

JWT HANDBOOK

By Sebastián Peyrott



Manual do JWT

Sebastián E. Peyrott, Auth0 Inc.

Versão 0.14.1, 2016-2018

Conteúdo

Agradecimentos especiais	4
1. Introdução	
1.1 O que é um JSON Web Token?	5 . 5 .
1.2 Que problema resolve?	6 . 6
1.3 Um pouco de história	
2 Aplicações Práticas	
2.1	
Sessões do Lado do Cliente/Sem Estado	8 . 8 .
2.1.1 Considerações de segurança.	9 . 9 .
2.1.1.1 Remoção de Assinatura	10
2.1.1.2 Falsificação de solicitação entre sites (CSRF)	11
2.1.1.3 Cross-Site Scripting (XSS)	11
2.1.2 As sessões do lado do cliente são úteis?	13 .
2.1.3 Exemplo	13 .
2.2 Identidade Federada.	16
2.2.1 Tokens de acesso e atualização.	18
2.2.2 JWTs e OAuth2.	19 .
2.2.3 JWTs e OpenID Connect	20 .
2.2.3.1 Fluxos e JWTs do OpenID Connect.	20 .
2.2.4 Exemplo	20 .
2.2.4.1 Configurando o bloqueio Auth0 para aplicativos Node.js	21
3 JSON Web Tokens em detalhes	
3.1 O	
cabeçalho	23 .
3.2 A Carga	24 .
3.2.1 Reivindicações Registradas.	25 .
3.2.2 Créditos Públicos e Privados.	25 .
3.3 JWTs não protegidos.	26 .
3.4 Criando um JWT não seguro	27 .
3.4.1 Exemplo de Código	27 .
3.5 Analisando um JWT não seguro	28 .
3.5.1 Código de Exemplo	28 . 29

4 Assinaturas Web JSON 4.1	
Estrutura de um JWT assinado	30 .
4.1.1 Visão Geral do Algoritmo para Serialização Compacta	30 .
4.1.2 Aspectos práticos dos algoritmos de assinatura.	32 .
4.1.3 Reivindicações de cabeçalho JWS.	33 . 36
4.1.4 Serialização JWS JSON	36
4.1.4.1 Serialização JWS JSON simplificada.	38
4.2 Assinando e validando tokens.	38 .
4.2.1 HS256: HMAC + SHA-256	39 .
4.2.2 RS256: RSASSA + SHA256	39 .
4.2.3 ES256: ECDSA usando P-256 e SHA-256	40
5 JSON Web Encryption (JWE)	
5.1 Estrutura de um JWT Criptografado.	41 .
5.1.1 Algoritmos de criptografia de chave.	44 .
5.1.1.1 Modos de gerenciamento de chaves	45 .
5.1.1.2 Chave de Criptografia de Conteúdo (CEK) e Chave de Criptografia JWE	46 .
5.1.2 Algoritmos de criptografia de conteúdo.	47 .
5.1.3 O Cabeçalho	48 .
5.1.4 Visão Geral do Algoritmo para Serialização Compacta	48 .
5.1.5 Serialização JWE JSON	49 .
5.1.5.1 Serialização JWE JSON simplificada	50 .
5.2 Criptografando e descriptografando tokens	52 .
5.2.1 Introdução: Gerenciando Chaves com node-jose	52 .
5.2.2 AES-128 Key Wrap (Chave) + AES-128 GCM (Conteúdo)	52 .
5.2.3 RSAES-OAEP (Chave) + AES-128 CBC + SHA-256 (Conteúdo)	54 .
5.2.4 ECDH-ES P-256 (Chave) + AES-128 GCM (Conteúdo)	54 . 55
5.2.5 JWT aninhado: ECDSA usando P-256 e SHA-256 (assinatura) + RSAES OAEP (chave criptografada) + AES-128 CBC + SHA-256 (conteúdo criptografado) .	55 . 56
5.2.6 Descriptografia	
6 JSON Web Keys (JWK)	58
6.1 Estrutura de uma Chave da Web JSON	59 .
6.1.1 Conjunto de Chaves da Web JSON	60
7 Algoritmos Web JSON 7.1	
Algoritmos Gerais	61 .
7.1.1 Base64	61 .
7.1.1.1 Base64-URL	61 .
7.1.1.2 Exemplo de Código	63 .
7.1.2 SHA.	63 .
7.2 Algoritmos de assinatura.	64 .
7.2.1 HMAC	69 .
7.2.1.1 HMAC + SHA256 (HS256)	69 .
7.2.2 RSA	71 .
7.2.2.1 Escolhendo e, d e n	73 .
7.2.2.2 Assinatura básica	75 . 76

7.2.2.3 RS256: RSASSA PKCS1 v1.5 usando SHA-256	76 .
7.2.2.3.1 Algoritmo	76 .
7.2.2.3.1.1 Primitiva EMSA-PKCS1-v1_5	78 .
7.2.2.3.1.2 OS2IP primitivo	79 .
7.2.2.3.1.3 Primitiva RSASP1	79 .
7.2.2.3.1.4 Primitiva RSAVP1	80 .
7.2.2.3.1.5 Primitiva I2OSP	80 .
7.2.2.3.2 Exemplo de código	81 .
7.2.2.4 PS256: RSASSA-PSS usando SHA-256 e MGF1 com SHA-256	86 .
7.2.2.4.1 Algoritmo	86 .
7.2.2.4.1.1 MGF1: a função de geração de máscaras	87 .
7.2.2.4.1.2 Primitiva EMSA-PSS-ENCODE	88 .
7.2.2.4.1.3 Primitiva EMSA-PSS-VERIFY	89 .
7.2.2.4.2 Exemplo de código	91 .
7.2.3 Curva Elíptica	94 .
7.2.3.1 Aritmética de Curvas Elípticas	96 .
7.2.3.1.1 Soma de Pontos	96 .
7.2.3.1.2 Duplicação de Pontos	97 .
7.2.3.1.3 Multiplicação escalar	97 .
7.2.3.2 Algoritmo de assinatura digital de curva elíptica (ECDSA)	98 .
7.2.3.2.1 Parâmetros do Domínio da Curva Elíptica	100 .
7.2.3.2.2 Chaves Públicas e Privadas	101 .
7.2.3.2.2.1 O problema do logaritmo discreto	101 .
7.2.3.2.3 ES256: ECDSA usando P-256 e SHA-256	101 .
7.3 Atualizações Futuras	104 .
8 Anexo A. Melhores Práticas Atuais	105
8.1 Armadilhas e Ataques Comuns	105
8.1.1 Ataque	105 .
“alg: nenhum”	106 .
8.1.2 Chave pública RS256 como ataque secreto HS256	108 .
8.1.3 Chaves HMAC fracas	109 .
8.1.4 Criptografia empilhada incorreta + suposições de verificação de assinatura	110 .
8.1.5 Ataques de curva elíptica inválidos	111 .
8.1.6 Ataques de Substituição	112 .
8.1.6.1 Destinatário Diferente	112 .
8.1.6.2 Mesmo Destinatário/JWT Cruzado	114 .
8.2 Mitigações e Melhores Práticas	115 .
8.2.1 Sempre realizar a verificação do algoritmo	115 .
8.2.2 Use algoritmos apropriados	116 .
8.2.3 Sempre realizar todas as validações	116 .
8.2.4 Valide sempre as entradas criptográficas	116 .
8.2.5 Escolher Chaves Fortes	117 .
8.2.6 Validar todas as reivindicações possíveis	117 .
8.2.7 Use a declaração typ para separar tipos de tokens	117 .
8.2.8 Use diferentes regras de validação para cada token	117 .
8.3 Conclusão	118 .

Agradecimentos especiais

Sem 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 trabalhava nele), **Matías Woloski** (por revisar as partes difíceis deste trabalho) , **Martín Gontovnikas** (por aturar meus pedidos e fazer de tudo para facilitar o trabalho), **Bárbara Mercedes Muñoz Cruzado** (por deixar tudo 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), **Federico Jack** (por manter tudo funcionando e ainda encontrar tempo para ouvir todos e cada um).

Capítulo 1

Introdução

JSON Web Token, ou JWT (“jot”) para abreviar, é um padrão para passar *reivindicações com segurança* em ambientes com restrição de espaço. Ele encontrou seu caminho em todos os 1 principais 2 frameworks web³ . Simplicidade, compacidade e usabilidade são as principais características de sua arquitetura. Embora sistemas⁴ 5 muito mais complexos ainda estejam em uso, os JWTs têm uma ampla gama de aplicações. Neste pequeno manual, abordaremos os aspectos mais importantes da arquitetura dos JWTs, incluindo sua representação binária e os algoritmos usados para construí-los, além de dar uma olhada em como eles são comumente usados na indústria.

1.1 O que é um JSON Web Token?

Um JSON Web Token se parece com isso (novas linhas inseridas para facilitar a leitura):

```
eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9.  
eyJzdWIoIxMjM0NTY3ODkwIwibmFtZSI6IkpvG4gRG9IiwiYWRtaW4iOnRydWV9.  
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

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

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

{
```

1<https://github.com/auth0/express-jwt>
 2<https://github.com/nsarno/knock>
 3<https://github.com/tymondesigns/jwt-auth>
 4<https://github.com/jpadilla/django-jwt-auth>
 5<https://en.wikipedia.org/wiki/Security Assertion Markup Language>

```

    "sub": "1234567890",
    "nome": "John Doe",
    "admin": verdadeiro
}

```

As reivindicações são *definições* ou *afirmações* feitas sobre uma determinada parte ou objeto. Algumas dessas declarações e seus significados são definidos como parte da especificação JWT. Outros são definidos pelo usuário. A mágica por trás dos JWTs é que eles padronizam certas declarações que são úteis no contexto de algumas operações comuns. Por exemplo, uma dessas operações comuns é estabelecer a identidade de determinada parte. Portanto, uma das declarações padrão encontradas nos JWTs é a declaração *sub* (de “assunto”). Daremos uma olhada mais profunda em cada uma das reivindicações padrão no [capítulo 3](#).

Outro aspecto fundamental dos JWTs é a possibilidade de assiná-los, usando JSON Web Signatures (JWS, RFC 75156), e/ou criptografá-los, usando JSON Web Encryption (JWE, RFC 75167). Juntamente com JWS e JWE, os JWTs fornecem uma solução poderosa e segura para muitos problemas diferentes.

1.2 Que problema resolve?

Embora o principal objetivo dos JWTs seja transferir reivindicações entre duas partes, sem dúvida o aspecto mais importante disso é o esforço de padronização na forma de um *formato de contêiner simples, opcionalmente validado e/ou criptografado*. Soluções ad hoc para esse mesmo problema foram implementadas tanto privada quanto publicamente no passado. Padrões mais antigos⁸ para estabelecer reivindicações sobre certas partes também estão disponíveis. O que o JWT traz para a mesa é um formato de contêiner padrão *simples* e útil.

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). Examinaremos mais de perto as 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 2011⁹. O objetivo do grupo era “padronizar o mecanismo de proteção de integridade (assinatura e MAC) e criptografia, bem como o formato de chaves e identificadores de algoritmos para suportar a interoperabilidade de serviços de segurança para protocolos que usam JSON”. No ano de 2013, uma série de rascunhos, incluindo um livro de receitas

⁶<https://tools.ietf.org/html/rfc7515>

⁷<https://tools.ietf.org/html/rfc7516>

⁸<https://en.wikipedia.org/wiki/Security Assertion Markup Language>

⁹<https://datatracker.ietf.org/wg/jose/history/>

com diferentes exemplos de uso das ideias produzidas pelo grupo. Esses rascunhos se tornariam mais tarde os RFCs JWT, JWS, JWE, JWK e JWA. A partir do ano de 2016, esses RFCs estão no processo de rastreamento de padrões e não foram encontradas erratas neles. O grupo está atualmente inativo.

Os principais autores por trás das especificações são Mike Jones¹⁰, Nat Sakimura¹¹, John Bradley¹² e Joe Hildebrand¹³.

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

Capítulo 2

Aplicações práticas

Antes de nos aprofundarmos na estrutura e construção de um JWT, veremos várias aplicações práticas. Este capítulo lhe dará uma noção da complexidade (ou simplicidade) das soluções comuns baseadas em JWT usadas no setor atualmente. Todo o código está disponível em repositórios públicos¹ para sua conveniência. Esteja ciente de que as demonstrações a seguir *não devem ser usadas em produção*. Casos de teste, log e práticas recomendadas de segurança são essenciais para o código pronto para produção. Essas amostras são apenas para fins educacionais e, portanto, permanecem simples e diretas.

2.1 Sessões do lado do cliente/sem estado

As chamadas sessões *sem estado* nada mais são do que dados do lado do cliente. O aspecto principal deste aplicativo 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, deve ser tratado com muito cuidado pelo back-end.

JWTs, em virtude de JWS e JWE, podem fornecer vários tipos de assinaturas e criptografia. As naturezas de assinatura são úteis para *validar* os dados contra adulteração. A criptografia é útil para *proteger* os dados de serem lidos por terceiros.

Na maioria das vezes, as sessões precisam apenas ser assinadas. Em outras palavras, não há preocupação com segurança ou privacidade quando os dados neles armazenados são lidos por terceiros. Um exemplo comum de uma reivindicação que geralmente pode ser lida com segurança por terceiros é a *subreivindicação* ("assunto"). A declaração de assunto geralmente identifica uma das partes para a outra (pense em IDs de usuário ou e-mails). Não é um requisito que esta reivindicação seja *única*. Em outras palavras, declarações adicionais podem ser necessárias para identificar exclusivamente um usuário. Isso é deixado para os usuários decidirem.

Uma reivindicação que não pode ser deixada em aberto apropriadamente pode ser uma reivindicação de "itens" que representa o carrinho de compras de um usuário. Este carrinho pode estar cheio de itens que o usuário está prestes a comprar e

¹<https://github.com/auth0/jwt-handbook-samples>

portanto, estão associados à sua sessão. Um terceiro (um script do lado do cliente) pode coletar esses itens se eles estiverem armazenados em um JWT não criptografado, o que pode gerar problemas de privacidade.

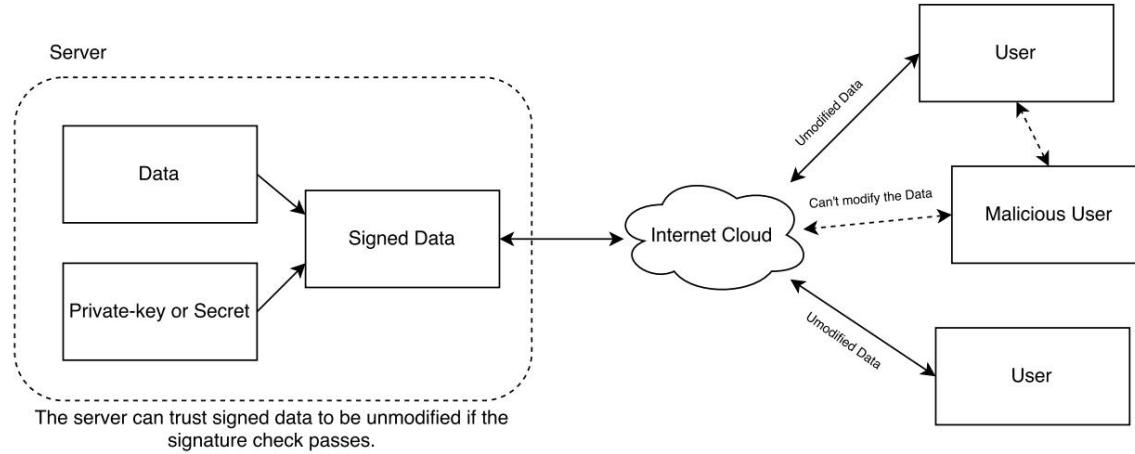


Figura 2.1: Dados assinados do lado do cliente

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: o cabeçalho, a carga útil e a assinatura. Essas três partes são codificadas separadamente. Dessa forma, é possível remover a assinatura e, em seguida, alterar o cabeçalho para afirmar que o JWT não está assinado. O uso descuidado de certas bibliotecas de validação JWT pode resultar em tokens não assinados sendo considerados tokens válidos, o que pode permitir que um invasor modifique a carga a seu critério. Isso é facilmente resolvido certificando-se de que o aplicativo que executa a validação não considere JWTs não assinados válidos.

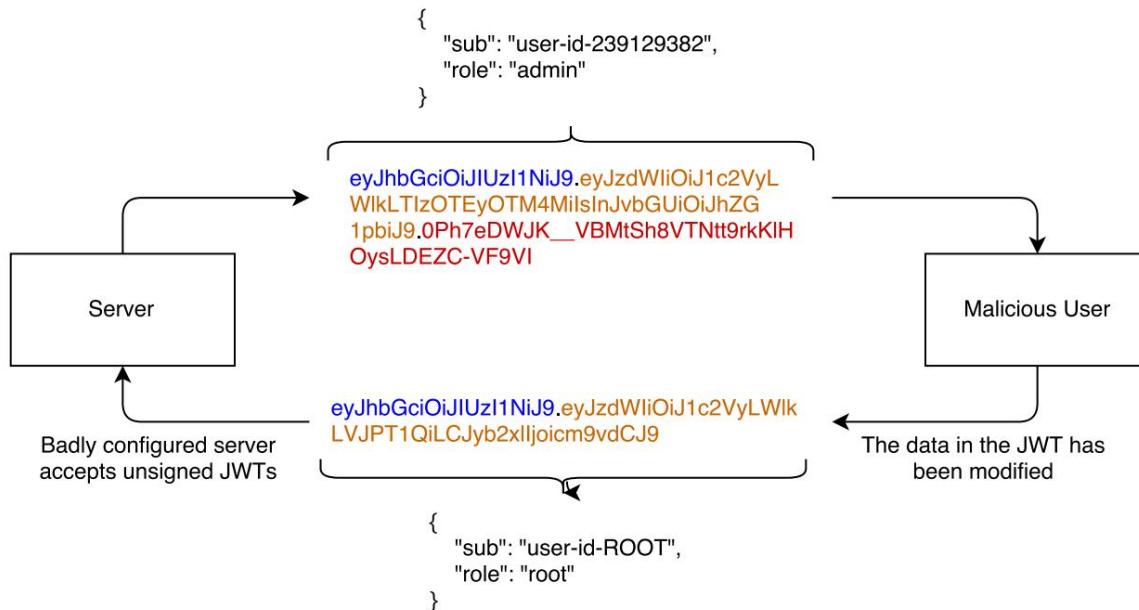


Figura 2.2: Remoção de Assinatura

2.1.1.2 Falsificação de solicitação entre sites (CSRF)

Os ataques de falsificação de solicitação entre sites tentam realizar solicitações contra sites nos quais o usuário está conectado, enganando o navegador do usuário para que envie uma solicitação de um site diferente. Para conseguir isso, um site (ou item) especialmente criado deve conter a URL para o destino. Um exemplo comum é uma tag incorporada em uma página maliciosa com o src apontando para o alvo do ataque. Por exemplo:

```
<!-- Isso está incorporado no site de outro domínio -->

```

A tag acima enviará uma solicitação para target.site.com toda vez que a página que a contém for carregada. Se o usuário tiver feito login anteriormente 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 qualquer outro dado do lado do cliente, podem ser armazenados como cookies.

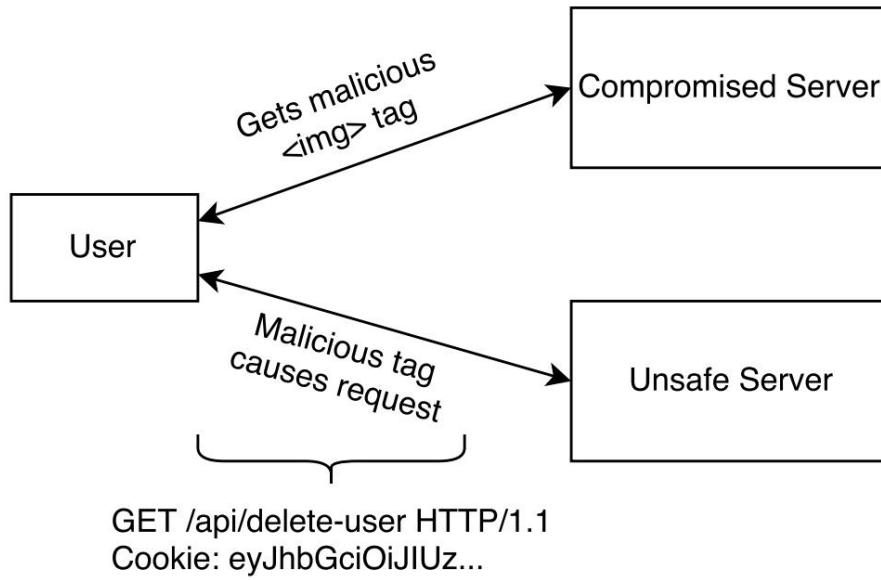


Figura 2.3: Falsificação de solicitação entre sites

JWTs de curta duração podem ajudar nesse caso. As técnicas comuns de mitigação de CSRF incluem cabeçalhos especiais que são adicionados às solicitações somente quando são executados a partir da origem correta, por cookies de sessão e por tokens de solicitação. Se os JWTs (e os dados da sessão) não forem armazenados como cookies, os ataques CSRF não serão possíveis. Ataques de script entre sites ainda são possíveis, no entanto.

2.1.1.3 Cross-Site Scripting (XSS)

Os ataques de script entre sites (XSS) tentam injetar JavaScript em sites confiáveis. O JavaScript injetado pode 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. Ataques XSS comuns geralmente são causados por validação imprópria de dados passados para o back-end (de maneira semelhante aos ataques de injeção de SQL).

Um exemplo de ataque XSS pode estar relacionado à seção de comentários de um site público. Sempre 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 limpar os comentários, um usuário mal-intencionado pode escrever um comentário de forma que possa ser interpretado pelo navegador como uma tag <script>. Assim, um usuário mal-intencionado pode inserir código JavaScript arbitrário e executá-lo no navegador de cada usuário, roubando credenciais armazenadas como cookies e no armazenamento local.

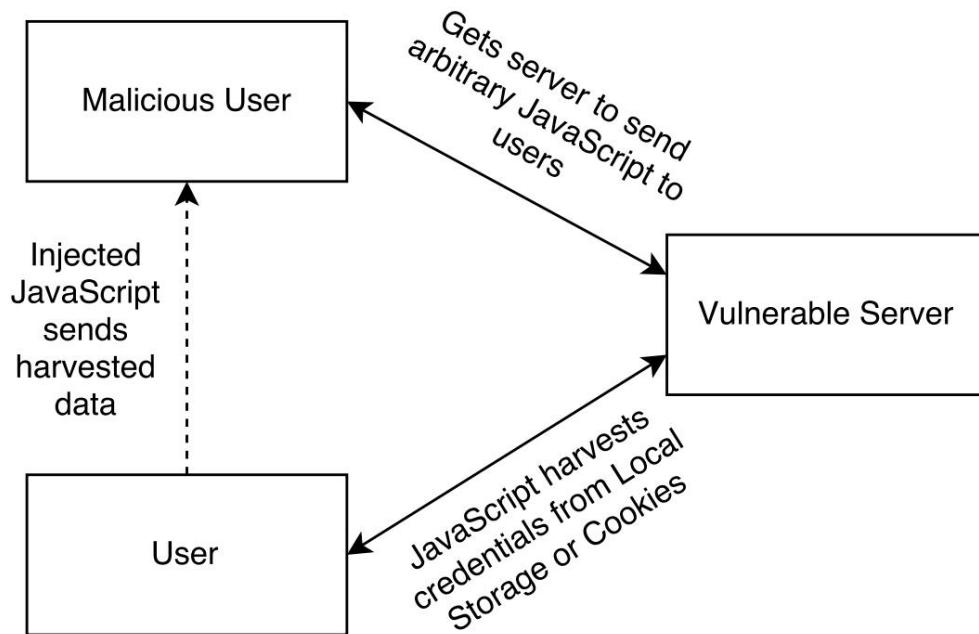


Figura 2.4: Script Cross Site Persistente

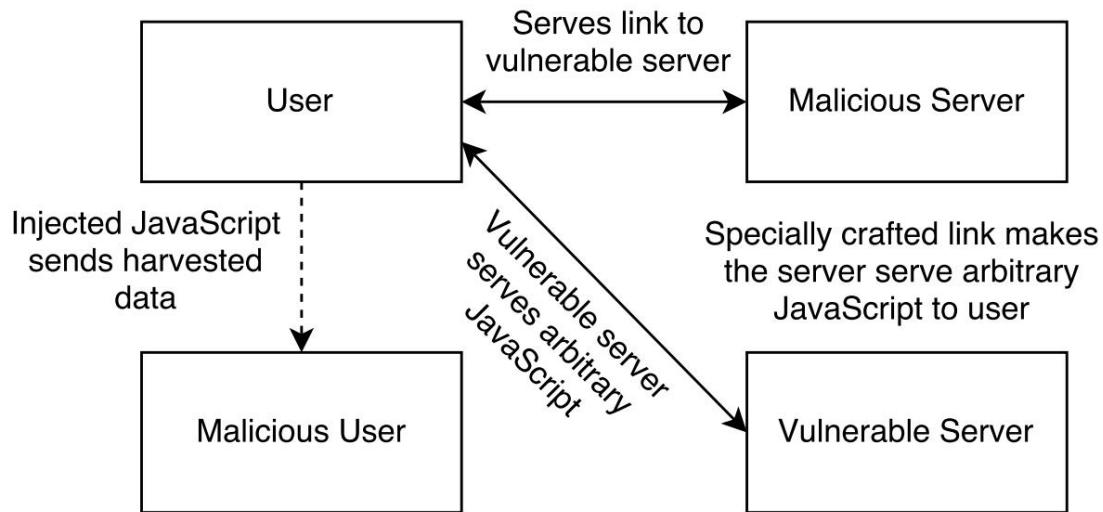


Figura 2.5: Script reflexivo entre sites

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 higienizados. Se cookies forem usados, é possível protegê-los de serem acessados por JavaScript definindo o sinalizador `HttpOnly2`. O sinalizador `HttpOnly`, embora útil, não protegerá o cookie de ataques CSRF.

