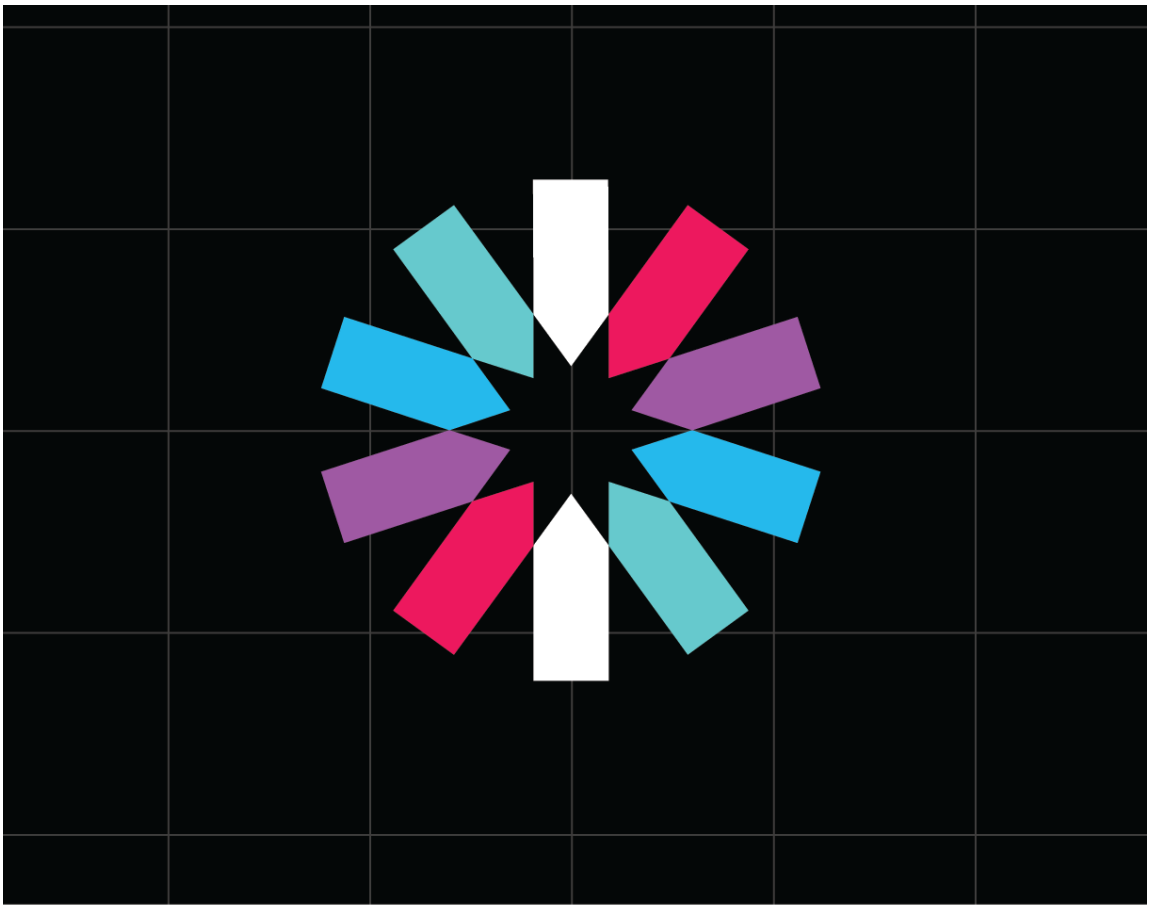

JWT HANDBOOK

By Sebastián Peyrott

Auth0

El manual de JWT

Sebastián E. Peyrott, Auth0 Inc.

Versión 0.12.0, 2016-2017

Contenido

Gracias especiales

Sin ningún orden especial:**Prosper Otemuyiwa**(por proporcionar el ejemplo de identidad federada del capítulo 2),**diego poza**(por revisar este trabajo y mantener mis manos libres mientras trabajaba en él), **Matías Woloski**(por revisar las partes difíciles de este trabajo),**Martín Gontovnikas**(por aceptar mis peticiones y hacer todo lo posible para que el trabajo sea más cómodo),**Bárbara Mercedes Muñoz Cruzado** (por hacer que todo se vea bien),**alejo fernandez**y**Víctor Fernández**(por hacer el trabajo de frontend y backend para distribuir este manual),**Sergio Fruto**(por hacer todo lo posible para ayudar a sus compañeros de equipo),**federico jack**(por mantener todo funcionando y aún así encontrar el tiempo para escuchar a todos y cada uno).

Capítulo 1

Introducción

JSON Web Token, o JWT ("jot") para abreviar, es un estándar para*sin peligro*paso*reclamación (es*en entornos con limitaciones de espacio. Ha encontrado su camino en todos[1]importante[2]web[3]marcos[4]. La simplicidad, la compacidad y la facilidad de uso son características clave de su arquitectura. Aunque sistemas mucho más complejos[5]todavía están en uso, los JWT tienen una amplia gama de aplicaciones. En este pequeño manual, cubriremos los aspectos más importantes de la arquitectura de los JWT, incluida su representación binaria y los algoritmos utilizados para construirlos, al tiempo que observamos cómo se usan comúnmente en la industria.

1.1 ¿Qué es un token web JSON?

