. A implementação abaixo segue passo a passo o algoritmo descrito na especificação. Todas as operações são executadas em números inteiros de 32 bits.

```
export default function sha256(message, returnBytes) {
    // Valores de hash iniciais
    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;
}
//(...)
}
```

A variável `k` contém uma série de constantes, que são definidas na especificação.

O resultado final está na variável `h_[0..7]`. A única etapa que falta é apresentá-lo em formato legível:

```
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;
}
```

Embora funcione, observe que a implementação acima não é ideal (e não suporta mensagens maiores que 2^{32}).

Outras variantes da família SHA-2 (como SHA-512) simplesmente alteram o tamanho do bloco processado em cada iteração e alteram as constantes e seu tamanho. Em particular, o SHA-512 requer que a matemática de 64 bits esteja disponível. Em outras palavras, para transformar a implementação de exemplo acima em SHA-512, uma biblioteca separada para matemática de 64 bits é necessária (já que o JavaScript suporta apenas operações bit a bit de 32 bits e matemática de ponto flutuante de 64 bits).

7.2 Algoritmos de assinatura

7.2.1 HMAC

Códigos de autenticação de mensagem baseados em hash (HMAC) fazem uso de uma função de hash criptográfica (como a família SHA discutida acima) e uma chave para criar um *código*

de autenticação para uma mensagem específica. Em outras palavras, um esquema de autenticação baseado em HMAC recebe uma função hash, uma mensagem e uma chave secreta como entradas e produz um código de autenticação como saída. A força da função de hash criptográfica garante que a mensagem não possa ser modificada sem a chave secreta. Assim, os HMACs servem a ambos os propósitos de *autenticação e integridade de dados*.

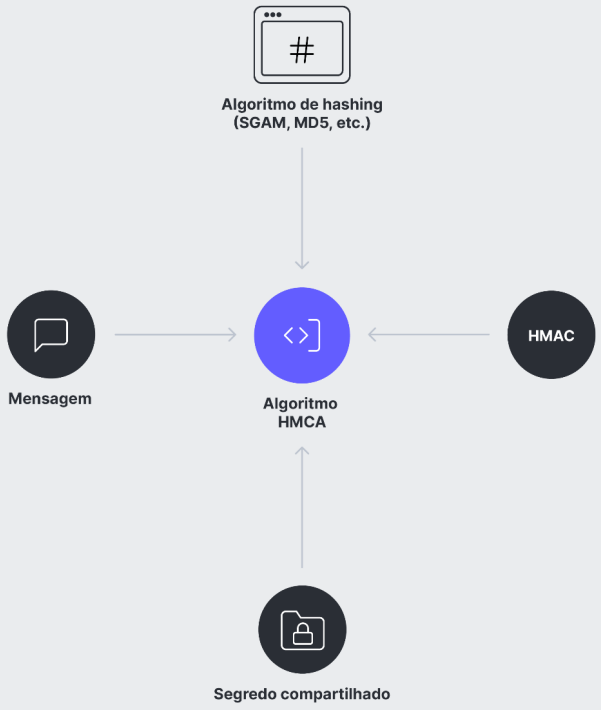


Figura 7.3: HMAC

Funções de hash fracas podem permitir que usuários mal-intencionados comprometam a validade do código de autenticação. Portanto, para que os HMACs sejam úteis, uma função hash forte deve ser escolhida. A família de funções SHA-2 ainda é forte o suficiente para os padrões de hoje, mas isso pode mudar no futuro. MD5, uma função hash criptográfica diferente usada extensivamente no passado, pode ser usada para HMACs. No entanto, pode ser vulnerável a colisões e ataques de prefixo. Embora esses ataques não tornem necessariamente o MD5 inadequado para uso com HMACs, algoritmos mais fortes estão prontamente disponíveis e devem ser considerados.

O algoritmo é simples o suficiente para caber em uma única linha

```
Considere H a função hash criptográfica
          B é o comprimento do bloco da função hash
            (quantos bits são processados por iteração)
          K é a chave secreta
          K' é a chave real usada pelo algoritmo HMAC
          L é o comprimento da saída da função hash
          ipad é o byte 0x36 repetido B vezes
          opad é o byte 0x5C repetido B vezes
          mensagem é a mensagem de entrada
          || é a função de concatenação

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

`K'` é calculado a partir da chave secreta `K` da seguinte forma:

Se `K` for menor que `B`, zeros são anexados até que `K` tenha o comprimento de `B`. O resultado é `K'`. Se `K` for maior que `B`, `H` é aplicado a `K`. O resultado é `K'`. Se `K` for exatamente `B` bytes, ele é usado como está (`K` é `K'`).

Eis aqui um exemplo de implementação em JavaScript:

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

    const blockSizeBytes = blockSizeBits / 8;

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

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

    const ipadSecret = ipad.map((value, index) => {
        return value ^ paddedSecret[index];
    });
    const opadSecret = opad.map((value, index) => {
        return value ^ paddedSecret[index];
    });
    // HMAC(message) = H(K' XOR opad || H(K' XOR ipad || message))
    const result = hashFn(
        append(opadSecret,
               uint32ArrayToUint8Array(hashFn(append(ipadSecret,
                                                     message), true))),
        returnBytes);

    return result;
}
```

Para verificar uma mensagem contra um HMAC, basta calcular o HMAC e comparar o resultado com o HMAC que acompanha a mensagem. Isso requer o conhecimento da chave secreta por todas as partes: quem produz a mensagem e quem só quer verificá-la.

7.2.1.1 HMAC + SHA256 (HS256)

Noções básicas sobre Base64-URL, SHA-256 e HMAC são tudo o que é necessário para implementar o algoritmo de assinatura HS256 a partir da especificação JWS. Com isso em mente, agora podemos combinar todo o código de exemplo desenvolvido até agora e construir um JWT totalmente assinado.

```
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}`;
}
```

Observe que esta função não executa nenhuma validação do cabeçalho ou do payload (além de verificar se são objetos). Você pode chamar essa função da seguinte forma:

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

Cole o JWT gerado no depurador do JWT.io e veja como ele é decodificado e validado.

Esta função é muito semelhante à usada no Capítulo 4 como uma demonstração para o algoritmo de assinatura. Do Capítulo 4:

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

Verificar também é fácil:

```
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]
    };
}
```

A assinatura é dividida do JWT e uma nova assinatura é computada. Se a nova assinatura corresponder à incluída no JWT, a assinatura é válida.

Você pode usar a função acima da seguinte forma:

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

Este código está disponível no arquivo `hs256.js` dos exemplos incluídos com este manual.

7.2.2 RSA

O RSA é um dos criptossistemas mais utilizados atualmente. Foi desenvolvido em 1977 por Ron Rivest, Adi Shamir e Leonard Adleman, cujas iniciais foram usadas para nomear o algoritmo. O aspecto principal do RSA está em sua assimetria: a chave usada para criptografar algo *não* é a chave usada para descriptografá-lo. Este esquema é conhecido como criptografia de chave pública (PKI), onde a chave pública é a chave de criptografia e a chave privada é a chave de descriptografia.

Quando se trata de assinaturas, a chave privada é usada para *assinar* uma informação e a chave pública é usada para *verificar* se a informação foi assinada por uma chave privada específica (sem realmente conhecê-la).

Há variações do algoritmo RSA para assinatura e criptografia. Vamos nos concentrar no algoritmo geral primeiro e, em seguida, vamos analisar as diferentes variações usadas com JWTs.

Muitos algoritmos criptográficos, e em particular o RSA, são baseados na dificuldade relativa de realizar certas operações matemáticas. O RSA escolhe a fatoração de número inteiro como sua principal ferramenta matemática. A fatoração de números inteiros é o problema matemático que tenta encontrar números que multiplicados entre si produzem o número

original como resultado. Em outras palavras, os fatores de um número inteiro são um conjunto de pares de números inteiros que, quando multiplicados, produzem o número inteiro original.

```
inteiro = fator_1 x fator_2
```

Este problema pode parecer fácil no início. E, para números pequenos, é mesmo. Tome, por exemplo, o número `35`:

```
35 = 7 x 5
```

Conhecendo as tabelas de multiplicação de 7 ou 5, é fácil encontrar dois números que produzem `35` quando multiplicados. Um algoritmo simplificado para encontrar fatores para um número inteiro pode ser:

1. Considere `n` o número que queremos fatorar.
2. Considere `x` um número entre `2` (inclusive) e `n/2` (inclusive).
3. Divida `n` por `x` e verifique se o resto é `0`. Se for, você encontrou um par de fatores (`x` e o quociente).
4. Continue executando a etapa 3, aumentando x em `1` em cada iterador até que x atinja seu limite superior `n/2`. Quando isso acontece, você encontrou todos os fatores possíveis de `n`.

Esta é essencialmente a abordagem de força bruta para encontrar fatores. Como pode imaginar, este algoritmo é terrivelmente ineficiente.

> Uma versão melhor deste algoritmo é chamada de `divisão por tentativa` e define condições mais rigorosas para `x`. Em particular, define o limite superior de x como `sqrt(n)`, e, em vez de aumentar `x` em `1` em cada iteração, faz `x` tomar o valor de números primos cada vez maiores. É trivial provar por que essas condições tornam o algoritmo mais eficiente enquanto mantêm sua precisão (embora fora do escopo deste texto).

Algoritmos mais eficientes existem, mas, por mais eficientes que sejam, mesmo com os computadores de hoje a fatoração de certos números é computacionalmente inviável. O problema é agravado quando certos números são escolhidos como n. Por exemplo, se n é o resultado da multiplicação de dois números primos, é muito mais difícil encontrar seus fatores (dos quais esses números primos são os únicos fatores possíveis).

> Se `n` fosse o resultado da multiplicação de dois números não primos, seria muito mais fácil encontrar seus fatores. Por quê? Porque os números não primos têm divisores diferentes de si mesmos (por definição), e esses divisores são, por sua vez, divisores de qualquer número multiplicado com eles. Em outras palavras, qualquer divisor para um fator de um número também é um fator do número. Ou, em outros termos, se n tiver fatores não primos, ele tem mais de dois fatores. Portanto, se n é o resultado da multiplicação de dois números primos, ele tem exatamente dois fatores (o menor número possível de fatores sem ser um número primo em si). Quanto menor o número de fatores, mais difícil será encontrá-los.