2.1.2 As sessões do lado do cliente são úteis?

Existem prós e contras em qualquer abordagem, e as sessões do lado do cliente não são uma exceção³. Alguns aplicativos podem exigir grandes sessões. Enviar esse estado para frente e para trás para cada solicitação (ou grupo de solicitações) pode facilmente superar os benefícios da tagarelice reduzida no back-end. É necessário um 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 mapeiam bem para 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, estude os benefícios de manter determinado estado do lado do cliente. Os JWTs são muito grandes? Isso tem um impacto na largura de banda? Essa largura de banda adicionada 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 sobre a abordagem correta.

2.1.3 Exemplo

Para o nosso exemplo faremos uma aplicação de compras simples. O carrinho de compras do usuário será armazenado no lado do cliente. Neste exemplo, há vários JWTs presentes. Nosso carrinho de compras será um deles.

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

Veja como o carrinho de compras fica quando decodificado:

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

Cada item é identificado por um ID numérico. O JWT codificado e assinado se parece com:

²<https://www.owasp.org/index.php/HttpOnly>

³<https://auth0.com/blog/stateless-auth-for-stateful-minds/>

```
eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9.
eyJpdGVtcyI6WzAsMiw0XSwiaWF0ljoxNDkzMjM5NjU5LCJleHAIoJE0OTMxNDMyNTI9.
932ZxtZzy1qhLXs932hd04J58lhbg5_g_rIj-Z16Js
```

Para renderizar os itens no carrinho, o frontend só precisa recuperá-los de seu cookie:

```
function populateCart() { const
  carrinhoElem = $('#carrinho');
  carrinhoElem.vazio();

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

}

carrinho const = jwt_decode(cartToken).items;

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

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

Aqui está a verificação de back-end para a validade do carrinho JWT implementado 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

  }; } pegar(e) {
    req.carrinho = { itens: [] };
  }
}

próximo();
}
```

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,
    cartSignJwtOptions);

res.cookie('carrinho', novoCarrinho,
    { maxAge: 1000 * 60 * 60
});

reenviar();

console.log(` ID do item ${req.query.id} adicionado ao carrinho.`);
);

```

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

```

app.use('/protegido', expressJwt({
    segredo: jwksClient.expressJwtSecret(jwksOpts), emissor:
    process.env.AUTH0_API_ISSUER, público:
    process.env.AUTH0_API_AUDIENCE, requestProperty:
    'accessToken', getToken: req => { return req.cookies['access_token'];

} ));

```

Em outras palavras, o terminal /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 por Auth0 ao nosso aplicativo. Isso requer a configuração de um cliente⁴ e um terminal de API⁵ usando o painel Auth0⁶. Estes são recuperados usando a biblioteca JavaScript Auth0, chamada pelo nosso frontend:

```

//Auth0 ID do cliente
const clientId = "t42WY87weXzepAdUlwMiHYRBQj9qWVAT"; //Auth0
Domain const domain = "speyrott.auth0.com";

```

```

const auth0 = new window.auth0.WebAuth({
    domínio: domínio,
    clientID: clientId, público:
    '/protected',

```

⁴<https://manage.auth0.com/#/clients>
⁵<https://manage.auth0.com/#/apis>
⁶<https://manage.auth0.com>

```

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

});

//(...)

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

```

A declaração de público deve corresponder àquela configurada para seu terminal de API usando o painel Auth0.

O servidor de autenticação e autorização Auth0 exibe uma tela de login com nossas configurações e redireciona de volta para 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 desse código pode ser encontrado no diretório samples/stateless-sessions.

2.2 Identidade Federada

Os sistemas federados de identidade⁷ permitem que partes diferentes, 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 gerenciamento de identidade federada: SAML⁸ e OpenID Connect⁹ são duas das mais comuns. Certas empresas fornecem produtos especializados que centralizam a autenticação e a autorização. Estes podem implementar um dos padrões mencionados acima ou usar algo completamente diferente. Algumas dessas empresas usam JWTs para essa finalidade.

A utilização de JWTs para autenticação e autorização centralizada varia de empresa para empresa, mas o fluxo essencial do processo de autorização é:

⁷<https://auth0.com/blog/2015/09/23/what-is-and-how-does-single-sign-on-work/>

⁸<http://saml.xml.org/saml-specifications> ⁹<https://openid.net/connect/>

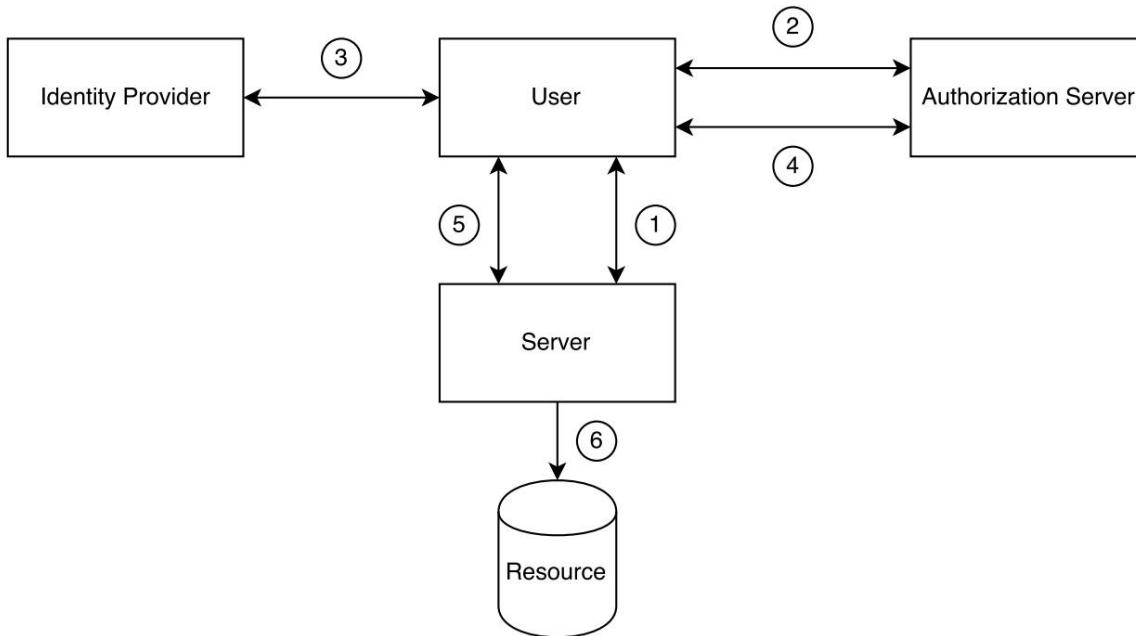


Figura 2.6: Fluxo de identidade federada comum

1. O usuário tenta acessar um recurso controlado por um servidor.
2. O usuário não possui 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 efetua 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 para servidor fluem pelo usuário sendo incorporados nas solicitações de redirecionamento (geralmente como parte da 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 por meio 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 por meio do Twitter, Facebook e um banco de dados de usuários comum.

2.2.1 Tokens de acesso e atualização

Tokens de acesso e atualização são dois tipos de tokens que você verá muito ao analisar diferentes soluções de identidade federada. Explicaremos brevemente o que são e como auxiliam no contexto de autenticação e autorização.

Ambos os conceitos geralmente são implementados no contexto da especificação OAuth210. A especificação OAuth2 define uma série de etapas necessárias para fornecer acesso aos 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 pela implementação*. Ou seja, as implementações OAuth2 concorrentes podem não ser interoperáveis. Por exemplo, o formato binário real dos tokens *não é especificado*. Sua finalidade e funcionalidade são.

Tokens de acesso são tokens que dão a quem os possui acesso a recursos protegidos. Esses tokens geralmente têm vida curta e podem ter uma data de validade incorporada a eles. Eles também podem conter ou estar associados a informações adicionais (por exemplo, um token de acesso pode conter o endereço IP 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, após a expiração de um token de acesso, um cliente pode solicitar um novo token de acesso ao servidor de autorização. Para que essa solicitação seja atendida, é necessário um token de atualização. Ao contrário dos tokens de acesso, os tokens de atualização geralmente são de longa duração.

¹⁰<https://tools.ietf.org/html/rfc6749#section-1.4>

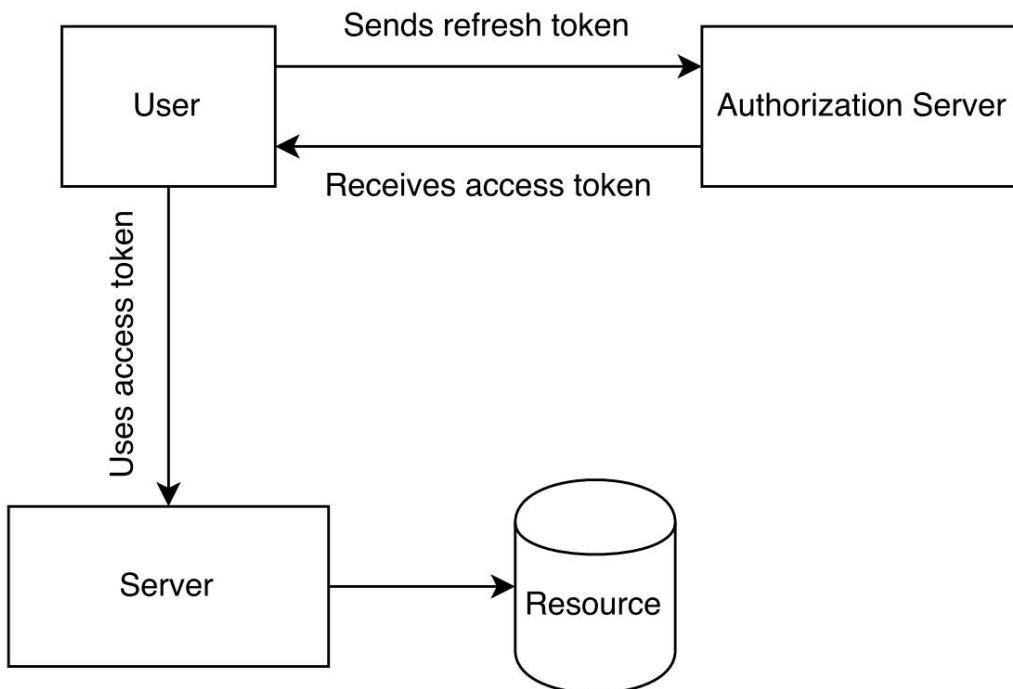


Figura 2.7: tokens de atualização e acesso

O aspecto chave da separação entre tokens de acesso e 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 servidor de recursos por conta própria. Não há necessidade de entrar em contato com o servidor de autorização para esse fim.

Os tokens de atualização, por outro lado, requerem acesso ao servidor de autorização. Ao manter a validação separada das consultas ao servidor de autorização, é possível obter melhor latência e padrões de acesso menos complexos. A segurança adequada em caso de vazamentos de token é obtida tornando os tokens de acesso o mais curto possível e incorporando verificações adicionais (como verificações de cliente) neles.

Os tokens de atualização, por serem de longa duração, devem ser protegidos contra vazamentos. No caso de um vazamento, a lista negra pode ser necessária no servidor (tokens de acesso de curta duração forçam tokens de atualização a serem usados eventualmente, protegendo assim o recurso depois que ele é colocado na lista negra e todos os tokens de acesso expiram).

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 maneira 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 ter uma data de expiração e são assinados para evitar consultas de validação no servidor de autorização. Vários provedores de identidade federados emitem tokens de acesso no formato JWT.

Os JWTs também podem ser usados para tokens de atualização. Há menos razão para usá-los para esta finalidade, no entanto. Como os tokens de atualização requerem acesso ao servidor de autorização, na maioria das vezes um UUID simples será suficiente, pois não há necessidade de o token carregar uma carga útil (embora possa ser assinado).

2.2.3 JWTs e OpenID Connect

OpenID Connect¹¹ é um esforço de padronização para trazer casos de uso típicos de OAuth2 sob uma especificação comum e bem definida. Como muitos detalhes por trás do OAuth2 são deixados para a escolha dos implementadores, o OpenID Connect tenta fornecer definições adequadas para as partes que faltam. Especificamente, o OpenID Connect define uma API e formato de dados para realizar fluxos de autorização OAuth2. Além disso, fornece uma camada de autenticação construída sobre esse fluxo. O formato de dados escolhido para algumas de suas partes é o JSON Web Token. Em particular, o ID token¹² é um tipo especial de token que carrega informações sobre o usuário autenticado.

2.2.3.1 Fluxos e JWTs do OpenID Connect

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 terminal de autorização (/autorizar). 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 terminal de autorização (/autorizar). 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 determinados tokens do terminal de autorização (/autorizar). Se um token de ID for solicitado, ele será retornado no formato JWT. Se um token de ID não for solicitado nesta etapa, ele poderá ser solicitado posteriormente diretamente do endpoint do token (/token).

2.2.4 Exemplo

Para este exemplo, usaremos Auth0¹³ como o servidor de autorização. Auth0 permite que diferentes provedores de identidade 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 identidade (como Twitter, Facebook, etc.). Os aplicativos não precisam se comprometer com provedores específicos depois de implantados. Então nosso exemplo

¹¹<https://openid.net/connect/>

¹²http://openid.net/specs/openid-connect-core-1_0.html#IDToken

¹³<https://auth0.com>

pode ser bastante simples. Configuramos a tela de login Auth0 usando a biblioteca Auth0.js¹⁴ em todos os nossos servidores de amostra. Depois que um usuário fizer login em um servidor, ele também terá acesso aos outros servidores (mesmo que não estejam interconectados).

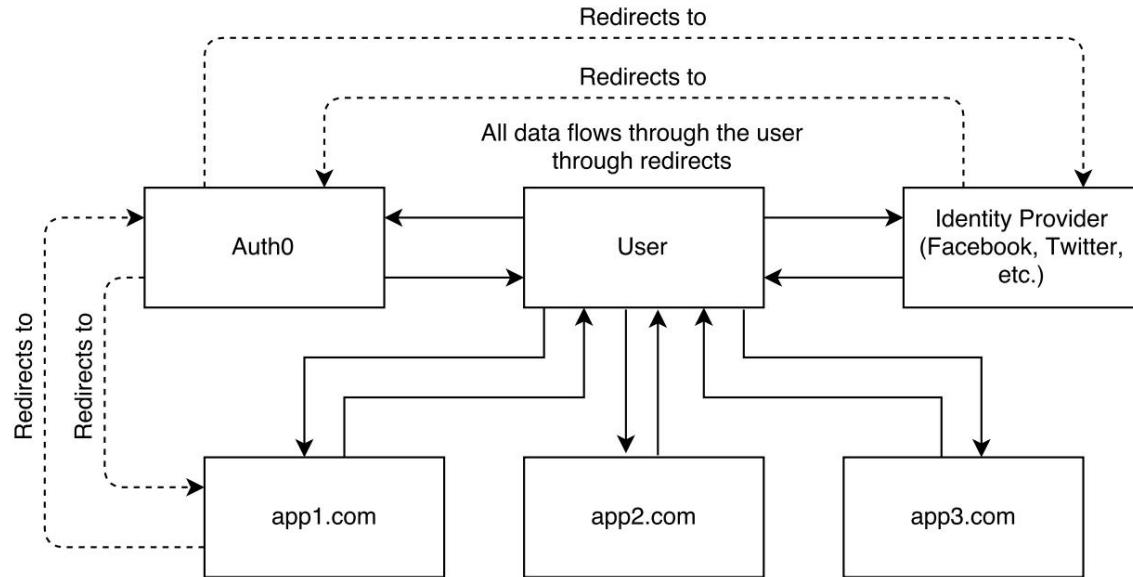


Figura 2.8: Auth0 como servidor de autorização

2.2.4.1 Configurando o bloqueio Auth0 para aplicativos Node.js

A configuração da biblioteca Auth0¹⁵ pode ser feita da seguinte maneira. Usaremos o mesmo exemplo usado para o exemplo de sessões sem estado:

```
const auth0 = new window.auth0.WebAuth({
    domínio: domínio,
    clientID: clientId, público:
    'app1.com/protected', escopo: 'openid
    profile purchase', responseType: 'id_token
    token', redirectUri: 'http://app1.com:3000/
    auth', responseMode: 'form_post'

});

//(...)
```

¹⁴<https://github.com/auth0/auth0.js>

¹⁵<https://github.com/auth0/auth0.js>

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

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 nenhum, se o usuário já deu autorização para um aplicativo usar suas credenciais para acessar um recurso protegido, o servidor de autorização simplesmente redirecionará de volta para o aplicativo. Isso parece para o usuário como se ele já estivesse logado no aplicativo.

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

Para testar isso, consulte o arquivo README para obter o exemplo localizado no diretório samples/single-sign-on-federated-identity para configurar os dois aplicativos e executá-los. Quando ambos estiverem em execução, acesse app1.com:3000¹⁶ e app2.com:3001¹⁷ e faça o login. Em seguida, saia de ambos os aplicativos. Agora tente fazer login em um deles. Em seguida, volte para o outro e faça o login. Você notará que a tela de login estará ausente em ambos os aplicativos. O servidor de autorização lembra 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, ele já está logado em ambos os aplicativos.

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

¹⁶<http://app1.com:3000>
¹⁷<http://app2.com:3001>

Capítulo 3

Tokens da Web JSON em detalhes

Conforme descrito no [capítulo 1](#), todos os JWTs são construídos a partir de três elementos diferentes: o cabeçalho, a carga útil e os dados de assinatura/criptografia. Os dois primeiros elementos são objetos JSON de uma determinada estrutura. O terceiro depende do algoritmo usado para assinatura ou criptografia e, no caso de JWTs *não criptografados*, é omitido. JWTs podem ser codificados em uma *representação compacta* conhecida como *JWS/JWE Compact Serialization*.

As especificações 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 baseada em URL Base641 dos bytes UTF-82 dos dois primeiros elementos JSON (o cabeçalho e a carga útil) e os dados, conforme necessário, para assinatura ou criptografia (que não é um objeto JSON em si). Esses dados também são codificados em Base64-URL. Esses três elementos são separados por pontos (".").

O JWT usa uma variante da codificação Base64 que é segura para URLs. Essa codificação basicamente substitui os caracteres "+" e "/" pelos caracteres "-" e "_", respectivamente.

O preenchimento também é removido. Essa variante é conhecida como `base64url3`. Observe que todas as referências à codificação Base64 neste documento referem-se a esta variante.

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

```
eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9.  
eyJzdWlOIixMjM0NTY3ODkwliwibmFtZSI6Ikpvag4gRG9IiwiYWRtaW4iOnRydWV9.  
TJVA95OrM7E2cBab30RMHrHDcEfijoYZgeFONFh7HgQ
```

Observe os pontos que separam os três elementos do JWT (na ordem: o cabeçalho, a carga útil e a assinatura).

Neste exemplo, o cabeçalho decodificado é:

¹<https://en.wikipedia.org/wiki/Base64>

²<https://en.wikipedia.org/wiki/UTF-8>

³<https://tools.ietf.org/html/rfc4648#section-5>

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

A carga útil decodificada é:

```
{
  "sub": "1234567890",
  "nome": "John Doe",
  "admin": verdadeiro
}
```

E o segredo necessário para verificar a assinatura é secreto.

JWT.io⁴ é um playground interativo para aprender mais sobre JWTs. Copie o token acima e veja o que acontece quando você o edita.

3.1 O Cabeçalho

Todo JWT carrega um cabeçalho (também conhecido como *cabeçalho JOSE*) com declarações sobre si mesmo. Essas declarações estabelecem os algoritmos usados, se o JWT é 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 cabeçalho. Por exemplo, JWTs criptografados carregam informações sobre os algoritmos criptográficos usados para criptografia de chave e criptografia de conteúdo. Esses campos não estão presentes para JWTs não criptografados.

A única declaração obrigatória para um cabeçalho JWT *não criptografado* é a declaração alg:

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

Para JWTs não criptografados, essa declaração deve ser definida como o valor nenhum.

As declarações de cabeçalho opcionais incluem as declarações type e cty:

- **type** : o tipo de mídia⁵ do próprio JWT. Este parâmetro destina-se apenas a ser usado como uma ajuda para usos em que os JWTs podem ser misturados com outros objetos que carregam um cabeçalho JOSE. Na prática, isso raramente acontece. Quando presente, essa declaração deve ser definida como o valor JWT.
- **cty**: o tipo de conteúdo. A maioria dos JWTs carrega declarações específicas e dados arbitrários como parte de sua carga útil. Nesse caso, a declaração de tipo de conteúdo *não deve* ser definida. Para instâncias em que a carga útil é um JWT em si (um JWT aninhado), essa declaração *deve* estar presente e conter o valor JWT. Isso informa à implementação que o processamento adicional do JWT aninhado é necessário. JWTs aninhados são raros, então a declaração cty raramente está presente nos cabeçalhos.

Portanto, para JWTs não criptografados, o cabeçalho é simplesmente:

⁴<https://jwt.io>

⁵<http://www.iana.org/assignments/media-types/media-types.xhtml>

```
{
  "alg": "nenhum"
}
```

que é codificado para:

eyJhbGciOiJub25lIn0

É possível adicionar declarações adicionais definidas pelo usuário ao cabeçalho. Isso geralmente é de uso limitado, a menos que determinados metadados específicos do usuário sejam necessários no caso de JWTs criptografados antes da descriptografia.

3.2 A carga útil

```
{
  "sub": "1234567890",
  "nome": "John Doe",
  "admin": verdadeiro
}
```

O payload é o elemento onde geralmente são adicionados todos os dados interessantes do usuário. Além disso, certas reivindicações definidas na especificação também podem estar presentes. Assim como o cabeçalho, a carga útil é um objeto JSON. Nenhuma reivindicação é obrigatória, embora reivindicações específicas tenham um significado definido. A especificação JWT especifica que declarações que não são compreendidas por uma implementação devem ser ignoradas. As reivindicações com significados específicos associados a elas são conhecidas como *reivindicações registradas*.

3.2.1 Reivindicações Registradas

- **iss**: da palavra *emissor*. Uma string ou URI que diferencia maiúsculas de minúsculas que identifica exclusivamente a parte que emitiu o JWT. Sua interpretação é específica da aplicação (não há autoridade central gerenciando os emissores).
- **sub**: da palavra *assunto*. Uma string ou URI que diferencia maiúsculas de minúsculas que identifica exclusivamente a parte sobre a qual este JWT carrega informações. Em outras palavras, as reivindicações contidas neste JWT são declarações sobre esta parte. A especificação JWT especifica que essa declaração deve ser exclusiva no contexto do emissor ou, nos casos em que isso não for possível, globalmente exclusiva. O tratamento desta reclamação é específico da aplicação.
- **aud**: da palavra *audiência*. Uma única string ou URI com distinção entre maiúsculas e minúsculas ou uma matriz de tais valores que identificam exclusivamente os destinatários pretendidos deste JWT. Ou seja, quando esta reclamação estiver presente, a parte que ler os dados deste JWT deve se encontrar na reclamação *aud* ou desconsiderar os dados contidos no JWT. Como no caso das reivindicações *iss* e *sub*, esta reivindicação é específica da aplicação.
- **exp**: da palavra *expiração (tempo)*. Um número que representa uma data e hora específicas no formato “seconds since epoch” conforme definido por POSIX6 . Esta reivindicação define o momento exato de

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

qual este JWT é considerado *inválido*. Algumas implementações podem permitir um certo skew entre os clocks (por considerar este JWT válido por alguns minutos após a data de expiração).

- **nbf:** de *não antes* (tempo). O oposto da reivindicação *exp*. Um número que representa uma data e hora específicas no formato “seconds since epoch” conforme definido por POSIX⁷. Esta reivindicação define o momento exato a partir do qual este JWT é considerado *válido*. A hora e a data atuais devem ser iguais ou posteriores a essa data e hora. Algumas implementações podem permitir uma certa inclinação.
- **iat:** de *emitido em* (tempo). Um número que representa uma data e hora específicas (no mesmo formato como *exp* e *nbf*) no qual este JWT foi emitido.
- **jti:** do *JWT ID*. Uma string que representa um identificador exclusivo para este JWT. Essa declaração pode ser usada para diferenciar JWTs de outros conteúdos semelhantes (evitando replays, por exemplo). Cabe à implementação garantir a exclusividade.

Como você deve ter notado, todos os nomes são curtos. Isso atende a um dos requisitos de design: manter os JWTs os menores possíveis.

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 Créditos Públicos e Privados

Todas as reivindicações que não fazem parte da seção de *reivindicações registradas* são reivindicações **públicas** ou **privadas**.

- Declarações **privadas** : são aquelas definidas pelos *usuários* (consumidores e produtores) dos JWTs. Em outras palavras, essas são afirmações ad hoc usadas para um caso particular. Como tal, deve-se ter cuidado para evitar colisões.
- Declarações **públicas** : são declarações *registradas* no registro IANA JSON Web Token Claims⁸ (um registro no qual os usuários podem registrar suas declarações e, assim, evitar colisões) ou nomeadas usando um nome resistente a colisões (por exemplo, anexando um namespace ao seu nome).

Na prática, a maioria das reivindicações são reivindicações registradas ou reivindicações privadas. Em geral, a maioria dos JWTs são emitidos com uma finalidade específica e um conjunto claro de possíveis usuários em mente. Isso simplifica a escolha de nomes resistentes a colisões.

Assim como nas regras de análise JSON, as declarações duplicadas (chaves JSON duplicadas) são tratadas mantendo apenas a última ocorrência como válida. A especificação JWT também permite que as implementações considerem JWTs com declarações duplicadas como *inválidos*. Na prática, se você não tiver certeza sobre a implementação que irá lidar com seus JWTs, tome cuidado para evitar declarações duplicadas.

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

3.3 JWTs não seguros

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