Un token web JSON se ve así (nuevas líneas insertadas para facilitar la lectura):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

Si bien esto parece un galimatías, en realidad es una muy**compacto**,**imprimible**representación de una serie de *reclamación (es*, junto con un**firma**para verificar su autenticidad.

```
{
  "algo":"HS256",
  "tipo":"JWT"
}

{
```

[1] https://github.com/auth0/express-jwt
[2] https://github.com/nsarno/golpe
[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",
  "nombre":"Juan Doe",
  "administración":verdadero
}
```

Las reclamaciones son*definiciones*o*afirmaciones*hecho sobre una determinada parte u objeto. Algunas de estas afirmaciones y su significado se definen como parte de la especificación JWT. Otros son definidos por el usuario. La magia detrás de los JWT es que estandarizan ciertos reclamos que son útiles en el contexto de algunas operaciones comunes. Por ejemplo, una de estas operaciones comunes es establecer la identidad de cierta parte. Entonces, una de las afirmaciones estándar que se encuentran en los JWT es la*sub*(de "sujeto") afirmación. Echaremos un vistazo más profundo a cada uno de los reclamos estándar enCapítulo 3.

Otro aspecto clave de los JWTs es la posibilidad de firmarlos, utilizando JSON Web Signatures (JWS, RFC 7515[6]), y/o encriptarlos, usando JSON Web Encryption (JWE, RFC 7516[7]). Junto con JWS y JWE, los JWT brindan una solución poderosa y segura para muchos problemas diferentes.

1.2 ¿Qué problema resuelve?

Aunque el propósito principal de los JWT es transferir reclamos entre dos partes, podría decirse que el aspecto más importante de esto es la*esfuerzo de estandarización*en forma de un*formato de contenedor simple, opcionalmente validado y/o encriptado*. Las soluciones ad hoc para este mismo problema se han implementado tanto de forma privada como pública en el pasado. Estándares más antiguos[8]para establecer reclamos sobre ciertas partes también están disponibles. Lo que JWT trae a la mesa es un*simple*, útil formato de contenedor estándar.

Aunque la definición dada es un poco abstracta hasta ahora, no es difícil imaginar cómo se pueden usar: sistemas de inicio de sesión (aunque son posibles otros usos). Echaremos un vistazo más de cerca a las aplicaciones prácticas enCapitulo 2. Algunas de estas aplicaciones incluyen:

- Autenticación
- Autorización
- Identidad federada
- Sesiones del lado del cliente (sesiones "sin estado")
- Secretos del lado del cliente

1.3 Un poco de historia

El grupo JSON Object Signing and Encryption (JOSE) se formó en el año 2011[9]. El objetivo del grupo era "*estandarizar el mecanismo para la protección de la integridad (firma y MAC) y el cifrado, así como el formato de las claves y los identificadores de algoritmos para respaldar la interoperabilidad de los servicios de seguridad para los protocolos que utilizan JSON*". Para el año 2013 una serie de borradores, incluido un recetario

[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/

con diferentes ejemplos del uso de las ideas producidas por el grupo, estaban disponibles. Estos borradores se convertirían más tarde en los RFC JWT, JWS, JWE, JWK y JWA. A partir del año 2016, estos RFC se encuentran en proceso de seguimiento de estándares y no se han encontrado erratas en ellos. El grupo está actualmente inactivo.

Los principales autores detrás de las especificaciones son Mike Jones.[10], Nat Sakimura[11], John Bradley[12]y joe hildebrand[13].

[10] http://autoemitido.info/
[11] https://nat.sakimura.org/
[12] https://www.linkedin.com/in/ve7jtb
[13] https://www.linkedin.com/in/hildjj

Capitulo 2

Aplicaciones prácticas

Antes de profundizar en la estructura y construcción de un JWT, veremos varias aplicaciones prácticas. Este capítulo le dará una idea de la complejidad (o simplicidad) de las soluciones comunes basadas en JWT que se usan en la industria hoy en día. Todo el código está disponible en repositorios públicos.[1] por su conveniencia. Tenga en cuenta que las siguientes demostraciones son *no* destinado a ser utilizado en la producción. Los casos de prueba, el registro y las mejores prácticas de seguridad son esenciales para el código listo para producción. Estas muestras son solo para fines educativos y, por lo tanto, siguen siendo simples y precisas.

2.1 Sesiones del lado del cliente/sin estado

La llamada *apátrida* de hecho, las sesiones no son más que datos del lado del cliente. El aspecto clave de esta aplicación radica en el uso de *firma* y posiblemente *cifrado* para autenticar y proteger el contenido de la sesión. Los datos del lado del cliente están sujetos a *manipulación*. Como tal, debe ser manejado con mucho cuidado por el backend.

Los JWT, en virtud de JWS y JWE, pueden proporcionar varios tipos de firmas y cifrado. Las firmas son útiles para *validar* los datos contra la manipulación. El cifrado es útil para *proteger* los datos sean leídos por terceros.

La mayoría de las sesiones solo necesitan ser firmadas. En otras palabras, no hay preocupación por la seguridad o la privacidad cuando los datos almacenados en ellos son leídos por terceros. Un ejemplo común de una reclamación que por lo general puede ser leída con seguridad por terceros es la *sub* pretensión ("sujeto"). El asunto del reclamo generalmente identifica a una de las partes con la otra (piense en ID de usuario o correos electrónicos). No es un requisito que esta reclamación sea *único*. En otras palabras, es posible que se requieran reclamos adicionales para identificar de manera única a un usuario. Esto se deja a los usuarios para decidir.

Un reclamo que no se puede dejar abierta de manera adecuada podría ser un reclamo de "artículos" que represente el carrito de compras de un usuario. Este carro puede estar lleno de artículos que el usuario está a punto de comprar y

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

por lo tanto, están asociados a su sesión. Un tercero (una secuencia de comandos del lado del cliente) podría recopilar estos elementos si están almacenados en un JWT sin cifrar, lo que podría generar problemas de privacidad.

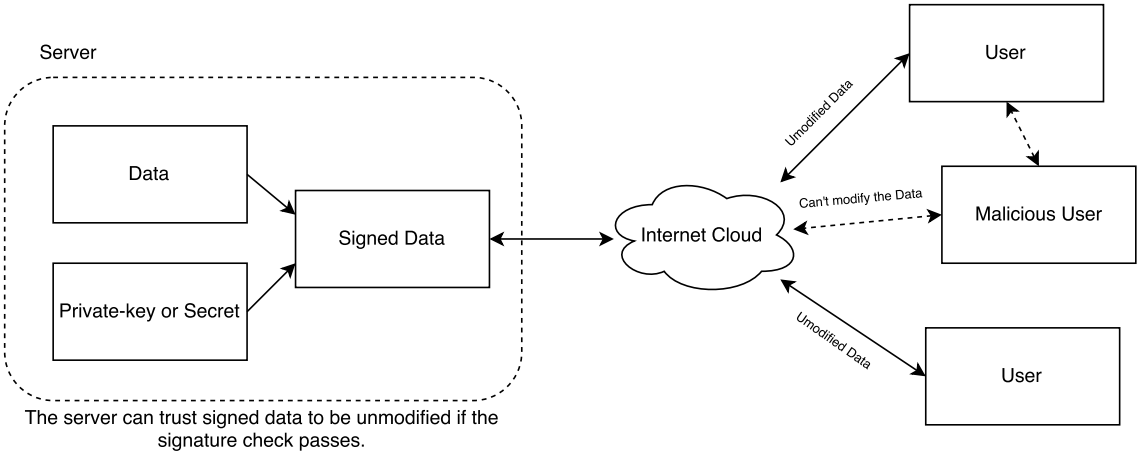

Figura 2.1: Datos firmados del lado del cliente

2.1.1 Consideraciones de seguridad

2.1.1.1 Eliminación de firmas

Un método común para atacar un JWT firmado es simplemente eliminar la firma. Los JWT firmados se construyen a partir de tres partes diferentes: el encabezado, la carga útil y la firma. Estas tres partes se codifican por separado. Como tal, es posible eliminar la firma y luego *cambio* el encabezado para reclamar el JWT es *no firmado*. El uso descuidado de ciertas bibliotecas de validación de JWT puede dar como resultado que los tokens sin firmar se tomen como tokens válidos, lo que puede permitir que un atacante modifique la carga útil a su discreción. Esto se soluciona fácilmente asegurándose de que la aplicación que realiza la validación no considere válidos los JWT sin firmar.

Figura 2.2: Eliminación de firmas

2.1.1.2 Falsificación de solicitud entre sitios (CSRF)

Los ataques de falsificación de solicitudes entre sitios intentan realizar solicitudes en sitios en los que el usuario ha iniciado sesión engañando al navegador del usuario para que envíe una solicitud desde un sitio diferente. Para lograr esto, un sitio (o elemento) especialmente diseñado debe contener la URL del destino. Un ejemplo común es un <img> etiqueta incrustada en una página maliciosa con elorigenapuntando al objetivo del ataque. Por ejemplo:

```
<!-- Esto está incrustado en el sitio de otro dominio -->
<imgorigen="http://target.site.com/add-user?user=name&subvención=administrador">
```

Lo anterior <imagen>etiqueta enviará una solicitud atarget.site.comcada vez que se carga la página que lo contiene. Si el usuario había iniciado sesión previamente entarget.site.comy el sitio usó una cookie para mantener activa la sesiónve, esta cookie será se no comobien. Si el sitio de destino d o no implementa ninguna técnica de mitigación de CSRF, la solicitud se manejará como una solicitud válida en nombre del usuario. Los JWT, como cualquier otro dato del lado del cliente, se pueden almacenar como cookies.

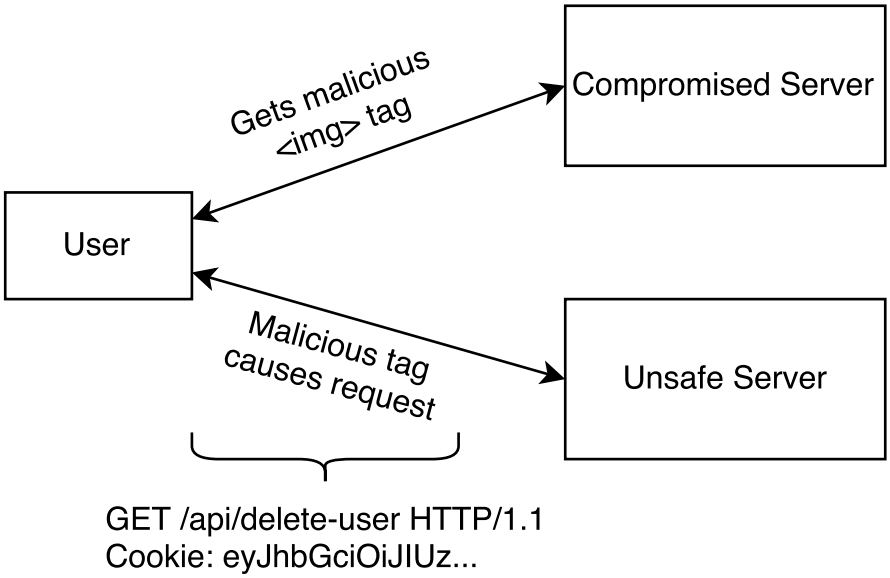

Figura 2.3: Falsificación de solicitud entre sitios

Los JWT de corta duración pueden ayudar en este caso. Las técnicas comunes de mitigación de CSRF incluyen encabezados especiales que se agregan a las solicitudes solo cuando se realizan desde el origen correcto, por cookies de sesión y por tokens de solicitud. Si los JWT (y los datos de sesión) no se almacenan como cookies, los ataques CSRF no son posibles. Sin embargo, los ataques de secuencias de comandos entre sitios aún son posibles.

2.1.1.3 Secuencias de comandos entre sitios (XSS)

Los ataques de secuencias de comandos entre sitios (XSS) intentan inyectar JavaScript en sitios de confianza. El JavaScript inyectado puede robar tokens de las cookies y el almacenamiento local. Si se filtra un token de acceso antes de que caduque, un usuario malintencionado podría usarlo para acceder a los recursos protegidos. Los ataques XSS comunes generalmente son causados por una validación incorrecta de los datos pasados al backend (de manera similar a los ataques de inyección SQL).

Un ejemplo de ataque XSS podría estar relacionado con la sección de comentarios de un sitio público. Cada vez que un usuario agrega un comentario, el backend lo almacena y lo muestra a los usuarios que cargan la sección de comentarios. Si el backend no desinfecta los comentarios, un usuario malintencionado podría escribir un comentario de tal manera que el navegador podría interpretarlo como un <guion>etiqueta. Entonces, un usuario malintencionado podría insertar código JavaScript arbitrario y ejecutarlo en el navegador de cada usuario, robando así las credenciales almacenadas como cookies y en loc. al almacenamiento.