Quando dois números primos diferentes e grandes são escolhidos e depois multiplicados, o resultado é outro número grande (chamado de semiprimo). Mas este número grande tem uma propriedade especial adicionada: ele é realmente difícil de fatorar. Mesmo os algoritmos de fatoração mais eficientes, como a peneira de campo de número geral, não podem fatorar, em um período de tempo razoável, números grandes que são o resultado da multiplicação de números primos grandes. Para exemplificar essa escala, em 2009 um número de 768 bits (232 dígitos decimais) foi fatorado após 2 anos de trabalho por um cluster de computadores. Aplicações típicas de RSA fazem uso de números de 2048 bits ou maiores.

> O algoritmo de Shor é um tipo especial de algoritmo de fatoração que pode mudar drasticamente as coisas no futuro. Enquanto a maioria dos algoritmos de fatoração é de natureza clássica (ou seja, eles operam em computadores clássicos), o algoritmo de Shor depende de computadores quânticos. Os computadores quânticos aproveitam a natureza de certos fenômenos quânticos para acelerar várias operações clássicas. Em particular, o algoritmo de Shor poderia acelerar a fatoração, trazendo sua complexidade para o domínio da complexidade do tempo polinomial (ao invés de exponencial). Isso é muito mais eficiente do que qualquer um dos algoritmos clássicos atuais. Especula-se que se tal algoritmo fosse executável em um computador quântico, as chaves RSA atuais se tornariam inúteis. Um computador quântico prático, conforme exigido pelo algoritmo de Shor, ainda não foi desenvolvido, mas esta é uma área de pesquisa ativa no momento.

Embora atualmente a fatoração de números inteiros seja computacionalmente inviável para semiprimos grandes, não há prova matemática de que esse deva ser o caso. Em outras palavras, no futuro, podem aparecer algoritmos que resolvem a fatoração de números inteiros em prazos razoáveis. O mesmo pode ser dito do RSA.

Com isso dito, podemos agora nos concentrar no algoritmo real. O princípio básico é capturado nesta expressão:

$$(me)d \equiv m (mod\ n)$$

Figura 7.4: Expressão básica do RSA

É computacionalmente viável encontrar três números inteiros muito grandes `e`, `d` e `n` que satisfaçam a equação acima. O algoritmo depende da dificuldade de encontrar `d` quando todos os outros números são conhecidos. Em outras palavras, essa expressão pode ser transformada em uma função unidirecional. `d` pode então ser considerado a chave privada, enquanto `n` e `e` são a chave pública.

7.2.2.1 Escolha de e, d e n

1. Escolha dois números primos distintos `p` e `q`.
    - Um gerador de números aleatórios criptograficamente seguro deve ser usado para escolher candidatos para `p` e `q`. Um RNG inseguro pode permitir que um invasor encontre um desses números.
    - Como não há como gerar números primos aleatoriamente, após dois números aleatórios serem escolhidos, eles devem passar por um teste de primalidade. Verificações de primalidade determinísticas podem ser caras, de modo que algumas implementações dependem de métodos probabilísticos. A probabilidade de encontrar um falso primo precisa ser considerada.
    - `p` e `q` devem ser semelhantes em magnitude, mas não idênticos, e devem diferir em comprimento por alguns dígitos.
2. `n` é o resultado de `p` vezes `q`. Este é o módulo da equação acima. Seu número de bits é o comprimento da chave do algoritmo.
3. Calcule a função totiente de Euler para `n`. Como `n` é um número semiprimo, isso é simples:

    `n - (p + q - 1)`. Vamos chamar esse valor de `phi(n)`.
4. Escolha um `e` que atenda aos seguintes critérios:
    - `1 < e < phi(n)`
    - `e` e `phi(n)` devem ser coprimos

5. Escolha um `d` que satisfaça a seguinte expressão:

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

Figura 7.5: Simetria entre e e d

Agora, você provavelmente está se perguntando como o RSA é seguro se publicarmos os valores `e` e `n`; não poderíamos usar esses valores para encontrar `d`? O problema da aritmética modular é que existem várias soluções possíveis. Desde que o `d` que escolhemos satisfaça a equação acima, qualquer valor é válido. Quanto maior o valor, mais difícil será encontrá-lo. Assim, o RSA funciona desde que apenas um dos valores e ou `d` seja conhecido por entidades públicas. Essa também é a razão pela qual um desses valores é escolhido: o valor público pode ser escolhido para ser o menor possível. Isso acelera o cálculo sem comprometer a segurança do algoritmo.

7.2.2.2 Assinatura básica

A assinatura no RSA é realizada da seguinte forma:

1. Um resumo da mensagem é produzido a partir da mensagem a ser assinada por uma função hash.
2. Este resumo é então elevado à potência de `d` módulo `n` (que é parte da chave privada).
3. O resultado é anexado à mensagem como a assinatura.

Quando um destinatário que possui a chave pública deseja verificar a autenticidade da mensagem, pode reverter a operação da seguinte forma:

1. A assinatura é elevada à potência de `e` módulo `n`. O valor resultante é o valor de resumo de referência.
2. Um resumo da mensagem é produzido a partir da mensagem usando a mesma função hash que na etapa de assinatura.
3. Os resultados das etapas 1 e 2 são comparados. Se eles corresponderem, a parte signatária deve estar na posse da chave privada.

Este esquema de assinatura/verificação é conhecido como “esquema de assinatura com apêndice” (SSA). Este esquema requer que a mensagem original esteja disponível para verificar

a mensagem. Em outras palavras, eles não permitem a recuperação da mensagem a partir da assinatura (mensagem e assinatura permanecem separadas).

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

Agora que temos uma noção básica de como o RSA funciona, podemos nos concentrar em uma variante específica: o PKCS#1 RSASSA v1.5 usando SHA-256, também conhecido como RS256 na especificação do JWA.

A especificação do Padrão de Criptografia de Chave Pública #1 (PKCS #1) define uma série de primitivas, formatos e esquemas de criptografia com base no algoritmo RSA. Esses elementos trabalham em conjunto para fornecer uma implementação detalhada do RSA utilizável em plataformas de computação modernas. O RSASSA é um dos esquemas nele definidos, e permite o uso do RSA para assinaturas.

7.2.2.3.1 Algoritmo

Para produzir uma assinatura:

1. Aplique a primitiva **EMSA-PKCS1-V1_5-ENCODE** à mensagem (um array de octetos). O resultado é a **mensagem codificada**. Essa primitiva usa uma função hash (geralmente uma função hash da família SHA, como SHA-256). Essa primitiva aceita um comprimento de mensagem codificada esperado. Neste caso, será o comprimento em octetos do número n do RSA (o comprimento da chave).
2. Aplique a primitiva **OS2IP** à mensagem codificada. O resultado é o **representante de mensagem de número inteiro**. OS2IP é o acrônimo para “Octet-String to Integer Primitive” (Primitiva de string de octeto para número inteiro).
3. Aplique a primitiva **RSASP1** ao representante de mensagem de número inteiro usando a chave privada. O resultado é o **representante de assinatura de número inteiro**.
4. Aplique a primitiva **I2OSP** para converter o representante da assinatura de número inteiro em um array de octetos (a **assinatura**). I2OSP é o acrônimo para “Integer to Octet-String Primitive” (Primitiva de número inteiro para string de octeto).

Uma possível implementação no JavaScript, considerando as primitivas mencionadas acima, poderia parecer com:

```
/**
 * Produz uma assinatura para uma mensagem usando o algoritmo RSA conforme definido
 * no PKCS#1.
 * @param {privateKey} Chave privada RSA, um objeto com
 *                     três membros: size (tamanho em bits), n (o módulo) e
 *                     d (o expoente privado), ambos bigInts
 *                     (biblioteca de inteiros grandes).
 * @param {hashFn} a função hash conforme exigido pelo PKCS#1,
 *                   deve pegar um Uint8Array e retornar um Uint8Array
 * @param {hashType} Um símbolo que identifica o tipo de função hash passada.
 *              Por enquanto, apenas "SHA-256" é compatível. Veja o objeto "hashTypes"
 *              para valores possíveis.
 * @param {message} Uma string ou Uint8Array com dados arbitrários para assinar
 * @return {Uint8Array} signature como um 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;
}
```

Para verificar uma assinatura:

1. Aplique a primitiva **OS2IP** à assinatura (um array de octetos). Este é o r**epresentante de assinatura de número inteiro**.
2. Aplique a primitiva **RSAVP1** ao resultado anterior. Essa primitiva também toma a chave pública como entrada. Este é o **representante de mensagem de número inteiro**.

3. Aplique a primitiva **I2OSP** ao resultado anterior. Essa primitiva assume um tamanho esperado como entrada. Este tamanho deve corresponder ao comprimento do módulo da chave em número de octetos. O resultado é a **mensagem codificada**.
4. Aplique a primitiva **EMSA-PKCS1-V1_5-ENCODE** à mensagem a ser verificada. O resultado é outra **mensagem codificada**. Essa primitiva usa uma função hash (geralmente uma função hash da família SHA, como SHA-256). Essa primitiva aceita um comprimento de mensagem codificada esperado. Neste caso, será o comprimento em octetos do número n do RSA (o comprimento da chave).
5. Compare ambas as mensagens codificadas (das etapas 3 e 4). Se corresponderem, a assinatura é válida, caso contrário, não é.