```
{
  "alg": "nenhum"
}
```

e uma carga útil definida pelo usuário. Por exemplo:

```
{
  "sub": "user123",
  "session": "ch72gsb320000udocl363eofy", "name":
  "Nome Bonito", "última página": "/views/settings"
}
```

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

```
eyJhbGciOiJub25lIn0.
eyJzdWIiOiJ1c2VyMTIzIiwi2Vzc2lvbiI6ImNoNzJnc2IzMjAwMDB1ZG9jbDM2M
2VvZnkiLCJuYW1lIjoiUHJldHR5IE5hbWUiLCJsYXN0cGFnZSI6Ii92aWV3cy9zZXR0aW5ncyJ9.
```

Um JWT não seguro como o mostrado acima pode ser adequado para uso do lado do cliente. Por exemplo, se o ID da sessão for um número difícil de adivinhar e o restante dos dados for usado apenas pelo cliente para construir uma exibição, o uso de uma assinatura é supérfluo. Esses dados podem ser usados por um aplicativo da Web de página única para construir uma exibição com o nome “bonito” para o usuário sem atingir o back-end enquanto ele é redirecionado para a última página visitada. Mesmo que um usuário mal-intencionado modifique esses dados, ele não ganhará nada.

Observe o ponto final (.) na representação compacta. Como não há assinatura, é simplesmente uma string vazia. O ponto ainda é adicionado, no entanto.

Na prática, no entanto, JWTs não garantidos são raros.

3.4 Criando um JWT não seguro

Para chegar à representação compacta das versões JSON do cabeçalho e da carga útil, execute as seguintes etapas:

1. Considere o cabeçalho como uma matriz de bytes de sua representação UTF-8. A especificação JWT *não* exige que o JSON seja minificado ou removido de caracteres sem sentido (como espaço em branco) antes da codificação.
2. Codifique a matriz de bytes usando o algoritmo Base64-URL, removendo os sinais de igual (=).
3. Considere a carga útil como uma matriz de bytes de sua representação UTF-8. A especificação JWT *não* exige que o JSON seja minificado ou removido de caracteres sem sentido (como espaço em branco) antes da codificação.

4. Codifique a matriz de bytes usando o algoritmo Base64-URL, removendo os sinais de igual (=).
5. Concatene as strings resultantes, colocando primeiro o cabeçalho, seguido de um “.” personagem, seguido pela carga útil.

A validação do cabeçalho e da carga útil (com relação à presença de declarações necessárias e ao uso correto de cada declaração) deve ser realizada antes da codificação.

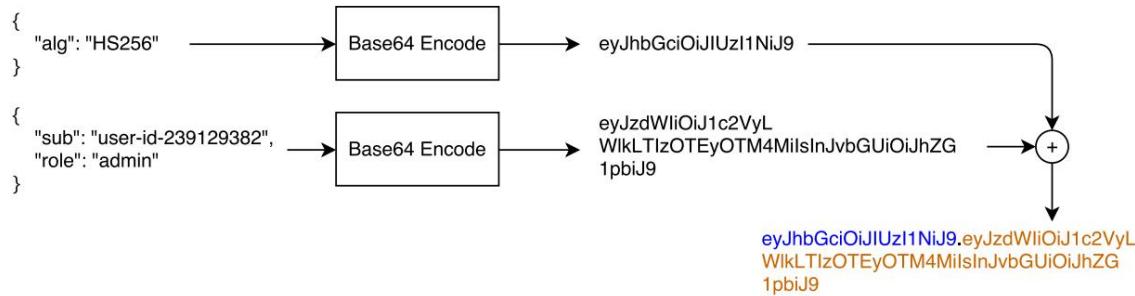


Figura 3.1: Geração JWT compacta e não segura

3.4.1 Exemplo de código

```

// variante segura de URL da função
Base64 b64(str) { return new
  Buffer(str).toString('base64') .replace(/=/g, '') .replace(/\+/g, '-') .replace(/\//g, '_');
}

função codificar(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 o acompanha.

3.5 Analisando um JWT não seguro

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

1. Encontre o primeiro ponto “.” personagem. Pegue a corda antes dela (sem incluí-la).

2. Decodifique a string usando o algoritmo Base64-URL. O resultado é o cabeçalho JWT.
3. Pegue a corda após o ponto da etapa 1.
4. Decodifique a string usando o algoritmo Base64-URL. O resultado é a carga JWT.

As strings JSON resultantes podem ser “embelezadas” adicionando espaços em branco conforme necessário.

3.5.1 Exemplo de Código

```
function decode(jwt) { const  
[headerB64, payloadB64] = jwt.split('.'); // Isso também suporta  
a análise da variante segura de URL de Base64. const headerStr = new Buffer(headerB64,  
'base64').toString(); const payloadStr = new Buffer(payloadB64, 'base64').toString();  
return { cabeçalho: JSON.parse(headerStr), payload: JSON.parse(payloadStr)  
  
};  
}
```

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

Capítulo 4

Assinaturas Web JSON

As assinaturas da Web JSON são provavelmente o recurso mais útil dos JWTs. Ao combinar 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 propósito de uma assinatura é permitir que uma ou mais partes estabeleçam a *autenticidade* do JWT. Autenticidade neste contexto significa que os dados contidos no JWT não foram adulterados. Em outras palavras, qualquer parte que possa realizar uma *verificação de assinatura* pode contar com o conteúdo fornecido pelo JWT. É importante ressaltar que a assinatura não impede que terceiros *leiam* o conteúdo do JWT. É para isso que a criptografia serve, e falaremos sobre isso mais adiante no [capítulo 5](#).

O processo de verificação da assinatura de um JWT é conhecido como *validação* ou *validação* de um token. Um token é considerado válido quando todas as restrições especificadas em seu cabeçalho e carga útil são atendidas.

Este é um aspecto *muito importante* dos JWTs: as implementações são necessárias para verificar um JWT até o ponto especificado por seu cabeçalho e sua carga útil (e, adicionalmente, o que o usuário exigir).

Portanto, um JWT pode ser considerado válido *mesmo que não tenha uma assinatura* (se o cabeçalho tiver a declaração *alg* definida como nenhuma). Além disso, mesmo que um JWT tenha uma assinatura válida, ele pode ser considerado inválido por outros motivos (por exemplo, pode ter expirado, de acordo com a reivindicação *exp*). Um ataque comum contra JWTs assinados depende da remoção da assinatura e da alteração do cabeçalho para torná-lo um JWT não seguro. É responsabilidade do usuário garantir que os JWTs sejam validados de acordo com seus próprios requisitos.

JWTs assinados são definidos na especificação JSON Web Signature, RFC 7515¹.

4.1 Estrutura de um JWT assinado

Cobrimos a estrutura de um JWT no [capítulo 3](#). Vamos revisá-lo aqui e dar atenção especial ao seu componente de assinatura.

¹<https://tools.ietf.org/html/rfc7515>

Um JWT assinado é composto por três elementos: o cabeçalho, a carga útil e a assinatura (novas linhas inseridas para facilitar a leitura):

```
eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9.  
eyJzdWliOilxMjM0NTY3ODkwiwibmFtZSI6IkpvG4gRG9IiwiYWRtaW4iOnRydWV9.  
TJVA95OrM7E2cBab30RMHrHDcEfijoYZgeFONFh7HgQ
```

O processo de decodificação dos dois primeiros elementos (o cabeçalho e o payload) é idêntico ao caso de JWTs não protegidos. O algoritmo e o código de amostra podem ser encontrados no final do [capítulo 3](#).

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

{
  "sub": "1234567890",
  "nome": "John Doe",
  "admin": verdadeiro
}
```

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

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

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

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

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

JWA é a especificação JSON Web Algorithms, RFC 75182 .

Esses algoritmos serão explicados em detalhes no [capítulo 7](#). Neste capítulo, vamos nos concentrar 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.

Estes são, essencialmente, variações dos três principais algoritmos requeridos e recomendados. O significado dessas siglas ficará mais claro no [capítulo 7](#).

²<https://tools.ietf.org/html/rfc7518>

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 UTF-8 codificação.
- **JSON.stringify**: uma função que pega um objeto JavaScript e o serializa na forma de string (JSON).
- **sha256**: uma função que recebe um array de octetos e retorna um novo array de octetos usando o Algoritmo SHA-256.
- **hmac**: uma função que recebe uma função SHA, um array de octetos e um segredo e retorna uma nova matriz de octetos usando o algoritmo HMAC.
- **rsassa**: uma função que recebe 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)));
assinatura const = 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)));
assinatura const = base64(rsassa(` ${encodedHeader}.${encodedPayload}`,
privateKey, sha256));
const jwt = `${encodedHeader}.${encodedPayload}.${signature}`;
```

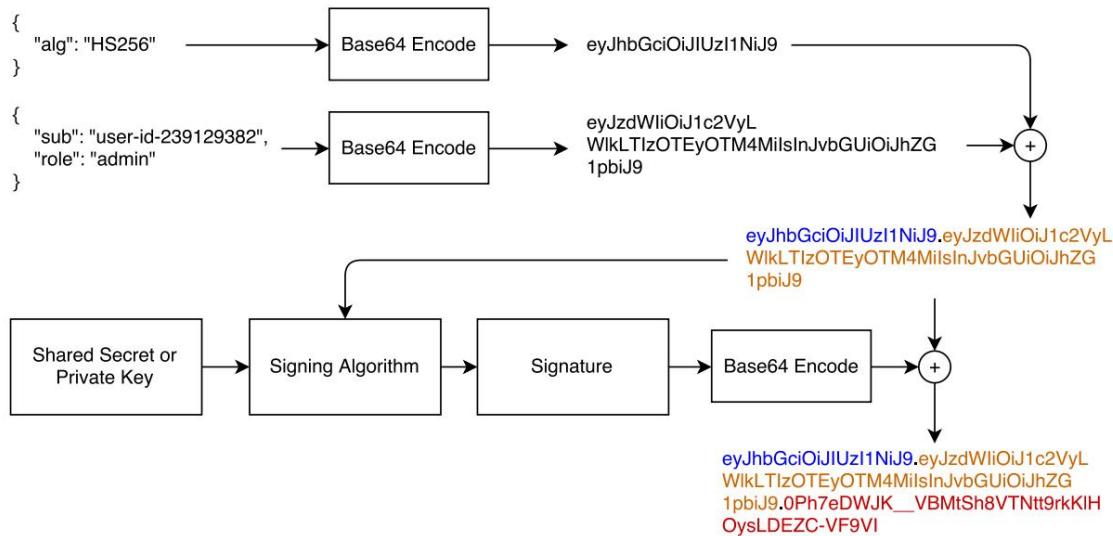


Figura 4.1: serialização compacta do JWS

Os detalhes completos desses algoritmos são mostrados no [capítulo 7](#).

4.1.2 Aspectos práticos dos algoritmos de assinatura

Todos os algoritmos de assinatura realizam a mesma coisa: eles fornecem uma maneira de estabelecer a autenticidade dos dados contidos no JWT. Como eles fazem isso varia.

Keyed-Hash Message Authentication Code (HMAC) é um algoritmo que combina uma determinada carga útil com um *segredo* usando uma função hash criptográfica³. O resultado é um código que pode ser usado para verificar uma mensagem *somente* se ambas as partes geradora e verificadora souberem o segredo. Em outras palavras, **os HMACs permitem que as mensagens sejam verificadas por meio de segredos compartilhados**.

A função hash criptográfica usada no HS256, o algoritmo de assinatura mais comum para JWTs, é SHA-256. O SHA-256 é explicado em detalhes no [capítulo 7](#). As funções 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 da saída da função. Desta forma, as funções 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 RSA4 (explicado no [capítulo 7](#)) adaptado para assinaturas. RSA é um algoritmo de chave pública. Algoritmos de chave pública geram chaves divididas: uma chave pública e uma privada

³https://en.wikipedia.org/wiki/Cryptographic_hash_function

⁴https://en.wikipedia.org/wiki/RSA_%28cryptosystem%29

chave. 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, ao contrário, só pode ser usada para verificar a autenticidade de uma mensagem. Assim, este esquema permite a distribuição segura de uma mensagem **um-para-muitos**. As partes receptoras 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 HMAC. Com HMAC + SHA-256, qualquer parte que pode verificar uma mensagem também pode *criar novas mensagens*. Por exemplo, se um usuário legítimo se tornasse malicioso, ele ou ela poderia modificar as mensagens sem que as outras partes percebessem. Com um esquema de chave pública, um usuário que se tornasse malicioso teria apenas a chave pública em sua posse e, portanto, não poderia criar novas mensagens assinadas com ela.

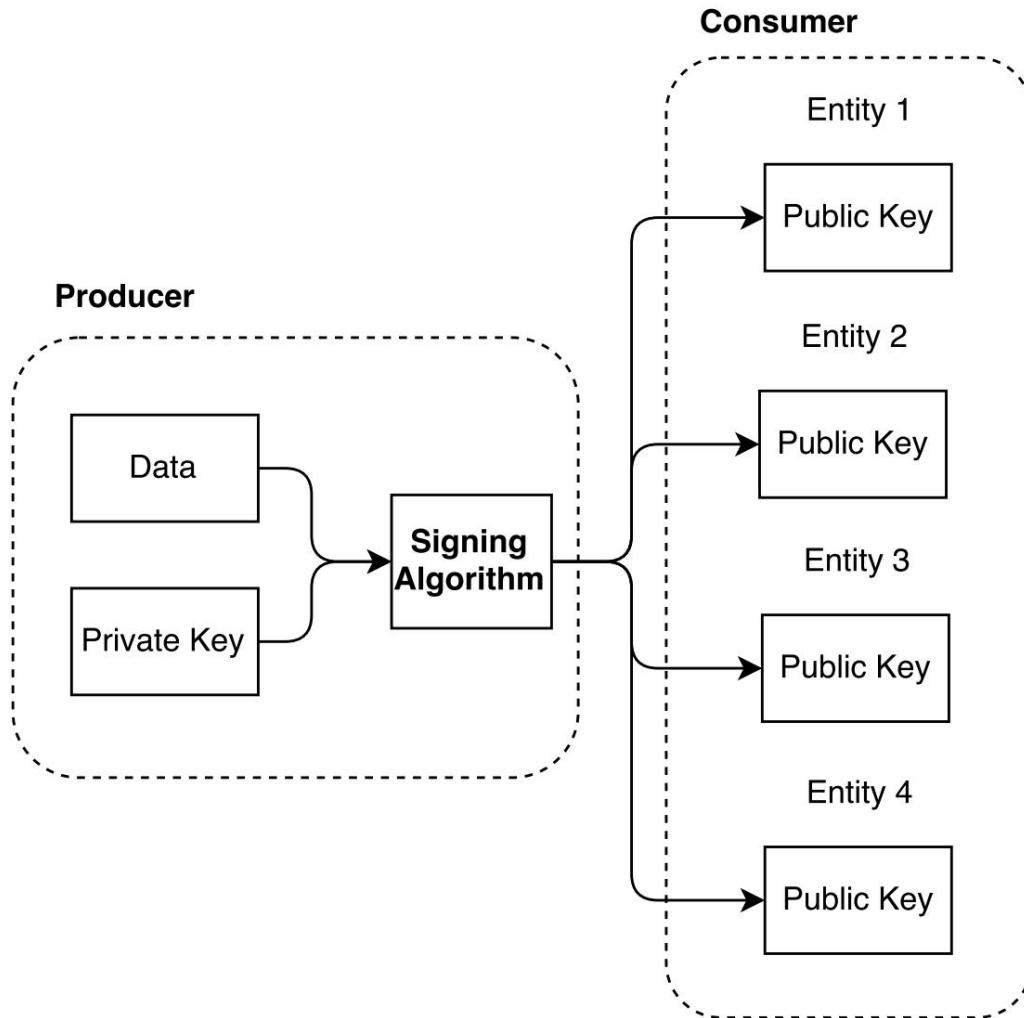


Figura 4.2: Assinatura um-para-muitos

A criptografia de chave pública⁵ 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 **muitos-para-um** seja construído. Essa variação é usada para JWTs criptografados, que são discutidos em

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

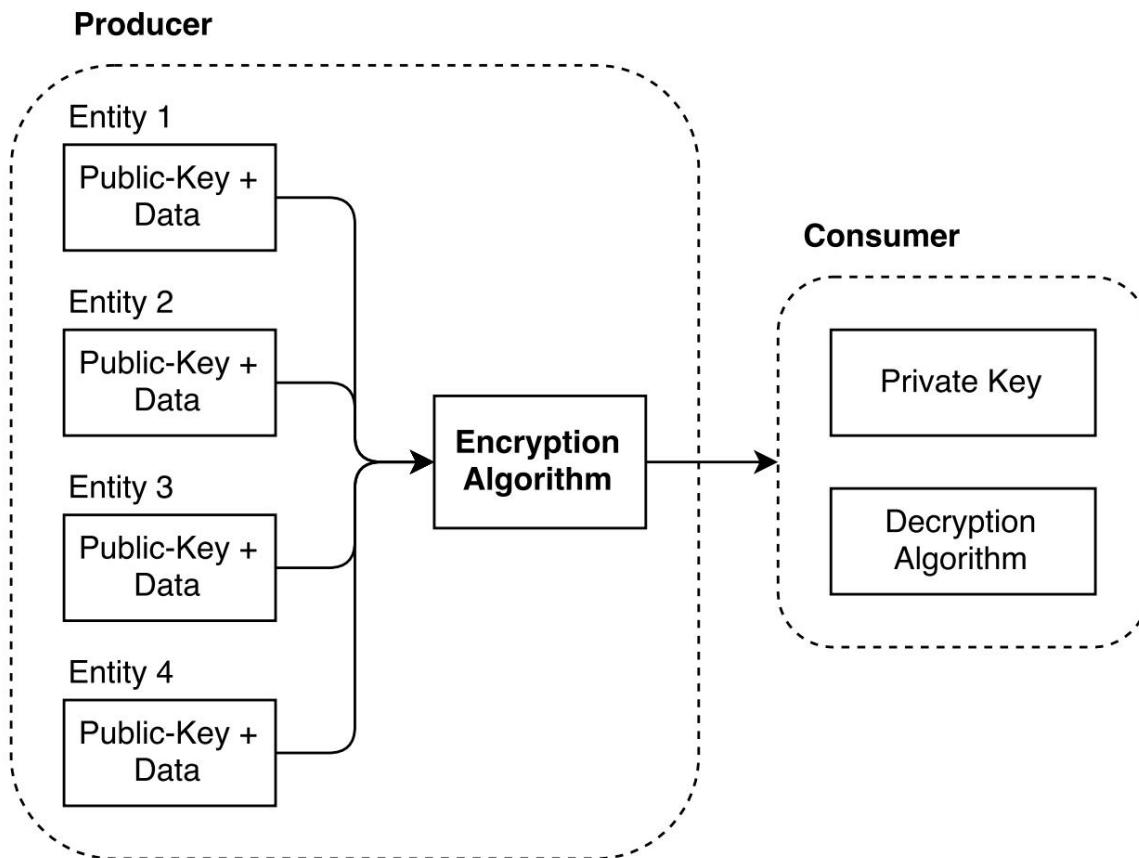


Figura 4.3: Criptografia de muitos para um

O algoritmo de assinatura digital de curva elíptica (ECDSA)⁶ é uma alternativa ao RSA. Esse algoritmo também gera um par de chaves pública e privada, mas a matemática por trás dele é diferente. Essa diferença permite requisitos de hardware menores do que o RSA para garantias de segurança semelhantes.

Estudaremos esses algoritmos com mais detalhes no [capítulo 7](#).

⁵https://en.wikipedia.org/wiki/Public-key_cryptography

⁶https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm

4.1.3 Reivindicações de cabeçalho JWS

O JWS permite casos de uso especiais que forcão o cabeçalho a carregar mais declarações. Por exemplo, para algoritmos de assinatura de chave pública, é possível incorporar a URL à chave pública como uma declaração. A seguir está a lista de declarações de cabeçalho registradas disponíveis para tokens JWS. Todas essas declarações são *adicionais* àquelas disponíveis para **JWTs não garantidos** e são opcionais, dependendo de como o JWT assinado deve ser usado.

- **jku**: URL de configuração da chave da Web JSON (JWK). Um URI apontando para um conjunto de chaves públicas codificadas em JSON usadas para assinar este JWT. A segurança de 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**: Chave da Web JSON. A chave usada para assinar este JWT no formato JSON Web Key (consulte o [capítulo 6](#)).
- **kid**: ID da chave. Uma string definida pelo usuário que representa uma única chave usada para assinar este JWT. Essa declaração é usada para sinalizar alterações de assinatura de chave para destinatários (quando várias chaves são usadas).
- **x5u**: URL X.509. Um URI apontando para um conjunto de certificados públicos X.509 (um padrão de formato de certificado) codificados no formato PEM. O primeiro certificado no conjunto deve ser aquele usado para assinar este JWT. Cada um dos certificados subseqüentes assina o anterior, completando assim a cadeia de certificados. X.509 é definido no RFC 52807 . A segurança de transporte é necessária para transferir os certificados.
- **x5c**: Cadeia de certificados X.509. Uma matriz JSON de certificados X.509 usados para assinar este JWS. Cada certificado deve ser o valor codificado em Base64 de sua representação DER PKIX. O primeiro certificado na matriz deve ser aquele 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 código X.509 DER certificado usado para assinar este JWT.
- **x5t#S256**: Idêntico a **x5t**, mas usa SHA-256 em vez de SHA-1. • **typ**: Idêntico ao valor **tip**
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 portadores de cabeçalho JOSE semelhantes são misturados com esse JWT em um único contêiner.
- **crit**: de *crítico*. Uma matriz de strings com os nomes das declarações que estão presentes nesse mesmo cabeçalho usado como extensões definidas pela implementação que devem ser tratadas pelos analisadores desse JWT. Ele deve conter os nomes das declarações 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. Essa representação permite várias assinaturas no mesmo JWT assinado. É conhecido como *serialização JWS JSON*.

⁷<https://tools.ietf.org/html/rfc5280>

No formulário JWS JSON Serialization, JWTs assinados são representados como texto imprimível com formato JSON (ou seja, o que você obteria ao chamar `JSON.stringify` em um navegador). É necessário um objeto JSON superior que carregue os seguintes pares chave-valor:

- **payload**: uma string codificada em Base64 do objeto de carga JWT real.
- **assinaturas**: uma matriz de objetos JSON que carregam as assinaturas. Esses objetos são definidos abaixo.

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

- **protected**: uma string codificada em Base64 do cabeçalho JWS. As reivindicações contidas neste cabeçalho são protegidas pela assinatura. Este cabeçalho é necessário apenas se não houver cabeçalhos desprotegidos.
Se houver cabeçalhos desprotegidos, esse cabeçalho pode ou não estar presente.
- **cabeçalho**: um objeto JSON contendo declarações de cabeçalho. Este cabeçalho está desprotegido pela assinatura.
Se nenhum cabeçalho protegido estiver presente, esse elemento será obrigatório. Se um cabeçalho protegido estiver presente, esse elemento será opcional.
- **assinatura**: Uma string codificada em Base64 da assinatura JWS.

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

Quando os cabeçalhos protegidos e desprotegidos estão presentes, o cabeçalho JOSE real é construído a partir da união dos elementos em ambos os cabeçalhos. Nenhuma reivindicação duplicada pode estar presente.

O exemplo a seguir foi retirado do JWS RFC8 :

```
{
  "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAIoJezMDA4MTkzODAsDQogIm
  h0dHA6Ly9leGFtcGxLmNvbS9pc19yb290Ijp0cnVlfQ", "assinuras": [ {
    "protected": "eyJhbGciOiJSUzI1NiJ9", "header":
    { "kid": "2010-12-29" }, "signature":

    "cC4hiUPoj9Eetdgtv3hF80EGrhB__dzERat0XF9g2VtQgr9Pjbu3XOiZj5RZmh7AA
    uHlm4Bh-0Qc_lF5YKt_O8W2Fp5ujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAyn
    RFdiuB--f_nZLgrnbyTyWzO5vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB
    _eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6Bfl7noOPqvhJ1phCnvWh6
    leYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLihWkWywlVmtVrBp0igcN_loypGIU
    PQGe77Rw"
  },
  {
    "protegido": "eyJhbGciOiJFUzI1NiJ9", "cabeçalho":
    { "criança": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
  }
}
```

⁸<https://tools.ietf.org/html/rfc7515#appendix-A.6>

```

    "assinatura": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDx
                  w5dJxLa8ISISApmWQxfKTUJqPP3-Kg6NU1Q"
}
]
}

```

Este exemplo codifica duas assinaturas para a mesma carga útil: uma assinatura RS256 e uma assinatura ES256.

4.1.4.1 Serialização JWS JSON simplificada

A serialização JWS JSON define um formulário simplificado para JWTs com apenas uma única assinatura. Essa forma é conhecida como *serialização JWS JSON simplificada*. A serialização nivelada remove a matriz de *assinaturas* e coloca os elementos de uma única assinatura no mesmo nível do elemento de *carga útil*.