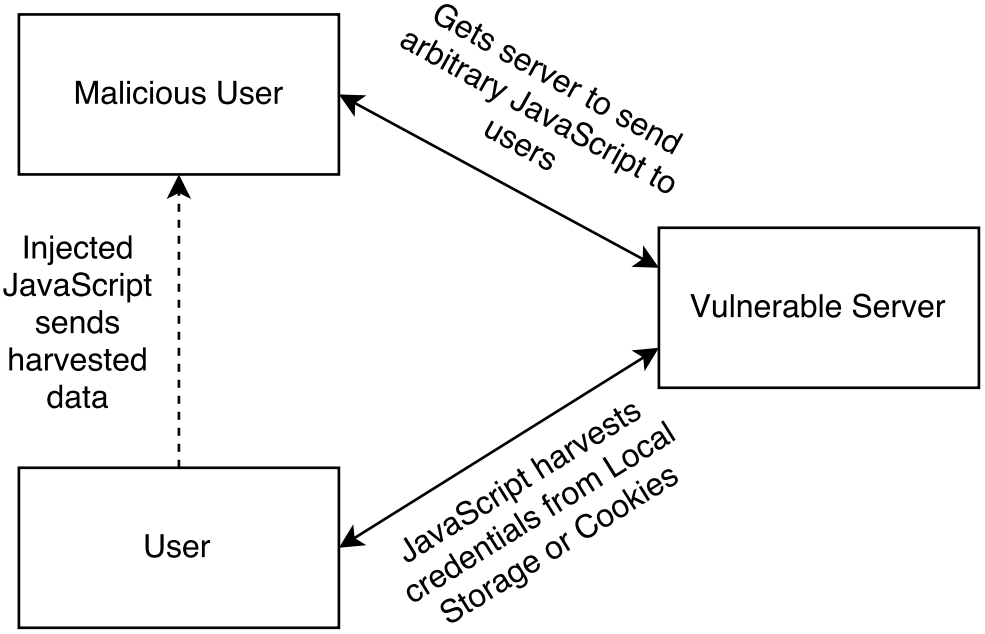

Figura 2.4: Secuencias de comandos entre sitios persistentes

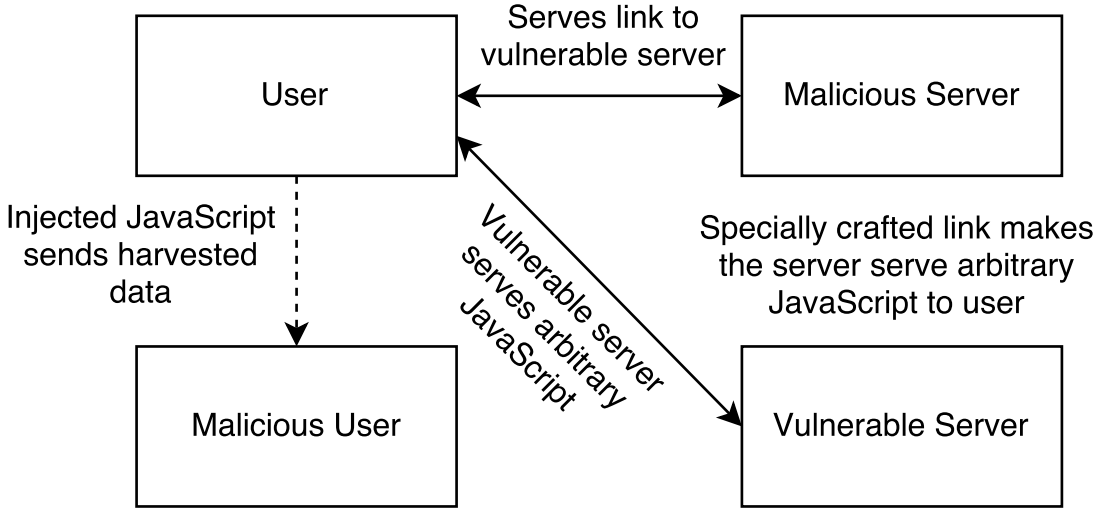

Figura 2.5: Re Cross Site Scriptin reflexivo gramo

Las técnicas de mitigación se basan en la validación adecuada de todos los datos pasados al backend. En particular, todos los datos recibidos de los clientes siempre deben ser desinfectados. Si se utilizan cookies, es posible protegerlas contra el acceso de JavaScript configurando el indicador HttpOnly[2]. El indicador HttpOnly, si bien es útil, no protegerá la cookie de los ataques CSRF.

2.1.2 ¿Son útiles las sesiones del lado del cliente?

Hay ventajas y desventajas en cualquier enfoque, y las sesiones del lado del cliente no son una excepción.[3]. Algunas aplicaciones pueden requerir grandes sesiones. Enviar este estado de un lado a otro para cada solicitud (o grupo de solicitudes) puede superar fácilmente los beneficios de la conversación reducida en el backend. Es necesario un cierto equilibrio entre los datos del lado del cliente y las búsquedas en la base de datos en el backend. Esto depende del modelo de datos de su aplicación. Algunas aplicaciones no se asignan bien a las sesiones del lado del cliente. Otros pueden depender completamente de los datos del lado del cliente. ¡La última palabra sobre este asunto es suya! Ejecute puntos de referencia, estudie los beneficios de mantener cierto estado del lado del cliente. ¿Los JWT son demasiado grandes? ¿Tiene esto un impacto en el ancho de banda? ¿Este ancho de banda agregado anula la latencia reducida en el backend? ¿Se pueden agregar solicitudes pequeñas en una sola solicitud más grande? ¿Estas solicitudes aún requieren búsquedas en grandes bases de datos? Responder a estas preguntas le ayudará a decidir el enfoque correcto.

2.1.3 Ejemplo

Para nuestro ejemplo, haremos una aplicación de compras simple. El carrito de compras del usuario se almacenará en el lado del cliente. En este ejemplo, hay varios JWT presentes. Nuestro carro de la compra será uno de ellos.

- Un JWT para el token de ID, un token que lleva la información del perfil del usuario, útil para la interfaz de usuario.
- Un JWT para interactuar con el backend de la API (el token de acceso).
- Un JWT para nuestro estado del lado del cliente: el carrito de compras.

Así es como se ve el carrito de compras cuando se decodifica:

```
{
  "elementos":[
    0,
    2,
    4
  ],
  "Yo en":1493139659,
  "Exp":1493143259
}
```

Cada elemento se identifica mediante un ID numérico. El JWT codificado y firmado se ve así:

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

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJpdGVtcyI6WzAsMiw0XSwiaWF0IjoxNDkzMTM5NjU5LCJleHAiOjE0OTMxNDMyNTl9.
932ZxtZzy1qhLXs932hd04J58Ihbg5_g_rIrj-Z16Js

Para representar los artículos en el carrito, la interfaz solo necesita recuperarlos de su cookie:

```
función poblarcarrito(){
    constante carritoElem=ps('#carro');
    carritoElem.vacío();

    constante carroToken=Galletas.obtener('carro')
    ; si(!carroToken){
        devolver;
    }

    constante carro=jwt_decode(cartToken).elementos;

    carro.para cada(Identificación del producto=> {
        constante nombre=elementos.encontrar(artículo=>artículo.identificación==Identificación del
        producto).nombre; carritoElem.adjuntar(`<li>${nombre}</li>`);
    });
}
```

Tenga en cuenta que la interfaz no verifica la firma, simplemente decodifica el JWT para que pueda mostrar su contenido. Los controles reales son realizados por el backend. Todos los JWT están verificados.

Aquí está la verificación de back-end para la validez del carro JWT implementado como un middleware Express:

```
función carritoValidador(requerido,resolución,Siguiente){
    si(!requerimiento.galletas.carro){
        requerimiento.carro= {elementos:
    []}; }más{
        probar{
            requerimiento.carro= {
                elementos:jwt.verificar(requerimiento.galletas.carro,
                                        proceso.env.AUTH0_CART_SECRET,
                                        cartVerifyJwtOptions).elementos
            };
        }captura(mi){
            requerimiento.carro= {elementos:[]};
        }
    }

