

MEI - MESTRADO EM ENGENHARIA DE INFORMÁTICA

Engenharia de Segurança

GRUPO 5 E 10



Carlos Carvalho PG47092



Beatriz Lacerda A89535



Tiago Freitas PG47687



João Rodrigues PG46534



Joana Sousa PG47282



Tiago Gomes PG47702

2 de Maio
2021/2022

PREFÁCIO

O seguinte relatório da Unidade Curricular Engenharia de Segurança, do curso de Engenharia Informática da Universidade do Minho, tem como objetivo a introdução de diversas especificações cada vez mais utilizadas no âmbito das técnicas criptográficas.

Para isso, será necessária uma pesquisa aprofundada não só de todas as especificações mas também todos os conceitos relacionados com a área das técnicas criptográficas. Este projeto irá incluir código de forma a visualizarmos todo o intuito deste trabalho desenvolvido em **Javascript/Node.js**, e como é também algo com que não estamos muito habituados a utilizar, será um desafio acrescido.

Assim sendo, além de darmos a conhecer todas estas especificações, pretendemos demonstrar a sua utilidade, quando deverão ser utilizadas e para que fins serão empregues.

JOSE

Conceito

A sigla **JOSE** tanto pode ser referida como *Javascript Object Signing and Encryption*, tal como *JSON Object Signing and Encryption*¹.

Foi proposto como um *standard* (pela comunidade e não pela IETF) desde maio de 2015 [RFC7520²], com o objetivo de ser uma solução alternativa e mais leve ao *XML Signature and Encryption*.

Esta solução foi projetada essencialmente para ambientes *web*, procurando promover transferências seguras entre agentes através de *URLs* seguros. Assim, dado este objetivo alvo, esta *framework* necessita de ser "leve" uma vez que foi projetada para ambientes com restrições de espaço, como é o caso maioritário de ambientes *web* (p.e.: cabeçalhos de autorização *HTTP*, parâmetros de queries para *URI*).

Além disso, esta *framework* também procura ser *self-contained*, de forma a ser facilmente utilizada em protocolos sem estado (como é o caso do *HTTP/1*, apesar deste apresentar algumas características com estado: *cookies*, *cache*, *TLS*), uma vez que toda a informação necessária estará dentro do respetivo *token*.

Assim, e mais concretamente, esta *framework* procura proporcionar uma série de métodos para assinaturas e cifragens de qualquer conteúdo, apesar de ter sido construído principalmente sobre *JSON* e *base64url* para facilmente ser usado em aplicações *web*.

Deste modo, trata-se de um *standard* assentado num aglomerado de *RFCs*, implicando uma constante evolução:

- **JWS** (RFC 7515³) - *JSON Web Signature*;
- **JWE** (RFC 7516⁴) - *JSON Web Encryption*;
- **JWK** (RFC 7517⁵) - *JSON Web Key*;
- **JWA** (RFC 7518⁶) - *JSON Web Algorithms*;
- **JWT** (RFC 7519⁷) - *JSON Web Token*;
- JOSE-Cookbook (RFC 7520) - com diversos exemplos de como proteger conteúdo utilizando *JOSE*;
- JWK-Thumbprint (RFC 7638⁸) - define um método para computacionar o valor de uma *hash* sobre um *JWK*;
- RFC 7797⁹ - *JSON Web Signature (JWS) Unencoded Payload Option*;
- RFC 8037¹⁰ - com a definição de como utilizar diferentes algoritmos no *JOSE*: algoritmos de Diffie-Hellman "X25519" e "X448"; algoritmos de assinaturas utilizando as curvas elípticas de *Edwards* "Ed25519" e "Ed448".

Os *standards* de *JWS*, *JWE*, *JWK*, *JWA* e *JWT* serão explorados de forma sucinta nas secções posteriores deste relatório, pelo que iremos começar por apresentar uma breve introdução¹¹):

¹<https://ldapwiki.com/wiki/Javascript%20Object%20Signing%20and%20Encryption>

²<https://datatracker.ietf.org/doc/rfc7520/>

³<https://datatracker.ietf.org/doc/rfc7515/>

⁴<https://datatracker.ietf.org/doc/rfc7516/>

⁵<https://datatracker.ietf.org/doc/rfc7517/>

⁶<https://datatracker.ietf.org/doc/rfc7518/>

⁷<https://datatracker.ietf.org/doc/rfc7519/>

⁸<https://datatracker.ietf.org/doc/rfc7638/>

⁹<https://datatracker.ietf.org/doc/rfc7797/>

¹⁰<https://datatracker.ietf.org/doc/rfc8037/>

¹¹fonte: https://www.youtube.com/watch?v=bW5pS4e_MX8

JSON Web Algorithm RFC 7518

"This specification registers cryptographic algorithms and identifiers to be used with the JWS, JWE, JWK specifications."

JSON Web Key RFC 7517

"A JSON Web Key (JWK) is a JavaScript Object Notation (JSON) data structure that represents a cryptographic key."

```
{
  "keys": [
    {
      "kty": "EC",
      "crv": "P-256",
      "x": "MKBCTN1cKUSDii1ly8s35261dZ8AiTo7Tu6KPAqv7D4",
      "y": "4Et16SRW2YiLurN5vfvHuhp7x8PxltmNWlbbM4IFyM",
      "use": "enc",
      "kid": "1"
    },
    {
      "kty": "RSA",
      "n": "0vx7agoebGcQGuuPiLJXZptN9nndrQmbXEpss2aiAFbNhM78LhNx
        4ccbFAAtVT86zuIRK7aPPFxuhDR116tsoC_BJECPebWKRXjBZC1F
        V4n3knjhMstn64tZ_2W-5JaCY4hc5n3yBXArw1931qt7_RN5w6Cf
        0h4QyQ5v-65YgjQR0_FDW2QvzqY368QOM1cAtasgzs8KJZgnYD9c7
        d0zgDAZHzu6qMqvRL5haJrn1n91cb0pbISD08qNLyrdkt-bFTWhAI
        4vMOPh6Nezu0EM41Fd2NcKwr3XPksINHaQ-G_xBniIqbw0Le1jF44
        -csFCur-kEgU8awapJzKngDKgw",
      "e": "AQAB",
      "alg": "RS256",
      "kid": "2011-04-29"
    }
  ]
}
```

Figura 3: Breve definição e simples exemplos de JWA e JWK.

JSON Web Signature RFC 7515

"JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures"

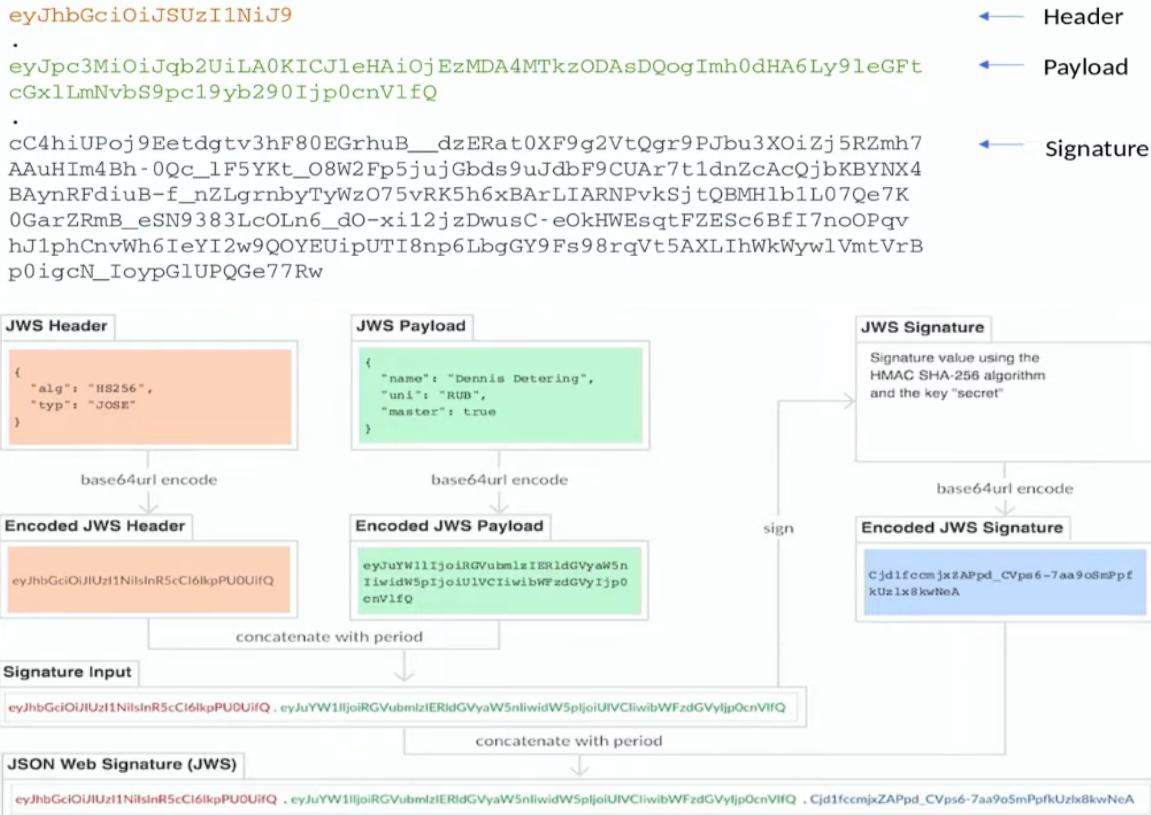


Figura 4: Breve definição e simples exemplo de construção de um JWS.

JSON Web Encryption RFC 7516

"JSON Web Encryption (JWE) represents encrypted content using JSON-based data structures."