Em JavaScript:

```
/**
 * Verifica uma assinatura para uma mensagem usando o algoritmo RSASSA conforme definido
 * no PKCS#1.
 * @param {privateKey} Chave privada RSA, um objeto com
 *                     três membros: size (tamanho em bits), n (o módulo) e
 *                     e (o expoente público), ambos bigInts
 *                     (biblioteca de inteiros grandes).
 * @param {hashFn} a função hash conforme exigido pelo PKCS#1,
 *                   deve pegar um Uint8Array e retornar um Uint8Array
 * @param {hashType} Um símbolo que identifica o tipo de função hash passada.
 *              Por enquanto, apenas "SHA-256" é compatível. Veja o objeto "hashTypes"
 *              para valores possíveis.
 * @param {message} A String ou Uint8Array com dados arbitrários para verificar
 * @param {signature} Uma Uint8Array com a assinatura
 * @return {Boolean} verdadeiro se a assinatura for válida, falso caso contrário.
 */
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 Primitiva EMSA-PKCS1-v1_5

Essa primitiva assume três elementos:

- A mensagem
- O comprimento pretendido do resultado
- A função hash a ser usada (que deve ser uma das opções da etapa 2)

1. Aplique a função hash selecionada à mensagem.
2. Produza a codificação DER para a seguinte estrutura ASN.1:

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

Onde `digest` é o resultado da etapa 1 e `DigestAlgorithm` é um dos seguintes:

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

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

3. Se o comprimento solicitado do resultado for menor do que o resultado da etapa 3 mais 11 (reqLength < step2Length + 11), então a primitiva falha em produzir o resultado e emite uma mensagem de erro ("comprimento da mensagem codificada pretendida muito curto").
4. Repita o octeto `0xFF` o seguinte número de vezes: `requested length` + `step2Length` - `3`. Este array de octetos é chamado `PS`.
5. Produza a mensagem codificada final (EM) como (|| é o operador de concatenação):

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

> Os OIDs ASN.1 são normalmente definidos em suas próprias especificações. Em outras palavras, você não encontrará o OID SHA-256 na especificação PKCS#1. Os OIDs SHA-1 e SHA-2 são definidos no RFC 3560.

7.2.2.3.1.2 Primitiva OS2IP

A primitiva OS2IP recebe um array de octetos e gera um representante de número inteiro.

- Considere que X_1, X_2, ..., X_n sejam os octetos do primeiro ao último da entrada.
- Calcule o resultado como:

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

Figura 7.7: Resultado da OS2IP

7.2.2.3.1.3 Primitiva RSASP1

A primitiva RSASP1 pega a chave privada e um representante da mensagem e produz um representante de assinatura.

- Considere `n` e `d` os números RSA para a chave privada.
- Considere `m` o representante da mensagem.

1. Verifique se o representante da mensagem está no intervalo: entre 0 e `n - 1`.
2. Calcule o resultado da seguinte forma:

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

Figura 7.8: Resultado de RSASP1

> O PKCS#1 define uma maneira alternativa e computacionalmente conveniente de armazenar a chave privada: em vez de manter `n` e `d` nela, uma combinação de diferentes valores pré-computados para certas operações é armazenada. Esses valores podem ser usados diretamente em certas operações e podem acelerar significativamente os cálculos. A maioria das chaves privadas é armazenada desta maneira. No entanto, armazenar a chave privada como `n` e `d` é válido.

7.2.2.3.1.4 Primitiva RSAVP1

A primitiva RSAVP1 pega uma chave pública e um representante de assinatura de número inteiro e produz um representante de mensagem de número inteiro.

- Considere `n` e `e` os números RSA para a chave pública.
- Considere o representante de assinatura de número inteiro.

1. Verifique se o representante da mensagem está no intervalo: entre 0 e `n - 1`.
2. Calcule o resultado da seguinte forma:

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

Figura 7.9: Resultado de RSAVP1

7.2.2.3.1.5 Primitiva I2OSP

A primitiva I2OSP pega um representante de número inteiro e produz um array de octetos.

- Considere `len` o comprimento esperado do array de octetos.
- Considere `x` o representante de número inteiro.

1. Se x > 256^{len}, então o número inteiro é muito grande e os argumentos estão errados.
2. Calcule a representação de base 256 do número inteiro:

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

Figura 7.10: Decomposição de I2OSP

3. Tome cada fator x_{len-i} de cada termo na ordem. Estes são os octetos para o resultado.

7.2.2.3.2 Código de exemplo

Como o RSA requer aritmética de precisão arbitrária, usaremos a biblioteca JavaScript de números inteiros grandes.

As primitivas `OS2IP` e `I2OSP` são bastante simples:

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

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

```
    });

    return result;
}

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

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

    return result;
}
```

A primitiva `I2OSP` essencialmente decompõe um número em seus componentes de base 256.

A primitiva `RSASP1` é essencialmente uma única operação e forma a base do algoritmo:

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

```
    }

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

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

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

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

Finalmente, a primitiva `EMSA-PKCS1-v1_5` executa a maior parte do trabalho árduo, transformando a mensagem em sua representação codificada e preenchida.

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

    const digest = hashFn(message, true);

    // DER é um  conjunto mais estrito  de BER, isto (felizmente) funciona:
    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);

    // T é o nome deste elemento no 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);
}
```

Para simplificar, apenas o SHA-256 é compatível. Adicionar outras funções de hash é tão simples quanto adicionar os OIDs certos.

A função `signPkcs1v1_5` reúne todas as primitivas para executar a assinatura:

```
/**
 * Produz uma assinatura para uma mensagem usando o algoritmo RSA conforme definido
 * no PKCS#1.
 * @param {privateKey} Chave privada RSA, um objeto com
 *                     três membros: size (tamanho em bits), n (o módulo) e
```

```
 *                    d (o expoente privado), ambos bigInts
 *                    (biblioteca de inteiros grandes).
 * @param {hashFn} a função hash conforme exigido pelo PKCS#1,
 *                  deve pegar um Uint8Array e retornar um Uint8Array
 * @param {hashType} Um símbolo que identifica o tipo de função hash passada.
 *              Por enquanto, apenas "SHA-256" é compatível. Veja o objeto "hashTypes"
 *              para valores possíveis.
 * @param {message} Uma string ou Uint8Array com dados arbitrários para assinar
 * @return {Uint8Array} A assinatura como um 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;
}
```

Para usar isso para assinar JWTs, é necessário um wrapper simples:

```
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}`;
}
```

Esta função é muito semelhante à função `jwtEncode` para `HS256` mostrada na seção HMAC. Verificar também é simples:

```
/**
 * Verifica uma assinatura para uma mensagem usando o algoritmo RSASSA conforme definido
 * no PKCS#1.
 * @param {privateKey} Chave privada RSA, um objeto com
 *                     três membros: size (tamanho em bits), n (o módulo) e
 *                     e (o expoente público), ambos bigInts
 *                     (biblioteca de inteiros grandes).
 * @param {hashFn} a função hash conforme exigido pelo PKCS#1,
 *                   deve pegar um Uint8Array e retornar um Uint8Array
 * @param {hashType} Um símbolo que identifica o tipo de função hash passada.
 *              Por enquanto, apenas "SHA-256" é compatível. Veja o objeto "hashTypes"
 *              para valores possíveis.
 * @param {message} A String ou Uint8Array com dados arbitrários para verificar
 * @param {signature} Uma Uint8Array com a assinatura
 * @return {Boolean} verdadeiro se a assinatura for válida, falso caso contrário.
 */
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);
}
```

Para usar isso para verificar JWTs, é necessário um wrapper simples:

```
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
    };
}
```

Para simplificar, as chaves privada e pública precisam ser passadas como objetos JavaScript com dois números separados: o módulo (`n`) e o expoente privado (`d`) para a chave privada, e o módulo (`n`) e o expoente público (`e`) para a chave pública. Isso contrasta com o formato usual codificado por PEM. Consulte o arquivo `rs256.js` para obter mais detalhes.

É possível usar OpenSSL para exportar esses números de uma chave PEM.

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

OpenSSL também pode ser usado para gerar uma chave RSA a partir do zero:

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

Você pode então exportar os números do formato PEM usando o comando mostrado acima.

Os números de chave privada incorporados no arquivo `testkey.js` são do arquivo `testkey.pem` no diretório de exemplos que acompanha este manual. A chave pública correspondente está no arquivo `pubtestkey.pem`.

Copie a saída da execução do exemplo rs256.js na área JWT de JWT.io. Em seguida, copie o conteúdo de `pubtestkey.pem` para a área de chave pública na mesma página e o JWT será validado com sucesso.

7.2.2.4 PS256: RSASSA-PSS usando SHA-256 e MGF1 com SHA-256

RSASSA-PSS é outro esquema de assinatura com apêndice baseado em RSA. "PSS" significa Esquema de Assinatura Probabilística, em contraste com a abordagem *determinística* usual. Este esquema usa um gerador de números aleatórios criptograficamente seguro. Se um gerador RNG seguro não estiver disponível, as operações de assinatura e verificação resultantes fornecerão um nível de segurança comparável às abordagens determinísticas. Desta forma, o RSASSA-PSS resulta em uma melhoria líquida em relação às assinaturas do PKCS v1.5 para os melhores cenários. De forma geral, no entanto, ambos os esquemas PSS e PKCS v1.5 permanecem ininterruptos.

RSASSA-PSS é definido no Padrão de Criptografia de Chave Pública#1 (PKCS #1) e não está disponível em versões anteriores do padrão.

7.2.2.4.1 Algoritmo

Para produzir uma assinatura:

1. Aplique a primitiva **EMSA-PSS-ENCODE** à mensagem. A primitiva recebe um parâmetro que deve ser o número de bits no módulo da chave menos 1. O resultado é a **mensagem codificada**.
2. Aplique a primitiva **OS2IP** à mensagem codificada. O resultado é o **representante de mensagem de número inteiro**. OS2IP é o acrônimo para “Octet-String to Integer Primitive” (Primitiva de string de octeto para número inteiro).
3. Aplique a primitiva **RSASP1** ao representante de mensagem de número inteiro usando a chave privada. O resultado é o **representante de assinatura de número inteiro**.

4. Aplique a primitiva **I2OSP** para converter o representante da assinatura de número inteiro em um array de octetos (a **assinatura**). I2OSP é o acrônimo para "Integer to Octet-String Primitive" (Primitiva de número inteiro para string de octeto).

Uma possível implementação no JavaScript, considerando as primitivas mencionadas acima, poderia parecer com:

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

Para verificar uma assinatura:

1. Aplique a primitiva **OS2IP** à assinatura (um array de octetos). Este é o **representante de assinatura de número inteiro**.
2. Aplique a primitiva **RSAVP1** ao resultado anterior. Essa primitiva também toma a chave pública como entrada. Este é o **representante de mensagem de número inteiro**.

3. Aplique a primitiva **I2OSP** ao resultado anterior. Essa primitiva assume um tamanho esperado como entrada. Este tamanho deve corresponder ao comprimento do módulo da chave em número de octetos. O resultado é a **mensagem codificada**.
4. Aplique a primitiva **EMSA-PSS-VERIFY** à mensagem a ser verificada e ao resultado da etapa anterior. Essa primitiva valida a assinatura ou não. Essa primitiva usa uma função hash (geralmente uma função hash da família SHA, como SHA-256). A primitivo recebe um parâmetro que deve ser o número de bits no módulo da chave menos 1.

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

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

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

7.2.2.4.1.1 MGF1: a função de geração de máscara

Funções de geração de máscara pegam uma entrada de qualquer comprimento e produzem uma saída de comprimento variável. Assim como as funções hash, elas são determinísticas: produzem a mesma saída para a mesma entrada. Em contraste com as funções hash, no

entanto, o comprimento da saída é variável. O algoritmo da Função de Geração de Máscara 1 (`MGF1`) é definido no Padrão de Criptografia de Chave Pública #1 (PKCS #1).

O `MGF1` recebe um valor de semente e o comprimento pretendido da saída como entradas. O comprimento máximo da saída é definido como 2^{32}. O `MGF1` usa internamente uma função hash configurável. O `PS256` especifica essa função hash como `SHA-256`.

5. Se o comprimento pretendido for maior que 2^{32}, pare com o erro "máscara muito longa".
6. Faça a iteração de 0 até o teto do comprimento pretendido dividido pelo comprimento da saída da função hash menos 1 (`ceiling(intendedLength / hashLength) - 1`) fazendo as seguintes operações:
    a. Considere `c = i2osp(counter, 4)` onde `counter` é o valor atual do contador de iteração.
    b. Considere `t = t.concat[hash(seed.concat(c)))` onde `t` é preservado entre iterações, `hash` é a função hash selecionada (`SHA-256`) e seed é o valor da semente de entrada.
7. Produza os octetos de comprimento mais à esquerda a partir do último valor de `t` como resultado da função.

7.2.2.4.1.2 Primitiva EMSA-PSS-ENCODE

A primitiva assume dois elementos:

- A mensagem a ser codificada como uma sequência de octetos.
- O comprimento máximo pretendido do resultado em bits.

Essa primitiva pode ser parametrizada pelos seguintes elementos:

- Uma função hash. No caso do `PS256`, este `SHA-256`.
- Uma função de geração de máscara. No caso do `PS256`, este é o `MGF1`.
- Um comprimento pretendido para o salt usado internamente.

Esses parâmetros são todos especificados pelo `PS256`, portanto, não são configuráveis e, para fins desta descrição, são considerados constantes.

Observe que o comprimento pretendido usado como entrada é expresso em bits. Para os exemplos a seguir, considere:

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

8. Se a entrada for maior que o comprimento máximo da função hash, pare. Caso contrário, aplique a função hash na mensagem.

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

9. Se a entrada for maior que o comprimento máximo da função hash, pare. Caso contrário, aplique a função hash na mensagem.

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

10. Gere uma sequência de octeto aleatória do comprimento do salt.
11. Concatene oito octetos de valor zero com o hash da mensagem e o salt.

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

12. Aplique a função hash no resultado da etapa anterior.

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

13. Gere uma sequência de octetos de valor zero de comprimento: comprimento máximo pretendido do resultado menos o comprimento do salt menos o comprimento do hash menos 2.

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

14. Concatene o resultado da etapa anterior com o octeto 0×01 e o salt.

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

15. Aplique a função de geração de máscara no resultado da etapa 5 e defina o comprimento pretendido desta função como o comprimento do resultado da etapa 7 (a função de geração de máscara aceita um parâmetro de comprimento pretendido).

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

16. Calcule o resultado da aplicação da operação XOR aos resultados das etapas 7 e 8.

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

17. Se o comprimento do resultado da operação anterior não for um múltiplo de 8, encontre a diferença no número de bits para torná-lo um múltiplo de 8 subtraindo bits e defina esse número de bits como 0 começando da esquerda.

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

18. Concatene o resultado da etapa anterior com o resultado da etapa 5 e o octeto 0xBC. Este é o resultado.

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

7.2.2.4.1.3 Primitiva EMSA-PSS-VERIFY

A primitiva assume três elementos:

- A mensagem a ser verificada.
- A assinatura como uma mensagem de número inteiro codificada.
- O comprimento máximo pretendido da mensagem de número inteiro codificada.

Essa primitiva pode ser parametrizada pelos seguintes elementos:

- Uma função hash. No caso do `PS256`, este `SHA-256`.
- Uma função de geração de máscara. No caso do `PS256`, este é o `MGF1`.
- Um comprimento pretendido para o salt usado internamente.

Esses parâmetros são todos especificados pelo `PS256`, portanto, não são configuráveis e, para fins desta descrição, são considerados constantes.

Observe que o comprimento pretendido usado como entrada é expresso em bits. Para os exemplos a seguir, considere:

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

1. Faça o hash da mensagem a ser verificada usando a função hash selecionada.

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

2. Se o comprimento esperado for menor que o comprimento do hash mais o comprimento do salt mais 2, considere a assinatura inválida.

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

3. Verifique se o último byte da mensagem codificada da assinatura tem o valor de `0xBC`

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

4. Divida a mensagem codificada em dois elementos. O primeiro elemento tem um comprimento de `expectedLength - hashLength - 1`. O segundo elemento começa no final do primeiro e tem um comprimento de `hashLength`.

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

5. Verifique se os bits `8 * expectedLength - expectedLengthBits` mais à esquerda (o comprimento esperado em bits menos o comprimento solicitado em bits) são 0.

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

6. Passe o segundo elemento extraído da etapa 4 (o resumo) para a função MGF selecionada. Solicite que o resultado tenha um comprimento de `expectedLength - hashLength - 1`.

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

7. Para cada byte do primeiro elemento extraído na etapa 4 (`mascarado`), aplique a função XOR usando o byte correspondente do elemento calculado na última etapa (`dbMask`).

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

8. Defina os bits `8 * expectedLength - expectedLengthBits` mais à esquerda do primeiro byte no elemento calculado na última etapa como 0.

```
const zeroBits = 8 * expectedLength - expectedLengthBits;
```

```
const zeroBitsMask = 0xFF >>> zeroBits;
db[0] &= zeroBitsMask;
```

9. Verifique se os 2 bytes `expectedLength - hashLength - saltLength - 2` mais à esquerda do elemento calculado na última etapa são 0. Verifique também se o primeiro elemento após o grupo de zeros é `0x01`.

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

10. Extraia o salt dos últimos octetos `saltLength` do elemento calculado na última etapa (`db`).

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

11. Calcule uma nova mensagem codificada concatenando oito octetos de valor zero, o hash calculado na etapa 1 e o salt extraído na última etapa.

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

12. Calcule o hash do elemento calculado na última etapa.

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

13. Compare o elemento calculado na última etapa com o segundo elemento extraído na etapa 4. Se eles corresponderem, a assinatura é válida, caso contrário, não é.

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

7.2.2.4.2 Código de exemplo

Como esperado de uma variante do RSASSA, a maior parte do código necessário para este algoritmo já está presente na implementação do RS256. As únicas diferenças são as adições das primitivas `EMSA-PSS-ENCODE`, `EMSA-PSS-VERIFY` e `MGF1`.

```
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 emsaPssEncode(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);
}
```

O exemplo completo está disponível nos arquivos `ps256.js`, `rsassa.js` e `pkcs.js`. Os números de chave privada incorporados no arquivo `testkey.js` são do arquivo `testkey.pem` no diretório de amostras que acompanha este manual. A chave pública correspondente está no arquivo `pubtestkey.pem`. Para obter ajuda na criação de chaves, consulte o exemplo RS256.

7.2.3 Curva elíptica

Algoritmos de Curva Elíptica (EC), assim como RSA, dependem de uma classe de problemas matemáticos que são intratáveis para certas condições. A intratabilidade refere-se à possibilidade de encontrar uma solução com recursos suficientes, mas que, na prática, é difícil de alcançar. Enquanto o RSA depende da intratabilidade do problema de fatoração (encontrar os fatores primos de um número coprimo grande), os algoritmos de curva elíptica dependem da intratabilidade do problema do logaritmo discreto da curva elíptica.

As curvas elípticas são descritas pela seguinte equação:

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

Figura 7.11: Equação da curva elíptica

Definindo a e b com valores diferentes, obtemos as seguintes curvas de amostra:

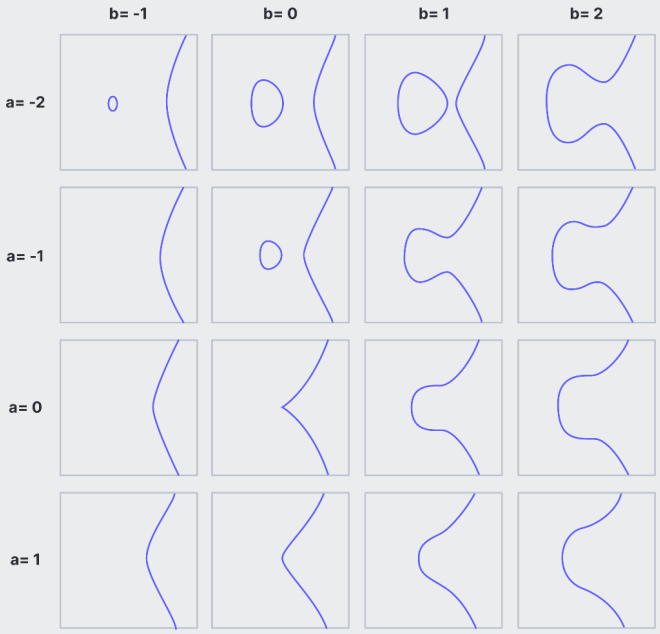


Figura 7.12: Algumas curvas elípticas

[1]

Em contraste com o RSA, algoritmos de curva elíptica são definidos para campos finitos específicos. De interesse para a criptografia EC são os campos binários e primos. A especificação JWA usa apenas campos primos, então nos concentraremos neles.

> Um campo, em termos matemáticos, é um conjunto de elementos para os quais as quatro operações aritméticas básicas são definidas: subtração, adição, multiplicação e divisão.

[1] Imagem de domínio público obtida de Wikimedia.

"Finito" significa que a criptografia de curva elíptica funciona em conjuntos finitos de números, em vez do conjunto infinito de números reais.

Um campo primo é um campo que contém um número primo `p` de elementos. Todos os elementos e operações aritméticas são implementados módulo `p` (o número primo de elementos).

Ao tornar o campo finito, os algoritmos usados para executar operações matemáticas mudam. Em particular, o logaritmo discreto precisa ser usado, em vez do logaritmo comum. Os logaritmos encontram o valor de `k` para expressões da seguinte forma:

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

Figura 7.13: Função exponencial

Não há algoritmo de propósito geral eficiente conhecido para calcular o logaritmo discreto. Essa limitação torna o logaritmo discreto ideal para criptografia. As curvas elípticas, conforme usadas pela criptografia, exploram essa limitação para fornecer operações seguras de criptografia e assinatura assimétricas.

A intratabilidade do problema do logaritmo discreto depende da escolha cuidadosa dos parâmetros do campo no qual ele será usado. Isso significa que, para que a criptografia de curva elíptica seja eficaz, certos parâmetros devem ser escolhidos com muito cuidado. Algoritmos de curva elíptica foram alvo de ataques passados devido a uso indevido.

Um aspecto interessante da criptografia de Curva Elíptica é que os tamanhos das chaves podem ser menores, fornecendo um nível semelhante de segurança em comparação com as chaves maiores usadas no RSA. Isso permite a criptografia mesmo em dispositivos com memória limitada. Em termos gerais, uma chave de curva elíptica de 256 bits é semelhante a uma chave RSA de 3072 bits em força criptográfica.

7.2.3.1 Aritmética da curva elíptica

Para fins de implementação de assinaturas de curva elíptica, é necessário implementar uma aritmética de curva elíptica. As três operações básicas são: adição de pontos, duplicação de

pontos e multiplicação escalar de pontos. Todas as três operações resultam em pontos válidos na mesma curva.

7.2.3.1.1 Adição de pontos

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

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

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

Figura 7.14: Adição de pontos

7.2.3.1.2 Duplicação de pontos

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

$\lambda \equiv$	$3x^p_2+a$	$\pmod{q}$
	$2 y_p$	

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

Figura 7.15: Duplicação de pontos

7.2.3.1.3 Multiplicação escalar

Para multiplicação escalar, o fator k é decomposto em sua representação binária.

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

Figura 7.16: Multiplicação escalar

Em seguida, o seguinte algoritmo é aplicado:

1. Considere `N` o ponto `P`.
2. Considere `Q` o ponto no infinito (`0, 0`).
3. Para `i` de `0` a `m` faça:
    1. Se `k~i~` for `1`, então `Q` será o resultado da adição de `Q` a `N` (adição de curva elíptica).
    2. Considere `N` o resultado da duplicação de `N` (duplicação da curva elíptica).
4. Retorne `Q`.

Exemplo de implementação em JavaScript:

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

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