Por exemplo, removendo uma das assinaturas do exemplo anterior, um objeto de serialização JSON nivelado seria:

```
{
  "carga útil": "eyJpc3MiOiJqb2UiLA0KICJleHAIoJezMDA4MTkzODAsDQog
                 Imh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ", "protected": {
    "eyJhbGciOiJFUzI1NiJ9", "header": { "kid": "e9bc097a-ce51-4036-9562-d2ade882db0d" },
    "signature": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFC
                  gfTjDxw5dJxLa8ISISApmWQxfKTUJqPP3-Kg6NU1Q"
  }
}
```

4.2 Assinatura e Validação de Tokens

Os algoritmos usados para assinar e validar tokens são explicados em detalhes no [capítulo 7](#). O uso de JWTs assinados é simples o suficiente na prática para que você possa aplicar os conceitos explicados até agora para usá-los com eficácia. Além disso, existem boas bibliotecas que você pode usar para implementá-las convenientemente. Examinaremos os algoritmos necessários e recomendados usando as mais populares dessas bibliotecas para JavaScript. Exemplos de outras linguagens e bibliotecas populares podem ser encontrados no código que acompanha.

Os exemplos a seguir fazem uso da popular biblioteca JavaScript jsonwebtoken.

```
importar jwt de 'jsonwebtoken'; //var jwt = require('jsonwebtoken');

carga const = {
  sub: "1234567890", nome:
  "John Doe",
  administrador: verdadeiro
};
```

4.2.1 HS256: HMAC + SHA-256

As assinaturas HMAC requerem um segredo compartilhado. Qualquer string fará:

```
const segredo = 'meu-secreto';

const assinado = jwt.sign(payload, secret, {
    algoritmo: 'HS256',
    expiresIn: '5s' // se omitido, o token não expirará
});
```

Verificar o token é igualmente fácil:

```
const decodificado = jwt.verify(assinado, segredo, {
    // Nunca se esqueça de tornar isso explícito para evitar // algoritmos
    // de ataques de remoção de assinatura: ['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 criação, ele seria considerado inválido e uma exceção seria lançada.

4.2.2 RS256: RSASSA + SHA256

Assinar e verificar tokens assinados RS256 é igualmente fácil. A única diferença está no uso de um par de chaves públicas/privadas em vez de um segredo compartilhado. Há muitas maneiras de criar chaves RSA.

OpenSSL é uma das bibliotecas mais populares para criação e gerenciamento de chaves:

```
# Gera uma chave privada openssl
genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits:2048 # Deriva a chave pública 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 de origem JavaScript e passado para a biblioteca jsonwebtoken.

```
// Você pode obter isso de private_key.pem acima. const
privateRsaKey = `<SUA-PRIVATE-RSA-KEY>`;

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

// Você pode obter isso em public_key.pem acima. const
publicRsaKey = `<SUA-CHAVE-RSA-PÚBLICA>`;

const decodificado = jwt.verify(assinado, publicRsaKey, {
```

```
// Nunca se esqueça de tornar isso explícito para evitar // ataques de
remoção de assinatura. algoritmos: ['RS256'],
});

});
```

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

Os algoritmos ECDSA também fazem uso de chaves públicas. A matemática por trás do algoritmo é diferente, portanto, as etapas para gerar as chaves também são diferentes. O “P-256” no nome desse algoritmo nos diz exatamente qual versão do algoritmo usar (mais detalhes sobre isso no [capítulo 7](#)). Também podemos usar o OpenSSL para gerar a chave: *# Gere uma chave privada (prime256v1 é o nome dos parâmetros usados # para gerar a chave, é o mesmo que P-256 na especificação JWA)*. openssl ecpparam -name prime256v1 -genkey -noout -out ecdsa_private_key.pem *# Derive a chave pública 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. Esse é um dos benefícios do ECDSA sobre o RSA (mais sobre isso no [capítulo 7](#)). Os arquivos gerados também estão no formato PEM, portanto, basta colá-los em sua fonte.

```
// Você pode obter isso de private_key.pem acima. const
privateEcdsaKey = `<SUA-PRIVATE-ECDSA-KEY>`;

const assinado = jwt.sign(payload, privateEcdsaKey, { algoritmo: 'ES256',
    expiresIn: '5s'

});

// Você pode obter isso em public_key.pem acima. const
publicEcdsaKey = `<SUA-TECLA-ECDSA-PÚBLICA>`;

const decoded = jwt.verify(signed, publicEcdsaKey, { // Nunca se esqueça
    de tornar isso explícito para evitar // ataques de remoção de
    assinaturas. algoritmos: ['ES256'],
});
```

Consulte o [capítulo 2](#) para aplicações práticas desses algoritmos no contexto de JWTs.

capítulo 5

Criptografia da Web JSON (JWE)

Enquanto JSON Web Signature (JWS) fornece um meio para *validar* dados, JSON Web Encryption (JWE) fornece uma maneira de manter os dados *opacos* para terceiros. Opaco, neste caso, significa *ilegível*.

Tokens criptografados não podem ser inspecionados por terceiros. Isso permite um uso interessante adicional casos.

Embora pareça que a criptografia oferece as mesmas garantias que a validação, com o recurso adicional de tornar os dados ilegíveis, nem sempre é esse o caso. Para entender por que, 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 conheçam um segredo compartilhado. Cada parte que detém o segredo compartilhado pode **criptografar** e **descriptografar** as 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 os tokens, e as partes que possuem a chave pública podem apenas verificar esses tokens, no JWE a parte que detém a chave privada é a única 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 que possuem a chave pública podem introduzir novos dados em uma troca. Em contraste, no JWS, as partes que possuem a chave pública podem apenas *verificar* os dados, mas não introduzir novos dados. Em termos diretos, o JWE não oferece 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-lo. O produtor detém a chave privada e os consumidores detêm a chave pública. Os dados só podem fluir dos detentores de chaves privadas para os 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. Nesse caso, os dados só podem fluir dos detentores de chaves públicas para os detentores de chaves privadas - os detentores de chaves públicas são os produtores e os detentores de chaves privadas são os consumidores:

JWS	JWE
Chave pública do produtor	Chave privada
Chave pública do consumidor	Chave privada

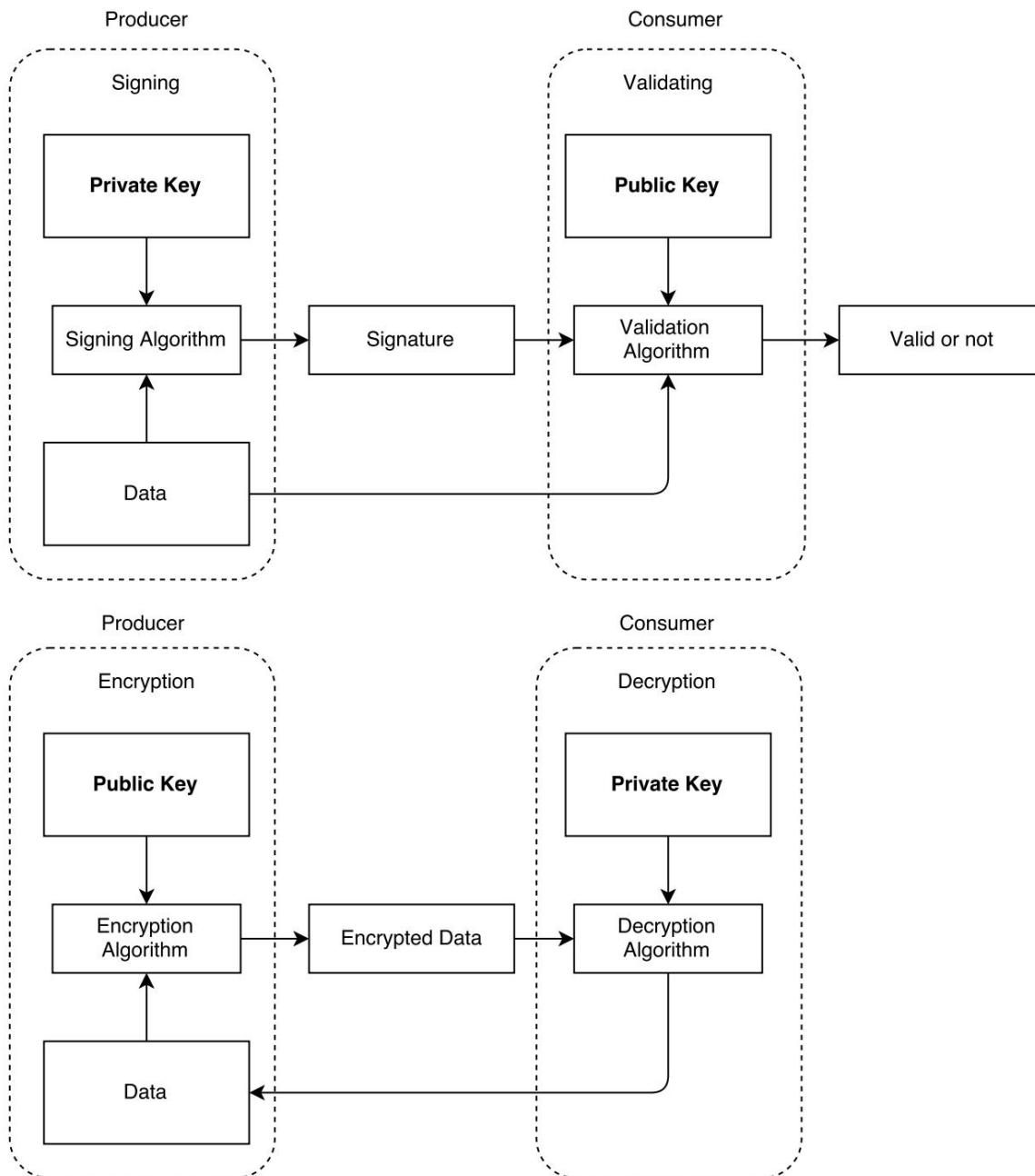


Figura 5.1: Assinatura versus criptografia usando criptografia de chave pública

Neste ponto algumas pessoas podem perguntar:

No caso do JWE, não poderíamos distribuir a chave privada para todas as partes que desejam enviar dados para um consumidor? Assim, se um consumidor pode descriptografar os dados, ele pode ter certeza de que também é válido (porque não é possível alterar os 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 de segredo 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. Desta forma, você obtém os benefícios de ambos.

Observe que tudo isso se aplica a situações em que os consumidores são entidades diferentes dos produtores. Se o produtor for a mesma entidade que consome os dados, um JWT criptografado com segredo compartilhado oferece as mesmas garantias que um JWT criptografado e assinado.

Os JWTs criptografados por JWE, independentemente de terem ou não um JWT assinado aninhado neles, carregam uma marca de autenticação. Essa tag permite que JWE JWTs sejam validados. No entanto, devido aos problemas mencionados acima, esta assinatura não se aplica aos mesmos casos de uso das assinaturas JWS. O objetivo dessa tag é impedir ataques de padding oracle¹ 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 facilitar a leitura):

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
UGhiOguC7luEvf_NPVaXsGMoLOmwvc1GyqIIKOK1nN94nHPoltGRhWhw7Zx0-kFm1NJn8LE9XShH59_
i8J0PH5ZyNfGy2xGdULU7sHNf6Gp2vPLgNZ__deLkxGHZ7PcHALUzoOegEI-8E66jX2E4zyJKx
YxzZltRzC5hIRrb6Y5Cl_p-ko3YvklysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWl5ng8Otv
zIV7elprCbuPhcCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP cFPgwCp6X-nZZd9OHBy-
B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGljb3RoZQ.
KDITtXchhZTGufMYmOYGS4HffxPSUrfmqCHXal9wOGY.
9hH0vgRFYgPnAHOd8stkvw
```

Embora possa ser difícil de ver no exemplo acima, JWE Compact Serialization tem cinco elementos.

Como no caso do JWS, esses elementos são separados por pontos e os dados contidos neles são codificados em Base64.

Os cinco elementos da representação compacta são, em ordem:

1. **O cabeçalho protegido:** um cabeçalho análogo ao cabeçalho JWS.
2. **A chave criptografada:** uma chave simétrica usada para criptografar o texto cifrado e outros dados criptografados. Essa chave é derivada da chave de criptografia real especificada pelo usuário e, portanto, é criptografada por ela.

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

3. **O vetor de inicialização:** alguns algoritmos de criptografia requerem dados adicionais (geralmente aleatórios) dados.
4. **Os dados criptografados (texto cifrado):** os dados reais que estão sendo criptografados.
5. **A etiqueta de autenticação:** dados adicionais produzidos pelos algoritmos que podem ser usados para validar o conteúdo do texto cifrado contra adulteração.

Como no caso de JWS e assinaturas únicas na serialização compacta, o JWE suporta uma única chave de criptografia em seu formato compacto.

Usar uma chave simétrica para executar o processo de criptografia real é uma prática comum ao usar criptografia assimétrica (criptografia de chave pública/privada). Os algoritmos de criptografia assimétrica geralmente são de alta complexidade computacional e, portanto, a criptografia de longas sequências de dados (o texto cifrado) não é 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, em seguida, 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 transmitidos a eles. Se o texto cifrado for modificado (mesmo sem ser descriptografado), os algoritmos podem processá-lo mesmo assim.

A etiqueta de autenticação pode ser usada para evitar isso, atuando essencialmente como uma assinatura. No entanto, isso 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 chave:

- **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:** curva elíptica Diffie-Hellman efêmera Acordo de chave estática usando concat KDF e variantes pré-empacotando a chave com qualquer uma das variantes não GCM AES acima.
- **Variantes PKCS #5:** PBES2 (criptografia baseada em senha) + HMAC (SHA-256 a 512) + variantes não GCM AES 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 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** •
- Curva Elíptica Diffie-Hellman Ephemeral Static (ECDH-ES)** usando Concat KDF (marcado para se tornar obrigatório no futuro)

- **Envoltório de chave ECDH-ES + AES-128**

- **Envoltório de chave ECDH-ES + AES-256**

Alguns desses algoritmos requerem parâmetros de cabeçalho 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 a carga útil é determinada. Em particular, a especificação JWE descreve estes modos de gerenciamento de chaves:

- **Envolvimento de chave:** a chave de criptografia de conteúdo (CEK) é criptografada para o destinatário pretendido usando um algoritmo de criptografia *simétrica*.

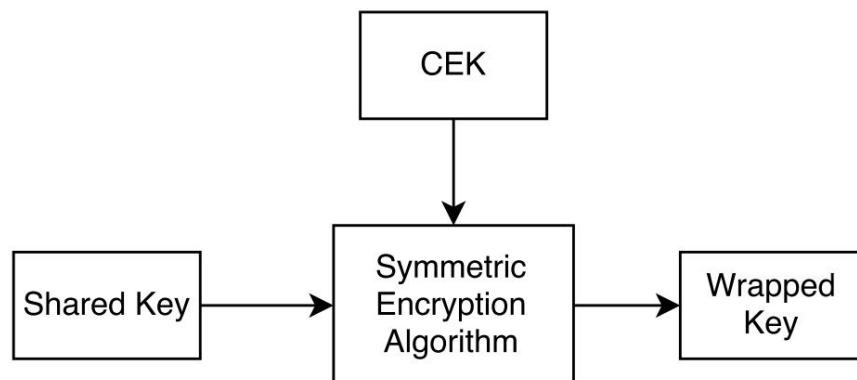


Figura 5.2: Envolvimento de chaves

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

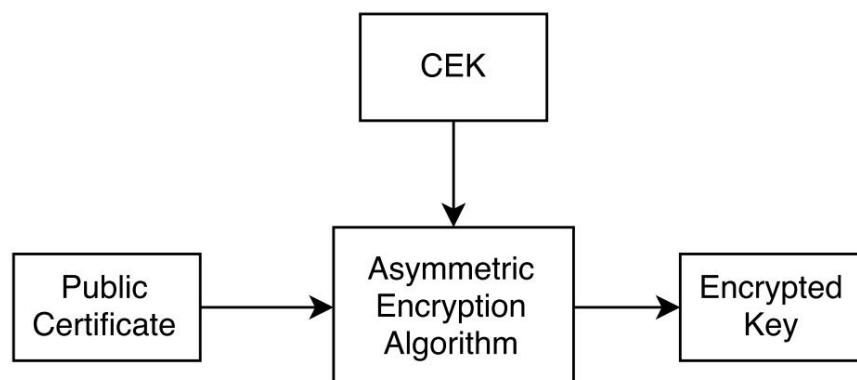


Figura 5.3: Criptografia de chave

- **Acordo de chave direta:** um algoritmo de acordo de chave é usado para escolher o CEK.

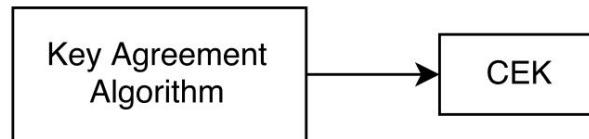


Figura 5.4: Contrato de chave direta

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

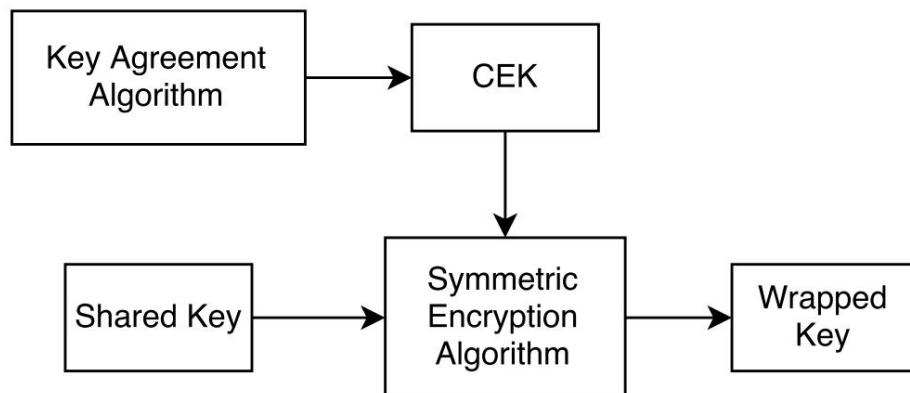


Figura 5.5: Contrato de chave direta

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

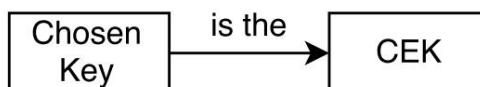


Figura 5.6: Contrato de chave direta

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 a chave de criptografia CEK e JWE.

A CEK é a chave real usada para criptografar a carga útil: um algoritmo de criptografia usa a CEK e o texto sem formatação para produzir o texto cifrado. Por outro lado, a chave de criptografia JWE é a forma criptografada da CEK ou uma sequência de octetos vazia (conforme exigido pelo algoritmo escolhido). Um

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 o CEK real (contrato de chave direta).

5.1.2 Algoritmos de criptografia de conteúdo

A seguir estão os algoritmos de criptografia de conteúdo, ou seja, aqueles usados para realmente criptografar a carga útil:

- **AES CBC + HMAC SHA:** AES 128 a 256 bits com Cipher Block Chaining e HMAC + SHA-256 a 512 para validação.
- **AES GCM:** AES 128 a 256 usando o modo Galois Counter.

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 Cabeçalho

Assim como o cabeçalho para JWS e JWTs não seguros, o cabeçalho carrega todas as informações necessárias para que o JWT seja processado corretamente pelas bibliotecas. A especificação JWE adapta os significados das declarações registradas definidas no JWS para seu próprio uso e adiciona algumas declarações próprias. Estas são as reivindicações novas e modificadas:

- **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 o CEK.
- **zip:** um algoritmo de compactação a ser aplicado aos dados criptografados antes da criptografia. Este parâmetro é opcional. Quando está ausente, nenhuma compressão é executada. Um valor usual para isso é DEF, o algoritmo comum deflate2 .
- **jku:** idêntico ao JWS, exceto que neste caso a reivindicação aponta para a chave pública usada para criptografar o CEK.
- **jkw:** idêntico ao JWS, exceto que neste caso a reivindicação aponta para a chave pública usada para criptografar o CEK.
- **kid:** idêntico ao JWS, exceto que neste caso a reivindicação aponta para a chave pública usada para criptografar o CEK.
- **x5u:** idêntico ao JWS, exceto que neste caso a reivindicação aponta para a chave pública usada para criptografar o CEK.
- **x5c:** idêntico ao JWS, exceto que neste caso a reivindicação aponta para a chave pública usada para criptografar o CEK.

²<https://tools.ietf.org/html/rfc1951>

- **x5t**: idêntico ao JWS, exceto que neste caso a reivindicação aponta para a chave pública usada para criptografar o CEK.
- **x5t#S256**: idêntico ao JWS, exceto que neste caso a reivindicação aponta para a chave pública usada para criptografar a CEK.
- **tipo**: 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 cabeçalho.

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, JWE Compact Serialization foi mencionado brevemente. É composto basicamente por cinco elementos codificados em forma de texto imprimível e separados por pontos (.). O algoritmo básico para construir uma serialização compacta JWE JWT é:

1. Se exigido pelo algoritmo escolhido (reivindicação alg), gere um número *aleatório* do tamanho necessário. É essencial cumprir certos requisitos criptográficos de aleatoriedade ao gerar esse valor. Consulte RFC 40863 ou use um gerador de números aleatórios criptograficamente validado.
2. Determine a chave de criptografia de conteúdo de acordo com o modo de gerenciamento de chaves⁴ :
 - Para **Acordo de chave direta**: 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 agrupamento de chave**: use o algoritmo de acordo de chave com o número aleatório para calcular a chave que será usada para agrupar o CEK.
 - Para **criptografia direta**: a CEK é a chave simétrica.
3. Determine a chave criptografada JWE de acordo com o modo de gerenciamento de chave:
 - Para **Contrato de Chave Direta e Criptografia Direta**: a Chave Criptografada JWE é vazio.
 - Para **encapsulamento de chave, criptografia de chave e acordo de chave com encapsulamento de chave**: criptografar a CEK para o destinatário. O resultado é a chave criptografada JWE.
4. Calcule um Vetor de Inicialização (IV) do tamanho requerido pelo algoritmo escolhido. Se não necessário, pule esta etapa.
5. Compace o texto sem formatação do conteúdo, se necessário (reivindicação de cabeçalho zip).
6. Criptografe os dados usando o CEK, o IV e os Dados Autenticados Adicionais (AAD).

O resultado é o conteúdo criptografado (texto cifrado JWE) e a etiqueta de autenticação. O AAD é usado apenas para serializações não compactas.
7. Construa a representação compacta como:

```
base64(cabeçalho) + '.' +
base64(encryptedKey) + '.' + // Etapas 2 e 3
```

³<https://tools.ietf.org/html/rfc4086>
45.1.1.1

```
base64(initializationVector) + '.' + // Etapa 4 base64(texto cifrado) +
'.' + // Passo 6 base64(authenticationTag)
// Passo 6
```

5.1.5 Serialização JWE JSON

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

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

- **protected**: objeto JSON codificado em Base64 do cabeçalho afirma ser protegido (validado, não criptografado) por este JWE JWT. Opcional. Pelo menos este elemento ou o cabeçalho desprotegido deve estar presente.
- **desprotegido**: declarações de cabeçalho que não são protegidas (validadas) como um objeto JSON (não codificado em Base64). Opcional. Pelo menos este elemento ou o cabeçalho protegido deve estar presente.
- **iv**: string Base64 do vetor de inicialização. Opcional (presente apenas 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, esse membro deverá estar ausente.
- **texto cifrado**: string codificada em Base64 dos dados criptografados.
- **tag**: string Base64 do tag de autenticação gerado pelo algoritmo de criptografia.
- **destinatários**: uma matriz JSON de objetos JSON, cada um contendo as informações necessárias para descriptografia por cada destinatário.

A seguir estão os membros dos objetos na matriz de destinatários:

- **cabeçalho**: um objeto JSON de declarações de cabeçalho desprotegidas.
- Opcional. • **cryptd_key** : chave criptografada JWE codificada em Base64. Presente apenas quando uma chave criptografada JWE En é usada.

O cabeçalho real usado para descriptografar um JWE JWT para um destinatário é construído a partir da união de cada cabeçalho presente. Não são permitidas reivindicações repetidas.

O formato das chaves criptografadas é descrito no [capítulo 6](#) (JSON Web Keys).

O exemplo a seguir foi retirado da RFC 7516 (JWE):

```
{
  "protégido": "eyJhbGciOiJBMTI4Q0JDLUhTMjU2In0",
  "desprotegido": {
    "jku": "https://server.example.com/keys.jwks"
  },
  "destinatários": [
    ...
  ]
}
```

```
{
  "header": { "alg": "RSA1_5", "kid": "2011-04-29" }, "encrypted_key": "UGhiOguC7IuEvf_NPVAXsGMoLOmwvc1GyqIIKOK1nN94nHPoltGRhWhw7Zx0-kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ_deLKxGHZ7PcHALUzoOegEl-8E66jX2E4zyJKx-YxzZlItRzC5hIRrb6Y5Cl_p-ko3YvkkyS2IFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPhcCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-cFPgwCp6X-nZZd9OHBv-B3mgSpc_Tb3mMRgSpc_Tb
},
{
  "header": { "alg": "A128KW", "kid": "7" }, "encrypted_key": "6KB707dM9YTlglHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ"
},
],
{
  "iv": "AxY8DCtDaGlzbGljb3RoZQ", "texto_cifrado": "KDITtXchhZTGufMYmOYGS4HffxPSUrfmqCHXal9wOGY", "tag": "Mz-VPPyU4RlcuYv1lwlvzw"
}
}
```

Este JWE JWT serializado em JSON transporta uma única carga útil para dois destinatários. O algoritmo de criptografia é AES-128 CBC + SHA-256, que você pode obter no cabeçalho protegido:

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

Ao realizar a união de todas as reivindicações para cada destinatário, o cabeçalho final de 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 JWE JSON simplificada

Assim como o JWS, o JWE define uma serialização JSON *simple*. Este formulário de serialização só pode ser usado para um único destinatário. Nesta forma, a matriz de destinatários é substituída por um cabeçalho e um par ou elementos criptografados_chave (isto é, as chaves de um único objeto da matriz de destinatários tomam seu lugar).

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

```
{
  "protegido": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",
  "desprotegido": { "jku": "https://server.example.com/keys.jwks" }, "header": {
    "alg": "RSA1_5", "kid": "2011-04-29" }, "encrypted_key": {

      "UGhIOguC7luEvf_NPVAXsGMoLOmwvc1GyqllKOK1nN94nHPoltGRhWhw7Zx0-
      kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPlgNZ_deLKx
      GHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-YxzZIltRzC5hlRirb6Y5Cl_p-ko3
      YvklysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWl5ng8OtvtzIV7elprCbuPh
      cCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-cFPg
      wCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A", "iv": "AxY8DCtDaGlzbGljb3RoZQ",
      "ciphertext": "KDITtXchhZTGufMYmOYGS4HffxPSUrfmqCHXal9wOGY", "tag": "Mz-
      VPPyU4RlcuYv1lwlvzw"
    }
}
```

5.2 Criptografia e Descriptografia de Tokens

Os exemplos a seguir mostram como executar a criptografia usando a popular biblioteca node-jose5. Essa biblioteca é um pouco mais complexa que o jsonwebtoken (usado para os exemplos do JWS), pois cobre muito mais terreno.

5.2.1 Introdução: Gerenciando Chaves com node-jose

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

Para criar um keystore vazio e adicionar algumas chaves de tipos diferentes:

```
// Cria um keystore vazio const
keystore = jose.JWK.createKeyStore();


---


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

```
// Gera algumas chaves. Você também pode importar chaves geradas de fontes // externas.
promessas const = [ keystore.generate('out', 128, { kid: 'example-1' }), keystore.generate('RSA',
2048, { kid: 'example-2' }), keystore.generate( 'EC', 'P-256', { garoto: 'exemplo-3' }),

];

```

Com o node-jose, a geração de chaves é uma questão bastante simples. Todos os tipos de chave utilizáveis com JWE e JWS são suportados. Neste exemplo, criamos três chaves diferentes: uma chave AES simples 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 chave pública/privada, a chave gerada é a chave *privada*. Para obter as chaves públicas, basta ligar para:

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

A chave pública será armazenada no formato JWK.