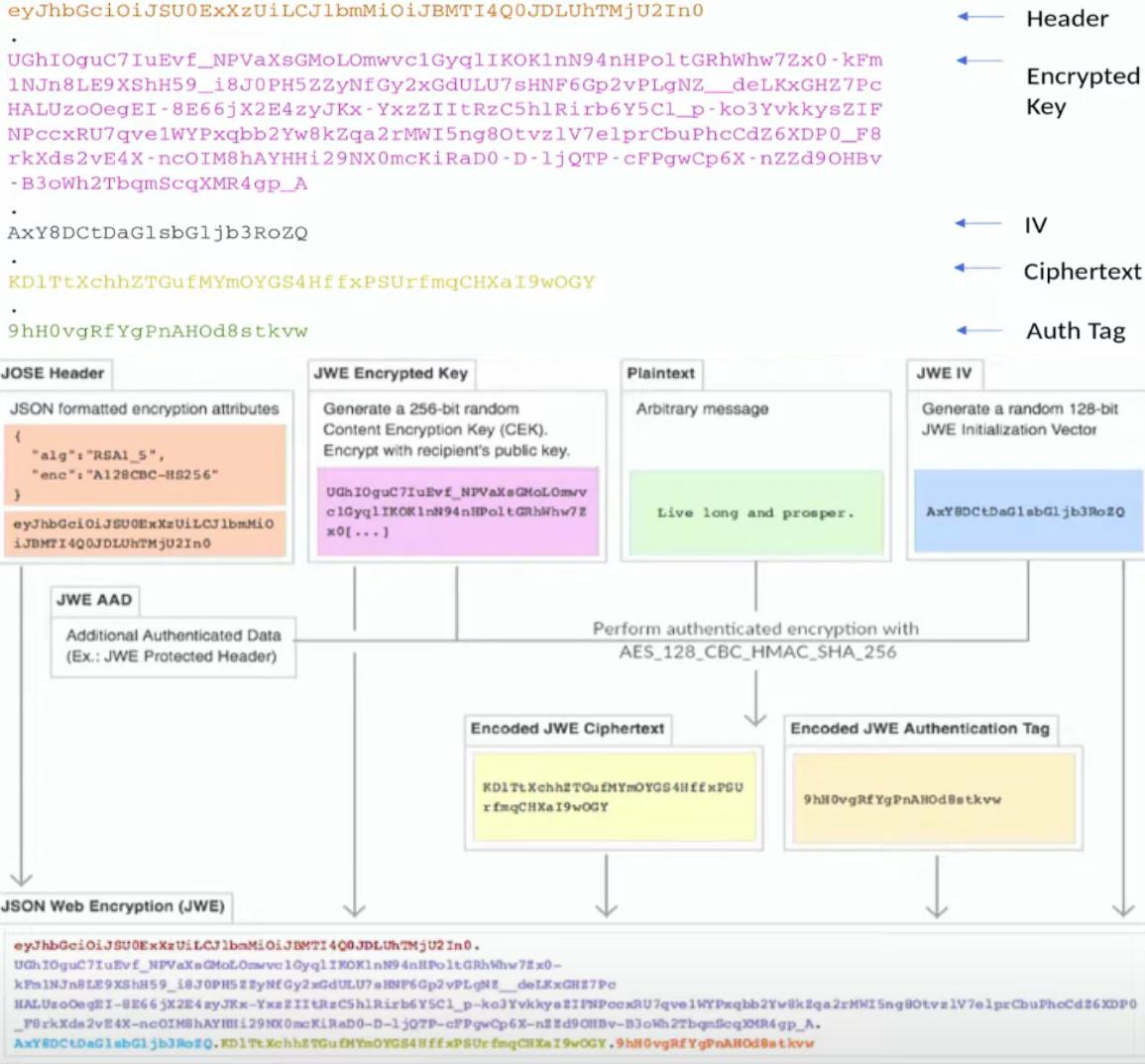


Figura 5: Breve definição e simples exemplo de construção de um JWE.

JSON Web Token RFC 7519

"... claims to be transferred between two parties [...] used as the payload of a JWS structure or as the plaintext of a JWE structure ..."

```
{
  "iss": "rub.de",
  "iat": 1478452995,
  "nbf": 1478452935,
  "exp": 1483228800,
  "username": "ddety",
  "is_admin": true
}
```

Figura 6: Breve definição e simples exemplo de um JWT.

Ataques e vulnerabilidades

Nesta secção iremos apresentar quatro diferentes ataques possíveis a algumas vulnerabilidades que estiveram expostas nesta *framework*. Para tal, iremos utilizar a apresentação de *Dennis Detering*, já referenciada anteriormente¹², que aprofunda mais concretamente estes ataques.

1 Exclusão de Assinatura

Durante a fase de especificação dos algoritmos a utilizar, existiu um parâmetro opcional que permitia não especificar nenhum algoritmo em concreto (em casos que a proteção seria feita por diferentes meios):

"alg" Param Value	Digital Signature or MAC Algorithm	Implementation Requirements
HS256	HMAC using SHA-256	Required
HS384	HMAC using SHA-384	Optional
HS512	HMAC using SHA-512	Optional
RS256	RSASSA-PKCS1-v1_5 using SHA-256	Recommended
RS384	RSASSA-PKCS1-v1_5 using SHA-384	Optional
RS512	RSASSA-PKCS1-v1_5 using SHA-512	Optional
ES256	ECDSA using P-256 and SHA-256	Recommended+
ES384	ECDSA using P-384 and SHA-384	Optional
ES512	ECDSA using P-521 and SHA-512	Optional
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256	Optional
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384	Optional
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512	Optional
none	No digital signature or MAC performed	Optional

Figura 7: Parâmetro *none* na especificação de algoritmos.

O problema desta decisão é que existiram imensas aplicações que não verificavam a existência deste parâmetro, permitindo aos atacantes uma forma de descartar a assinatura aplicando este parâmetro, levando às aplicações a validar a assinatura como estando correta.

Trata-se de um exemplo de um ataque já resolvido, porém, foi uma vulnerabilidade exposta em diversas bibliotecas durante um tempo considerável.

2 Confusão de Chaves

Agora na fase de verificação das assinaturas, existe a questão sobre qual o tipo de verificação a utilizar: *RSA* ou *HMAC*? De uma forma geral em todas imensas bibliotecas, esta especificação foi parametrizada como:

```
verify(string token, string verificationKey)
```

No caso de se aplicar *HMAC*, o segundo parâmetro tratar-se-ia da respetiva chave privada. Já no caso de se aplicar *RSA*, tratar-se-ia da sua respetiva chave pública.

¹²fonte: https://www.youtube.com/watch?v=bW5pS4e_MX8

E como é que o sistema teria de verificar qual mecanismo a utilizar? Através do respetivo cabeçalho da especificação do *JOSE*. O primeiro problema é exatamente neste ponto, uma vez que o cabeçalho é controlado pelo utilizador. Assim, o atacante poderia fornecer uma chave privada ao sistema, pensando este que se trataria de uma chave pública, ou vice-versa. Como teriam sido geradas de forma diferente, a verificação das chaves seria positiva, permitindo ultrapassar os mecanismos de assinaturas do sistema.

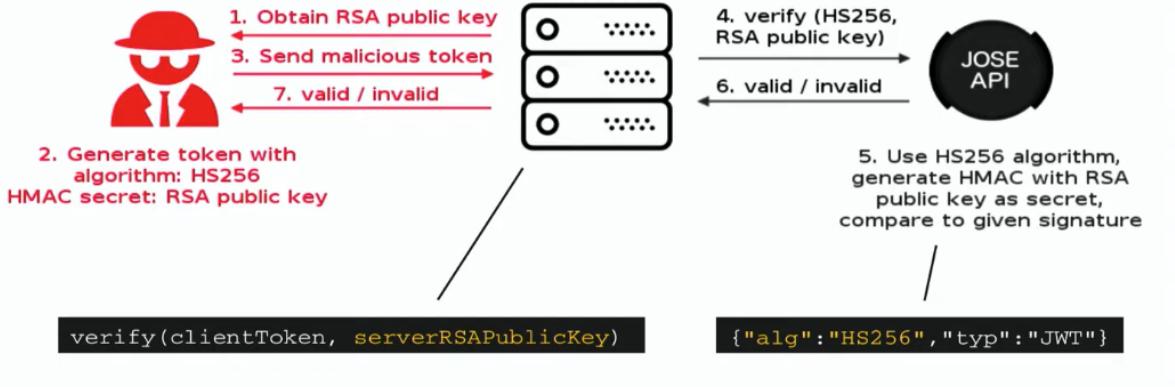


Figura 8: Confusão de chaves para os sistemas.

3 Bleichenbacher MMA

Este ataque é deveras complexo, procurando detetar vulnerabilidades num sistema (analisando tempos de resposta, por exemplo) durante uma fase de inundação de *ciphertexts*.

- Described in 1998 by Daniel Bleichenbacher
- Used to break:
 - SSL (1998)
 - XML Encryption (2012)
 - PKCS#11 (2012)
 - TLS (2016)
- Recommended algorithm in JSON Web Encryption

"alg" Param Value	Key Management Algorithm	More Header Params	Implementation Requirements
RSA1_5	RSAES-PKCS1-v1_5	(none)	Recommended-
RSA-OAEP	RSAES OAEP using default parameters	(none)	Recommended+
RSA-OAEP-256	RSAES OAEP using SHA-2 56 and MGF1 with SHA-256	(none)	Optional
A128KW	AES Key Wrap with default initial value using 128-bit key	(none)	Recommended
A192KW	AES Key Wrap with default initial value using 192-bit key	(none)	Optional
A256KW	AES Key Wrap with default initial value using 256-bit key	(none)	Recommended
dir	Direct use of shared symmetric key as the CEK	(none)	Recommended
ECDH-ES	Elliptic Curve Diffie-Hellman Ephemeral Static key agreement using Concat KDF	"epk", "apu", "apv"	Recommended+
ECDH-ES+A128KW	ECDH-ES using Concat KDF and CEK wrapped with	"epk", "apu", "apv"	Recommended

- Error- and time-based padding oracle

```
throw new JOSE_Exception_DecryptionFailed('Master key encryption failed');
throw new JOSE_Exception_DecryptionFailed('Encryption/Mac key derivation failed');
throw new JOSE_Exception_DecryptionFailed('Payload encryption failed');
throw new JOSE_Exception_UnexpectedAlgorithm('Algorithm not supported');
throw new JOSE_Exception_UnexpectedAlgorithm('Invalid authentication tag');
```

- Attack scenario

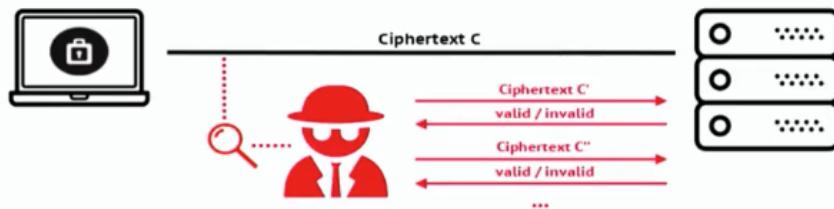


Figura 9: Ataque de Bleichenbacher.

4 Ataque de *timing*

Para efetivamente aplicar este ataque, o atacante necessita de analisar uma série de aspectos cuidadosamente. Um desses aspectos, trata-se de saber qual a linguagem aplicada no sistema, durante o mecanismo de análise e verificação das assinaturas.