    Siguiente();
}
```

Cuando se agregan elementos, el backend construye un nuevo JWT con el nuevo elemento y una nueva firma:

```
aplicación.obtener('/protegido/añadir_elemento',validador de id,carritoValidador,(requerido,resolución)=> {
```

```
    requerimiento.carro.elementos.empujar(parseInt(requerimiento.consulta.identificación));

    constantecarro nuevo=jwt.señal(requerimiento.carro,
                                   proceso.env.AUTH0_CART_SECRET,
                                   cartSignJwtOptions);

    resolución.Galleta('carro',carro nuevo, {
        MaxAge:1000*60*60
    });

    resolución.final();

    consola.Iniciar sesión(Id. de artículo ${requerimiento.consulta.identificación} añadido al carrito.`);
});
```

Tenga en cuenta que las ubicaciones con el prefijo /protegidotambién están protegidos por el token de acceso a la API. Esta es la configuración usandoexpress-jwt:

```
aplicación.usar('/protegido',expresoJwt({
    secreto:jwksCliente.expresoJwtSecreto(jwksOpts), editor:
    proceso.env.AUTH0_API_EMISOR, audiencia:proceso.env.
    AUTH0_API_AUDIENCIA, solicitudPropiedad:'token de
    acceso', obtenerToken:requerimiento=> {

        devolverrequerimiento.galletas['token_de_acceso'];
    }
}));
```

En otras palabras, el /protegido/añadir_elementoel punto final primero debe pasar el paso de validación del token de acceso antes de validar el carrito. Un token valida el acceso (autorización) a la API y el otro token valida la integridad de los datos del lado del cliente (el carrito).

El token de acceso y el token de ID son asignados por Auth0 a nuestra aplicación. Esto requiere configurar un cliente[4]y un punto final de API[5]utilizando el panel de Auth0[6]. Luego, estos se recuperan utilizando la biblioteca JavaScript Auth0, llamada por nuestra interfaz:

```
//Auth0 ID de cliente
constanteIdentificación del cliente="t42WY87weXzepAdUlwMiHYRBQj9qWVAT"; //
Auth0 Dominio
constantedominio="speyrott.auth0.com";

constanteautor0=nuevoventana.autor0.WebAuth({
    dominio:dominio, Identificación del
    cliente:Identificación del cliente,
    audiencia:'/protegido',
```

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

```
        alcance:'compra de perfil openid', tipo de respuesta:
        'ficha_id_token', redirigirUri:'http://localhost:3000/
        autorización/', modo de respuesta:'formulario_post'

});

//(...)

ps('#botón-iniciar sesión').en('hacer clic',función(evento){
        autor0.autorizar();
});
```

losaudienciala reclamación debe coincidir con la configuración de su punto de conexión de la API mediante el panel Auth0.

El servidor de autenticación y autorización Auth0 muestra una pantalla de inicio de sesión con nuestra configuración y luego redirige a nuestra aplicación en una ruta específica con los tokens que solicitamos. Estos son manejados por nuestro backend que simplemente los configura como cookies:

```
aplicación.correo('/autenticación',(requerido,resolución)=> {
        resolución.Galleta('token_de_acceso',requerimiento.cuerpo.token_de_acceso, {
                solo http:verdadero,
                MaxAge:requerimiento.cuerpo.expira en*1000
        });
        resolución.Galleta('id_token',requerimiento.cuerpo.id_token, {
                MaxAge:requerimiento.cuerpo.expira en*1000
        });
        resolución.redirigir('/');
});
```

La implementación de técnicas de mitigación de CSRF se deja como un ejercicio para el lector. El ejemplo completo de este código se puede encontrar en elmuestras/sesiones sin estadodirectorio.

2.2 Identidad federada

identidad federada[7]Los sistemas permiten que diferentes partes, posiblemente no relacionadas, compartan servicios de autenticación y autorización con otras partes. En otras palabras, la identidad de un usuario está centralizada. Existen varias soluciones para la gestión de identidades federadas: SAML[8]y Open ID Connect[9]son dos de los más comunes. Ciertas empresas ofrecen productos especializados que centralizan la autenticación y la autorización. Estos pueden implementar uno de los estándares mencionados anteriormente o usar algo completamente diferente. Algunas de estas empresas utilizan JWT para este fin.

El uso de JWT para la autenticación y autorización centralizadas varía de una empresa a otra, pero el flujo esencial del proceso de autorización es:

[7]https://auth0.com/blog/2015/09/23/qué-es-y-cómo-funciona-el-inicio-de-inicio-de-single/
[8]http://saml.xml.org/saml-especificaciones
[9]https://openid.net/conectar/

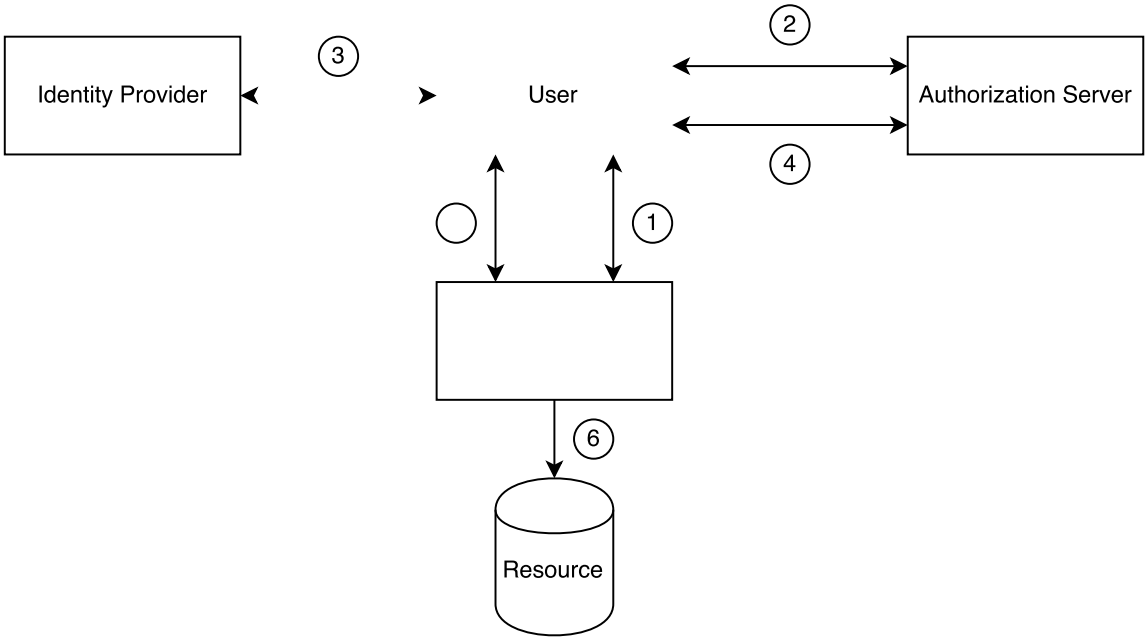

Figura 2.6: Flujo de identidad federada común

1. El usuario intenta acceder a un recurso controlado por un servidor.
2. El usuario no tiene las credenciales adecuadas para acceder al recurso, por lo que el servidor redirige al usuario al servidor de autorización. El servidor de autorización está configurado para permitir que los usuarios inicien sesión con las credenciales administradas por un proveedor de identidad.
3. El servidor de autorización redirige al usuario a la pantalla de inicio de sesión del proveedor de identidad.
4. El usuario inicia sesión con éxito y es redirigido al servidor de autorización. El servidor de autorización utiliza las credenciales proporcionadas por el proveedor de identidad para acceder a las credenciales requeridas por el servidor de recursos.
5. El usuario es redirigido tiene al servidor de recursos por El servidor de autorización. La e solicitud ahora las credenciales correctas Th tials necesarios para acceder fuente.
6. usuario obtiene acceso t o al recurso con éxito.

Todos datos pasados desde los servidores r al servidor fluye a través del usuario al estar integrado el redireccionamiento solicita (generalmente como parte de la URL). Esto hace que tra en t nsport security (TLS) y seguridad de datos básico.

Las credenciales devueltas de th El servidor de autorización para el usuario se puede codificar como un JWT. Si el servidor de autorización permite inicios de sesión a través de un proveedor de identidad (como es el caso en este ejemplo), se puede decir que el servidor de autorización proporciona una interfaz unificada y datos unificados (el JWT) al usuario.

Para nuestro ejemplo más adelante en este sección, usaremos Auth 0 como servidor de autorización y manejar la base de inicio de sesión a través de Twitter, Faceb ook, y unr fuera del molino datos de usuarios.

2.2.1 Tokens de acceso y actualización

Los tokens de acceso y actualización son dos tipos de tokens que verá mucho al analizar diferentes soluciones de identidad federada. Explicaremos brevemente qué son y cómo ayudan en el contexto de la autenticación y autorización.

Ambos conceptos suelen implementarse en el contexto de la especificación OAuth2[10]. La especificación OAuth2 define una serie de pasos necesarios para brindar acceso a los recursos al separar el acceso de la propiedad (en otras palabras, permite que varias partes con diferentes niveles de acceso accedan al mismo recurso). Varias partes de estos pasos son *implementación definida*. Es decir, es posible que las implementaciones de OAuth2 de la competencia no sean interoperables. Por ejemplo, el formato binario real de los tokens es *no especificado*. Su propósito y funcionalidad es.