Também é possível importar chaves preexistentes:

```
// onde a entrada é uma: // *
// instância jose.JWK.Key // *
// representação do 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 exclusivo });

// onde input é a: // * Serialização
de string de um JSON JWK/(base64-encoded)
// PEM/(codificado em binário) DER // *
Buffer de um JSON JWK/(codificado em base64) PEM/(codificado em binário) DER // o
formulário é um: // * "json" para um JWK stringificado em JSON // * "pkcs8" para uma chave
privada PKCS8 codificada em DER (não criptografada!) // * "spki" para uma chave pública
SPKI codificada em DER // * "pkix" para um certificado PKIX X.509 codificado em DER // *
"x509" para um certificado PKIX X.509 codificado por DER // * "pem" para um PEM
codificado por PKCS8 / SPKI / PKIX jose.JWK.asKey(input, form). then(function(result)
{ // {result} é um jose.JWK.Key // {result.keystore} é um jose.JWK.KeyStore exclusivo});
```

5.2.2 Envoltório de chave AES-128 (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 criptografar e descriptografar. A chave para “example-1” que geramos antes é uma dessas chaves. No AES-128 Key Wrap, essa chave é usada para agrupar uma chave gerada aleatoriamente, que é usada para criptografar o conteúdo usando o algoritmo AES-128 GCM. Também seria possível usar esta chave diretamente (modo Direct Encryption).

```
função criptografar(chave, opções, texto simples) {
    return jose.JWE.createEncrypt(opções, chave) .update(texto
        simples) .final();

}

function a128gcm(compact) { chave
    const = keystore.get('example-1'); const opções =
    { formato: compacto ? 'compacto' : 'geral', contentAlg:
        'A128GCM'

    };

    return encrypt(chave, opções, JSON.stringify(payload));
}
```

A biblioteca node-jose trabalha principalmente com promessas⁶. O objeto retornado por a128gcm é uma promessa. A função createEncrypt pode criptografar qualquer conteúdo passado para ela. Em outras palavras, não é necessário que o conteúdo seja um JWT (embora na maioria das vezes seja). É por esse motivo 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 públicas/privadas. 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.

```
função criptografar(chave, opções, texto simples) {
    return jose.JWE.createEncrypt(opções, chave) .update(texto
        simples) .final();

}
```

```
function rsa(compact) {
```

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

```

chave const = keystore.get('example-2'); const
opções = { formato: compacto ? 'compacto' : 'geral',
            contentAlg: 'A128CBC-HS256'

};

return encrypt(chave, opções, 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 à do RSA:

```

função criptografar(chave, opções, texto simples) {
    return jose.JWE.createEncrypt(opções, chave) .update(texto
        simples) .final();

}

function ecdhes(compact) { const
    key = keystore.get('example-3'); const opções =
    { formato: compacto ? 'compacto' : 'geral', contentAlg:
        'A128GCM'

};

return encrypt(chave, opções, 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)

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

```

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

```

```

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

const promessa = nova Promessa((resolver, rejeitar) => {

    signaturePromise.then(result => { const
        options = { format: compact ?
            'compact' : 'general', contentAlg: 'A128CBC-HS256'

        };
        resolve(encrypt(encryptionKey, opções, JSON.stringify(resultado))); }, erro =>
    { rejeitar(erro);

    });

    promessa de retorno ;
}

```

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 jsonwebtoken) para esse fim. No entanto, dada a necessidade do node-jose, não faz sentido adicionar dependências e usar APIs inconsistentes.

Executar a etapa de assinatura primeiro só é possível porque o 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. É também a razão pela qual o JWE exige a presença de uma marca de autenticação.

5.2.6 Descriptografia

A descriptografia é tão simples quanto a criptografia. Assim como na criptografia, a carga útil deve ser convertida explicitamente entre diferentes formatos de dados.

```

// 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(` Resultado
            descriptografado: ${JSON.stringify(decrypted)} ); }, error =>
        { console.log(error);
    }

```

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

A descriptografia dos algoritmos RSA e Curva Elíptica é análoga, usando a chave privada em vez da chave simétrica. Se você tiver um armazenamento de chaves com as declarações infantis certas, é possível simplesmente passar o armazenamento de chaves 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) //basta passar o keystore
    aqui .decrypt(result) .then(decrypted => { decrypted.payload =
        JSON.parse(decrypted.payload); console.log(` Resultado
            descriptografado : ${ JSON.stringify(decrypted)}`); }, error =>
    { console.log(error);

});
```

Capítulo 6

Chaves da Web JSON (JWK)

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

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

As chaves são especificadas em diferentes declarações de cabeçalho. JWKs literais são colocados sob a reivindicação `jwk`. A reivindicação `jku`, por outro lado, pode apontar para um *conjunto* de chaves armazenadas em uma URL. Ambas as reivindicações estão no formato JWK.

Um exemplo de JWK:

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "MKBCTNlcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
  "y": "4EtI6SRW2YiLUrN5vfvVHuhp7x8PxItmWWIbbM4IFyM",
  "d": "870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",
  "use": "enc", "criança": "1"
}
```

6.1 Estrutura de uma chave da Web JSON

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 de chave. Os parâmetros comuns são:

- **kty**: “tipo de chave”. Esta afirmação diferencia os tipos de chaves. Os tipos suportados são EC, para chaves de curva elíptica; RSA para chaves RSA; e oct para chaves simétricas. Esta declaração é necessária.
- **use**: esta declaração especifica o uso pretendido da chave. Existem dois usos possíveis: sig (para assinatura) e enc (para criptografia). Esta reivindicação é opcional. A mesma chave pode ser usada para criptografia e assinaturas, caso em que esse membro não deve estar presente.
- **key_ops**: uma matriz de valores de string que especifica usos detalhados para a chave. Os valores possíveis são: assinar, verificar, criptografar, descriptografar, wrapKey, unwrapKey, derivaKey, derivaBits. Certas operações não devem ser usadas juntas. Por exemplo, assinar e verificar são apropriados para a mesma chave, enquanto assinar e criptografar não são. Esta declaração é opcional e não deve ser usada ao mesmo tempo que a declaração de uso. Nos casos em que ambos estão presentes, seu conteúdo deve ser consistente.
- **alg**: “algoritmo”. O algoritmo a ser usado com esta chave. Pode ser qualquer um dos algoritmos admitidos para operações JWE ou JWS. Esta reivindicação é opcional.
- **kid**: “id da chave”. Um identificador exclusivo para esta chave. Ele pode ser usado para corresponder uma chave a uma declaração infantil no cabeçalho JWE ou JWS ou para escolher uma chave de um conjunto de chaves de acordo com a lógica do aplicativo. Esta reivindicação é opcional. Duas chaves no mesmo conjunto de chaves podem carregar o mesmo kid apenas se tiverem reivindicações de kty diferentes e forem destinadas ao mesmo uso.
- **x5u**: uma URL que aponta para um certificado de chave pública X.509 ou cadeia de certificados no formato codificado PEM. Se outras declarações opcionais estiverem presentes, elas devem ser consistentes com o conteúdo do certificado. Esta reivindicação é opcional.
- **x5c**: um certificado X.509 DER codificado em Base64-URL ou cadeia de certificados. Uma cadeia de certificados é representada como uma matriz de tais certificados. O primeiro certificado deve ser o certificado referido por este JWK. Todas as outras reivindicações presentes neste JWK devem ser consistentes com os valores do primeiro certificado. Esta reivindicação é opcional.
- **x5t**: uma impressão digital SHA-1 codificada em Base64-URL da codificação DER de um certificado X.509. O certificado para o qual esta impressão digital aponta deve ser consistente com as reivindicações neste JWK. Esta reivindicação é opcional.
- **x5t#S256**: idêntico à reivindicação 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 do algoritmo de chave. As chaves RSA, por outro lado, carregam parâmetros como n, e, dp, etc. O significado desses parâmetros ficará claro no [capítulo 7](#), onde cada algoritmo de chave é explicado em detalhes.

6.1.1 Conjunto de Chaves Web JSON

A especificação JWK admite grupos de chaves. Estes 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 da Web JSON é simplesmente um objeto JSON com um membro de chaves. Este membro é uma matriz JSON de JWKs.

Exemplo de conjunto JWK:

```
{
  "chaves": [ {
    "kty": "EC",
    "crv": "P-256",
    "x": "MKBCTNlcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
    "y": "4EtI6SRW2YiLUrN5vfvVHuhp7x8PxltmWWIbbM4IFyM",
    "use": "enc", "kid": "1"

  },
  {
    "kty": "RSA", "n":
    "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
      4ccbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMs
      tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
      QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbl
      SD08qNLyrdkt-bFTWhAl4vMQFh6WeZu0fM4IFd2NcRwr3XPksINHaQ-G_xBnilqb
      w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",
    "e": "AQAB",
    "alg": "RS256",
    "criança": "2011-04-29"
  }
]
}
```

Neste exemplo, duas chaves públicas estão disponíveis. A primeira é do tipo curva elíptica e está limitada a operações de *criptografia* pela declaração de uso. O segundo é do tipo RSA e está associado a um algoritmo específico (RS256) pela reivindicação alg. Isso significa que essa segunda chave deve ser usada para *assinaturas*.

Capítulo 7

Algoritmos Web JSON

Você provavelmente notou que há muitas referências a este capítulo ao longo deste manual. A razão é que grande parte da mágica por trás dos JWTs está nos algoritmos empregados com eles. 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 atualmente. Compreendê-los em profundidade não é necessário para usar os JWTs de forma eficaz e, portanto, este capítulo é destinado a mentes curiosas que desejam entender a última peça do quebra-cabeça.

7.1 Algoritmos Gerais

Os algoritmos a seguir têm muitos aplicativos diferentes dentro das especificações JWT, JWS e JWE. Alguns algoritmos, como 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 de binário para texto. Seu principal objetivo é transformar uma sequência de octetos em uma sequência de caracteres imprimíveis, ao custo de aumentar o tamanho. Em termos matemáticos, Base64 transforma uma sequência de números de base 256 em uma sequência de números de base 64. A palavra *base* pode ser usada no lugar de *radix*, daí o nome do algoritmo.

Nota: Base64 não é realmente usado pela especificação JWT. É a variante *Base64-URL* descrita posteriormente neste capítulo, que é usada pelo JWT.

Para entender como o Base64 pode transformar uma série de números arbitrários em texto, primeiro é necessário estar familiarizado com os sistemas de codificação de texto. Sistemas de codificação de texto mapeiam números para caracteres. Embora esse mapeamento seja arbitrário e, no caso de Base64, possa ser definido pela implementação, o padrão de fato para a codificação Base64 é RFC 4648¹.

¹<https://tools.ietf.org/rfc/rfc4648.txt>

```

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

```

Na codificação Base64, cada caractere representa 6 bits dos dados originais. A codificação é realizada em grupos de quatro caracteres codificados. Portanto, 24 bits de dados originais são reunidos e codificados como quatro caracteres Base64. Como se espera que os dados originais sejam 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 valores de 8 bits -> dados concatenados de 24 bits -> 4 caracteres de 6 bits

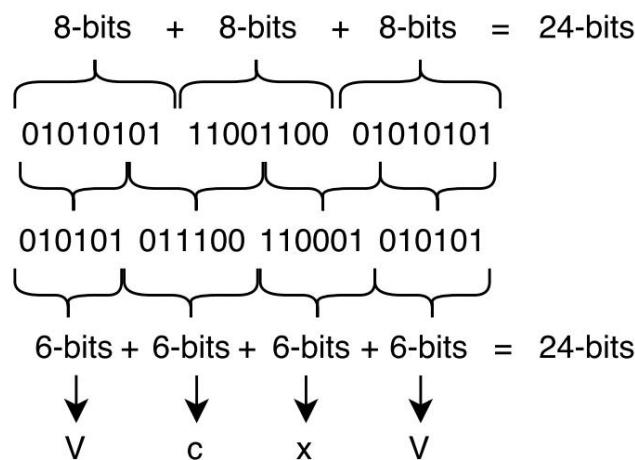


Figura 7.1: Codificação Base64

Se o número de octetos nos dados de entrada não for divisível por três, a última porção de dados a codificar terá menos de 24 bits de dados. Quando este for o caso, zeros são adicionados aos dados de entrada concatenados para formar um número inteiro 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 produzirá 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 URL. Base64 é uma codificação conveniente para passar dados arbitrários em campos de texto. Como apenas dois caracteres de Base64 são problemáticos como parte da URL, uma variante segura de URL é fácil de implementar. O caractere + e o caractere / são substituídos pelo caractere - e o _.

7.1.1.2 Exemplo de código

O exemplo a seguir implementa um codificador de URL Base64 burro. O exemplo foi escrito pensando na simplicidade, e não na velocidade.

```
tabela const = [
  '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 * @returns
 * uma String com os valores codificados *
 */

export function encode(input) { let result
  = "";
  for(let i = 0; i < input.length; i += 3) { const restante =
    input.length - i;
```

```

deixe concat = input[i] << 16; resultado
+= (tabela[concat >>> (24 - 6)]);

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

if(restante > 2) { concat |=
input[i + 2]; resultado +=
tabela[(concat >>> (24 - 18)) & 0x3F] +
tabela[concat & 0x3F];
} else
{ resultado += tabela[(concat >>> (24 - 18)) & 0x3F] + "=";

} } else
{ resultado += tabela[(concat >>> (24 - 12)) & 0x3F] + "==";
}
}

resultado de retorno ;
}

```

7.1.2 SHA

O Secure Hash Algorithm (SHA) usado nas especificações do JWT é definido no FIPS-1802 . Não deve ser confundido com a família de algoritmos SHA-13, que está obsoleta desde 2010. Para diferenciar esta família da anterior, esta família às vezes é chamada de *SHA-2*.

Os algoritmos no RFC 4634 são SHA-224, SHA-256, SHA-384 e SHA-512. De importância para JWT são SHA-256 e SHA-512. Vamos nos concentrar na variante SHA-256 e explicar suas diferenças em relação às outras variantes.

Assim como muitos algoritmos de hash, o SHA funciona processando a entrada em blocos de tamanho fixo, aplicando uma série de operações matemáticas e acumulando o resultado executando uma operação com os resultados da iteração anterior. Depois que todos os blocos de entrada de tamanho fixo são processados, diz-se que o resumo foi calculado.

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 esta 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:

²<http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>

³<https://en.wikipedia.org/wiki/SHA-1>

```

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

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

função maj(x, y, z) {
    retornar (x & y) ^ (x & z) ^ (y & z);
}

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

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

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

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

```

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

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

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 anexado para que o comprimento resultante da mensagem seja a solução a esta equação:

$$\begin{aligned} L &= \text{Comprimento da mensagem em bits} \\ 0 &= (65 + N + L) \bmod 512 \end{aligned}$$

3. Em seguida, o número de bits na mensagem original é anexado como um 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
 acolchoada: 01011111 01010101 10101010 00111100 10000000 ...0... 00000000 00100000

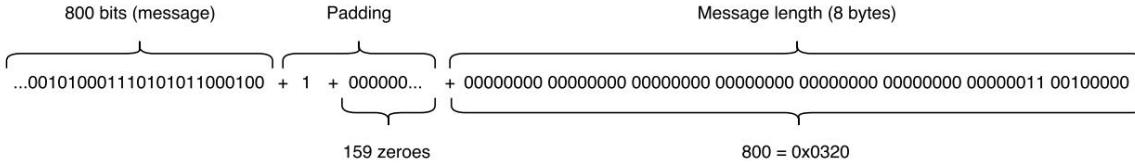


Figura 7.2: Preenchimento SHA

Uma implementação simples em JavaScript poderia ser:

```
function padMessage(message) { if(!
  (message instanceof Uint8Array) && !(message instanceof Int8Array)) {
  lançar novo erro("contêiner de mensagem não suportado");
}

const bitLength = message.length * 8; const
fullLength = bitLength + 65; //Extra 1 + tamanho da mensagem. 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: suporta mensagens
com bitLength maior que 2^32 padded[padded.length - 1] = bitLength;

retorno acolchoado;
}
```

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

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

```

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

const padded = padMessage(mensagem);
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(deixe t = 16; t < 64; ++t) {
    w[t] = ssig1(w[t - 2]) + w[t - 7] + ssig0(w[t - 15]) + w[t - 16];
}

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

for(let t = 0; t < 64; ++t) { let t1 = h +
bsig1(e) + ch(e, f, g) + k[t] + w[t]; deixe 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 de forma legível:

```

if(returnBytes) { resultado
  const = new Uint8Array(h_.length * 4); h_.forEach((valor,
índice) => {
  const i = índice * 4;
  resultado[i] = (valor & 0xFF24) | resultado[i + 1] = (valor >>>
  16) & 0xFF; resultado[i + 2] = (valor >>> 8) & 0xFF;
  resultado[i + 3] = (valor >>> 0) & 0xFF;

});

  resultado de
retorno ; } else { function
toHex(n) { let str = (n >>>
0).toString(16); deixe resultado = ""; for(let i =
str.length; i < 8; ++i) { resultado += "0";

}

  return resultado + str;
}

deixe resultado = "";
h_.forEach(n =>
{ resultado += toHex(n);
});

  resultado de retorno ;
}

```

Embora funcione, observe que a implementação acima não é ideal (e não suporta mensagens maiores que 232).

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, é necessária uma biblioteca separada para matemática de 64 bits (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

Os Códigos de Autenticação de Mensagens Baseados em Hash (HMAC)⁴ fazem uso de uma função 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 usa uma função hash, uma mensagem e uma chave secreta como entradas e produz um código de autenticação como saída. A força da função hash criptográfica garante que a mensagem não possa ser modificada sem a chave secreta.

Assim, os HMACs atendem a propósitos de *autenticação* e *integridade de dados*.

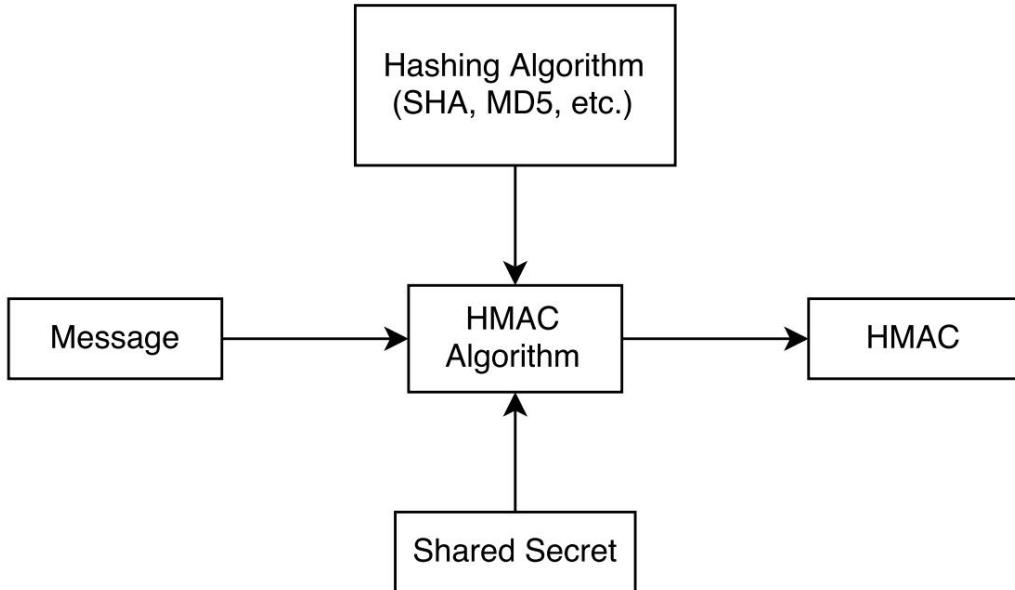


Figura 7.3: HMAC

Funções hash fracas podem permitir que usuários mal-intencionados comprometam a validade do código de autenticação. Portanto, para que os HMACs sejam úteis, uma função de hash forte deve ser escolhida. A família de funções SHA 2 ainda é forte o suficiente para os padrões atuais, 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 ataques de colisão e prefixo. Embora esses ataques não necessariamente tornem 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:

Seja H a função hash criptográfica

B ser o comprimento do bloco da função hash

⁴<https://tools.ietf.org/html/rfc2104>

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

$\text{HMAC}(\text{mensagem}) = H(K' \text{ XOR } \text{opad} \parallel H(K' \text{ XOR } \text{ipad} \parallel \text{mensagem}))$

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

Se K for menor que B, zeros serão adicionados até que K seja de comprimento 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, será usado como está (K é K').

Aqui está um exemplo de implementação em JavaScript:

```
função padrão de exportação hmac(hashFn, blockSizeBits, segredo, mensagem, returnBytes) { if(!message
  instanceof Uint8Array) { throw new Error('message must be of Uint8Array');

}

const blockSizeBytes = blockSizeBits / 8;

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

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

}

const ipadSecret = ipad.map((valor, índice) => {
  valor de retorno paddedSecret[índice];
});
const opadSecret = opad.map((valor, índice) => {
  valor de retorno paddedSecret[índice];
});

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

```

const result =
    hashFn( append(opadSecret,
        uint32ArrayToUint8Array(hashFn(append(ipadSecret,
            mensagem), verdadeiro))),
    returnBytes);

resultado de retorno ;
}

```

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

7.2.1.1 HMAC + SHA256 (HS256)

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

```

função padrão de exportação jwtEncode(cabeçalho, carga útil, segredo) {
    if(typeof header != 'object' || typeof payload != 'object') { throw new Error('header
        and payload must be objects');
    }
    if(tipo de segredo != 'string') {
        throw new Error("o segredo deve ser uma string");
    }

    cabeçalho.alg = 'HS256';

    const encHeader = b64(JSON.stringify(cabeçalho)); const
    encPayload = b64(JSON.stringify(payload)); const jwtUnprotected
    = `${encHeader}.${encPayload}`; assinatura const =
    b64(uint32ArrayToUint8Array(
        hmac(sha256, 512, segredo, stringToUtf8(jwtUnprotected), verdadeiro)));

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

```

Observe que esta função não executa nenhuma validação do cabeçalho ou carga útil (além de verificar se eles são objetos). Você pode chamar essa função assim:

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

Cole o JWT gerado no depurador5 do JWT.io e veja como ele é decodificado e validado.

Essa função é muito semelhante àquela usada no [capítulo 4](#) como demonstração do algoritmo de assinatura. Do [capítulo 4](#):

⁵<https://jwt.io>

```

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

A verificação é igualmente fácil:

função de exportação 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) {
  lançar novo erro ('Formato JWT inválido');
}

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

const jwtDesprotegido = `${split[0]}.${split[1]}'; assinatura const =
b64(hmac(sha256, 512, segredo, stringToUtf8(jwtUnprotected),
  verdadeiro));

return
  { cabeçalho:
    cabeçalho, carga útil: JSON.parse(unb64(split[1])),
    válido: assinatura == split[2]
  };
}

```

A assinatura é separada 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 maneira:

```

const segredo = 'segredo';
const codificado = jwtEncode({}, {sub: "teste@teste.com"}, segredo); const
decodificado = jwtVerifyAndDecode(codificado, segredo); Este código está disponível
no arquivo hs256.js das amostras incluídas6 neste manual.