Deste modo, o atacante poderá começar a atacar o sistema e tirar partido de alguns aspectos, dependendo da linguagem aplicada. É dado o exemplo da linguagem *PHP* que utiliza a função *memcmp* da linguagem *C*, que itera cada caractere até encontrar algum diferente. Isto implica que quantos mais caracteres estejam certos, mais tempo esta verificação irá demorar.

- Vulnerability

```
return $this->signature == hash_hmac($this->digest(), $signature_base_string, $public_key_or_secret, true);
```

- Underlying C implementation

```
int memcmp(const void *s1, const void *s2, size_t n) {
    unsigned char u1, u2;

    for ( ; n-- ; s1++, s2++) {
        u1 = *(unsigned char *) s1;
        u2 = *(unsigned char *) s2;

        if ( u1 != u2 ) {
            return (u1-u2);
        }
    }
    return 0;
}
```

```

(Original) 73702ca3cf26f97b69d6891bff8f93a06d0bcc68
0.0000013119118 0000000000000000000000000000000000000000
0.0000013932216 10000000000000000000000000000000000000000
0.0000016285241 20000000000000000000000000000000000000000
0.0000015981053 30000000000000000000000000000000000000000
0.0000013780439 40000000000000000000000000000000000000000
0.0000014892939 50000000000000000000000000000000000000000
0.0000012960518 60000000000000000000000000000000000000000
0.0000021717087 70000000000000000000000000000000000000000
0.0000014749665 80000000000000000000000000000000000000000
0.0000016347097 90000000000000000000000000000000000000000
0.0000016016298 a0000000000000000000000000000000000000000
0.0000016616338 b0000000000000000000000000000000000000000
0.0000016631538 c0000000000000000000000000000000000000000
0.0000015649927 d0000000000000000000000000000000000000000
0.0000014099265 e0000000000000000000000000000000000000000
0.0000016158320 f00000000000000000000000000000000000000000

```

Figura 10: Análise dos tempos consoante o número de caracteres certos.

Biblioteca node-jose

Quanto a uma implementação desta especificação (*JOSE*) em *Javascript/Node.js*, fomos deparados com uma biblioteca¹³ fornecida pela **Cisco**¹⁴.

Trata-se de uma biblioteca deveras bem documentada, capaz de fornecer imensos exemplos da sua utilização, pelo que apenas iremos apresentar uma pequena seleção de exemplos.

¹³<https://github.com/cisco/node-jose>

¹⁴<https://en.wikipedia.org/wiki/Cisco>

1 Tratamento de chaves

Keys and Key Stores

The `jose.JWK` namespace deals with JWK and JWK-sets.

- `jose.JWK.Key` is a logical representation of a JWK, and is the "raw" entry point for various cryptographic operations (e.g., sign, verify, encrypt, decrypt).
- `jose.JWK.KeyStore` represents a collection of Keys.

Creating a JWE or JWS ultimately require one or more explicit Key objects.

Processing a JWE or JWS relies on a KeyStore.

Obtaining a KeyStore

To create an empty keystore:

```
keystore = jose.JWK.createKeyStore();
```

To import a JWK-set as a keystore:

```
// {input} is a String or JSON object representing the JWK-set
jose.JWK.asKeyStore(input).
  then(function(result) {
    // {result} is a jose.JWK.KeyStore
    keystore = result;
  });
});
```

Exporting a KeyStore

To export the public keys of a keystore as a JWK-set:

```
output = keystore.toJSON();
```

To export all the keys of a keystore:

```
output = keystore.toJSON(true);
```

To generate a new Key:

```
// first argument is the key type (kty)
// second is the key size (in bits) or named curve ('crv') for "EC"
keystore.generate("oct", 256).
  then(function(result) {
    // {result} is a jose.JWK.Key
    key = result;
  });

// ... with properties
var props = {
  kid: 'gBdaS-G8RLax2qg0bTD94w',
  alg: 'A256GCM',
  use: 'enc'
};
keystore.generate("oct", 256, props).
  then(function(result) {
    // {result} is a jose.JWK.Key
    key = result;
  });

```

2 Assinaturas

Signatures

Keys Used for Signing and Verifying

When signing content, the key is expected to meet one of the following:

1. A secret key (e.g, `"kty":"oct"`)
2. The **private** key from a PKI (`"kty":"EC"` or `"kty":"RSA"`) key pair

When verifying content, the key is expected to meet one of the following:

1. A secret key (e.g, `"kty":"oct"`)
2. The **public** key from a PKI (`"kty":"EC"` or `"kty":"RSA"`) key pair

Signing Content

At its simplest, to create a JWS:

```
// {input} is a Buffer
jose.JWS.createSign(key).
  update(input).
  final().
  then(function(result) {
    // {result} is a JSON object -- JWS using the JSON General Serialization
  });
});
```

The JWS is signed using the preferred algorithm appropriate for the given Key. The preferred algorithm is the first item returned by `key.algorithms("sign")`.

To create a JWS using another serialization format:

```
jose.JWS.createSign({ format: 'flattened' }, key).
  update(input).
  final().
  then(function(result) {
    // {result} is a JSON object -- JWS using the JSON Flattened Serialization
  });

jose.JWS.createSign({ format: 'compact' }, key).
  update(input).
  final().
  then(function(result) {
    // {result} is a String -- JWS using the Compact Serialization
  });
});
```

To create a JWS using a specific algorithm:

```
jose.JWS.createSign({ alg: 'PS256' }, key).
  update(input).
  final().
  then(function(result) {
    // ....
  });
});
```

To create a JWS for a specified content type:

```
jose.JWS.createSign({ fields: { cty: 'jwk+json' } }, key).
  update(input).
  final().
  then(function(result) {
    // ....
  });
});
```

3 Cifragem

Encryption

Keys Used for Encrypting and Decrypting

When encrypting content, the key is expected to meet one of the following:

1. A secret key (e.g, "kty":"oct")
2. The public key from a PKI ("kty":"EC" or "kty":"RSA") key pair

When decrypting content, the key is expected to meet one of the following:

1. A secret key (e.g, "kty":"oct")
2. The private key from a PKI ("kty":"EC" or "kty":"RSA") key pair

Encrypting Content

At its simplest, to create a JWE:

```
// {input} is a Buffer
jose.JWE.createEncrypt(key).
  update(input).
  final().
  then(function(result) {
    // {result} is a JSON Object -- JWE using the JSON General Serialization
});
```

How the JWE content is encrypted depends on the provided Key.

- If the Key only supports content encryption algorithms, then the preferred algorithm is used to encrypt the content and the key encryption algorithm (i.e., the "alg" member) is set to "dir". The preferred algorithm is the first item returned by `key.algorithms("encrypt")`.
- If the Key supports key management algorithms, then the JWE content is encrypted using "A128CBC-HS256" by default, and the Content Encryption Key is encrypted using the preferred algorithms for the given Key. The preferred algorithm is the first item returned by `key.algorithms("wrap")`.

To create a JWE using a different serialization format:

```
jose.JWE.createEncrypt({ format: 'compact' }, key).
  update(input).
  final().
  then(function(result) {
    // {result} is a String -- JWE using the Compact Serialization
});

jose.JWE.createEncrypt({ format: 'flattened' }, key).
  update(input).
  final().
  then(function(result) {
    // {result} is a JSON Object -- JWE using the JSON Flattened Serialization
});
```

4 Decifragem

Decrypting a JWE

To decrypt a JWE, and retrieve the plaintext:

```
jose.JWE.createDecrypt(keystore).
  decrypt(input).
  then(function(result) {
    // {result} is a Object with:
    // * header: the combined 'protected' and 'unprotected' header members
    // * protected: an array of the member names from the "protected" member
    // * key: Key used to decrypt
    // * payload: Buffer of the decrypted content
    // * plaintext: Buffer of the decrypted content (alternate)
  });
});
```

To decrypt a JWE using an implied key:

```
jose.JWE.createDecrypt(key).
  decrypt(input).
  then(function(result) {
    // ...
  });
});
```

Allowing (or Disallowing) Encryption Algorithms

To restrict what encryption algorithms are allowed when verifying, add the `algorithms` member to the `options` Object. The `algorithms` member is either a string or an array of strings, where the string value(s) can be one of the following:

- `"*"`: accept all supported algorithms
- `<alg name>` (e.g., `"A128KW"`): accept the specific algorithm (can have a single '*' to match a range of similar algorithms)
- `!<alg name>` (e.g., `"!RSA1_5"`): do not accept the specific algorithm (can have a single '*' to match a range of similar algorithms)

The negation is intended to be used with the wildcard accept string, and disallow takes precedence over allowed.

JWT

Conceito

JSON Web Token(JWT) é um padrão aberto (RFC 7519) que define uma maneira compacta e independente de transmitir informações com segurança entre as partes como um objeto JSON. Essas informações podem ser verificadas e confiáveis porque são assinadas digitalmente. Os JWTs podem ser assinados usando um segredo (com o algoritmo HMAC) ou um par de chaves pública/privada usando RSA ou ECDSA.

Embora os JWTs possam ser encriptados para fornecer sigilo entre as partes, focamos apenas nos tokens assinados. Os mesmos podem verificar a integridade das declarações contidas nele, enquanto os tokens cifrados ocultam essas declarações de outras partes. Quando os tokens são assinados usando pares de chave pública/privada, a assinatura também certifica que apenas a parte que possui a chave privada é a que a assinou.

Além disso, é importante referir que JWT é apenas utilizado para autorização e não para autenticação. Autorização é a garantia que um utilizador envia um pedido para o servidor é o mesmo utilizador que deu log durante a autenticação.

1 Autenticação

A autenticação é o processo de verificação de identificação de um utilizador através da aquisição de credenciais e o uso dessas credenciais para confirmar a identidade do mesmo. O processo de autorização começa se as credenciais forem legítima e segue sempre o procedimento de autenticação.

2 Autorização

A autorização é o processo de permitir que os utilizadores autenticados accedam a recursos que determinam as permissões de acesso ao sistema. A autorização permite que se controle os privilégios de acesso, ao conceder ou negar licenças específicas ao mesmo.

Assim, a autorização ocorre após o sistema autenticar a identidade, concedendo um acesso completo a recursos como informações, arquivos, bancos de dados, fundos, locais, etc. Dito isto, a autorização afeta a capacidade de aceder ao sistema e até o ponto se pode fazer.