```
    return Q;
}
```

Algo a observar é que, na aritmética modular, a divisão é implementada como a multiplicação entre o numerador e o inverso do divisor.

Versões JavaScript dessas operações podem ser encontradas no repositório de amostras no arquivo `ecdsa.js`. Essas implementações ingênuas, embora funcionais, são vulneráveis a ataques de tempo. Implementações prontas para produção usam algoritmos diferentes que levam esses ataques em consideração.

7.2.3.2 Algoritmo de assinatura digital de curva elíptica (ECDSA)

O algoritmo de assinatura digital de curva elíptica (ECDSA) foi desenvolvido por um comitê para o American National Standards Institute (ANSI). A norma é X9.63. A norma especifica todos os parâmetros necessários para o uso adequado de curvas elípticas para assinaturas de maneira segura. A especificação JWA depende desta especificação (e FIPS 186-4) para escolher parâmetros de curva e especificar o algoritmo.

Para uso com JWTs, o JWA especifica que a entrada para o algoritmo de assinatura é o cabeçalho e payload codificados com `Base64`, assim como qualquer outro algoritmo de assinatura, mas o resultado é dois números inteiros `r` e `s`, em vez de um. Esses números inteiros devem ser convertidos em sequências de 32 bytes em ordem big-endiana, que são então concatenadas para formar uma única assinatura de 64 bytes.

```
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}`;
}
```

Este código é semelhante ao que foi usado para assinaturas RSA e HMAC. A principal diferença reside na conversão dos dois números de assinatura `r` e `s` em octetos de 32 bytes. Para isso, podemos usar a função `i2osp` do PKCS, que também usamos para RSA.

A verificação da assinatura requer a recuperação dos parâmetros `r` e `s`:

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

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

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

```
    const signature = verify(publicKey,
                             sha256,
                             sha256.hashType,
                             jwtUnprotected,
                             ecSignature);
    return {
        header: header,
        payload: JSON.parse(unb64(split[1])),
        valid: valid
    };
}
```

Novamente, o procedimento para verificar a validade da assinatura é semelhante ao RSA e ao HMAC. Neste caso, os valores `r` e `s` precisam ser recuperados da assinatura JWT de 64 bytes. Os primeiros 32 bytes são o elemento `r`, e os 32 bytes restantes são o elemento `s`. Para converter esses valores em números, podemos usar a primitiva `os2ip` do PKCS.

7.2.3.2.1 Parâmetros de domínio de curva elíptica

As operações de curva elíptica usadas pelo ECDSA dependem de alguns parâmetros principais:

1. `p` ou `q`: o número primo usado para definir o campo primo no qual as operações aritméticas são realizadas. As operações de campo primo usam aritmética modular.
2. `a`: coeficiente de `x` na equação da curva.
3. `b`: constante na equação da curva (interceptação y).
4. `G`: um ponto de curva válido usado como `ponto de base` para operações de curva elíptica. O ponto de base é usado em operações aritméticas para obter outros pontos na curva.
5. `n`: a ordem do ponto de base `G`. Este parâmetro é o número de pontos válidos na curva que podem ser construídos usando o ponto `G` como ponto de base.

Para que as operações de curva elíptica sejam seguras, esses parâmetros devem ser escolhidos com cuidado. No contexto do `JWA`, existem apenas três curvas que são consideradas válidas: `P-256`, `P-384` e `P-521`. Essas curvas são definidas no FIPS 186-4 e outros padrões associados.

Para nossa amostra de código, usaremos a curva P-256:

```
const p256 = {
    q: bigInt('00ffffffff0000001000000000000' +
              '000000000000ffffffffffffffffff' +
              'ffffff', 16),
    // order of base point
    n: bigInt('115792089210356248762697446949407573529996955224135760342' +
              '422259061068512044369'),
    // base point
    G: {
        x: bigInt('6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0' +
                  'f4a13945d898c296', 16),
        y: bigInt('4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ece' +
                  'cbb6406837bf51f5', 16)
    },
    //a: bigInt(-3)
    a: bigInt('00ffffffff0000001000000000000' +
              '000000000000ffffffffffffffffff' +
              'fffffc', 16),
    b: bigInt('5ac635d8aa3a93e7b3ebbd55769886' +
              'bc651d06b0cc53b0f63bce3c3e27d2' +
              '604b', 16)
};
```

7.2.3.2.2 Chaves públicas e privadas

Construir chaves públicas e privadas usando curvas elípticas é realmente simples.

Uma chave privada pode ser criada escolhendo-se um número aleatório entre 1 e a ordem n do ponto de base `G`. Em outras palavras:

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

E pronto! Simples assim.

A chave pública pode ser calculada a partir da chave privada multiplicando-se o ponto de base `G` pela chave privada:

```
// ecMultiply é a operação de multiplicação escalar da curva elíptica
const publicKey = ecMultiply(G, privateKey);
```

Em outras palavras, a chave pública é um ponto na curva elíptica, enquanto a chave privada é simplesmente um valor escalar.

7.2.3.2.2.1 O problema do logaritmo discreto

Dado que a derivação da chave pública da chave privada é simples, fazer o oposto também parece ser simples. Queremos encontrar um número d tal que G multiplicado por ele produza a chave pública `Q`.

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

Figura 7.17: Chave privada como o logaritmo da chave pública

No contexto de um grupo aditivo, tal como o campo primo escolhido para curvas elípticas, a computação de k é o problema do logaritmo discreto. Não há algoritmo de propósito geral conhecido que possa calcular isso de forma eficiente. Para números de 256 bits como os usados para a curva P-256, a complexidade está bem além das capacidades computacionais atuais. É aqui que reside a força da criptografia de curva elíptica.

7.2.3.2.3 ES256: ECDSA usando P-256 e SHA-256

O algoritmo de assinatura em si é simples e requer operações modulares aritméticas e de curva elíptica:

1. Calcule o resumo da mensagem a assinar usando uma função hash criptograficamente segura. Considere este número como `e`.
2. Use um gerador de números aleatórios criptograficamente seguro para escolher um número `k` no intervalo `1` até `n - 1`.
3. Multiplique o ponto de base `G` por `k` (mod `q`).
4. Considere `r` o resultado de tomar a coordenada `x` do ponto da etapa anterior módulo a ordem de `G` (`n`).
5. Se `r` for zero, repita as etapas 2 a 5 até que não seja zero.
6. Considere `d` a chave privada e `s` o resultado de:

$$(x_p,\ y_p)+(x_p,\ y_p)\equiv(x_r,\ y_r)\ (mod\ q)$$

$$\lambda\equiv\frac{3x^p{}_2+a}{2\ y_p}\ (mod\ q)$$

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

Figura 7.18: s

7. Se `s` for zero, repita as etapas 2 a 7 até que não seja zero.

A assinatura é a tupla `r` e `s`. Para fins de JWA, `r` e `s` são representados como duas sequências de octetos de 32 bytes concatenadas (primeiro `r` e depois `s`).

Implementação de exemplo:

```
export function sign(privateKey, hashFn, hashType, message)  {
    if(hashType !== hashTypes.sha256) {
```

```
        throw new Error('unsupported hash type');
    }

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

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

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

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

A verificação também é simples. Para uma determinada assinatura (`r,s`):

1. Calcule o resumo da mensagem a assinar usando uma função hash criptograficamente segura. Considere este número como `e`.
2. Considere `c` o inverso multiplicativo de `s` módulo a ordem `n`.
3. Considere `u1` como `e`, multiplicado por `c` módulo `n`.
4. Considere `u2` como `r`, multiplicado por `c` módulo `n`.
5. Considere o ponto `A` o ponto de base `G` multiplicado por `u1` módulo `q`.
6. Considere o ponto `B` a chave pública `Q` multiplicada por `u2` módulo `q`.
7. Considere o ponto `C` a adição da curva elíptica dos pontos `A` e `B` (módulo `q`).
8. Considere `v` a coordenada `x` do ponto `C` módulo `n`.
9. Se `v` for igual a `r`, a assinatura é válida, caso contrário, não é.

Implementação de exemplo:

```
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;
}
```

Uma parte importante do algoritmo que muitas vezes é ignorada é a verificação da validade da chave pública. Esta foi uma fonte de ataques no passado. Se um invasor controla a chave pública que uma parte verificadora usa para validação da assinatura da mensagem e a chave pública não é validada como um ponto na curva, o invasor pode criar uma chave pública especial que pode ser usada para vazar informações para o invasor. Isso é particularmente importante à luz dos principais protocolos de acordo de chaves, alguns dos quais são usados para criptografar JWTs.

Esta implementação de amostra pode ser encontrada no repositório de amostras no arquivo `ecdsa.js`.

7.3 Atualizações futuras

A especificação do JWA tem muitos mais algoritmos. Em versões futuras deste manual, vamos revisar os algoritmos restantes.

Capítulo 8

Anexo A. Práticas recomendadas atuais

Desde seu lançamento, os JWTs têm sido usados em muitos lugares diferentes. Isso expôs os JWTs e as implementações de bibliotecas a uma série de ataques. Nós mencionamos alguns deles nos capítulos anteriores. Nesta seção, vamos analisar as melhores práticas atuais para trabalhar com JWTs.

Esta seção é baseada no rascunho JWT Best Current Practices (Melhores Práticas Atuais do JWT) do IETF OAuth Working Group. A versão do rascunho usada nesta seção é 00, datada de 19 de julho de 2017.