```

⁶<https://github.com/auth0/jwt-handbook-samples>

7.2.2 RSA

O RSA é um dos sistemas criptográficos mais usados atualmente. Foi desenvolvido em 1977 por Ron Rivest, Adi Shamir e Leonard Adleman, cujas iniciais foram usadas para nomear o algoritmo. O principal aspecto do RSA está em sua assimetria: a chave usada para criptografar algo *não* é a chave usada para descriptografá-lo. Esse 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 ela foi assinada por uma chave privada específica (sem realmente conhecê-la).

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

Muitos algoritmos criptográficos, e em particular o RSA, são baseados na relativa dificuldade de realizar certas operações matemáticas. O RSA escolhe a fatoração inteira⁷ como sua principal ferramenta matemática. A fatoração inteira 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, resultam no número inteiro original.

$\text{inteiro} = \text{fator_1} \times \text{fator_2}$

Esse problema pode parecer fácil no começo. E para números pequenos, é. Tomemos por exemplo o número 35:

$$35 = 7 \times 5$$

Conhecendo as tabelas de multiplicação de 7 ou 5, é fácil encontrar dois números que resultam em 35 quando multiplicados. Um algoritmo ingênuo para encontrar fatores para um número inteiro poderia ser:

1. Seja n o número que queremos fatorar.
2. Seja x um número entre 2 (inclusive) en / 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 realizando a etapa 3 aumentando x em 1 em cada iteração até que x atinja seu limite superior $n / 2$. Quando isso acontecer, você terá encontrado todos os fatores possíveis de n .

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

Uma versão melhor desse algoritmo é chamada de divisão de tentativas e define condições mais estritas para x . Em particular, ele define o limite superior de x como \sqrt{n} e, em vez de aumentar x em 1 a cada iteração, faz x assumir o valor de números primos cada vez maiores. É trivial provar por que essas condições tornam o algoritmo mais eficiente, mantendo-o correto (embora fora do escopo deste texto).

Algoritmos mais eficientes existem, mas, por mais eficientes que sejam, mesmo com os computadores de hoje, certos números são computacionalmente impossíveis de fatorar. O problema é agravado quando certos

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

números são escolhidos como n . Por exemplo, se n é o resultado da multiplicação de dois números primos⁸, isso 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 que? Porque os números não primos têm divisores diferentes deles mesmos (por definição), e esses divisores são, por sua vez, divisores de qualquer número multiplicado por eles. Em outras palavras, qualquer divisor de um fator de um número também é um fator do número.

Ou, em outros termos, se n possui fatores não primos, possui mais de dois fatores. Então, 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 é encontrá-los.

Quando dois números primos grandes e diferentes são escolhidos e depois multiplicados, o resultado é outro número grande (chamado de semiprimo). Mas esse grande número tem uma propriedade especial adicional: é realmente difícil de fatorar. Mesmo os algoritmos de fatoração mais eficientes, como o crivo de campo numérico geral⁹, não podem fatorar números grandes resultantes da multiplicação de grandes números primos em prazos razoáveis. Para dar uma noção de escala, em 2009 um número de 768 bits (232 dígitos decimais) foi fatorado¹⁰ após 2 anos de trabalho por um cluster de computadores. Aplicações típicas de RSA usam números de 2048 bits ou maiores.

O algoritmo¹¹ de Shor é um tipo especial de algoritmo de fatoração que pode mudar drasticamente as coisas no futuro. Embora a maioria dos algoritmos de fatoração sejam de natureza clássica (ou seja, eles operam em computadores clássicos), o algoritmo de Shor depende de computadores quânticos¹². 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 de tempo polinomial (em vez 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 como requerido pelo algoritmo de Shor ainda não foi desenvolvido, mas esta é uma área de pesquisa ativa no momento.

Embora atualmente a fatoração inteira seja computacionalmente impraticável para grandes semiprimos, não há nenhuma prova matemática de que esse seja o caso. Em outras palavras, no futuro podem surgir algoritmos que resolvam a fatoração inteira em prazos razoáveis. O mesmo pode ser dito do RSA.

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

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

Figura 7.4: Expressão básica RSA

⁸https://en.wikipedia.org/wiki/Prime_number

⁹https://en.wikipedia.org/wiki/General_number_field_sieve

¹⁰<http://eprint.iacr.org/2010/006> ¹¹https://en.wikipedia.org/wiki/Shor%27s_algorithm ¹²https://en.wikipedia.org/wiki/Quantum_computing

É computacionalmente viável encontrar três 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 Escolhendo 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 fazer com que um invasor encontre um desses números.
 - Como não há como gerar aleatoriamente números primos, depois que dois números aleatórios são escolhidos, eles devem passar por um teste de primalidade. Verificações de primalidade determinísticas podem ser caras, então algumas implementações dependem de métodos probabilísticos. Quão provável é encontrar um primo falso precisa ser considerado.
 - 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 totiente¹³ de Euler para n . Como n é um número semiprimo, isso é tão simples quanto: $n - (p + q - 1)$. Chamaremos esse valor de $\phi(n)$.
4. Escolha um e que atenda aos seguintes critérios:
 - $1 < e < \phi(n)$
 - e e $\phi(n)$ devem ser primos primos
5. Escolha um anúncio que satisfaça a seguinte expressão:

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

Figura 7.5: Expressão básica RSA

A chave pública é composta pelos valores n e e . A chave privada é composta pelos valores n e d . Os valores p , q e $\phi(n)$ devem ser descartados ou mantidos em segredo, pois podem ser usados para ajudar a encontrar d . A partir das equações acima, é evidente que e e d são matematicamente simétricos. Podemos reescrever a equação do passo 5 como:

$$d \cdot e \equiv 1 \pmod{\phi(n)}$$

Figura 7.6: 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 é encontrá-lo. Portanto, o RSA funciona desde que apenas um dos valores e ou d seja conhecido

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

festas públicas. Esta é também a razão pela qual um desses valores é escolhido: o valor público pode ser escolhido para ser o menor possível. Isso acelera a computação sem comprometer a segurança do algoritmo.

7.2.2.2 Assinatura Básica

O login no RSA é feito da seguinte forma:

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

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

1. A assinatura é elevada à potência de e modulo n. O valor resultante é o resumo de referência valor.
2. Um resumo da mensagem é produzido a partir da mensagem usando a mesma função de hash da assinatura etapa.
3. Os resultados das etapas 1 e 2 são comparados. Se forem iguais, a parte signatária deve estar de 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 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: PKCS#1 RSASSA v1.5 usando SHA-256, também conhecido como RS256 na especificação JWA.

A especificação do Padrão de Criptografia de Chave Pública nº 1 (PKCS nº 1)¹⁴ define uma série de primitivos, formatos e esquemas de criptografia baseados no algoritmo RSA. Esses elementos trabalham juntos para fornecer uma implementação detalhada do RSA utilizável em plataformas de computação modernas. O RSASSA é um dos esquemas definidos nele e permite o uso de RSA para assinaturas.

7.2.2.3.1 Algoritmo

Para produzir uma assinatura:

1. Aplique a primitiva **EMSA-PKCS1-V1_5-ENCODE** à mensagem (uma matriz de octetos).
O resultado é a **mensagem codificada**. Essa primitiva faz uso de uma função de hash (geralmente uma função de hash da família SHA, como SHA-256). Esta primitiva aceita um comprimento de mensagem codificada esperado.
Neste caso, será o comprimento em octetos do número RSA n (o comprimento da chave).

¹⁴<https://www.ietf.org/rfc/rfc3447.txt>

2. Aplique a primitiva **OS2IP** à mensagem codificada. O resultado é o **representante da mensagem inteira**. OS2IP é a sigla para “Octet-String to Integer Primitive”.
3. Aplique a primitiva **RSASP1** ao representante da mensagem inteira usando a chave privada. O resultado é o **representante da assinatura inteira**.
4. Aplique a primitiva **I2OSP** para converter o representante da assinatura inteira em uma matriz de octetos (a **assinatura**). I2OSP é a sigla para “Integer to Octet-String Primitive”.

Uma possível implementação em JavaScript, dadas as primitivas mencionadas acima, poderia ser:

```
/*
 * Produz uma assinatura para uma mensagem usando o algoritmo RSA conforme definido * em
 * PKCS#1.
 * @param {privateKey} Chave privada RSA, um objeto com
 * três membros: size (tamanho em bits), n (o módulo) ed (o expoente
 * privado), ambos bigInts (biblioteca de inteiros grandes). * @param
 * {hashFn} a função hash conforme exigido pelo PKCS#1, deve receber
 * um Uint8Array e retornar um Uint8Array * @param {hashType} Um símbolo que identifica o tipo de
 * função hash passada.
 *
 * Por enquanto, apenas "SHA-256" é suportado. Consulte o objeto "hashTypes"
 * para obter os valores possíveis.
 * @param {message} Uma String ou Uint8Array com dados arbitrários para assinar *
 * @return {Uint8Array} A assinatura como um Uint8Array */
function sign(privateKey, hashFn, hashType, message) { const
  encodedMessage =
    emsaPkcs1v1_5(hashFn, hashType, privateKey.size / 8, mensagem); const
  intMessage = os2ip(encodedMessage); const intSignature = rsasp1(privateKey,
  intMessage); assinatura const = i2osp(intSignature, privateKey.size / 8); assinatura de
  retorno ;
}

Para verificar uma assinatura:
```

1. Aplique a primitiva **OS2IP** à assinatura (uma matriz de octetos). Este é o **sinal inteiro representante da natureza**.
2. Aplique a primitiva **RSAVP1** ao resultado anterior. Essa primitiva também recebe a chave pública como entrada. Este é o **representante da mensagem inteira**.
3. Aplique a primitiva **I2OSP** ao resultado anterior. Essa primitiva recebe um tamanho esperado como entrada. Esse 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 faz uso de uma função de hash (geralmente uma função de hash da família SHA, como SHA-256). Esta primitiva aceita um comprimento de mensagem codificada esperado. Neste caso, será o comprimento em octetos do número RSA n (o comprimento da chave).
5. Compare as duas mensagens codificadas (das etapas 3 e 4). Se coincidirem, a assinatura é válida,

caso contrário, não é.

Em JavaScript:

```
/**
 * Verifica uma assinatura para uma mensagem usando o algoritmo RSASSA conforme definido *
 * em PKCS#1.
 * @param {publicKey} Chave privada RSA, um objeto com
 * três membros: size (tamanho em bits), n (o módulo) ee (o expoente
 * público), ambos bigInts (biblioteca de inteiros grandes). * @param
 * {hashFn} a função hash conforme exigido pelo PKCS#1, deve receber
 * um Uint8Array e retornar um Uint8Array * @param {hashType} Um símbolo que identifica o tipo de
 * função hash passada.

 *
 * Por enquanto, apenas "SHA-256" é suportado. Consulte o objeto
 * "hashTypes" para obter os valores possíveis.
 * @param {message} Uma String ou Uint8Array com dados arbitrários para verificar *
 * @param {signature} Um Uint8Array com a assinatura * @return {Boolean} true se a
 * assinatura for válida, false caso contrário. */

função de exportação VerifyPkcs1v1_5(publicKey,
    hashFn,
    hashType,
    mensagem,
    assinatura)
{ if(signature.length !== publicKey.size / 8) {
    lançar novo erro('comprimento de assinatura inválido');
}

const intAssinatura = os2ip(assinatura); const
intVerification = rsavp1(publicKey, intSignature); mensagem de verificação
const = i2osp(intVerification, publicKey.size / 8);

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

return uint8ArrayEquals (mensagem codificada, mensagem de verificação);
}
```

7.2.2.3.1.1 Primitiva EMSA-PKCS1-v1_5

Este primitivo leva três elementos:

- A mensagem •
 - O comprimento pretendido do resultado
 - E a função hash a usar (que deve ser uma das opções do passo 2)
1. Aplique a função hash selecionada à mensagem.

2. Produza a codificação DER para a seguinte estrutura ASN.1:

```
DigestInfo ::= SEQUENCE
  { digestAlgorithm DigestAlgorithm, resumo
    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 PARÂMETROS NULL }
  { OID id-md5 PARÂMETROS NULL }
  { OID id-sha1 PARÂMETROS NULL }
  { OID id-sha256 PARÂMETROS NULL }
  { OID id-sha384 PARÂMETROS NULL }
  { OID id-sha512 PARÂMETROS NULL }
}
```

3. Se o comprimento solicitado do resultado for menor 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 pretendido muito curto").

4. Repita o octeto 0xFF o seguinte número de vezes: comprimento solicitado + step2Length - 3. Essa matriz de octetos é chamada de PS.

5. Produza a mensagem final codificada (EM) como (|| é o operador de concatenação):

EM = 0x00 || 0x01 || PS || 0x00 || step2result

Os OIDs ASN.1 geralmente são definidos em suas próprias especificações. Em outras palavras, você não encontrará o SHA-256 OID na especificação PKCS#1. Os OIDs SHA-1 e SHA-2 são definidos no RFC 356015 .

7.2.2.3.1.2 OS2IP primitivo

A primitiva OS2IP pega uma matriz de octetos e gera um representante inteiro.

- Seja X1, X2, ..., Xn os octetos do primeiro ao último da entrada. • Calcule o resultado como:

$$\text{resultado} = X_1 \cdot 2^{8(n-1)} + X_2 \cdot 2^{8(n-2)} + \dots + X_n \cdot 2^0$$

Figura 7.7: Resultado OS2IP

7.2.2.3.1.3 RSASP1 primitivo

¹⁵<https://tools.ietf.org/html/rfc3560.html>

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

- Sejam n e d os números RSA da chave privada. • Seja 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:

$$ds = m \pmod{n}$$

Figura 7.8: resultado 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 são armazenadas dessa maneira. No entanto, armazenar a chave privada como n e d é válido.

7.2.2.3.1.4 RSAVP1 primitivo

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

- Sejam n e e os números RSA da chave pública. • Seja s o representante da assinatura inteira.

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

$$e m = s \pmod{n}$$

Figura 7.9: resultado RSAVP1

7.2.2.3.1.5 I2OSP primitivo

A primitiva I2OSP pega um representante inteiro e produz uma matriz de octetos.

- Seja len o comprimento esperado do array de octetos. • Seja x o representante inteiro.

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

$$x = x_1 \cdot 256^{\lfloor \log_2 len \rfloor} + x_2 \cdot 256^{\lfloor \log_2 len \rfloor - 1} + \dots + x_{\lfloor \log_2 len \rfloor} \cdot 256 + x_{\lfloor \log_2 len \rfloor + 1}$$

Figura 7.10: Decomposição I2OSP

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

7.2.2.3.2 Exemplo de código

Como o RSA requer aritmética de precisão arbitrária, usaremos a biblioteca JavaScript big-integer¹⁶¹⁷.

As primitivas OS2IP e I2OSP são bastante simples:

```
função os2ip(bytes) {
    deixe resultado = bigint();

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

resultado de retorno ;
```

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

resultado const = new Uint8Array(comprimento esperado);
deixe resto = bigint(intRepr); for(let i = esperadoLength - 1; i >=
0; --i) { const position = bigint(256).pow(i); const quotrem =
    resto.divmod(position); resto = quotrem.restante;
    resultado[resultado.comprimento - 1 - i] =
    quotrem.quotem.valueOf();
```

```
}
```

resultado **de retorno** ;

}

A primitiva I2OSP basicamente decompõe um número em seus componentes básicos 256¹⁷.

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

16¹⁶<https://www.npmjs.com/package/big-integer>

17¹⁷https://en.wikipedia.org/wiki/Positional_notation

```

function rsasp1(privateKey, intMessage)
{ if(intMessage.isNegative() ||
   intMessage.greaterOrEquals(privateKey.n)) { throw new
   Error("mensagem representativa fora do intervalo");
}

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

```

Para verificações, a primitiva RSAVP1 é usada:

```

export function rsavp1(publicKey, intSignature)
{ if(intSignature.isNegative() ||
   intSignature.greaterOrEquals(publicKey.n)) { throw new
   Error("mensagem representativa fora do intervalo");
}

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

```

Por fim, a primitiva EMSA-PKCS1-v1_5 executa a maior parte do trabalho duro, transformando a mensagem em sua representação codificada e preenchida.

```

função emsaPkcs1v1_5(hashFn, hashType, esperadoLength, mensagem) {
  if(hashType != hashTypes.sha256) { throw new
  Error("Tipo de hash não suportado");
}

  resumo const = hashFn(mensagem, verdadeiro);

  // DER é um conjunto mais restrito de BER, isso (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); berWriter.endSequence();

  // T é o nome deste elemento no RFC 3447 const t =
  berWriter.buffer;

  if(expectedLength < (t.length + 11)) { throw new
  Error('comprimento pretendido da mensagem codificada muito curto');
}

```

```

    }

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 SHA-256 é suportado. Adicionar outras funções de hash é tão simples quanto adicionar os IDs corretos.

A função signPkcs1v1_5 junta todas as primitivas para realizar a assinatura:

```

/*
 * Produz uma assinatura para uma mensagem usando o algoritmo RSA conforme definido *
 * em PKCS#1.
 * @param {privateKey} Chave privada RSA, um objeto com
 * três membros: size (tamanho em bits), n (o módulo) ed (o expoente privado),
 * ambos bignums (biblioteca de inteiros grandes). * @param {hashFn} a
 * função hash conforme exigido pelo PKCS#1, deve receber um Uint8Array
 * e retornar um Uint8Array * @param {hashType} Um símbolo que identifica o tipo de função hash passada.
 */

*
 * Por enquanto, apenas "SHA-256" é suportado. Consulte o objeto
 * "hashTypes" para obter os valores possíveis.
 * @param {message} Uma String ou Uint8Array com dados arbitrários para assinar * @return
 * {Uint8Array} A assinatura como um Uint8Array */

```

```

função de exportação signPkcs1v1_5(privateKey, hashFn, hashType, mensagem) {
  const encodedMessage =
    emsaPkcs1v1_5(hashFn, hashType, privateKey.size / 8, mensagem); const
    intMessage = os2ip(encodedMessage); const intSignature = rsasp1(privateKey,
    intMessage); assinatura const = i2osp(intSignature, privateKey.size / 8); assinatura de
    retorno ;
}

}

```

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

```

função padrão de exportação jwtEncode(header, payload, privateKey) {
  if(typeof header != 'object' || typeof payload != 'object') { throw new Error('header
    and payload must be objects');
}

cabeçalho.alg = 'RS256';

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

```

```

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

```

Essa função é muito semelhante à função jwtEncode para HS256 mostrada na seção HMAC.

A verificação é igualmente simples:

```

/**
 * Verifica uma assinatura para uma mensagem usando o algoritmo RSASSA conforme definido *
 * em PKCS#1.
 * @param {publicKey} Chave privada RSA, um objeto com
 *                   três membros: size (tamanho em bits), n (o módulo) ee (o expoente público),
 *                   ambos bigInts (biblioteca de inteiros grandes). * @param {hashFn} a
 *                   função hash conforme exigido pelo PKCS#1, deve receber um Uint8Array
 * e retornar um Uint8Array * @param {hashType} Um símbolo que identifica o tipo de função hash
 *                   passada.
 *
 * Por enquanto, apenas "SHA-256" é suportado. Consulte o objeto
 * "hashTypes" para obter os valores possíveis.
 * @param {message} Uma String ou Uint8Array com dados arbitrários para verificar * @param
 * {signature} Um Uint8Array com a assinatura * @return {Boolean} true se a assinatura for válida,
 * false caso contrário. */

```

```

função de exportação VerifyPkcs1v1_5(publicKey,
                                         hashFn,
                                         hashType,
                                         mensagem,
                                         assinatura)
{ if(signature.length !== publicKey.size / 8) {
    lançar novo erro('comprimento de assinatura inválido');
}

const intAssinatura = os2ip(assinatura); const
intVerification = rsavp1(publicKey, intSignature); mensagem de verificação
const = i2osp(intVerification, publicKey.size / 8);

const encodedMessage =
emsapkcs1v1_5(hashFn, hashType, publicKey.size / 8, mensagem);

return uint8ArrayEquals (mensagem codificada, mensagem de verificação);
}

```

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

```
função de exportação jwtVerifyAndDecode(jwt, publicKey) { if(isString(jwt))  
  { throw new TypeError('jwt deve ser uma string');  
  
}  
  
const split = jwt.split('.'); if(split.length !=  
== 3) {  
  lançar novo erro ('Formato JWT inválido');  
}  
  
cabeçalho const = JSON.parse(unb64(split[0])); if(header.alg !=  
== 'RS256') { throw new Error('Algoritmo errado: ${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  
{ cabeçalho:  
  cabeçalho, carga útil: JSON.parse(unb64(split[1])),  
  válido: válido  
};  
}
```

Para simplificar, as chaves privada e pública devem 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 está em contraste com o formato usual PEM Encoded¹⁸. Consulte o arquivo rs256.js para obter mais detalhes.

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

openssl rsa -text -noout -in testkey.pem

O OpenSSL também pode ser usado para gerar uma chave RSA 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 embutidos no arquivo testkey.js são do arquivo testkey.pem no diretório samples que acompanha este manual. A chave pública correspondente está no arquivo pubtestkey.pem.

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

Copie a saída da execução do rs256.js sample19 na área JWT em JWT.io²⁰. 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 Prova bilistic Signature Scheme, em contraste com a abordagem *determinística* usual . Este esquema faz uso de um gerador de números aleatórios criptograficamente seguro. Se um RNG seguro não estiver disponível, as operações de assinatura e verificação resultantes fornecem um nível de segurança comparável a abordagens determinísticas. Dessa forma, o RSASSA-PSS resulta em uma melhoria líquida em relação às assinaturas PKCS v1.5 para os melhores cenários. Na natureza, no entanto, os esquemas PSS e PKCS v1.5 permanecem ininterruptos.

RSASSA-PSS é definido no Padrão de Criptografia de Chave Pública nº 1 (PKCS nº 1)²¹ e não está disponível em versões anteriores da norma.

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 da mensagem inteira**. OS2IP é a sigla para “Octet-String to Integer Primitive”.
3. Aplique a primitiva **RSASP1** ao representante da mensagem inteira usando a chave privada. O resultado é o **representante da assinatura inteira**.
4. Aplique a primitiva **I2OSP** para converter o representante da assinatura inteira em uma matriz de octetos (a **assinatura**). I2OSP é a sigla para “Integer to Octet-String Primitive”.

Uma possível implementação em JavaScript, dadas as primitivas mencionadas acima, poderia ser:

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

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

¹⁹<https://github.com/auth0/jwt-handbook-samples/blob/master/rs256.js>

²⁰<https://jwt.io> ²¹<https://www.ietf.org/rfc/rfc3447.txt>

```

const intSignature = rsasp1(privateKey, intMessage); assinatura const
= i2osp(intSignature, privateKey.size / 8); assinatura de retorno ;
}

}

```

Para verificar uma assinatura:

1. Aplique a primitiva **OS2IP** à assinatura (uma matriz de octetos). Este é o **sinal inteiro representante da natureza**.
 2. Aplique a primitiva **RSAVP1** ao resultado anterior. Essa primitiva também recebe a chave pública como entrada. Este é o **representante da mensagem inteira**.
 3. Aplique a primitiva **I2OSP** ao resultado anterior. Essa primitiva recebe um tamanho esperado como entrada. Esse 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. Esta primitiva mostra se a assinatura é válida ou não. Essa primitiva faz uso de uma função de hash (geralmente uma função de hash da família SHA, como SHA-256).
- A primitiva recebe um parâmetro que deve ser o número de bits no módulo da chave menos 1.

```

função de exportação VerifyPss(publicKey, hashFn, hashType, mensagem, assinatura) { if(signature.length !
== publicKey.size / 8) {
    lançar novo erro('comprimento de assinatura inválido');
}

const intAssinatura = os2ip(assinatura); const
intVerification = rsavp1(publicKey, intSignature); mensagem de verificação
const = i2osp(intVerification, Math.ceil( (publicKey.size - 1) / 8 ));

return emsaPssVerify(hashFn,
                      hashType,
                      mgf1.bind(null, hashFn), 256/8 ,
                      publicKey.size - 1 ,

                      mensagem,
                      mensagem de verificação);
}

```

7.2.2.4.1.1 MGF1: a função de geração de máscara

As funções de geração de máscara recebem entrada de qualquer comprimento e produzem saída de comprimento variável. Como as funções hash, elas são determinísticas: produzem a mesma saída para a mesma entrada. Em contraste com as funções de hash, porém, o comprimento da saída é variável. O algoritmo Função de Geração de Máscara 1 (MGF1) é definido no Padrão de Criptografia de Chave Pública nº 1 (PKCS nº 1)²².

²²<https://www.ietf.org/rfc/rfc3447.txt>

MGF1 usa um valor de semente e o comprimento pretendido da saída como entradas. O comprimento máximo da saída é definido como 232 . MGF1 usa internamente uma função hash configurável. PS256 especifica esta função hash como SHA-256.