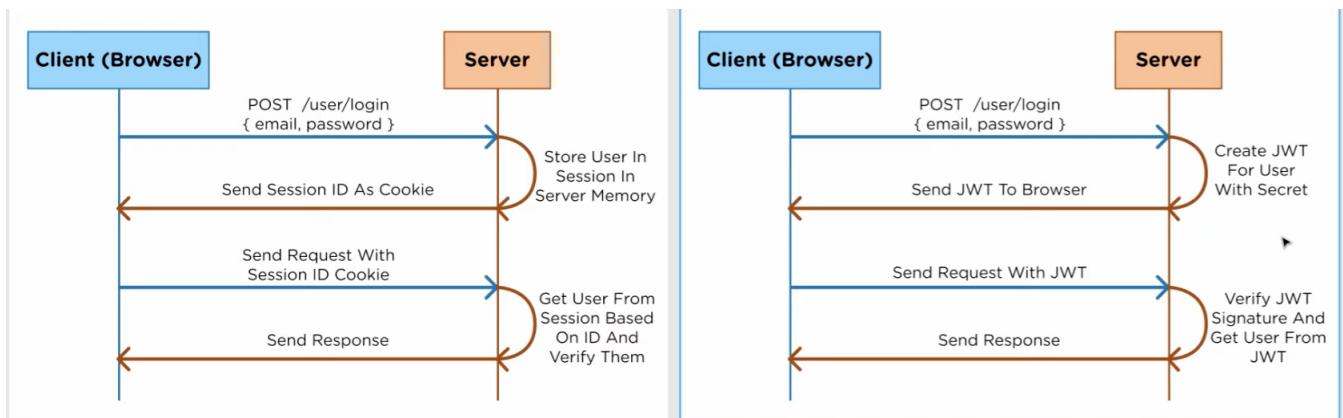


Figura 11: Exemplo JWT

Como verificamos pela figura acima, do lado esquerdo temos um exemplo de utilização de memória do servidor, por outro lado, no lado direito observa-se a criação de um JW Token desse servidor com uma segredo, que de seguida é recebido um pedido com outro JWT e por fim, dentro do mesmo servidor é executada a confirmação de que a assinatura JSON Web Token recebido corresponde ao JWT criado pelo mesmo utilizador.

3 Encoded/Decoded

Como já referido na secção anterior,**JOSE** procura proporcionar uma série de métodos para assinaturas e cifragens de qualquer conteúdo, incluindo o próprio JWT. Como também acontece com outros métodos, este consiste numa divisão em 3 (três) partes, separadas elas por pontos:

- Header
- Payload
- Signature

Maioritariamente, o **Header** é constituído pelo tipo de token associado, neste caso específico, JWT, e o algoritmo de assinatura que foi utilizado, como HMAC,SHA256 ou até RSA. Numa segunda fase, está o **Payload**, que contém todas as informações úteis, ou seja, declarações, ou até outros dados adicionais. Existem 3 tipos de reclamações:

- Registradas: Reclamações não obrigatórias, mas recomendadas de forma a fornecer um conjunto de queixas úteis, como iss (emissor), exp (tempo de expiração), sub (assunto), aud (público), entre outros.
- Públicas: Reclamações que podem ser definidas à vontade por aqueles que usam tokens. Mas, para evitar colisões, eles devem ser definidos no IANA JSON Web Token Registry ou definidos como um URI que contém um namespace resistente a colisões.
- Privadas: Reclamações criadas para compartilhar informações entre as partes que concordam em usá-las e não são reclamações registadas ou públicas.

Por último, **verify signature** é dos aspetos mais importantes, isto porque, esta assinatura permite-nos verificar que o token não foi alterado ou modificado pelo cliente ou utilizador antes de ser novamente enviado para nós, ou seja servidor. A forma como o faz é utilizando as duas primeiras porções do Token, ou seja, pega no header e codifica para base64, adiciona um .(ponto) e codifica em base64 a seguinte porção, o payload. De seguida utilizada o algoritmo mencionado no Header e uma chave secreta que é definida na assinatura verificada. Basicamente o algoritmo tem a função de codificar a porção com os dados, e seguidamente pega em toda essa informação de header e payload e codifica utilizando a chave secreta específica.

Por fim, a secção azul visualizada na seguinte figura é nada mais nada menos do que uma versão codificada de todos os dados que foram enviados para o cliente. Porém, se o cliente alterar os dados, é sabido que a assinatura vai alterar, isto porque, quando o servidor receber o código codificado do cliente, vai combinar as duas secções, codifica-las e de seguida vai comparar com a última secção da chave. No caso de ser idêntica temos certezas que não existiu alterações, caso contrário sabe-se que os dados do JSON Web Token já não se encontram válidos.

Desta forma, é preceitivel todo o conceito por detrás do **Decoded**. Pela figura adiante, é de fácil compreensão que **Encoded** é o próprio JWT, que tal como referido anterioremnte é dividido por pontos (.), sendo a primeira secção o header, seguido do payload, finalizando com a assinatura.

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG91IiwiWF0IjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE
{ "alg": "HS256", "typ": "JWT" }
PAYOUT: DATA
{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }
VERIFY SIGNATURE
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) □ secret base64 encoded

Figura 12: Encoded and Decoded

Na verdade, um JWT não existe em si, ou seja é como uma classe abstrata. Assim, esta framework é dividida em duas implementações concretas:

- JWS - JSON Web Signature
- JWE - JSON Web Encryption

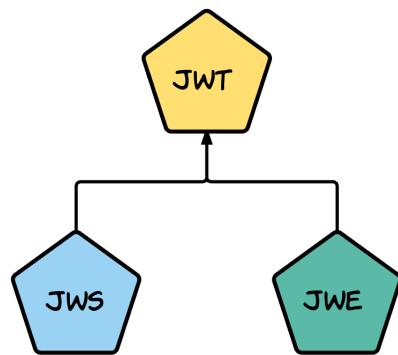


Figura 13: Divisão de JWT

4 Utilidade

Depois de passarmos um pouco pelo conceito, é importante entendermos a importância e utilidade. Imaginemos que no mesmo sistema/aplicação existem dois servidores distintos. Numa situação banal, imaginemos termos um servidor que tem acesso a todas as informações pessoais e outro que apenas é relativo a promoções, o cliente/utilizador tem que fazer log in para entrar em nos dois servidores, isto porque a informação será mantida apenas no servidor que efetuou o log.

Porém utilizando JWT e partilhando a mesma chave secreta entre os dois servidores, é apenas necessário enviar o mesmo JWT do utilizador para os dois servidores e será autenticado ao mesmo tempo em ambos, sem ser necessário fazer log in duas vezes. Desta forma, desde que os dois servidores continuem a partilhar a mesma chave secreta e a informação do utilizador seja mantida no token do mesmo, não será necessário efetuar a mesma operação duas vezes.

5 JWT não seguro

Um JWT pode ser um objeto JWS ou JWE, como se observa na imagem anterior. O JWT não seguro é um objeto JWS no qual no cabeçalho JOSE o valor do elemento alg é definido como none. Em outras palavras, um JWT não seguro é um JWS sem assinatura, ou seja, enquanto se divulga informações de direitos e a identidade do utilizador no JWT não seguro, espera-se que o transporte subjacente forneça uma garantia sobre a integridade e a confidencialidade do token.

6 JWT Autenticação em Node.js

- CORRER O COMANDO `npm run devStart`

- ENVIAR PEDIDOS E OBTER RESULTADOS

A linha de comando do npm tem perguntas como nome, licença, scripts, descrição, autor, palavras-chave, versão, arquivo principal, etc. Depois de o npm terminar o projeto, um arquivo package.json ficará visível na pasta do mesmo.

- INSTALAR OS MÓDULOS NECESSÁRIOS

Após a criação do projeto, o próximo passo é incorporar os pacotes e módulos a serem utilizados.

- CRIAR ARQUIVO DE CONFIGURAÇÃO (.ENV)

Este arquivo contém as variáveis que são necessárias passar para o ambiente da aplicação.

- CRIAR ROTA PARA GERAR JWT

É necessário gerar um pedido 'post' que envia o token JWT em resposta.

```
JS authServer.js > ...
1  require('dotenv').config()
2
3  const express = require('express')
4  const app = express()
5  const jwt = require('jsonwebtoken')
6
7  app.use(express.json())
8
9  let refreshTokens = []
10
11 app.post('/token', (req, res) => {
12   const refreshToken = req.body.token
13   if (refreshToken == null) return res.sendStatus(401)
14   if (!refreshTokens.includes(refreshToken)) return res.sendStatus(403)
15   jwt.verify(refreshToken, process.env.REFRESH_TOKEN_SECRET, (err, user) => {
16     if (err) return res.sendStatus(403)
17     const accessToken = generateAccessToken({ name: user.name })
18     res.json({ accessToken: accessToken })
19   })
20 })
21
22 app.delete('/logout', (req, res) => {
23   refreshTokens = refreshTokens.filter(token => token !== req.body.token)
24   res.sendStatus(204)
25 })
26
27 app.post('/login', (req, res) => {
28   // Authenticate User
29
30   const username = req.body.username
31   const user = { name: username }
32
33   const accessToken = generateAccessToken(user)
34   const refreshToken = jwt.sign(user, process.env.REFRESH_TOKEN_SECRET)
35   refreshTokens.push(refreshToken)
36   res.json({ accessToken: accessToken, refreshToken: refreshToken })
37 })
38
39 function generateAccessToken(user) {
40   return jwt.sign(user, process.env.ACCESS_TOKEN_SECRET, { expiresIn: '15s' })
41 }
42
43 app.listen(4000)
```

- CRIAR ROTA PARA VALIDAR JWT

É necessário gerar um pedido 'get' que contém o token JWT no cabeçalho e envia o status de verificação como resposta.

```
requests.rest > ⚙ GET /posts
Send Request
1  GET http://localhost:3000/posts
2  Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1IjoiSmltIiwiaWF0IjoxNTY4NzU50DEyLCJleHAiOjE1Njg3NTk
3
4  ###
5
6  Send Request
7  DELETE http://localhost:4000/logout
8  Content-Type: application/json
9
10 {
11   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1IjoiSmltIiwiaWF0IjoxNTY4NzU50TIyfQ.RT6wszuCeFLwC_6ksmNMIELxiC"
12 }
13 ###
14
15 Send Request
16 POST http://localhost:4000/token
17 Content-Type: application/json
18
19 {
20   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1IjoiSmltIiwiaWF0IjoxNTY4NzU50TIyfQ.RT6wszuCeFLwC_6ksmNMIELxiC"
21 }
22 ###
23
24 Send Request
25 POST http://localhost:4000/login
26 Content-Type: application/json
27
28 {
29   "username": "Jim"
```

```
JS server.js > ...
1  require('dotenv').config()
2
3  const express = require('express')
4  const app = express()
5  const jwt = require('jsonwebtoken')
6
7  app.use(express.json())
8
9  const posts = [
10    {
11      username: 'Kyle',
12      title: 'Post 1'
13    },
14    {
15      username: 'Jim',
16      title: 'Post 2'
17    }
18  ]
19
20  app.get('/posts', authenticateToken, (req, res) => {
21    res.json(posts.filter(post => post.username === req.user.name))
22  })
23
24  function authenticateToken(req, res, next) {
25    const authHeader = req.headers['authorization']
26    const token = authHeader && authHeader.split(' ')[1]
27    if (token == null) return res.sendStatus(401)
28
29    jwt.verify(token, process.env.ACCESS_TOKEN_SECRET, (err, user) => {
30      console.log(err)
31      if (err) return res.sendStatus(403)
32      req.user = user
33      next()
34    })
35  }
36
37  app.listen(3000)
```

JWS

Conceito

JSON Web Signature representa um conteúdo (payload) seguro com uma assinatura digital ou MAC, garante a integridade dos dados, autenticação, não repúdio (no caso de utilização de algoritmos assimétricos), porém tem algumas desvantagens como, o conteúdo do JWS token não está cifrado mas sim codificado em base64url, o que quer dizer que qualquer pessoa pode ver o seu conteúdo, o mesmo não pode ser alterado pois consegue-se verificar se o conteúdo foi alterado com a assinatura digital ou MAC.