8.1 Armadilhas e ataques comuns

Antes de dar uma olhada no primeiro ataque, é importante observar que muitos desses ataques estão relacionados à implementação, em vez do design, dos JSON Web Tokens. Isso não os torna menos críticos. É discutível se alguns desses ataques podem ser mitigados ou eliminados com a alteração do design subjacente. No momento, a especificação e o formato do JWT estão definidos de maneira fixa; portanto, a maioria das mudanças acontece no espaço de implementação (alterações em bibliotecas, APIs, práticas de programação e convenções).

Também é importante ter uma ideia básica da representação mais comum para JWTs: o formato Serialização compacta do JWS. JWTs não serializados têm dois objetos JSON principais: `header` e `payload`.

O objeto `header` contém informações sobre o próprio JWT: o tipo de token, o algoritmo de assinatura ou criptografia usado, a ID da chave, etc.

O objeto `payload` contém todas as informações relevantes transportadas pelo token. Existem vários claims padrão, como `sub` (assunto) ou `iat` (emitido em), mas quaisquer claims personalizados podem ser incluídos como parte do payload.

Esses objetos são codificados usando o formato de Serialização compacta do JWS para produzir algo assim:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

Este é um JWT assinado. JWTs assinados em formato compacto são simplesmente os objetos header e payload codificados usando codificação Base64-URL e separados por um ponto (.). A última parte da representação compacta é o objeto signature. Em outras palavras, o formato é:

```
[Header codificado por Base64-URL].[Payload codificado por Base64-URL].[Signature]
```

Isso se aplica a tokens assinados. Os tokens criptografados têm um formato compacto serializado diferente que também depende da codificação Base64-URL e campos separados por pontos.

> Se você quiser praticar com JWTs e ver como eles são codificados/decodificados, veja o arquivo JWT.io.

8.1.1 Ataque “alg: none”

Como mencionamos anteriormente, os JWTs carregam dois objetos JSON com informações importantes: `header` e `payload`. O header inclui informações sobre o algoritmo usado pelo JWT para assinar ou criptografar os dados contidos nele. JWTs assinados assinam tanto o header quanto o payload, enquanto JWTs criptografados criptografam apenas o payload (o header deve estar sempre legível).

No caso de tokens assinados, embora a assinatura (signature) proteja o header e o payload contra adulteração, é possível *reescrever* o JWT sem usar a assinatura e alterar os dados contidos nele. Como isso funciona?

Tome, por exemplo, um JWT com um determinado header e payload. Algo como isto:

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

```
payload: {
  sub: "joe"
  role: "user"
}
```

Agora, digamos que você codifique esse token em sua forma serializada compacta com uma assinatura e uma chave de assinatura de valor "secret". Podemos usar o JWT.io para isso. O resultado é (novas linhas inseridas para legibilidade):

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

Vá em frente e cole isso no JWT.io.

Agora, como este é um token assinado, estamos livres para lê-lo. Isso também significa que poderíamos construir um token semelhante com dados ligeiramente alterados nele, embora nesse caso não seríamos capazes de assiná-lo a menos que soubéssemos a chave de assinatura. Digamos que um invasor não sabe a chave de assinatura – o que ele ou ela poderia fazer? Nesse tipo de ataque, o usuário mal-intencionado pode tentar usar um token sem assinatura! Como isso funciona?

Primeiro, o invasor modifica o token. Por exemplo:

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

Codificado (novas linhas inseridas para legibilidade):

`eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOiJqb2UiLCJyb2xlIjoiYWRtaW4ifQ.`

Observe que este token não inclui uma assinatura (`"alg": "none"`) e que o claim `role` do payload foi alterado. Se o invasor conseguir usar esse token com sucesso, ele ou ela pode aprimorar um ataque de privilégio! Por que um ataque como este funcionaria? Vamos analisar como uma hipotética biblioteca JWT poderia funcionar. Imagine uma função de decodificação que se parece com isto:

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

Esta função recebe um token codificado e um segredo e tenta verificar o token; em seguida, retorna os dados decodificados nele. Se a verificação falhar, ela abrirá uma exceção. Para escolher o algoritmo certo para a verificação, a função depende do claim `alg` do header. É aqui que o ataque é bem sucedido. No passado, muitas bibliotecas confiaram nesse claim para escolher o algoritmo de verificação e, como você deve ter adivinhado, em nosso token mal-intencionado o claim `alg` é `none` (nenhum). Isso significa que não há algoritmo de verificação e a etapa de verificação sempre é bem-sucedida.

Como pode ver, este é um exemplo clássico de um ataque que depende de uma certa ambiguidade da API de uma biblioteca específica, em vez de uma vulnerabilidade na própria especificação. Mesmo assim, este é um ataque real que foi possível em várias implementações diferentes no passado. Por esta razão, muitas bibliotecas hoje relatam tokens `"alg": "none"` como inválidos, mesmo que não haja nenhuma assinatura implantada. Existem outras mitigações possíveis para esse tipo de ataque; a mais importante é sempre verificar o algoritmo especificado no header antes de tentar verificar um token. Outra é usar bibliotecas que exigem o algoritmo de verificação como uma entrada para a função de verificação, em vez de confiar no claim `alg`.

8.1.2 Ataque de chave pública RS256 como segredo HS256

Este ataque é semelhante ao ataque `"alg": "none"` e também depende da ambiguidade na API de certas bibliotecas JWT. Nosso token de exemplo será semelhante ao token daquele ataque. Nesse caso, no entanto, em vez de remover a assinatura, construiremos uma assinatura válida que a biblioteca de verificação também considerará válida, contando com

uma brecha em muitas APIs. Primeiro, considere a assinatura de função típica de algumas bibliotecas JWT para a função de verificação:

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

Como você pode ver aqui, esta função é essencialmente idêntica à do ataque `"alg": "none"`. Se a verificação for bem sucedida, o token será decodificado e retornado; caso contrário, uma exceção será lançada. Neste caso, no entanto, a função também aceita uma chave pública como o segundo parâmetro. De certa forma, isso faz sentido: tanto a chave pública quanto o segredo compartilhado geralmente são strings ou matrizes de bytes, então do ponto de vista dos tipos necessários para esse argumento de função, um único argumento pode representar uma chave pública (para algoritmos RS, ES ou PS) e um segredo compartilhado (para algoritmos HS). Esse tipo de assinatura de função é comum para muitas bibliotecas JWT.

Agora, suponha que o invasor receba um token codificado assinado com um par de chaves RSA. É parecido com este (novas linhas inseridas para legibilidade):

`eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJqb2UiLCJyb2xlIjoidXNlciJ9.QDjcv11Kcb69THVLKMErYqzy9htWlCDtBdonVR5SX4geZa_R8StjwUuuskveUsdJVgjgXwMso7puAJZzoE9LEr9XCxau7SF1ddws4ONiqxSVXZbO0pSgbKm3FpkVz4Jyy4oNTs-bIYyE0xf8snFlT1MbBWcG5psnuG04IEle4s`

Descodificado:

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

Este token é assinado com um par de chaves RSA. As assinaturas RSA são produzidas com a chave privada, enquanto a verificação é feita com a chave pública. Quem verificar o token no futuro pode fazer uma chamada para nossa função hipotética `jwtDecode` anterior desta forma:

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

Mas aqui está o problema: a chave pública é, como o nome indica, geralmente pública. O invasor pode colocar as mãos nela, e isso não deve causar problemas. Mas e se o invasor criasse um novo token usando o seguinte esquema? Primeiro, o invasor modifica o header e escolhe `HS256` como o algoritmo de assinatura:

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

Em seguida, o invasor amplia as permissões alterando o claim de `função` no payload:

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

Este é o ataque: o invasor cria um JWT recém-codificado usando a chave pública, que é uma sequência de caracteres simples, como o segredo compartilhado HS256! Em outras palavras, como o segredo compartilhado para HS256 pode ser qualquer string, mesmo uma string como a chave pública para o algoritmo RS256 pode ser usada para isso.

Agora, se voltarmos ao nosso uso hipotético da função `jwtDecode` de antes:

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

Podemos ver claramente o problema: o token será considerado válido! A chave pública será passada para a função `jwtDecode` como o segundo argumento, mas em vez de ser usada como uma chave pública para o algoritmo RS256, ela será usada como um segredo compartilhado para o algoritmo HS256. Isso é causado pela função `jwtDecode` que depende do claim `alg` do header para escolher o algoritmo de verificação para o JWT. E o invasor mudou isso:

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

Assim como no caso `"alg": "none"`, confiar no claim `alg` combinado com uma API ruim ou confusa pode resultar em um ataque bem-sucedido de um usuário mal-intencionado.

Mitigações contra esse ataque incluem passar um algoritmo explícito para a função `jwtDecode`, verificar o claim `alg` ou usar APIs que separam algoritmos de chave pública de algoritmos de segredos compartilhados.

8.1.3 Chaves HMAC fracas

Os algoritmos HMAC dependem de um segredo compartilhado para produzir e verificar assinaturas. Algumas pessoas assumem que os segredos compartilhados são semelhantes a senhas e, em certo sentido, eles são: devem ser mantidos em segredo. Entretanto, é aí que as semelhanças terminam. Para senhas, embora o comprimento seja uma propriedade importante, o comprimento mínimo necessário é relativamente pequeno em comparação com outros tipos de segredos. Esta é uma consequência dos algoritmos de hash que são usados para armazenar senhas (junto com um salt) que evitam ataques de força bruta em prazos razoáveis.

Por outro lado, os segredos compartilhados HMAC, como usados pelos JWTs, são otimizados para velocidade. Isso permite que muitas operações de assinatura/verificação sejam executadas com eficiência, mas facilita os ataques de força bruta. Portanto, o comprimento do segredo compartilhado para HS256/384/512 é de extrema importância. Na verdade, os algoritmos JSON Web definem o comprimento mínimo da chave como igual ao tamanho em bits da função hash usada junto com o algoritmo HMAC:

> “Uma chave do mesmo tamanho que a saída de hash (por exemplo, 256 bits para ”HS256“) ou maior PRECISA ser usada com este algoritmo.” - Algoritmos JSON Web (RFC 7518), 3.2 HMAC com Funções SHA-2

Em outras palavras, muitas senhas que poderiam ser usadas em outros contextos simplesmente não são boas o suficiente para uso com JWTs assinados por HMAC. 256 bits é igual a 32 caracteres ASCII; portanto, se você estiver usando algo legível, considere esse número como o número mínimo de caracteres a serem incluídos no segredo. Outra boa opção é mudar para RS256 ou outros algoritmos de chave pública, que são muito mais robustos e flexíveis. Este não é simplesmente um ataque hipotético; já foi demonstrado que os ataques de força bruta para HS256 são simples o suficiente para terem sucesso se o segredo compartilhado for curto demais.