1. Se o comprimento pretendido for maior que 232, pare com o erro “máscara muito longa”.
2. Itere de 0 até o teto do comprimento pretendido dividido pelo comprimento da saída da função hash menos 1 ($\text{ceiling}(\text{intendedLength} / \text{hashLength}) - 1$) fazendo a seguinte operaçãoções:
 1. Seja $c = \text{i2osp}(\text{counter}, 4)$ onde counter é o valor atual do contador de iteração.
 2. Seja $t = t.\text{concat}(\text{hash}(\text{seed}.\text{concat}(c)))$ onde t é preservado entre iterações, hash é a função de hash selecionada (SHA-256) e seed é o valor de semente de entrada.
3. Emite os octetos de comprimento pretendidos mais à esquerda do último valor de t como resultado da função.

7.2.2.4.1.2 Primitiva EMSA-PSS-ENCODE

O primitivo leva dois elementos:

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

Esta 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 sal usado internamente.

Estes parâmetros são todos especificados pelo PS256, pelo que não são configuráveis e para efeitos desta descrição são considerados constantes.

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

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

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

const hash1 = sha256(inputMessage);

2. Se o comprimento pretendido da mensagem for menor que o comprimento do hash mais o comprimento do sal mais 2, pare com um erro.

  if(intendedLength < (hashed1.length + pretendidoSaltLength + 2)) {
    throw new Error('Erro de codificação');
  }

3. Gere uma sequência aleatória de octetos do comprimento do sal.

4. Concatene oito octetos de valor zero com o hash da mensagem e o sal.

const m = [0,0,0,0,0,0,0, ...hashed1, ...sal];

5. Aplique a função hash ao resultado da etapa anterior.
```

```
const hash2 = sha256(m);
```

6. Gere uma sequência de octetos com valor zero de comprimento: comprimento máximo pretendido do resultado menos comprimento do sal menos comprimento do hash menos 2.

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

7. Concatene o resultado do passo anterior com o octeto 0x01 e o salt.

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

8. Aplique a função de geração de máscara ao resultado da etapa 5 e defina o comprimento pretendido dessa 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);
```

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

```
const maskedDb = db.map((valor, índice) => { ^
    valor de retorno dbMask[índice];
});
```

10. 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 os bits e, em seguida, defina esse número de bits para 0 começando pela esquerda.

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

11. Concatene o resultado do passo anterior com o resultado do passo 5 e o octeto 0xBC. Esse é o resultado.

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

7.2.2.4.1.3 Primitiva EMSA-PSS-VERIFY

O primitivo leva três elementos:

- A mensagem a ser verificada. •
- A assinatura como uma mensagem inteira codificada.
- O comprimento máximo pretendido da mensagem inteira codificada.

Esta 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 sal usado internamente.

Estes parâmetros são todos especificados pelo PS256, pelo que não são configuráveis e para efeitos desta descrição são considerados constantes.

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

```
const comprimento esperado = Math.ceil(comprimento esperadoBits / 8);

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

const digest1 = hashFn(mensagem, verdadeiro);

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

if(expectedLength < (digest1.length + saltLength + 2)) {
    retorna falso;
}

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

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

}

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

const maskedLength = comprimento esperado - digest1.length - 1; const
masked = notificationMessage.subarray (0, maskedLength); const digest2 =
verificaçãoMessage.subarray(maskedLength,
                                maskedLength + digest1.length);

5. Verifique se os 8 * esperadosLength - esperadoLengthBits mais à esquerda (o comprimento esperado em bits menos o comprimento solicitado em bits) os bits mascarados são 0.

zeroBits const = 8 * comprimento esperado - comprimento esperadoBits;
const zeroBitsMask = 0xFF >>> zeroBits; if((masked[0] & (~zeroBitsMask)) !=
== 0) { return false;

}

6. Passe o segundo elemento extraído do passo 4 (o resumo) para a função MGF selecionada.
Solicite que o resultado tenha um comprimento esperadoLength - 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 $8 * \text{esperadoLength} - \text{esperadoLengthBits}$ mais à esquerda do primeiro byte no elemento calculado na última etapa para 0.

```
zeroBits const = 8 * comprimento esperado - comprimento esperadoBits;
const zeroBitsMask = 0xFF >>> zeroBits; db[0] &= zeroBitsMask;
```

9. Verifique se os bytes esperados - hashLength - saltLength - 2 bytes 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 = esperadoLength - (digest1.length + saltLength + 2); if(!db.subarray(0,
zeroCheckLength).every(v => v === 0) ||
    db[zeroCheckLength] !== 0x01) { return
    false;
}
```

10. Extraia o sal 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 com colocado na etapa 1, e o sal extraído na última etapa.

```
const m = Uint8Array.of(0, 0, 0, 0, 0, 0, 0, ...digestão1, ...sal);
```

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

```
const esperadoDigest = hashFn(m, verdadeiro);
```

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

```
return uint8ArrayEquals(resumo2, esperadoResumo);
```

7.2.2.4.2 Exemplo de código

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.

```
função de exportação mgf1(hashFn, comprimento esperado, seed)
  { if(comprimento esperado > Math.pow(2, 32)) { throw new
    Error('máscara muito longa');
  }

  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);
```

```

        valor const = hashFn(Uint8Array.of(...seed, ...c), true); result.set(value, i *
        hashSize);

    } return resultado.subarray(0, comprimento esperado);
}

função de exportação emsaPssEncode(hashFn,
                                    hashType,
                                    mgf,
                                    saltLength,
                                    esperadoLengthBits,
                                    mensagem) { const
esperadoLength = Math.ceil(expectedLengthBits / 8);

const digest1 = hashFn(mensagem, verdadeiro);
if(expectedLength < (digest1.length + saltLength + 2)) { throw new Error('erro
de codificação');
}

const salt = crypto.randomBytes(saltLength); const m =
Uint8Array.of(...(novo Uint8Array(8)), ...digest1, ...salt); const
digest2 = hashFn(m,
verdadeiro); 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((valor, índice) => valor ^ dbMask[índice]);

zeroBits const = 8 * comprimento esperado - comprimento esperadoBits;
const zeroBitsMask = 0xFF >>> zeroBits; masked[0] &= zeroBitsMask;

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

função de exportação emsaPssVerify(hashFn,
                                    hashType,
                                    mgf,
                                    saltLength,
                                    esperadoLengthBits,
                                    mensagem,
                                    verificaçãoMensagem) { const
esperadoLength = Math.ceil(expectedLengthBits / 8);

const digest1 = hashFn(mensagem, verdadeiro);
if(expectedLength < (digest1.length + saltLength + 2)) {

```

```

        retorna false;
    }

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

    }

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

    }

    const maskedLength = comprimento esperado - digest1.length - 1; const
    masked = notificationMessage.subarray (0, maskedLength); const digest2 =
    verifica&gt;oMessage.subarray(maskedLength,
                                maskedLength + digest1.length);

    zeroBits const = 8 * comprimento esperado - comprimento esperadoBits;
    const zeroBitsMask = 0xFF >>> zeroBits; if((masked[0] & (~zeroBitsMask)) !
    == 0) { return false;

    }

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

    const zeroCheckLength = esperadoLength - (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, ...digest&gt;o1, ...sal); const esperadoDigest = hashFn(m,
    verdadeiro);

    return uint8ArrayEquals(resumo2, esperadoResumo);
}

```

O exemplo23 completo está disponível nos arquivos ps256.js, rsassa.js e pkcs.js. Os números de chave privada embutidos no arquivo testkey.js são do arquivo testkey.pem no diretório samples que acompanha este manual. A chave pública correspondente está no arquivo pubtestkey.pem. Para obter ajuda na criação de chaves, consulte o exemplo [RS256](#).

²³<https://github.com/auth0/jwt-handbook-samples>

7.2.3 Curva Elíptica

Os algoritmos de Curva Elíptica (EC), assim como o RSA, dependem de uma classe de problemas matemáticos que são intratáveis em 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 fatoração²⁴ (encontrar os fatores primos de um grande número coprimo), os algoritmos de curva elíptica dependem da intratabilidade do problema de 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 para valores diferentes, obtemos as seguintes curvas de amostra:

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

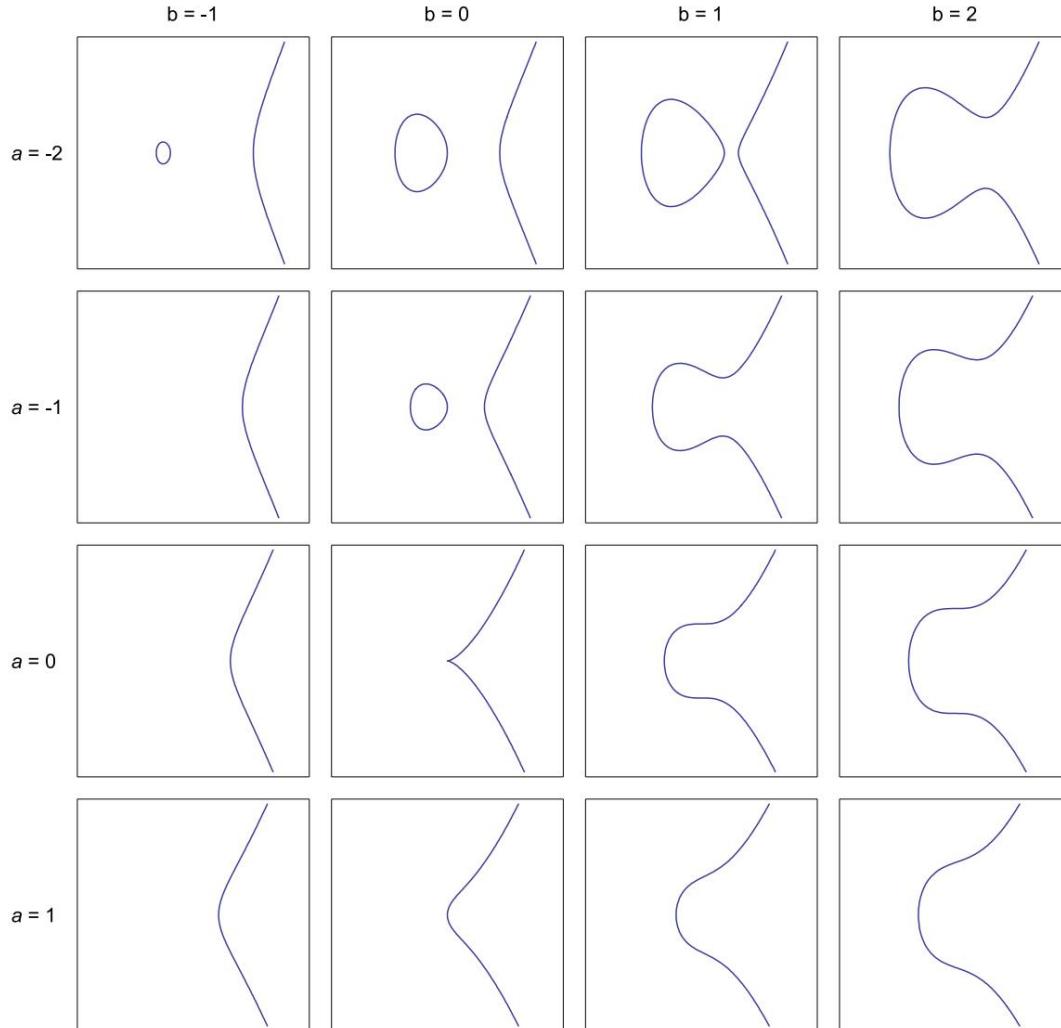


Figura 7.12: Algumas curvas elípticas

Imagen de domínio público retirada da Wikimedia.²⁵

Ao contrário do RSA, os 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 vamos nos concentrar neles.

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

“Finite” significa que a criptografia de curva elíptica funciona em conjuntos finitos de números, em vez do conjunto infinito de números reais.

²⁵<https://commons.wikimedia.org/wiki/File:EllipticCurveCatalog.svg>

Um campo primo é um corpo 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 realizar as operações matemáticas mudam. Em particular, o logaritmo discreto²⁶ deve ser usado em vez do logaritmo comum. Os logaritmos encontram o valor de k para expressões da seguinte forma:

$$\begin{aligned} a^k &= c \\ \log(a) &= k \end{aligned}$$

Figura 7.13: Função exponencial

Não há nenhum 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 criptografia assimétrica segura e operações de assinatura.

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. Os algoritmos de curva elíptica foram alvo de ataques anteriores devido ao uso indevido²⁷.

Um aspecto interessante da criptografia de curva elíptica é que os tamanhos de chave podem ser menores enquanto fornecem um nível de segurança semelhante em comparação com chaves maiores usadas em 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 de Curvas Elípticas

Para fins de implementação de assinaturas de curvas elípticas, é necessário implementar a aritmética de curvas elípticas. 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

²⁶ https://en.wikipedia.org/wiki/Discrete_logarithm
²⁷ <https://safecurves.cr.yp.to/>

$$\begin{aligned}
 & P+Q \circ R \pmod{q} \quad (P \circ Q) \\
 & (x_{p \circ v \circ c \circ p}) + (x_q, y_q) \circ (x_r, y_r) \pmod{q} \\
 & \quad \quad \quad \frac{y \circ y \circ y \circ p}{x \circ q \circ x \circ p} \pmod{q} \\
 & \quad \quad \quad x \circ r \circ y \circ 2 \circ y \circ x \circ p \circ y \circ x \circ q \pmod{q} \quad (x \\
 & \quad \quad \quad a \circ n \circ o \circ y \circ p \circ y \circ x \circ r \circ y \circ y \pmod{q})
 \end{aligned}$$

Figura 7.14: Adição de ponto

7.2.3.1.2 Duplicação de pontos

$$\begin{aligned}
 & P+Q \circ R \pmod{q} \quad (P=Q) \\
 & 2 \circ P \circ R \pmod{q} + \\
 & (x_{p \circ v \circ c \circ p}) \circ (x_{p \circ v \circ c \circ p}) \pmod{q} \\
 & \quad \quad \quad \frac{3 \circ x_{p \circ v \circ c \circ p}^2 + a}{2 \circ a \circ n \circ o \circ p} \pmod{q} \\
 & \quad \quad \quad x \circ r \circ y \circ 2 \circ 2 \circ x_{p \circ v \circ c \circ p} \pmod{q} \\
 & \quad \quad \quad a \circ n \circ o \circ y \circ (x \circ p \circ y \circ x \circ r \circ y \circ y_{p \circ v \circ c \circ p}) \pmod{q}
 \end{aligned}$$

Figura 7.15: Duplicação de pontos

7.2.3.1.3 Multiplicação escalar

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

$$\begin{aligned}
 & kP \circ R \pmod{q} \\
 & k = k_0 + 2k_1 + 2^2 k_2 + \dots + 2^m k_m \text{ onde } [k_0 \dots k_m] \in \{0,1\}
 \end{aligned}$$

Figura 7.16: Multiplicação escalar

Em seguida, o seguinte algoritmo é aplicado:

1. Seja N o ponto P.
2. Seja Q o ponto no infinito (0, 0).
3. Para i de 0 a m faça:

1. Se $k \sim i \sim 1$, então seja Q o resultado da adição de Q a N (adição de curva elíptica).
2. Seja N o resultado da duplicação de N (duplicação da curva elíptica).
4. Retorno Q .

Exemplo de implementação em JavaScript:

```
function ecMultiply(P, k, modulus) { let N =
  Object.assign({}, p); deixe 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 = ecDuplo(N, módulo);
}

retornar Q;
}
```

Uma coisa a notar 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 amostras²⁸ no arquivo ecdsa.js. Essas implementações ingênuas, embora funcionais, são vulneráveis a ataques de temporização. As implementações prontas para produção usam diferentes algoritmos que levam em consideração esses ataques.

7.2.3.2 Algoritmo de assinatura digital de curva elíptica (ECDSA)

O Elliptic-Curve Digital Signature Algorithm (ECDSA) foi desenvolvido por um comitê do American National Standards Institute (ANSI)²⁹. O padrão é X9.6330. A norma especifica todos os parâmetros necessários para o uso adequado de curvas elípticas para assinaturas de forma segura. A especificação JWA depende dessa especificação (e FIPS 186-431) para escolher os parâmetros da curva e especificar o algoritmo.

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

```
função padrão de exportação jwtEncode(header, payload, privateKey) {
  if(typeof header != 'object' || typeof payload != 'object') {

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

```

        throw new Error('header e payload devem ser objetos');
    }

cabeçalho.alg = 'ES256';

const encHeader = b64(JSON.stringify(cabeçalho)); 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); assinatura const =
b64(Uint8Array.of(...ecR, ...ecS));

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

```

Esse código é semelhante ao usado para assinaturas RSA e HMAC. A principal diferença está 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:

```

função de exportação jwtVerifyAndDecode(jwt, publicKey) { cabeçalho
    const = JSON.parse(unb64(split[0])); if(header.alg != 'ES256')
    { throw new Error('Algoritmo errado: ${header.alg}');

}

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

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

};

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

return
{ cabeçalho:
cabeçalho, carga útil: JSON.parse(unb64(split[1])),

```

```

    válido: válido
};

}

```

Novamente, o procedimento para verificar a validade da assinatura é semelhante ao RSA e ao HMAC. Nesse caso, os valores r e s devem 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 do Domínio da Curva Elíptica

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

- p ou q: o primo usado para definir o campo primo32 no qual as operações aritméticas são realizadas.
Operações de campo primário usam aritmética modular³².
- a: coeficiente de x na equação da curva. • b:
constante na equação da curva (intersecção y). • G: um ponto de curva válido usado como ponto base para operações de curva elíptica. O ponto base é usado em operações aritméticas para obter outros pontos na curva. • n: a ordem do ponto base G. Este parâmetro é o número de pontos válidos na curva que pode ser construído 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 consideradas válidas: P-256, P-384 e P-521. Essas curvas são definidas no FIPS 186-434 e outros padrões associados.

Para nosso exemplo de código, usaremos a curva P-256:

```

const p256 = { q:
  bigint('0xffffffff00000001000000000000' +
    '000000000000ffffffffff00000000ffff' +
    'ffff', 16), // ordem

  do ponto base n:
  bigint('115792089210356248762697446949407573529996955224135760342' +
    '422259061068512044369'), // ponto base

  G:
  { x: bigint('6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0' + 'f4a13945d898c296',
    16), y: bigint('4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ece' +
    'cbb6406837bf51f5', 16)

  }, //a: bigint(-3) a:
  bigint('00ffffffff00000001000000000000' +
    '000000000000ffffffffff0000ffff' +

```

32 https://en.wikipedia.org/wiki/Finite_field

33https://en.wikipedia.org/wiki/Modular_arithmetic

34<http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>

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

7.2.3.2.2 Chaves Públicas e Privadas

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

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

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

É isso! Tão simples como isso.

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

```
// ecMultiply é a operação de multiplicação escalar de 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.

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

Figura 7.17: Chave privada como o logaritmo da chave pública

No contexto de um grupo aditivo³⁵, como o campo principal escolhido para curvas elípticas, computar k é o problema do logaritmo discreto. Não há algoritmo de propósito geral conhecido que possa computar isso de forma eficiente. Para números de 256 bits como os usados para a curva P-256, a complexidade está muito 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 aritméticas e curvas elípticas modulares:

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

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

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

Figura 7.18: s

7. Se s for zero, repita os passos 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).

Exemplo de implementação:

```
função de exportação sign(privateKey, hashFn, hashType, message) { if(hashType != hashTypes.sha256) { throw new Error('tipo de hash não suportado'); }

}

// Algoritmo conforme descrito em ANS X9.62-1998, 5.3

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

deixe
r;
vamos ;
    faça
    { deixe
        k; do { // Aviso: use um RNG seguro aqui k =
            bigInt.randBetween(1, p256.nMin1); ponto const =
            ecMultiply(p256.G, k, p256.q); r = ponto.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());

retornar {
```

```

    r: r,
    s: s
};

}

```

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

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

Exemplo de implementação:

```

função de exportação Verify(publicKey, hashFn, hashType, mensagem, assinatura) {
    if(hashType != hashTypes.sha256) { throw new
        Error('tipo de hash não suportado');
    }

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

    // Verifica se a chave pública é um ponto de curva válido if(isValidPoint(publicKey.Q))
    { return false;

    }

    // Algoritmo conforme descrito em ANS X9.62-1998, 5.4

    const e = BigInt(hashFn(mensagem), 16);

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

    const pointA = ecMultiply(p256.G, u1, p256.q); ponto constB =
        ecMultiply (publicKey.Q, u2, p256.q); ponto const = ecAdd(pontoA, pontoB,
        p256.q);

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

```

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

Uma parte importante do algoritmo que muitas vezes é negligenciada é a verificação da validade da chave pública. Esta tem sido uma fonte de ataques no passado³⁶. Se um invasor controlar a chave pública que uma parte verificadora usa para validação da assinatura da mensagem e a chave pública não for validada como um ponto na curva, o invasor pode criar uma chave pública especial que pode ser usada para vazar informações para o atacante. Isso é particularmente importante à luz dos principais protocolos de acordo, alguns dos quais são usados para criptografar JWTs.

Essa implementação de amostra pode ser encontrada no repositório de amostras³⁷ no arquivo ecdsa.js.

7.3 Atualizações Futuras

A especificação JWA tem muito mais algoritmos. Em versões futuras deste manual, examinaremos os algoritmos restantes.

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

Capítulo 8

Anexo A. Melhores Práticas Atuais

Desde seu lançamento, os JWTs têm sido usados em muitos lugares diferentes. Isso expôs JWTs e implementações de biblioteca a vários ataques. Mencionamos alguns deles nos capítulos anteriores. Nesta seção, veremos as práticas recomendadas atuais para trabalhar com JWTs.

Esta seção é baseada no rascunho para JWT Best Current Practices¹ do IETF OAuth Working Group². A versão da minuta utilizada nesta seção é 00, datada de 19 de julho de 2017³.

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, e não ao design, de JSON Web Tokens. Isso não os torna menos críticos. É discutível se alguns desses ataques poderiam ser mitigados ou removidos alterando o design subjacente. No momento, a especificação e o formato do JWT são imutáveis, portanto, a maioria das alterações ocorre no espaço de implementação (alterações em bibliotecas, APIs, práticas e convenções de programação).

Também é importante ter uma ideia básica da representação mais comum para JWTs: o formato [JWS Compact Serialization](#). Os JWTs não serializados têm dois objetos JSON principais: o cabeçalho e a carga útil.

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

O objeto de carga útil contém todas as informações relevantes transportadas pelo token. Existem várias declarações padrão, como sub (assunto) ou iat (emitido em), mas qualquer declaração personalizada pode ser incluída como parte da carga útil.

¹<https://tools.ietf.org/wg/oauth/draft-ietf-oauth-jwt-bcp/>

²<https://tools.ietf.org/wg/oauth/> ³<https://tools.ietf.org/html/draft-ietf-oauth-jwt-bcp-00>

Esses objetos são codificados usando o formato JWS Compact Serialization para produzir algo assim:

```
eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9.  
eyJzdWliOlxMjM0NTY3ODkwliwbtZSI6Ikpvag4gRG9lliwiaWF0ljoxNTE2MjM5MDlyfQ.  
XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

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

[Cabeçalho codificado em URL Base64].[Carga útil codificada em URL Base64].[Assinatura]

Isso se aplica apenas 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 jogar com JWTs e ver como eles são codificados/decodificados, verifique [JWT.io](https://jwt.io) .

8.1.1 Ataque “alg: nenhum”

Como mencionamos anteriormente, os JWTs carregam dois objetos JSON com informações importantes, o cabeçalho e o payload. O cabeçalho inclui informações sobre o algoritmo usado pelo JWT para assinar ou criptografar os dados nele contidos. Os JWTs assinados assinam o cabeçalho e a carga útil, enquanto os JWTs criptografados criptografam apenas a carga útil (o cabeçalho sempre deve ser legível).

No caso de tokens assinados, embora a assinatura proteja o cabeçalho e o payload contra adulteração, é possível reescrever o JWT sem usar a assinatura e alterar os dados contidos nela. Como é que isso funciona?

Tomemos, por exemplo, um JWT com um determinado cabeçalho e carga útil. Algo assim:

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

Agora, digamos que você codifique esse token em seu formato serializado compacto com uma assinatura e uma chave de assinatura de valor “secret”. Podemos usar o [JWT.io](https://jwt.io)5 para isso. O resultado é (novas linhas inseridas para facilitar a leitura):

```
eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9.  
eyJzdWliOjzb2UiLCJyb2xlioidXNlcj9. vqf3WzGLAxHW-  
X7UP-co3bU_ISUDvVjF2MKtLtsU1kzU
```

Vá em frente e cole isso em [JWT.io](https://jwt.io)⁶ .

4<https://jwt.io>
5<https://jwt.io>
6<https://jwt.io>

Agora, como este é um token assinado, podemos lê-lo. Isso também significa que poderíamos construir um token semelhante com dados ligeiramente alterados, embora, nesse caso, não pudéssemos assiná-lo, a menos que soubéssemos a chave de assinatura. Digamos que um invasor não conheça 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", type:  
  "JWT" }, payload:  
{ sub: "joe" role:  
"admin"  
}
```

Codificado (novas linhas inseridas para facilitar a leitura):

```
eyJhbGciOiJub25lIwidHlwIjoiSl0UIn0.eyJzdWliOiJqb2UiLCJyb2xlljoiYWRtaW4ifQ.
```

Observe que esse token não inclui uma assinatura ("alg": "none") e que a declaração de função da carga útil foi alterada. Se o invasor conseguir usar esse token com sucesso, ele poderá realizar um ataque de escalada de privilégio! Por que um ataque como esse funcionaria? Vamos dar uma olhada em como alguma biblioteca JWT hipotética poderia funcionar. Digamos que temos uma função de decodificação semelhante a esta:

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

Essa função pega um token codificado e um segredo e tenta verificar o token e, em seguida, retornar os dados decodificados nele. Se a verificação falhar, ela lançará uma exceção. Para escolher o algoritmo certo para verificação, a função depende da declaração alg do cabeçalho. É aqui que o ataque é bem-sucedido. No passado⁷, muitas bibliotecas dependiam dessa declaração para escolher o algoritmo de verificação e, como você deve ter adivinhado, em nosso token malicioso, a declaração alg é nenhuma. Isso significa que não há algoritmo de verificação e a etapa de verificação sempre é bem-sucedida.