Tal como o JWE são definidas duas serializações estreitamente relacionadas para os JWS. A serialização compacta JWS que é destinada para ambientes com restrições de espaço, e a serialização JWS JSON que representa JWS como um objecto JSON que permite que o mesmo conteúdo seja assinado por múltiplas partes.

7 JWS Compact Serialization

Com a serialização compacta JWS, um token JWS é construído com 3 componentes-chave, cada um separado por um período (‘.’):

- JOSE header
- JWS Payload
- JWS Signature



Figura 14: Estrutura de um JWS token formado por JWS compact serialization

7.1 JOSE Header

A primeira componente da serialização compacta JWS é o JOSE Header. É um simples objecto JSON que descreve a assinatura digital ou MAC a ser aplicado JWS Header e JWS Payload, o único parâmetro obrigatório neste objecto é o parâmetro "alg" que identifica o algoritmo utilizado para assinar JWS Header e JWS Payload, o resto dos parâmetros opcionais podem ser consultados na [Seção 4.1](#) do documento [\[RFC 7515\]](#).

7.2 JWS Payload

A segunda componente da serialização compacta JWS é o conteúdo (payload) que se pretende assinar. Este conteúdo pode ser de qualquer tipo desde que seja convertido para base64url.

7.3 JWS Signature

A ultima componente da serialização compacta JWS é o JWS Signature. Que é o resultado de assinar digitalmente (ou MAC) da mensagem formada a partir do JOSE Header e o Payload (ASCII(BASE64URL-ENCODE(UTF8(JOSE Header)) ‘.’ BASE64URL-ENCODE(JWS Payload))).

8 JWS Compact Serialization — Processo

De seguida vamos apresentar o processo passo por passo para criar, um JWS Token com serialização compacta JWS:

- Criar um representação UTF-8 do JOSE Header, o único elemento obrigatório é o elemento ‘alg’ que identifica o algoritmo que vai ser utilizado para a assinatura do conteúdo, caso o valor do elemento alg seja ‘none’, o Header é considerado um JWS Unprotect Header.
- Converter o objeto JSON Header e o JWS Payload que desejamos assinar para base64url.
- Assinar a representação UTF-8 do JWS Signing Input (que é a concatenação JWS Header , o período (:), e JWS Payload codificados em base64url) com a algoritmo escolhido no JOSE Header.
- Converter o resultado do ponto anterior para base64url.
- Concatenar as tres partes codificadas em base64url que formam o JWS token separadas com o período(:).

Segue um pequeno exemplo da criação de um JWS token utilizando a serialização compacta JWS em Node.js com o package node-jose, já identificado anteriormente neste trabalho.

```
var jose = require("node-jose");

// função de converte um objecto json em uma string Base64
function json2Base64(jsonObject) {
    var buff = Buffer.from(JSON.stringify(jsonObject));
    return buff.toString('utf8');
}

async function pd1(){
    try {
        const keystore = [
            "kty": "EC",
            "d": "N_102Ta0zGq6jpvEyW8Lm0YLjClaodenL8TfGLLnTn8",
            "use": "sig",
            "crv": "P-256",
            "kid": "zdFjSAXRK_xMKdclIAQWzSYaTmyZ1an045bGiYXm8Ic",
            "x": "yfyuh0683zDMNIu4yEc0-b23dPLMW5DnWnWTjG9gDZY",
            "y": "B00200b4d1wZR_456ZzQfSwK7Xu3H8crdwLGfZqwC",
            "alg": "ES256"
        ];

        const ks = await jose.JWK.asKeyStore(keystore);
        const rawKey = ks.get(keystore[0].kid);
        const key = await jose.JWK.asKey(rawKey);

        // Payload (não tem de ser um objecto JSON)
        var payload = {"name": "bob",
                      "exp": 2313212121,
                      "http://example.com": true};

        // converter payload para base64url
        var buffer_payload_utf = json2Base64(payload);

        var format = {format: 'compact'}; // 'flattened', 'compact' ou ''
        const token = await jose.JWS.createSign(format, key).update(buffer_payload_utf, 'utf8').final();

        console.log(token);

    }catch (err) {
        console.log(err);
    }
}

pd1();
```

Figura 15: Exemplo código Node-js, JWS Compact Serialization

```
(base) joao@Air-di-Joao:~/PD1% node jws.js
eyJhbGciOiJFUzI1NiIsImtpZCI6InpEZmpTQVhSS194tUtky2xJQVFxeINZYVRteVoxYW4wNFN1R215WG04SMiFQ.eyJuYw1Iijoim9iIiwiZXhwIjoxMzEzMjEyMTIxLCJodHRwOi8vZXhhbXBsZS5jb201OnRydWV9.uN616ss0DPCMmcUa9XMZeQPxB6L6bgp0lic724CjkJagG63iFVqj9a8NDHlx9IinTIVlxj4Z-JyYsYYdG8pPjxA
```

Figura 16: Token JWS Compact Serialization

Neste caso como queremos a serialização compacta é necessário definir o formato na criação da assinatura.

9 JWS JSON Serialization

Ao contrário da serialização compacta JWS, a serialização JWS JSON pode produzir várias assinaturas sobre a mesmo JWS payload juntamente com vários Header JOSE. Este objecto JSON divide-se em dois elementos principais payload e signatures, cada elemento do array signatures é composto por dois elementos protected (que tem a informação relativa ao JOSE Header, ex. o algoritmo utilizado para a assinatura) e a signature.

10 JWS JSON Serialization — Processo

De seguida vamos apresentar o processo passo por passo para criar, um JWS Token com serialização JSON JWS.

- Converter o JWS Payload que desejamos assinar para base64url.
- Decidir o número de assinaturas, e criar um Header com os elementos necessários para cada assinatura.
- Converter os JWS Header para para base64url.
- Para cada JWS Header assinar a representação UTF-8 do JWS Signing Input.
- Depois que todas as assinaturas forem computadas, o array das JSON signatures poderá ser construída e concluirá a serialização JWS JSON.

Segue um pequeno exemplo da criação de um JWS token utilizando a serialização JWS JSON em Node.js com o package node-jose, já “identificada” anteriormente neste trabalho.

```
async function pd1(){
  try {
    const keystore = [
      {
        "kty": "EC",
        "d": "N_i02Ta0zGq6jpvEyW8Lm0YLjClaodenL8TfGLnTn8",
        "use": "sig",
        "crv": "P-256",
        "kid": "zDfjSAXRK_xMKdclIAQWzSYaTmyZ1an04SbGuyXm8Ic",
        "x": "yfyuh0683zDMNIu4yEc0-b23dpUMw5DnWnw1jG9g0ZY",
        "y": "_B000200b4dIwZR_456ZzQfSwK7Xu3H8crdwLGfZqwc",
        "alg": "ES256"
      },
      {
        "kty": "oct",
        "use": "sig",
        "kid": "8FzNFdht6g1yaNrWCF5UGlDarqZalEJzz--z-gWcVY",
        "k": "2oFCr-_PkW69mvuxEob4DGbftBx0kn0YTa8Sy-KxGrjT_d19BgG_K0lowq\ERPg1ruBvs5JUgRe_K6-hbJS5Wm0jYS5unElw-44ZZ26v7f4LB-RZSrGwTr4z141b5GTaCe0Hu-Zm1R_Y7NNLw0DM4aeUmaG6\Bs0BW8ZDZvrrosTqbAqljg1FsopoBluc871uoj9jLAY8LVq_Ym1oRe2qU7SGi4jgi1Yv0xx2ZD0J4bMvh3JFR0jDaBe96Hi0R\ImLwQX5-I3x-b7Y3tMHA0ZMi0azQGeQ6TigsFjGI6u-c0rkY92h_cYzTym2129462d7RA",
        "alg": "HS256"
      }
    ],
    const ks = await jose.JWK.asKeyStore(keystore);
    const rawKey = ks.get(keystore[0].kid);
    const key = await jose.JWK.asKey(rawKey);

    const keys = ks.all();

    // Payload (não tem de ser um objecto JSON)
    var payload = {"name": "bob",
                  "exp": 23132121,
                  "http://example.com": true};

    // converter payload para base64url
    var buffer_payload_utf = json2Base64(payload);

    var format = {format: ''}; // 'flattened', 'compact' ou ''
    const token = await jose.JWS.createSign(format, keys).update(buffer_payload_utf, 'utf8').final();
    console.log(token);

  } catch (err) {
    console.log(err);
  }
}

pd1();
```

Figura 17: Exemplo código Node-js, JSON Serialization

```
(base) joao@Air-di-Joao PD1 % node jws.js
{
  payload: 'eyJJuYW1lIjoiYm9iIiwiZXhwIjoyMzEzMjEyMTIxLCJodHRwOi8vZXhhbXBsZS5jb20iOnRydWV9',
  signatures: [
    {
      protected: 'eyJhbGciOiJFUzI1NiIsImtpZCI6InpEZmpTQVhSS194TUtkY2xJQVFxeIHZYVRteVoxYW4wNFNiR2l5WG04SWMifQ',
      signature: 'bWh14Uv1SY1Y-1KPQgy51_zEAjjfJ30dTKFg8C16IevBx13h5Y9YLP186b_eX5xdqMK7jP7FJKwhsHChOyoM6Q'
    },
    {
      protected: 'eyJhbGciOiJIUzI1NiIsImtpZCI6IjhGek5GZGh0NmcxewFOclDRjVVR2xEYXJxWmFsRUp6enotLXotZ1djVlkifQ',
      signature: 'D6vgy3WugUHWHYdErwLwtyTrsn0nmV-17EwTwoBd01Q'
    }
  ]
}
```

Figura 18: Token JSON Serialization

11 The JWS Protected Header

O JWS Protected Header é um objecto JSON em que todos os Headers tem de ser totalmente protegidos por uma assinatura ou um algoritmo MAC, no caso da serialização JSON JWS podem existir vários JWS Protected Header mas tem de ser assinados de forma diferente.