8.1.4 Pressuposições erradas de verificação empilhada de criptografia + assinatura

Assinaturas fornecem proteção contra adulteração. Ou seja, embora não impeçam a leitura dos dados, garantem sua imutabilidade: quaisquer alterações aos dados resultam em uma assinatura inválida. A criptografia, por outro lado, torna os dados ilegíveis, a menos que você conheça a chave compartilhada ou a chave privada.

Para muitas aplicações, as assinaturas são tudo o que é necessário. No entanto, para dados confidenciais, pode ser necessário fazer a criptografia. Os JWTs aceitam assinaturas e criptografia.

É muito comum supor erroneamente que a criptografia também fornece proteção contra adulteração em todos os casos. A justificativa para essa suposição geralmente é algo assim: “se os dados não podem ser lidos, como um invasor seria capaz de modificá-los para seu benefício?”. Infelizmente, isso subestima os invasores e seu conhecimento dos algoritmos envolvidos no processo.

Alguns algoritmos de criptografia/descriptografia produzem saída independentemente da validade dos dados passados para eles. Em outras palavras, mesmo que os dados criptografados tenham sido modificados, algo resultará do processo de descriptografia. A modificação cega dos dados geralmente resulta em lixo como saída, mas para um invasor mal-intencionado isso pode ser suficiente para obter acesso a um sistema. Por exemplo, considere um payload JWT que se pareça com isto:

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

Como podemos ver aqui, o claim admin é simplesmente um booleano. Se um invasor conseguir produzir uma alteração nos dados descriptografados que inverta o valor booleano, pode executar com sucesso um ataque de escalonamento de privilégios. Em particular, os invasores com tempo suficiente para executar ataques podem tentar quantas alterações aos dados criptografados quiserem, sem que o sistema descarte o token como inválido antes de processá-lo. Outros ataques podem envolver o fornecimento de dados inválidos para subsistemas que esperam que os dados já estejam higienizados nesse ponto, desencadeando bugs e falhas ou servindo como ponto de entrada para outros tipos de ataques.

Por esse motivo, os Algoritmos JSON Web definem apenas algoritmos de criptografia que também incluem a verificação da integridade dos dados. Em outras palavras, desde que o algoritmo de criptografia seja um dos algoritmos sancionados pelo JWA, pode não ser necessário que seu aplicativo empilhe um JWT criptografado sobre um JWT assinado. No entanto, se você criptografar um JWT usando um algoritmo não padrão, deve garantir que a integridade dos dados seja fornecida por esse algoritmo, ou precisará aninhar os JWTs, usando um JWT assinado como o JWT mais interno para garantir a integridade dos dados.

Os JWTs aninhados são explicitamente definidos e suportados pela especificação. Embora incomuns, eles também podem aparecer em outros cenários, como o envio de um token emitido por alguma parte através de um sistema de terceiros que também use JWTs.

Um erro comum nesses cenários está relacionado à validação do JWT aninhado. Para garantir que a integridade dos dados seja preservada e que todos os dados sejam decodificados corretamente, todas as camadas dos JWTs precisam passar por todas as validações relacionadas aos algoritmos definidos em seus headers. Em outras palavras, mesmo que o JWT mais externo possa ser descriptografado e validado, também é necessário validar (ou descriptografar) todos os JWTs mais internos. Não fazer isso, especialmente no caso de um JWT criptografado mais externo carregando um JWT assinado mais interno, pode resultar no uso de dados não verificados, com todos os problemas de segurança relacionados.

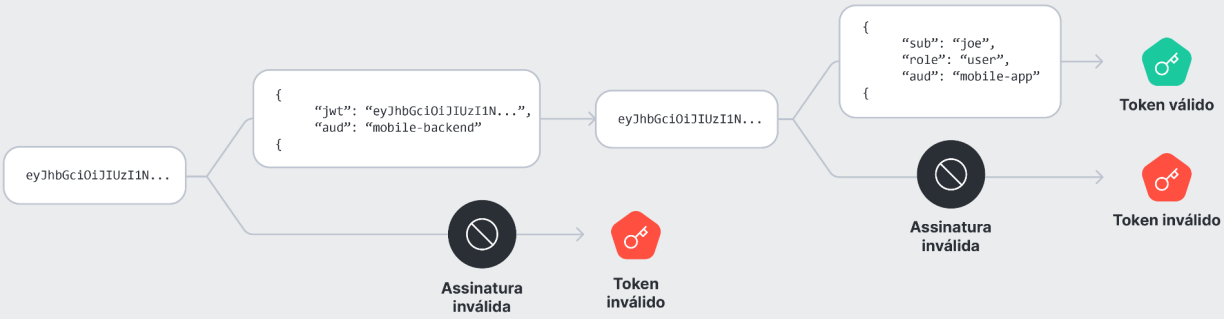


Figura 8.1: Validação do JWT aninhado

8.1.5 Ataques de curva elíptica inválida

A criptografia de curva elíptica é uma das famílias de algoritmos de chave pública suportadas pelos Algoritmos JSON Web. A criptografia de curva elíptica depende da intratabilidade do problema do logaritmo discreto da curva elíptica, um problema matemático que não pode ser resolvido em tempos razoáveis para números grandes o suficiente. Esse problema impede a recuperação da chave privada a partir de uma chave pública, de uma mensagem criptografada e seu texto simples. Quando comparadas ao RSA (outro algoritmo de chave pública que também é suportado por Algoritmos JSON Web), as curvas elípticas fornecem um nível semelhante de força, exigindo chaves menores.

As curvas elípticas, conforme exigido para operações criptográficas, são definidas em campos finitos. Em outras palavras, eles operam em conjuntos de números discretos (em vez de todos os números reais). Isso significa que todos os números envolvidos em operações criptográficas de curva elíptica são números inteiros.

Todas as operações matemáticas de curvas elípticas resultam em pontos válidos sobre a curva. Em outras palavras, a matemática para curvas elípticas é definida de tal maneira que pontos inválidos simplesmente não são possíveis. Se um ponto inválido for produzido, então há um erro nas entradas das operações. As principais operações aritméticas nas curvas elípticas são:

- **Adição de pontos:** adição de dois pontos na mesma curva, resultando em um terceiro ponto na mesma curva.
- **Duplicação de pontos:** adição de um ponto a ele próprio, resultando em um novo ponto na mesma curva.
- **Multiplicação escalar:** multiplicação de um único ponto na curva por um número escalar, definido como a adição repetida desse número a si mesmo `k` vezes (onde `k` é o valor escalar).

Todas as operações criptográficas em curvas elípticas dependem dessas operações aritméticas. Algumas implementações, no entanto, não conseguem validar as entradas para elas. Na criptografia de curva elíptica, a chave pública é um ponto na curva elíptica, enquanto a chave privada é simplesmente um número que fica dentro de um intervalo especial, mas muito grande. Se as entradas para essas operações não forem validadas, as operações aritméticas podem produzir resultados aparentemente válidos, mesmo quando não são. Esses resultados, quando usados no contexto de operações criptográficas, como a descriptografia, podem ser usados para recuperar a chave privada. Este ataque foi demonstrado no passado. Essa classe de ataques é conhecida como ataques de curva inválida. Implementações de boa qualidade sempre verificam se todas as entradas passadas para qualquer função pública são válidas. Isso inclui verificar se as chaves públicas são um ponto de curva elíptica válido para a curva escolhida e se as chaves privadas estão dentro do intervalo de valores válido.

8.1.6 Ataques de substituição

Ataques de substituição são uma classe de ataques em que um invasor consegue interceptar pelo menos dois tokens diferentes. O invasor então consegue usar um ou ambos os tokens para fins diferentes daquele para o qual foram destinados.

Há dois tipos de ataques de substituição: mesmo destinatário (chamado de JWT cruzado no rascunho) e destinatário diferente.

8.1.6.1 Destinatário diferente

Ataques de destinatários diferentes funcionam enviando um token destinado a um destinatário para um diferente. Digamos que exista um servidor de autorização que emita tokens para um serviço de terceiros. O token de autorização é um JWT assinado com o seguinte payload:

```
{
```

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

Este token pode ser usado em uma API para executar operações autenticadas. Além disso, pelo menos quando se trata deste serviço, o usuário `joe` tem privilégios de nível de administrador. No entanto, há um problema com este token: não há destinatário pretendido ou mesmo um emissor nele. O que aconteceria se outra API, diferente do destinatário pretendido para o qual esse token foi emitido, usasse a assinatura como a única verificação de validade? Digamos que exista também um usuário `joe` no banco de dados para esse serviço ou API. O invasor pode enviar esse mesmo token para esse outro serviço e ganhar instantaneamente os privilégios de administrador!

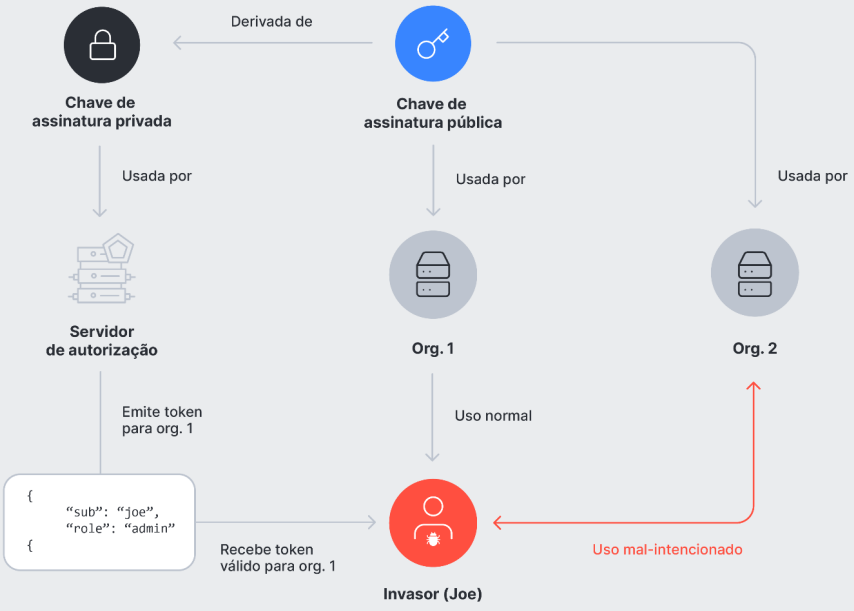


Figura 8.2: Ataque de substituição de JWT com destinatário diferente

Para evitar esses ataques, a validação de token deve contar com chaves ou segredos exclusivos por serviço ou claims específicos. Por exemplo, este token pode incluir um claim `aud`

especificando o público-alvo. Desta forma, mesmo que a assinatura seja válida, o token não pode ser usado em outros serviços que compartilham o mesmo segredo ou chave de assinatura.