Como você pode ver, este é um exemplo clássico de ataque que depende de uma certa ambigüidade da API de uma biblioteca específica, ao invés de uma vulnerabilidade na própria especificação. Mesmo assim, este é um ataque real que foi possível em várias implementações diferentes no passado. Por esse motivo, muitas bibliotecas relatam hoje os tokens "alg": "none" como inválidos, mesmo que não haja assinatura.

Existem outras mitigações possíveis para esse tipo de ataque, sendo a mais importante sempre verificar o algoritmo especificado no cabeçalho antes de tentar verificar um token. Outra é usar bibliotecas que requerem o algoritmo de verificação como uma entrada para a função de verificação, em vez de confiar na declaração alg.

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

8.1.2 Chave pública RS256 como ataque secreto HS256

Esse ataque é semelhante ao ataque "alg": "none" e também depende da ambiguidade na API de certas bibliotecas JWT. Nosso token de amostra será semelhante ao desse 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 é decodificado e retornado, caso contrário, uma exceção é lançada. Nesse caso, porém, a função também aceita uma chave pública como segundo parâmetro. De certa forma, isso faz sentido: tanto a chave pública quanto o segredo compartilhado geralmente são strings ou matrizes de bytes, portanto, do ponto de vista dos tipos necessários para esse argumento de função, um único argumento pode representar tanto uma chave pública (para RS, algoritmos 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 obtenha um token codificado assinado com um par de chaves RSA. Fica assim (novas linhas inseridas para facilitar a leitura):

```
eyJhbGciOiJSUzI1NilsInR5cCl6IkpxXVCJ9.eyJzdWliOiJqb2UiLCJyb2xIjjoidXNIciJ9.

ODjcv11Kcb69THVLKMErYqzy9htWICDtBdonVR5SX4geZa_R8StjwUuuskvveUsdjVgjgXwMso7p uAJZzoE9LEr9XCxau7SF1ddws4ONiqxSVXZbO0pSgbKm3FpkVz4Jyy4oNTs blpsyE0xTbO0pSgbKm3FpkVz4Jyy4oNTs blpsyE0xTbO0pSgbKm3FpkVz4Jyy4oNTs blpsyE0xTbO0pSgbKm3FpkVz4Jyy4oNTs blpsyE0xTb5MsBlpsyE0xTb5MsblpsyE0xTf8Ms
```

Decodificado:

```
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 poderia fazer uma chamada para nossa função `jwtDecode` hipotética de antes assim:

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

Mas aqui está o problema: a chave pública é, como o nome indica, geralmente pública. O invasor pode colocar as mãos nele, e tudo bem. Mas e se o invasor criasse um novo token usando o seguinte esquema? Primeiro, o invasor modifica o cabeçalho e escolhe HS256 como o algoritmo de assinatura:

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

Em seguida, ele escala as permissões alterando a reivindicação de função na carga útil:

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

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

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

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

Agora 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 da declaração alg do cabeçalho para escolher o algoritmo de verificação para o JWT. E o atacante mudou isso:

```
header:  
  { "alg": "HS256", // <-- alterado pelo invasor de RS256 "typ": "JWT"  
  }
```

Assim como no caso "alg": "none", confiar na declaração alg combinada com uma API ruim ou confusa pode resultar em um ataque bem-sucedido por um usuário mal-intencionado.

Mitigações contra esse ataque incluem passar um algoritmo explícito para a função jwtDecode, verificar a declaração alg ou usar APIs que separam algoritmos de chave pública de algoritmos de segredo compartilhado.

8.1.3 Chaves HMAC fracas

Os algoritmos HMAC dependem de um segredo compartilhado para produzir e verificar assinaturas. Algumas pessoas assumem que segredos compartilhados são semelhantes a senhas e, de certa forma, são: eles devem ser mantidos em segredo. No entanto, 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. Isso é uma consequência dos algoritmos de hash usados para armazenar senhas (junto com um sal) que impedem ataques de força bruta em prazos razoáveis.

Por outro lado, os segredos compartilhados do HMAC, conforme usados pelos JWTs, são otimizados para velocidade. Isso permite que muitas operações de assinatura/verificação sejam executadas com eficiência, mas torna os ataques de força bruta mais fáceis⁸. Portanto, o comprimento do segredo compartilhado para HS256/384/512 é de extrema importância. Na verdade, JSON Web Algorithms⁹ define o comprimento mínimo da chave 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 DEVE ser usada com este algoritmo." - Algoritmos Web JSON (RFC 7518), 3.2 HMAC com SHA-2 Functions¹⁰

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 por humanos, 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, foi demonstrado que os ataques de força bruta para HS256 são simples o suficiente para executar¹¹ se o segredo compartilhado for muito curto.

8.1.4 Criptografia Empilhada Incorreta + Suposições de Verificação de Assinatura

As assinaturas fornecem proteção contra adulteração. Ou seja, embora não impeçam a leitura dos dados, eles os tornam imutáveis: qualquer alteração nos dados resulta 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, a criptografia pode ser necessária. Os JWTs suportam ambos: assinaturas e criptografia.

É muito comum presumir erroneamente que a criptografia também fornece proteção contra adulteração em todos os casos. A justificativa para essa suposição geralmente é algo como: "se os dados não podem ser lidos, como um invasor poderia 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 sairá do processo de descriptografia. A modificação cega de 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 uma carga JWT semelhante a esta:

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

Como podemos ver aqui, a declaração admin é simplesmente um booleano. Se um invasor conseguir produzir uma alteração nos dados descriptografados que resulte na inversão desse valor booleano, ele poderá

⁸<https://auth0.com/blog/brute-forcing-hs256-is-possible-the-importance-of-using-strong-keys-to-sign-jwts/>

⁹<https://tools.ietf.org/html/rfc7518> ¹⁰<https://tools.ietf.org/html/rfc7518#section-3.2> ¹¹<https://auth0.com/blog/brute-forcing-hs256-is-possible-a-importancia-de-usar-chaves-fortes-para-assinar-jwts/>

executar com sucesso um ataque de escalação de privilégios. Em particular, os invasores que têm amplas janelas de tempo para realizar ataques podem tentar quantas alterações quiserem nos dados criptografados, sem que o sistema descarte o token como inválido antes de processá-lo. Outros ataques podem envolver a alimentação de dados inválidos para subsistemas que esperam que os dados já estejam sanitizados naquele ponto, desencadeando bugs, falhas ou servindo como ponto de entrada para outros tipos de ataques.

Por esse motivo, JSON Web Algorithms¹² define apenas algoritmos de criptografia que também incluem verificação de integridade de dados. Em outras palavras, desde que o algoritmo de criptografia seja um dos algoritmos sancionados pelo JWA¹³, 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, deverá garantir que a integridade dos dados seja fornecida por esse algoritmo ou precisará aninhar JWTs, usando um JWT assinado como o JWT mais interno para garantir a integridade dos dados.

JWTs¹⁴ 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 por meio de um sistema de terceiros que também usa JWTs.

Um erro comum nesses cenários está relacionado à validação do JWT aninhado. Para garantir que a integridade dos dados seja preservada e que os dados sejam decodificados adequadamente, todas as camadas dos JWTs devem passar por todas as validações relacionadas aos algoritmos definidos em seus cabeçalhos. 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. Deixar de 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 associados a isso.

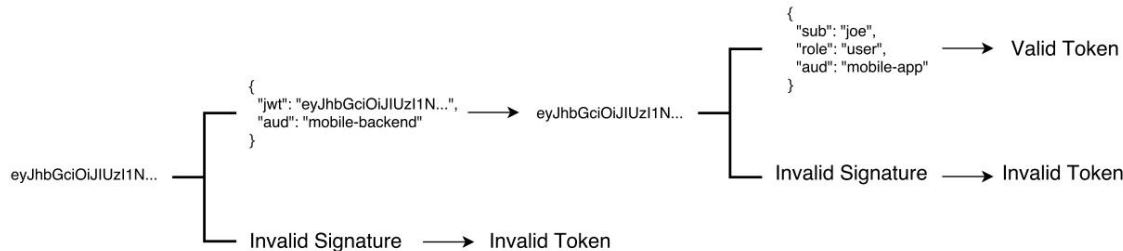


Figura 8.1: Validação do Nested JWT

8.1.5 Ataques de curva elíptica inválidos

A criptografia de curva elíptica é uma das famílias de algoritmos de chave pública suportadas por JSON Web Algorithms¹⁵. A criptografia de curva elíptica depende da intratabilidade do [problema de logaritmo discreto de curva elíptica](#), um problema matemático que não pode ser resolvido em tempos razoáveis para grandes

¹²<https://tools.ietf.org/html/rfc7518>

¹³<https://tools.ietf.org/html/rfc7518>

¹⁴<https://tools.ietf.org/html/rfc7519#section-2>

¹⁵<https://tools.ietf.org/html/rfc7518>

números suficientes. Esse problema impede a recuperação da chave privada de uma chave pública, uma mensagem criptografada e seu texto sem formatação. Quando comparadas ao RSA, outro algoritmo de chave pública que também é compatível com JSON Web Algorithms, as curvas elípticas fornecem um nível semelhante de força, embora requeiram chaves menores.

As curvas elípticas, necessárias para operações criptográficas, são definidas sobre 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 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 forma que pontos inválidos simplesmente não são possíveis.

Se um ponto inválido for produzido, há um erro nas entradas das operações. As principais operações aritméticas em curvas elípticas são:

- **Adição de pontos:** somando dois pontos na mesma curva resultando em um terceiro ponto na mesma curva.
- **Duplicação de pontos:** adicionar um ponto a si mesmo, resultando em um novo ponto na mesma curva. •
- Multiplicação escalar:** multiplicar um único ponto na curva por um número escalar, definido como adicionar repetidamente esse 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, falham em validar as entradas para eles. 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 forem. Esses resultados, quando usados no contexto de operações criptográficas como descriptografia, podem ser usados para recuperar a chave privada. Este ataque foi demonstrado no passado¹⁶. Essa classe de ataques é conhecida como ataques de curva inválida¹⁷.

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 válido de valores.

8.1.6 Ataques de Substituição

Os 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 daqueles para os quais foram destinados.

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

8.1.6.1 Destinatário Diferente

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

¹⁶<http://blogs.adobe.com/security/2017/03/critical-vulnerability-uncovered-in-json-encryption.html> ¹⁷<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.3920&rep=rep1&type=pdf>

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

Esse 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 administrador. No entanto, há um problema com esse token: não há um destinatário pretendido ou mesmo um emissor nele. O que aconteceria se uma API diferente, diferente do destinatário pretendido para o qual este token foi emitido, usasse a assinatura como a única verificação de validade? Digamos que também haja um usuário joe no banco de dados para esse serviço ou API. O invasor pode enviar esse mesmo token para esse outro serviço e obter privilégios de administrador instantaneamente!

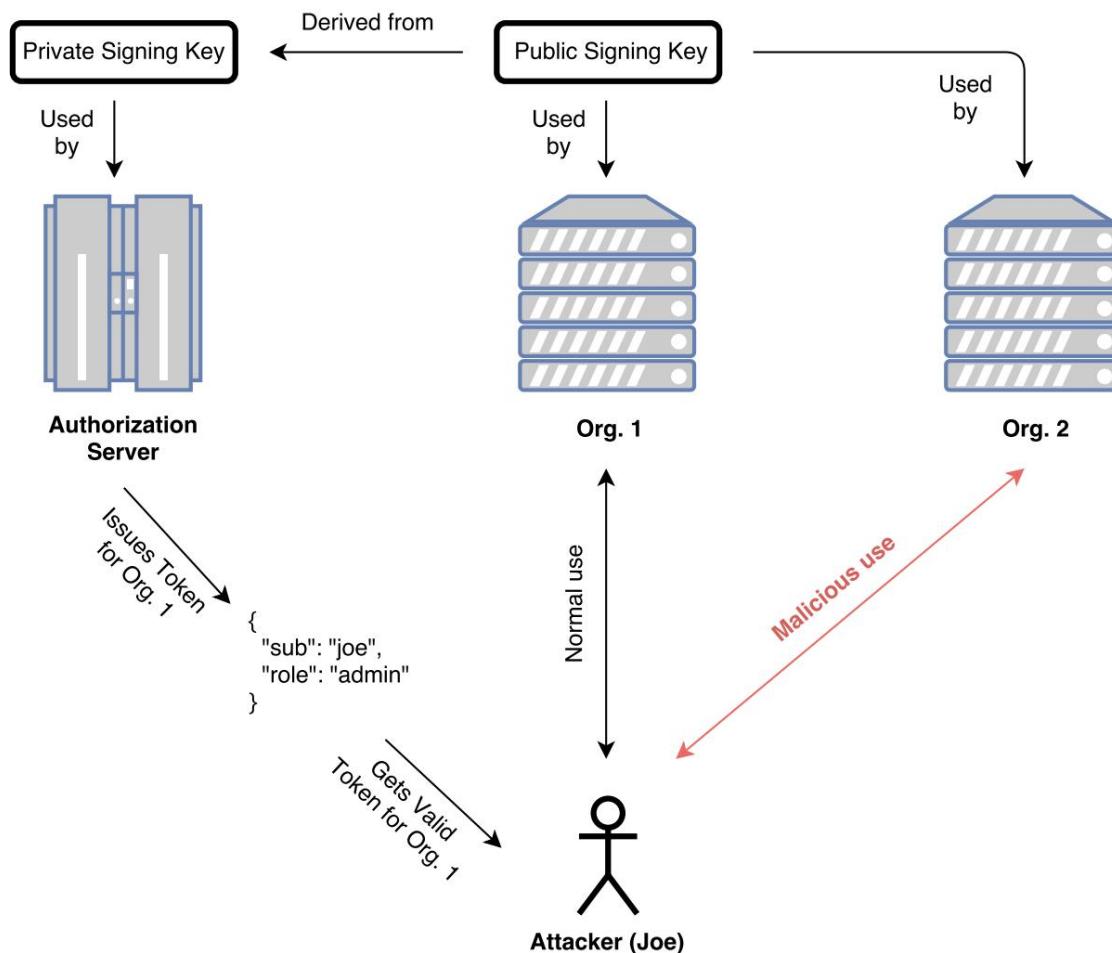


Figura 8.2: Ataque de substituição JWT de destinatário diferente

Para evitar esses ataques, a validação de token deve contar com chaves ou segredos exclusivos por serviço ou declarações específicas. Por exemplo, esse token pode incluir uma declaração aud especificando o público-alvo. Dessa forma, mesmo que a assinatura seja válida, o token não poderá ser utilizado em outros serviços que compartilhem o mesmo segredo ou chave de assinatura.

8.1.6.2 Mesmo Destinatário/JWT Cruzado

Este ataque é semelhante ao anterior, mas ao invés de contar com um token emitido para um destinatário diferente, neste caso, o destinatário é o mesmo. O que muda neste caso é que o atacante envia o token para um serviço diferente daquele a que se destina (dentro da mesma empresa ou prestador de serviço).

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 uma declaração de emissor (iss), uma declaração de público (aud) e uma declaração de permissão (perm). A API para a qual esse token foi emitido verifica todas essas declarações, mesmo que a assinatura do token seja válida. Dessa forma, mesmo que o invasor consiga colocar as mãos em um token assinado com a mesma chave privada ou segredo, ele não poderá utilizá-lo para operar nesse serviço se não for destinado a ele.

No entanto, a cool-company tem outros serviços públicos. Um desses serviços, o serviço cool-company/item-database, foi atualizado recentemente para verificar reivindicações junto com a assinatura do token. No entanto, durante os upgrades, a equipe responsável por selecionar as reivindicações que seriam validadas cometeu um erro: não validaram corretamente a reivindicação aud. Em vez de verificar uma correspondência exata, eles decidiram verificar a presença da string de empresa legal. 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 de banco de dados do usuário no local para o token do serviço de banco de dados de itens. Isso concederia ao invasor permissões de gravação no banco de dados de itens, quando ele deveria ter permissões de gravação apenas no banco de dados do usuário!

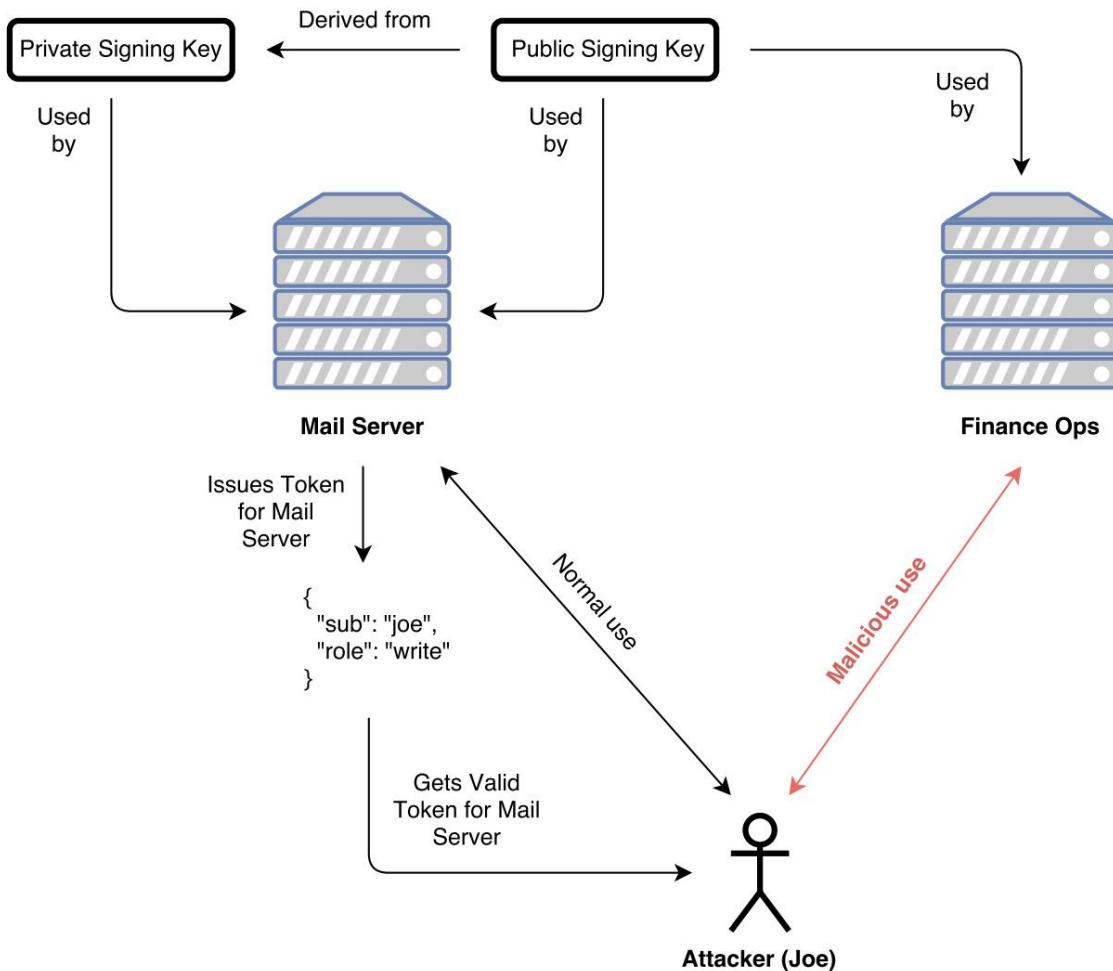


Figura 8.3: Ataque de substituição JWT do mesmo destinatário

8.2 Mitigações e Melhores Práticas

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

8.2.1 Sempre Realizar Verificação de Algoritmo

O ataque "alg": "none" e o ataque "chave pública RS256 como segredo compartilhado HS256" podem ser evitados por essa mitigação. Toda vez que um JWT deve ser validado, o algoritmo deve ser explicitamente

selecionado para impedir que os invasores tenham controle. As bibliotecas costumavam confiar no cabeçalho alg para selecionar o algoritmo para validação. A partir do momento em que ataques como esses foram vistos à solta¹⁸, as bibliotecas passaram a fornecer pelo menos a opção de especificar explicitamente os algoritmos selecionados para validação, desconsiderando o que é especificado no cabeçalho. Ainda assim, algumas bibliotecas fornecem a opção de usar o que for especificado no cabeçalho, portanto, os desenvolvedores devem tomar cuidado para sempre usar a seleção explícita de algoritmos.

8.2.2 Use Algoritmos Apropriados

Embora a especificação JSON Web Algorithms declare uma série de algoritmos recomendados e necessários, escolher o correto para um cenário específico ainda depende dos usuários. Por exemplo, um JWT assinado com uma assinatura HMAC pode ser suficiente para armazenar um pequeno token de seu aplicativo da Web de servidor único e página única no navegador de um usuário. Em contraste, um algoritmo de segredo compartilhado seria extremamente inconveniente em um cenário de identidade federada.

Outra maneira de pensar sobre isso é considerar todos os JWTs inválidos, a menos que o algoritmo de validação seja aceitável para o aplicativo. Em outras palavras, mesmo que o validador 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 a aplicação. Essa também é outra forma de dizer o que mencionamos em nossa recomendação anterior: sempre realizar a verificação do algoritmo.

8.2.3 Sempre realizar todas as validações

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

8.2.4 Sempre Valide as Entradas Criptográficas

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

No caso de operações de curva elíptica, nosso exemplo anterior, as bibliotecas devem sempre validar as chaves públicas antes de usá-las (ou seja, confirmar que elas representam um ponto válido na curva selecionada). Esses tipos de verificações são normalmente tratados pela biblioteca criptográfica subjacente. Os desenvolvedores devem certificar-se de que sua biblioteca de escolha executa essas validações ou devem adicionar o código necessário para executá-las no nível do aplicativo. Deixar de fazer isso pode resultar no comprometimento de sua(s) chave(s) privada(s).

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

8.2.5 Escolher Chaves Fortes

Embora essa recomendação se aplique a qualquer chave criptográfica, ela ainda é ignorada muitas vezes. Como mostramos acima, o comprimento mínimo necessário para os segredos compartilhados do HMAC geralmente é ignorado.

Mas mesmo que o segredo compartilhado fosse longo o suficiente, ele também deveria ser totalmente aleatório. Uma chave longa com um nível ruim de aleatoriedade (também conhecido como “entropia”) ainda pode ser forçada ou adivinhada. Para garantir que esse não seja o caso, as bibliotecas de geração de chaves devem contar com geradores de números pseudo-aleatórios (PRNGs) de qualidade criptográfica devidamente propagados durante a inicialização. Na melhor das hipóteses, um gerador de número de hardware pode ser usado.

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

8.2.6 Validar todas as reivindicações possíveis

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

8.2.7 Use a reivindicação typ para separar tipos de tokens

Embora na maioria das vezes a declaração de tipo tenha um único valor (JWT), ela também pode ser usada para separar diferentes tipos de JWTs específicos do aplicativo. Isso pode ser útil caso seu sistema precise lidar com muitos tipos diferentes de tokens.

Essa declaração também pode impedir o uso indevido de um token em um contexto diferente por meio de uma verificação de declaração adicional. O [padrão JWS permite explicitamente](#) valores específicos do aplicativo da declaração de tipo.

8.2.8 Usar Regras de Validação Diferentes para Cada Token

Esta prática resume muitas das que foram enumeradas antes. Para evitar ataques, é de fundamental importância garantir que cada token emitido tenha regras de validação muito claras e específicas.

Isso não significa apenas usar a declaração type quando apropriado, ou validar todas as declarações possíveis, como iss ou aud, mas também implica evitar a reutilização de chaves para diferentes tokens sempre que possível ou usar diferentes declarações personalizadas ou formatos de declaração. Dessa forma, os tokens destinados a serem usados em um único local 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 diferentes chaves privadas para cada subsistema de sua arquitetura. Você também pode fazer reivindicações

mais específico, especificando um determinado formato interno para eles. A declaração iss, por exemplo, poderia ser uma URL do subsistema que emitiu aquele token, ao invés do nome da empresa, dificultando sua reutilização.

8.3 Conclusão

JSON Web Tokens são uma ferramenta que faz uso de criptografia. Como todas as ferramentas que fazem isso, e especialmente aquelas usadas para lidar com informações confidenciais, elas devem ser usadas com cuidado. Sua aparente simplicidade pode confundir alguns desenvolvedores e fazê-los pensar que usar JWTs é apenas uma questão de escolher o segredo compartilhado correto ou o algoritmo de chave pública. Infelizmente, como vimos acima, esse não é o caso. É de extrema importância seguir as práticas recomendadas para cada ferramenta em sua caixa de ferramentas, e os JWTs não são exceção. Isso inclui escolher bibliotecas de alta qualidade testadas em batalha; validar reivindicações de carga útil e cabeçalho; escolher os algoritmos certos; certificando-se de que chaves fortes sejam geradas; atentar para as sutilezas de cada API; entre outras coisas. Se tudo isso parecer assustador, considere transferir alguns dos encargos para provedores externos. Auth0¹⁹ é um desses provedores. Se você não puder fazer isso, considere estas recomendações com cuidado e lembre-se: não role seu próprio crypto²⁰, confie em código testado e comprovado.

¹⁹<https://auth0.com>

²⁰<https://security.stackexchange.com/questions/18197/why-shouldnt-we-roll-our-own>