12 JWS Unprotect Header

O JWS Unprotect Header é um objecto JSON que não tem os elementos do Header totalmente protegidos por uma assinatura ou algoritmo MAC. Em outras palavras é um JOSE Header com o elemento ‘alg’ = ‘none’ o que quer dizer que não foi assinado. Isto só pode estar presente no caso da serialização JSON JWS.

JWE

Conceito

JSON Web Encryption (JWE) representa conteúdo encriptado usando JSON-based data structures [RFC7159]. Os mecanismos criptográficos JWE são utilizados para encriptar e fornecer proteção de integridade para uma sequência arbitrária de octetos.

São definidas duas serializações estreitamente relacionadas para os JWE. A Serialização Compacta dos JWE é uma representação compacta, segura para URL, destinada para ambientes com restrições de espaço, tais como cabeçalhos de autorização HTTP e parâmetros de consulta URI. A Serialização JWE JSON representa JWEs como objectos JSON e permite que o mesmo conteúdo seja encriptado para múltiplas partes. Ambas partilham os mesmos fundamentos criptográficos.

13 JWE Compact Serialization

Com a serialização compacta JWE, um token JWE é construído com cinco componentes-chave, cada um separado por um período :

- JOSE header
- JWE Encrypted Key
- JWE initialization vector
- JWE Additional Authentication Data (AAD)
- JWE Ciphertext
- JWE Authentication Tag.



Figura 19: Estrutura de um JWE token formado por JWE compact serialization

13.1 JOSE header

O JOSE header é o primeiro elemento do JWE Token produzido sob serialização compacta. A estrutura do cabeçalho JOSE é a mesma da JWS, com algumas exceções. A especificação JWE introduz dois novos elementos (enc e zip), que estão incluídos no cabeçalho JOSE do símbolo JWE, para além do que é definido pela especificação JSON Web Signature (JWS).

13.2 JWE Encrypted Key

Para compreender a secção JWE Encrypted Key do JWE, precisamos primeiro de compreender como é que um JSON é encriptado. O elemento *enc* do cabeçalho JOSE define o algoritmo de encriptação do conteúdo e deve ser um algoritmo simétrico de Authenticated Encryption with Associated Data (AEAD). O elemento *alg* do cabeçalho JOSE define o algoritmo de encriptação para encriptar o Content Encryption Key (CEK). Este algoritmo também pode ser definido como o algoritmo de encapsulamento da chave, uma vez que envolve o CEK.

Nota:

As chaves simétricas são principalmente utilizadas para a encriptação de conteúdos e são muito mais rápidas do que a encriptação de chaves assimétricas. Ao mesmo tempo, a encriptação de chaves assimétricas não pode ser utilizada para encriptar mensagens grandes.

13.2.1 AEAD

Authenticated Encryption with Associated Data (AEAD) é um modo de operação em blocos cifrados que fornece simultaneamente garantias de confidencialidade, integridade e autenticidade dos dados; a decifra é combinada numa única etapa com a verificação da integridade.

13.3 JWE initialization vector

Alguns algoritmos de encriptação, que são utilizados para a encriptação, requerem um vector de inicialização aleatório, durante o processo de encriptação. O vector de inicialização é um número gerado aleatoriamente, que é utilizado juntamente com uma chave secreta para encriptar dados. Isto adicionará aleatoriedade aos dados encriptados, o que evitaria a repetição, mesmo que os mesmos dados sejam encriptados usando a mesma chave secreta uma e outra vez. Para decifrar a mensagem na extremidade receptora do token, tem de conhecer o vector de inicialização, daí que esteja incluído no token JWE, sob o elemento Vector de Inicialização JWE. Se o algoritmo de encriptação do conteúdo não requer um vector de inicialização, então o valor deste elemento deve ser mantido vazio.

13.4 JWE Ciphertext

O quarto elemento de um JWE Token é o valor de base64url-encoded do texto cifrado JWE. O texto cifrado JWE é calculado através da encriptação do JSON em texto simples utilizando a Content Encryption Key (CEK), o JWE initialization vector e o valor dos Additional Authentication Data (AAD), com o algoritmo de encriptação definido pelo elemento de cabeçalho *enc*. O algoritmo definido pelo elemento de cabeçalho *enc* deve ser um algoritmo simétrico de Autenticação Autenticada com Dados Associados (AEAD). O algoritmo AEAD, que é utilizado para encriptar a carga útil em texto simples, também permite especificar Dados Autenticados Adicionais (DAA).

13.5 JWE Authentication Tag

O valor de base64url-encoded da JWE Authenticated Tag é o elemento final do JWE Token. Como referido anteriormente na secção referente a AEAD, o valor da Tag deve ser produzido durante o processo de encriptação AEAD, juntamente com o texto cifrado. A etiqueta de autenticação assegura a integridade do texto cifrado e dos Additional Authenticated Data (AAD).

14 JWE Compact Serialization — Processo

De seguida vamos apresentar o processo passo por passo para criar, um JWE Token com serialização compacta JWE:

- Definir o algoritmo utilizado para determinar o valor da Chave de Criptografia de Conteúdo (CEK). Este algoritmo é definido pelo elemento `alg` no cabeçalho JOSE. Existe apenas um elemento de `alg` por JWE Token.
- Calcular o CEK e calcular a Chave Encriptada JWE com base no modo de gestão de chaves, escolhido no paço anterior. O CEK é posteriormente utilizado para cifrar a carga útil do JSON. Existe apenas um elemento da Chave Encriptada JWE no token JWE.
- Cálculo do valor base64url-encriptado da Chave Encriptada JWE, produzido na etapa anterior. Este é o segundo elemento do token JWE.
- Gerar um valor aleatório para o Vector de Inicialização JWE. Este é o 3º elemento do símbolo JWE.
- Se necessária a compressão do token, o text JSON deve ser comprimido seguindo o algoritmo de compressão definido sob o elemento de cabeçalho `zip`.
- Construir a representação JSON do cabeçalho JOSE e encontrar o valor base64url-encoded do cabeçalho JOSE codificado com UTF-8. Este é o 1º elemento do JWE Token.
- Para encriptar o JSON, precisamos do CEK previamente estabelecido, do JWE Initialization Vector , e dos Additional Authenticated Data (AAD). Calcular o valor ASCII do cabeçalho codificado JOSE do ponto anterior e utilizá-lo como AAD.
- Encriptar o JSON do ponto anterior utilizando o CEK, o Vector de Inicialização JWE e os Dados Additional Authenticated Data (AAD), seguindo o algoritmo de encriptação de conteúdo definido pelo elemento `enc` do cabeçalho.
- O algoritmo definido pelo elemento `enc` do cabeçalho de encabeçamento é um algoritmo AEAD e após o processo de encriptação, produz o texto cifrado e a Authentication Tag.
- Calcular o valor base64url-encoded do ciphertext. Este é o 4º elemento do JWE Token.
- Calcular o valor de base64url-encoded da Authentication Tag. Este é o 5º elemento do token JWE.
- Assim temos todos os elementos para construir o token JWE

15 JWE JSON Serialization

Ao contrário da JWE compact serialization, a JWE JSON serialization pode produzir dados encriptados dirigidos a múltiplos destinatários sobre o mesmo JSON. A derradeira forma serializada sob JWE JSON serialization representa encriptação de todo um payload num objecto JSON. Este objecto JSON inclui seis elementos:

- JWE Protected Header
- JWE Shared Unprotected Header
- JWE Per-Recipient Unprotected Header
- JWE Initialization Vector
- JWE Ciphertext
- JWE Authentication Tag

Segue-se um exemplo de um JWE Token, serializado usando JWE JSON serialization.

```
{  
  "protected": "eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0",  
  "unprotected": [  
    "jku": "https://server.example.com/keys.jwks"  
,  
    "recipients": [  
      {  
        "header": {  
          "alg": "RSA1_5",  
          "kid": "2011-04-29"  
        },  
        "encrypted_key": "UGhIOguC7IuEvf_NPVaXsGMoL0mwvc1GyqlI9XShH59_i8J0PH5ZZyNfGy2xGd"  
      },  
      {  
        "header": {  
          "alg": "A128KW",  
          "kid": "7"  
        },  
        "encrypted_key": "6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDThzBC2IlrT1o0Q"  
      }  
,  
      {"iv": "AxY8DCtDaGlsbGljb3RoZQ",  
       "ciphertext": "KDlTtXchhZTGufMYm0YGS4HffxPSUrfmqCHxaI9w0GY",  
       "tag": "Mz-VPPyU4RlcuYv1IwIvw"  
    ]  

```

Figura 20: Estrutura de um JWE token formado por JWE JSON Serialization

15.1 JWE Protected Header

O JWE protected header é um objecto JSON que inclui os elementos de cabeçalho que têm de ser protegidos pela operação de encriptação autenticada (AEAD). Os elementos dentro do cabeçalho protegido JWE são aplicáveis a todos os destinatários do JWE Token. O elemento protegido na forma serializada JSON representa o valor de base64url-encoded do JWE protected header. Só pode haver um elemento protegido num JWE protected header ao nível da raiz e quaisquer elementos do cabeçalho que discutimos anteriormente sob o JOSE header também podem ser utilizados sob JWE protected header.

15.2 JWE Shared Unprotected Header

O JWE shared unprotected header é um objecto JSON que inclui os elementos do cabeçalho que não estão protegidos. Os elementos dentro do JWE shared unprotected header são aplicáveis a todos os destinatários do JWE Token. O elemento desprotegido na forma de JSON serializado representa o cabeçalho JWE shared unprotected header. Só pode haver um elemento desprotegido num código JWE, ao nível da raiz e qualquer elemento de cabeçalho que discutimos anteriormente sob o código JOSE também pode ser utilizado sob o código JWE partilhado desprotegido.