8.1.6.2 Mesmo destinatário/JWT cruzado

Este ataque é semelhante ao anterior, mas em vez de depender de um token emitido para um destinatário diferente, o destinatário é o mesmo. O que muda neste caso é que o invasor envia o token para um serviço diferente, em vez do pretendido (dentro da mesma empresa ou provedor de serviços).

Vamos imaginar um token com o seguinte payload:

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

Este token parece muito mais seguro. Temos um claim de emissor (`iss`), um de público-alvo (`aud`) e um de permissões (`perm`). A API para a qual esse token foi emitido verifica todos esses claims, mesmo que a assinatura do token seja válida. Desta forma, mesmo que o invasor consiga colocar as mãos em um token assinado com a mesma chave privada ou segredo, ele não pode usá-lo para operar este serviço se não for destinado a ele.

No entanto, a `cool-company` tem outros serviços públicos. Um desses serviços, o `cool-company/item-database`, foi atualizado recentemente para verificar os claims junto com a assinatura do token. No entanto, durante as atualizações, a equipe responsável por selecionar os claims que seriam validados cometeu um erro: não validou o claim `aud` corretamente. Em vez de verificar uma correspondência exata, eles decidiram verificar a presença da string `cool-company`. Acontece que o outro serviço, o hipotético serviço `cool-company/user-database`, emite tokens que também passam nessa verificação. Em outras palavras, um invasor pode usar o token destinado ao serviço `user-database` em vigor para o

token do serviço `item-database`. Isso concederia ao invasor permissões de gravação para o item-database quando só deveria ter permissões de gravação para o user-database!

Figura 8.3: Ataque de substituição de JWT com mesmo destinatário

8.2 Mitigações e práticas recomendadas

Demos uma olhada em ataques comuns usando JWTs; agora, vamos dar uma olhada na lista atual de práticas recomendadas. Todos esses ataques podem ser evitados com sucesso seguindo estas recomendações.

8.2.1 Executar sempre a verificação de algoritmo

O ataque `"alg": "none"` e o ataque "chave pública RS256 como segredo compartilhado HS256" podem ser evitados por esta mitigação. Toda vez que um JWT precisar ser validado, o algoritmo deve ser explicitamente selecionado para evitar dar controle aos invasores. As bibliotecas costumavam confiar no claim `alg` do header para selecionar o algoritmo para validação. A partir do momento em que ataques como esses passaram a ocorrer, as

bibliotecas começaram a pelo menos fornecer a opção de especificar explicitamente os algoritmos selecionados para validação, desconsiderando o que é especificado no header. Ainda assim, algumas bibliotecas fornecem a opção de usar o que estiver especificado no header, por isso os desenvolvedores devem tomar cuidado para sempre usar a seleção de algoritmo explícita.

8.2.2 Usar algoritmos adequados

Embora a especificação de Algoritmos JSON Web declare uma série de algoritmos recomendados e necessários, escolher o correto para um cenário específico ainda depende dos usuários. Por exemplo, um JWT com uma assinatura HMAC pode ser suficiente para armazenar um pequeno token de um aplicativo simples no navegador de um usuário. Em contraste, um algoritmo de segredo compartilhado seria extremamente inconveniente em um cenário de identidades federadas.

Outra maneira de pensar sobre isso é considerar todos os JWTs inválidos, a menos que esse algoritmo de validação seja aceitável para o aplicativo. Em outras palavras, mesmo que a parte validadora tenha as chaves e os meios necessários para validar um token, ele ainda deve ser considerado inválido se o algoritmo de validação não for o correto para o aplicativo. Essa também é outra maneira de dizer o que mencionamos em nossa recomendação anterior: sempre realize a verificação do algoritmo.

8.2.3 Sempre realizar todas as validações

No caso de tokens aninhados, é necessário sempre executar todas as etapas de validação conforme declarado nos headers de cada token. Em outras palavras, não é suficiente descriptografar ou validar o token mais externo e depois ignorar a validação dos tokens internos. Mesmo no caso de usar apenas JWTs assinados, é necessário validar todas as assinaturas. Esta é uma fonte de erros comuns em aplicativos que usam JWTs para transportar outros JWTs emitidos por partes externas.

8.2.4 Sempre validar as entradas criptográficas

Como já mostramos na seção de ataques, certas operações criptográficas não são bem definidas para entradas fora de seu alcance. Essas entradas inválidas podem ser exploradas para produzir resultados inesperados ou para extrair informações confidenciais que podem levar a um comprometimento total (ou seja, com os invasores obtendo uma chave privada).

No caso de operações de curva elíptica, nosso exemplo de antes, as bibliotecas devem sempre validar chaves públicas antes de usá-las (ou seja, confirmando que representam um ponto válido na curva selecionada). Esses tipos de verificações são normalmente tratados pela biblioteca criptográfica subjacente. Os desenvolvedores precisam garantir que a biblioteca escolhida execute essas validações, ou terão que adicionar o código necessário para executá-las no nível do aplicativo. Deixar de fazer isso pode resultar no comprometimento das chaves privadas.

8.2.5 Escolher chaves fortes

Embora esta recomendação se aplique a qualquer chave criptográfica, ela ainda é ignorada muitas vezes. Como mostramos acima, o comprimento mínimo necessário para segredos compartilhados do HMAC é muitas vezes ignorado. Mas mesmo que o segredo compartilhado seja longo o suficiente, ele também precisa ser totalmente aleatório. Uma chave longa com um nível ruim de aleatoriedade (também conhecida como "entropia") ainda pode ser forçada ou adivinhada. Para garantir que esse não seja o caso, as bibliotecas geradoras de chaves devem contar com geradores de números pseudo-aleatórios de qualidade criptográfica (PRNGs) devidamente semeados durante a inicialização. Na melhor das hipóteses, um gerador de número de hardware pode ser usado.

Esta recomendação se aplica a algoritmos de chave compartilhada e algoritmos de chave pública. Além disso, no caso dos algoritmos de chave compartilhada, as senhas legíveis por humanos não são consideradas boas o suficiente e são vulneráveis a ataques de dicionário.

8.2.6 Validar todos os claims possíveis

Alguns dos ataques que discutimos dependem de suposições de validação incorretas. Em particular, dependem da validação de assinatura ou descriptografia como o único meio de validação. Alguns invasores podem ter acesso a tokens assinados ou criptografados corretamente que podem ser usados para fins maliciosos, geralmente usando-os em contextos inesperados. A maneira certa de evitar esses ataques é considerar apenas um token válido quando a assinatura e o conteúdo são válidos. Por esse motivo, claims como `sub` (assunto), `exp` (tempo de expiração), `iat` (emitido em), `aud` (público-alvo), `iss` (emissor), `nbf` (não válido antes de) são de extrema importância e devem ser sempre validados quando presentes. Se você estiver criando tokens, considere adicionar quantos claims forem necessários para evitar seu uso em diferentes contextos. Em geral, `sub`, `iss`, `aud` e `exp` são sempre úteis e devem estar presentes.

8.2.7 Usar o claim typ para separar tipos de tokens

Embora na maioria das vezes o claim `typ` tenha um único valor (`JWT`), ele também pode ser usado para separar diferentes tipos de JWTs específicos de aplicativos. Isso pode ser útil no caso de seu sistema precisar lidar com muitos tipos diferentes de tokens. Esse claim também pode evitar o uso indevido de um token em um contexto diferente por meio de uma verificação de claim adicional. O padrão JWS permite explicitamente valores do claim `typ` específicos para o aplicativo.

8.2.8 Usar regras de validação diferentes para cada token

Essa prática resume muitas das que foram enumeradas anteriormente. Para evitar ataques, é de fundamental importância garantir que cada token emitido tenha regras de validação muito claras e específicas. Isso não apenas significa usar o claim `typ` quando apropriado, ou validar todos os claims possíveis, como `iss` ou `aud`, mas também implica evitar a reutilização de chaves para diferentes tokens, quando possível, ou usar diferentes claims ou formatos de claim personalizados. Desta forma, os tokens que se destinam a ser usados em um único lugar não podem ser substituídos por outros tokens com requisitos muito semelhantes.

Em outras palavras, em vez de usar a mesma chave privada para assinar todos os tipos de tokens, considere usar chaves privadas diferentes para cada subsistema de sua arquitetura. Você também pode tornar os claims mais específicos, determinando um formato interno para eles. O claim `iss`, por exemplo, pode ser um URL do subsistema que emitiu esse token, em vez do nome da empresa, tornando mais difícil ser reutilizado.

8.3 Conclusão

Os JSON Web Tokens são uma ferramenta que usa criptografia. Como todas as ferramentas que fazem isso, e especialmente aquelas que são usadas para lidar com informações confidenciais, eles precisam ser usados com cuidado. Sua aparente simplicidade pode confundir alguns desenvolvedores e fazê-los pensar que o uso dos JWTs é apenas uma questão de escolher o algoritmo de chave pública ou segredo compartilhado certo. Infelizmente, como vimos acima, esse não é o caso. É de extrema importância seguir as melhores práticas para cada ferramenta em sua caixa de ferramentas, e os JWTs não são exceção. Isso inclui escolher bibliotecas testadas em campo e de alta qualidade; validar claims de payload e header; escolher os algoritmos certos; garantir que chaves fortes sejam geradas; prestar atenção às sutilezas de cada API; entre outras coisas. Se tudo isso parecer assustador,

considere transferir parte da responsabilidade para provedores externos. A Auth0 é um desses provedores. Se você não puder fazer isso, considere essas recomendações com cuidado e lembre-se: não aplique sua própria criptografia, confie em código testado e comprovado.

Auth0, uma unidade de produto dentro da Okta, fornece uma plataforma para autenticar, autorizar e proteger o acesso de aplicativos, dispositivos e usuários.
As equipes de segurança e desenvolvimento contam com fatores como a simplicidade, extensibilidade e experiência da Auth0 para fazer a identidade funcionar para todos.
Resguardando mais de 4,5 bilhões de transações de login a cada mês, a Auth0 protege identidades para que os inovadores possam inovar e capacitar empresas globais a fornecer a melhor e mais confiável experiência digital para seus clientes em todo o mundo.

Para obter mais informações, visite **https://auth0.com** ou siga a **@auth0** no Twitter.