15.3 JWE Per-Recipient Unprotected Header

O JWE per-recipient unprotected header é um objecto JSON que inclui os elementos do cabeçalho que não são protegidos pela integridade. Os elementos dentro do JWE per-recipient unprotected header são aplicáveis apenas a um determinado destinatário do JWE Token. No JWE Token, estes elementos de cabeçalho são agrupados sob o elemento destinatário. O elemento destinatário representa um conjunto de destinatários do JWE Token. Cada membro contém um elemento de cabeçalho e um elemento cipher_key.

15.3.1 Header

O elemento de cabeçalho, que está dentro de cada array de destinatários JSON, representa o valor dos elementos do JWE header correspondentes que não são protegidos por encriptação autenticada para cada destinatário.

15.3.2 Chave encriptada

O elemento chave encriptada representa o valor base64url-encoded da chave encriptada. Esta é a chave utilizada para encriptar o payload da mensagem. A chave pode ser encriptada de diferentes maneiras para cada destinatário.

15.4 JWE Initialization Vector

Tendo o mesmo significado como explicado anteriormente no âmbito do JWE compact serialization. O elemento iv no Token JWE representa o valor do vector de inicialização utilizado para a encriptação.

15.5 JWE Ciphertext

Tal como referido na JWE compact serialization anteriormente, o elemento de texto cifrado no JWE Token contém o valor de base64url-encoded do texto cifrado JWE.

15.6 JWE Authentication Tag

Tal como referido na JWE compact serialization anteriormente. O elemento da tag no JWE Tokwn é o valor base64url-encoded da etiqueta autenticada JWE, sendo o resultado do processo de encriptação usando um algoritmo AEAD.

16 JWE JSON Serialization — Processo

Segue-se uma lista do processo de encriptação de um JWE sob a JSON serialization.

- Definir o algoritmo utilizado para determinar o valor da Chave de Criptografia de Conteúdo (CEK). Este algoritmo é definido pelo elemento alg no JOSE header. Sob a JWE JSON serialization, o JOSE header é construído pela união de todos os elementos definidos sob o JWE Protected Header, JWE Shared Unprotected Header e Per-Recipient Unprotected Header. Uma vez incluído no Per-Recipient Unprotected Header, o elemento alg pode ser definido por destinatário.
- Calcular o CEK e calcular a JWE Encrypted Key atendendo ao algoritmo escolhido no passo anterior. O CEK é posteriormente utilizado para encriptar o payload do JSON.
- Calcular o valor base64url-encriptado da JWE Encrypted Key, que é produzida na etapa anterior. Mais uma vez, este é calculado por destinatário e o valor resultante é incluído na chave encriptada.
- Executar os três passos anteriores para cada destinatário do JWE Token. Cada iteração irá produzir um elemento no conjunto de destinatários JSON do token JWE.
- Gerar um valor aleatório para o JWE Initialization Vector. Independentemente da técnica de serialização, o token JWE terá o valor de base64url-encoded do JWE Initialization Vector.
- Se for necessária a compressão do token, o payload JSON em plaintext, deve ser comprimido seguindo o algoritmo de compressão definido sob o elemento de cabeçalho zip. O valor do elemento de cabeçalho zip pode ser definido sob o cabeçalho JWE Protected Header ou JWE Shared Unprotected Header.
- Construir a representação JSON do JWE Protected Header, JWE Shared Unprotected Header and Per-Recipient Unprotected Headers.
- Calcular o valor base64url-encoded do JWE Protected Header com a codificação UTF-8. Este valor é representado pelo elemento protegido no token JWE serializado. O JWE Protected Header é opcional e, se presente, só pode existir um. Caso contrário, o valor do elemento protegido estará vazio.
- Gerar um valor para os Additional Authenticated Data (AAD) e calcular o valor base64url-encoded dos mesmos. Este é um passo opcional e se for feito, então o valor AAD de base64url-encoded será usado como elemento de input para cifrar o payload do JSON.
- Para encriptar o payload do JSON, precisamos do CEK, do JWE Initialization Vector, e dos Additional Authenticated Data (AAD). Calcular o valor ASCII do JWE Protected Header codificado e utilizá-lo como AAD.
- Encriptar o payload comprimido do JSON usando o CEK, o JWE Initialization Vector e os Additional Authenticated Data (AAD), seguindo o algoritmo de encriptação definido pelo elemento *enc* do cabeçalho.
- O algoritmo definido pelo elemento de cabeçalho *enc* é um algoritmo AEAD e, após o processo de encriptação, produz o texto cifrado e a Authentication Tag.
- Cálculo do valor base64url-encoded do ciphertext, produzido na etapa anterior.
- Calcular o valor base64url-encoded da Authentication Tag, também produzida anteriormente.

17 Message Decryption

O processo de decifra da mensagem é o inverso do processo de cifra. A ordem das etapas não é significativa nos casos em que não há dependências entre os inputs e outputs entre passos. Porém se qualquer uma destas etapas falhar, o conteúdo cifrado não pode ser validado.

Quando existem múltiplos destinatários, é feita uma decisão de candidatura para que a JWE seja aceite. Em alguns casos, o conteúdo encriptado para todos os destinatários deve validar com sucesso, ou a JWE será considerada inválida. Noutros, apenas o conteúdo encriptado para um único destinatário chega para este ser validado com sucesso. No entanto, em todos os casos, o conteúdo encriptado deve ser validado com sucesso ou a JWE é considerado inválido.

18 Vulnerabilidades e Questões de Segurança

Todas as questões de segurança que são pertinentes a qualquer criptografia e devem ser tidas em conta também para JWS/JWE/JWK. Entre estes problemas estão, proteger as chaves assimétricas privadas e simétricas do utilizador empregando contramedidas para vários ataques.

Há porém alguns aspetos a ter em conta e ataques conhecidos tais como:

18.1 Key Protection

As implementações que realizam a encriptação devem proteger chave de encriptação e a CEK. O comprometimento da chave de encriptação pode resultar na divulgação de todos os conteúdos protegidos com essa chave. Da mesma forma, o compromisso da CEK pode resultar na revelação do conteúdo encriptado associado.

18.2 Using Matching Algorithm Strengths

Algoritmos de forças equivalentes devem ser utilizados em conjunto sempre que possível. Por exemplo, quando se utiliza o invólucro de chave AES com um determinado tamanho de chave, recomenda-se a utilização do mesmo tamanho de chave quando se utiliza também o GCM AES. Se a chave de encriptação e os algoritmos de encriptação de conteúdo forem diferentes, a segurança efectiva é determinada pelo mais fraco dos dois algoritmos.

18.3 Adaptive Chosen-Ciphertext Attacks

Ao decifrar, deve ser tomado especial cuidado para não permitir que o destinatário JWE seja utilizado como um oráculo para decifrar mensagens. RFC deve ser consultado para contra-medidas específicas a ataques em RSAES-PKCS1-v1_5. Um atacante pode modificar o conteúdo do Header Parameter "alg" de "RSA-OAEP" a "RSA1_5", a fim de gerar um erro de formatação que pode ser detectado e utilizado para recuperar o CEK mesmo que a RSAES-OAEP tenha sido utilizada para encriptar o CEK. É portanto, particularmente importante reportar todos os erros de formatação ao CEK, Additional Authenticated Data, ou ciphertext como um único erro quando o conteúdo encriptado é rejeitado.

Além disso, este tipo de ataque pode ser evitado através da restrição da utilização de uma chave para um conjunto limitado de algoritmos. Isto significa, por exemplo, que se a chave for marcada como sendo para apenas "RSA-OAEP", qualquer tentativa de decifrar uma mensagem usando o algoritmo "RSA1_5" com essa chave deve falhar imediatamente devido à utilização inválida da chave.

JWK

Conceito

JSON Web Key (JWK), é um objeto JSON que representa uma chave criptográfica utilizada para validar a assinatura de um JWT emitido pelo servidor de autorização. Se o servidor emissor do JWT utilizou uma chave assimétrica para assinar o JWT, é provável a existência de um ficheiro JSON Web Key Set (JWKS). Este ficheiro terá, obrigatoriamente, um campo *keys*, que, por sua vez, será uma lista de objetos JWK. Deste modo, o JWKS expõem as chaves públicas.



Figura 21: Exemplo JWKS

Formato e Parâmetros da Chave

É possível gerar:

- uma **Chave RSA**
- uma **EC Key Pair**, baseadas em curvas elípticas
- uma **Octect Key Pair**, usadas para representar chaves de curvas de Edwards e que terão o tipo OKP
- uma **Octet Sequence Key**, usadas para representar chaves secretas usadas em HMAC ou AES

Dependendo do tipo de chave pretendida, esta terá diferentes propriedades.

Na tabela abaixo estão representadas os vários membros pertencentes a uma JSK. Os membros marcados a vermelho correspondem a membros pertencentes a todo o tipo de chave, independentemente do seu tipo. Os membros pertencentes exclusivamente a EC (chaves de curvas elípticas) estão marcadas a verde. Os membros pertencentes exclusivamente à representação de uma chave RSA estão, por sua vez, marcados a azul. Adicionalmente, são apresentados exemplos de cada uma das chaves referidas.

Membro	Descrição
kty	Identifica a família do algoritmo criptográfico utilizado que poderá ser EC ou RSA. Membro Obrigatório.
alg	Identifica o algoritmo criptográfico utilizado. Membro Opcional.
use	Identifica o propósito da chave. Os valores poderão ser "sig"(signature) ou "enc"(encryption) Membro Opcional
kid	Membro que usado para fazer correspondência pelo que cada chave deve ter um diferente. Pode ser usado para escolher uma chave específica num JWKS. Membro Opcional.
key_ops	Este membro é um array que contem valores de operações de chave. Este valores podem ser, por exemplo, "verify"(verificar assinatura digital). Membro Opcional.
crv	Udado para identificar a curva criptográfica usada com a chave. Os valores podem ser P-256, P-384 ou P-521.
x	Contem as coordenadas X do ponto da curva.
y	Contem as coordenadas X do ponto da curva.
n	Contem o valor do modulus da chave pública RSA.
e	Contem o valor do expoente para a chave pública RSA.

```

1   {
2     "kty": "RSA",
3     "e": "AQAB",
4     "use": "sig",
5     "kid": "sig-1649987873",
6     "alg": "RS256",
7     "n": "iu5M3_EwtAwXSkwCiB2HDlxHfdYy19Zzf4qk5HX93Hz0c"
8   }

```

Listing 1: Exemplo de JWK - RSA

```

1   {
2     "kty": "EC",
3     "use": "sig",
4     "crv": "P-256",
5     "kid": "sig-1649991262",
6     "x": "cF1C6rJd_Ycvjw8S9t2gUbAycuqmKAHNOwCwdTwIi_I",
7     "y": "vZKEQX1UlGKMmkWSVHAYKOQUsuBSvEDaXGU_1xVh-S8",
8     "alg": "ES256"
9   }

```

Listing 2: Exemplo de JWK - EC

```

1   {
2     "kty": "OKP",
3     "use": "sig",
4     "crv": "Ed25519",
5     "kid": "sig-1649991487",
6     "x": "9Q9hN9deUMw9iBOPDePumzqwo8rc8nOkKjWBVFARpOo",
7     "alg": "EdDSA"
8   }

```

Listing 3: Exemplo de JWK - Octet Key Pair

```

1   {
2     "keys": [
3       {
4         "kty": "oct",
5         "use": "sig",
6         "kid": "sig-1649991849",
7         "k": "UHfd0zc",
8         "alg": "HS256"
9       }
10    ]
11  }

```

Listing 4: Exemplo de JWKS - Octet Sequence Key

Módulos e Geração das Chaves

A validação de um JWT através de um JWKS em node.js é um processo que engloba vários passos, sendo que o primeiro é a geração das chaves, isto é, do ficheiro JWKS. Existem múltiplas opções para a criação das chaves:

- OpenSSL e biblioteca **rsa-pem-to-jwk**

Um dos modos de criação das chaves é através da execução de comandos openssl que geram chaves no formato PEM. Para a criação de uma chave privada RSA de 2048 bit pode ser utilizado o comando:

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

Para a criação de uma chave pública RSA pode ser utilizado:

```
openssl rsa -in private.pem -RSAPublicKey_out -out public.pem
```

Após a geração destas chaves, é possível utilizar a biblioteca **rsa-pem-to-jwk** para converter estas chaves do formato PEM para o formato JWK:

```

1  var fs = require('fs');
2
3  var rsaPemToJwk = require('rsa-pem-to-jwk');
4
5  var pem = fs.readFileSync('privateKey.pem');

```

```

6     var jwk = rsaPemToJwk(pem, {use: 'sig'}, 'public');
7

```

Listing 1: Conversão PEM para JWK

De modo a utilizar esta biblioteca deve ser instalada com o *npm*: `npm install rsa-pem-to-jwk`.

- **Biblioteca node-jose**

A biblioteca node-jose, instalável através do *npm* (`npm install node-jose`), oferece métodos para a implementação do JSON Object Signing and Encryption (JOSE) que possibilitam, por exemplo, a geração de chaves JWK.

```

1  const fs = require("fs");
2
3  const jose = require("node-jose");
4
5  const keyStore = jose.JWK.createKeyStore();
6
7  keyStore.generate("RSA", 2048, { alg: "RS256", use: "sig" }).then((result) => {
8    fs.writeFileSync(
9      "Keys.json",
10     JSON.stringify(keyStore.toJSON(true), null, "  ")
11   );
12 });

```

Listing 2: Conversão PEM para JWK

- **Geradores Online**

Existem também disponíveis online várias ferramentas que geram automaticamente uma JWK. Entre estas ferramentas destaca-se, por exemplo, a `mkjwk` (<https://mkjwk.org/>) através da qual é possível gerar vários tipos diferentes de chave diferentes de forma fácil e intuitiva.

Como funciona a validação através de JWKS

Após a geração da chave, o JWT deve ser gerado e assinado com a chavepropriada. Esta assinatura é realizada através de métodos da JWS. Seguidamente, pode dar-se o processo de verificação. Bibliotecas como `jsonwebtoken` e `jwk-to-pem` facilitam este processo. Deste modo, o token tem que ser decodificado para que seja possível aceder ao *kid*. O *kid* irá permitir saber qual a chave pública no ficheiro JWKS correspondente ao token cuja assinatura queremos validar. Dado o acesso à chave pública, a verificação é feita através do método *verify* da biblioteca `jsonwebtoken`.

```

1 router.post("/verify", async (req, res) => {
2   let resourcePath = "token/jwks";
3
4   let token = req.body;
5
6   let decodedToken = jwt.decode(token, { complete: true });
7
8   let kid = decodedToken.headers.kid;
9
10  return new Promise(function (resolve, reject) {
11    var jwksPromise = config.request("GET", resourcePath);
12
13    jwksPromise
14      .then(function (jwksResponse) {
15        const jwktopem = require("jwk-to-pem");
16
17        const jwt = require("jsonwebtoken");
18
19        const [firstKey] = jwksResponse.keys(kid);
20        const publicKey = jwktopem(firstKey);

```

```

21     try {
22         const decoded = jwt.verify(token, publicKey);
23         resolve(decoded);
24     } catch (e) {
25         reject(e);
26     }
27 }
28 .catch(function (error) {
29     reject(error);
30 });
31 });
32 });

```

Vulnerabilidades

Validar um JWT com JWKS é o método de autenticação mais seguro dado que a assinatura ocorre como uma operação criptográfica.

No entanto, é um método que não está completamente isento de vulnerabilidades. Um dos ataques possíveis à chave é através de um dos parâmetros do *header* do JWT: **jku**. Este parâmetro pode ser utilizado para especificar o JSON Web Key Set URL, isto é, indica onde a aplicação pode encontrar a JWK usada para verificar a assinatura do token.

```

1  {
2      "alg": "RS256",
3      "typ": "JWT",
4      "jku": "https://example.com/key.json"
5  }

```

Listing 5: Exemplo de um *header* de um JWT com o parâmetro jku

Dentro do ficheiro key.json indicado no parâmetro jku está a chave pública em formato json, que poderá incluir-se em qualquer um dos tipos apresentados acima. Este pode ser um ponto de vulnerabilidade no sentido em que um atacante poderá alterar esta URL para que o jku passe a apontar para uma chave feita pelo próprio atacante em vez da chave válida. Desta modo, o atacante consegue assinar tokens maliciosos com a sua chave privada que serão verificados.

Existem, contudo, formas de contornar este problema através de *URL filtering*.

JWA

Conceito

A especificação JSON Web Algorithms (JWA) regista os algoritmos e os identificadores necessários para o JWS, JWK e JWE. Ter os algoritmos e os identificadores registos à parte, tem a vantagem de se poder fazer alterações nestes sem influencias as 3 especificações mencionada e vice-versa, ou seja, na necessidade de fazer alterações no JWS, JWE ou JWK este "documento" não é influenciado.

alg Parameter Value	Digital Signature or MAC Algorithm	Implementation Requirements
HS256	HMAC using SHA-256 hash algorithm	REQUIRED
HS384	HMAC using SHA-384 hash algorithm	OPTIONAL
HS512	HMAC using SHA-512 hash algorithm	OPTIONAL
RS256	RSASSA using SHA-256 hash algorithm	RECOMMENDED
RS384	RSASSA using SHA-384 hash algorithm	OPTIONAL
RS512	RSASSA using SHA-512 hash algorithm	OPTIONAL
ES256	ECDSA using P-256 curve and SHA-256 hash algorithm	RECOMMENDED+
ES384	ECDSA using P-384 curve and SHA-384 hash algorithm	OPTIONAL
ES512	ECDSA using P-512 curve and SHA-512 hash algorithm	OPTIONAL
none	No digital signature or MAC value included	REQUIRED

Figura 22: Algoritmos usados em assinaturas

alg Parameter Value	Key Management Algorithm	Implementation Requirements
RSA1_5	RSAES-PKCS1-V1_5 [RFC3447]	REQUIRED
RSA-OAEP	RSAES using Optimal Asymmetric Encryption Padding (OAEP) [RFC3447], with the default parameters specified by RFC 3447 in Section A.2.1	OPTIONAL
A128KW	Advanced Encryption Standard (AES) Key Wrap Algorithm [RFC3394] using the default initial value specified in Section 2.2.3.1 and using 128 bit keys	RECOMMENDED
A256KW	AES Key Wrap Algorithm using the default initial value specified in Section 2.2.3.1 and using 256 bit keys	RECOMMENDED
dir	Direct use of a shared symmetric key as the Content Encryption Key (CEK) for the block encryption step (rather than using the symmetric key to wrap the CEK)	RECOMMENDED
ECDH-ES	Elliptic Curve Diffie-Hellman Ephemeral Static [RFC6090] key agreement using the Concat KDF, as defined in Section 5.8.1 of [NIST.800-56A], with the agreed-upon key being used directly as the Content Encryption Key (CEK) (rather than being used to wrap the CEK), as specified in Section 4.7	RECOMMENDED+

Figura 23: Algoritmos usados na produção da JWE Encrypted Key

kty Parameter Value	Key Type	Implementation Requirements
EC	Elliptic Curve [DSS] key type	RECOMMENDED+
RSA	RSA [RFC3447] key type	REQUIRED
oct	Octet sequence key type (used to represent symmetric keys)	RECOMMENDED+

Figura 24: Identificadores do tipo de chave (JWK)

O uso do '+' nos requisitos indica que é provável que no futuro esse requisito aumente de "força".

Parte I

Conclusão

Sendo este o trabalho prático de grupo da Unidade Curricular de Engenharia de Segurança, foi, de certa forma, mais desafiante, uma vez que nos deparamos com algo diferente de qualquer outro trabalho prático, numa linguagem que dentro do curso era um pouco desconhecida.

Apesar de neste momento já estarmos mais ambientados àquilo que são as frameworks Web, foram, inevitavelmente, surgindo algumas dificuldades no que toca à descoberta de estratégias que melhor se ajustassem à implementação de alguns requisitos e estruturação do projeto.

A nossa maior dificuldade foi conseguir entender de certa forma como todas estas especificações estavam interligadas. No entanto, depois de alguma dedicação e esforço, conseguimos ultrapassar esta dificuldade e várias outras com que mais à frente nos deparámos.

É de notar que foi o primeiro projeto que envolveu dois grupos com um número de pessoas mais elevado que o normal o que significou uma organização e comunicação entre os elementos levando a uma melhor realização do projeto.

Assim, a realização do mesmo foi bastante importante para continuar a adquirir experiência e conhecimento, à medida que se entende são conhecimentos com o que vamos encontrar futuramente. Conseguimos, desta forma, consolidar não só todos os conhecimentos adquiridos nas aulas práticas e teóricas, como também vários outros que foram sendo necessários durante a realização deste projeto.