fichas de accesoson tokens que dan a quienes los tienen acceso a recursos protegidos. Estos tokens suelen ser de corta duración y pueden tener una fecha de caducidad incrustada. También pueden llevar o estar asociados con información adicional (por ejemplo, un token de acceso puede llevar la dirección IP desde la que se permiten las solicitudes). Estos datos adicionales están definidos por la implementación.

Fichas de actualización, por otro lado, permite a los clientes solicitar nuevos tokens de acceso. Por ejemplo, después de que haya caducado un token de acceso, un cliente puede realizar una solicitud de un nuevo token de acceso al servidor de autorización. Para satisfacer esta solicitud, se requiere un token de actualización. A diferencia de los tokens de acceso, los tokens de actualización suelen ser de larga duración.

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

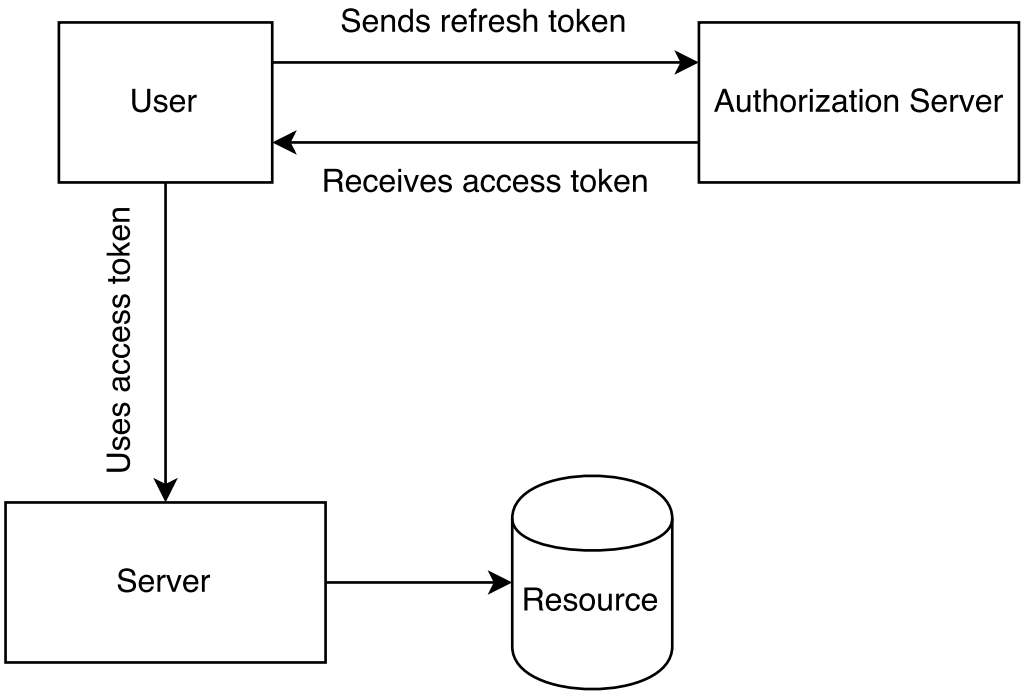

Figura 2.7: Tokens de actualización y acceso

El aspecto clave de la separación entre tokens de acceso y actualización radica en la posibilidad de hacer que los tokens de acceso sean fáciles de validar. Un token de acceso que lleva una firma (como un JWT firmado) puede ser validado por el servidor de recursos por sí solo. No es necesario ponerse en contacto con el servidor de autorizaciones para este fin.

Los tokens de actualización, por otro lado, requieren acceso al servidor de autorización. Al mantener la validación separada de las consultas al servidor de autorización, es posible una mejor latencia y patrones de acceso menos complejos. La seguridad adecuada en caso de fugas de tokens se logra haciendo que los tokens de acceso tengan una vida útil lo más breve posible e incorporando verificaciones adicionales (como verificaciones de clientes) en ellos.

Los tokens de actualización, en virtud de su larga duración, deben protegerse contra filtraciones. En el caso de una fuga, puede ser necesaria la inclusión en la lista negra en el servidor (los tokens de acceso de corta duración obligan a usar tokens de actualización) eventualmente, protegiendo así el recurso después de que se incluye en la lista negra y caducan todos los tokens de acceso).

> Nota: los conceptos de token de acceso y el token de actualización eran producido en OAuth2. OAuth 1.0 y 1.0a usan la palabra *simbólico* d intrínsecamente diferentes.

2. 2.2 JWT y OAuth2

Aunque OAuth2 no menciona el formato de sus tokens, los JWT son una buena combinación para sus requisitos. Los JWT firmados son buenos tokens de acceso, ya que pueden codificar todos los datos necesarios

para diferenciar los niveles de acceso a un recurso, pueden llevar una fecha de caducidad y están firmados para evitar consultas de validación contra el servidor de autorizaciones. Varios proveedores de identidad federados emiten tokens de acceso en formato JWT.

Los JWT también se pueden usar para tokens de actualización. Sin embargo, hay menos razones para usarlos para este propósito. Dado que los tokens de actualización requieren acceso al servidor de autorización, la mayoría de las veces bastará con un UUID simple, ya que no es necesario que el token lleve una carga útil (aunque puede estar firmado).

2.2.3 JWT y OpenID Connect

Conexión de identificación abierta[11]es un esfuerzo de estandarización para traer casos de uso típicos de OAuth2 bajo una especificación común bien definida. Dado que muchos detalles detrás de OAuth2 se dejan a la elección de los implementadores, OpenID Connect intenta proporcionar definiciones adecuadas para las partes que faltan. Específicamente, OpenID Connect define una API y un formato de datos para realizar flujos de autorización de OAuth2. Además, proporciona una capa de autenticación construida sobre este flujo. El formato de datos elegido para algunas de sus partes es JSON Web Token. En particular, el token de identificación[12]es un tipo especial de token que lleva información sobre el usuario autenticado.

2.2.3.1 Flujos de OpenID Connect y JWT

OpenID Connect define varios flujos que devuelven datos de diferentes maneras. Algunos de estos datos pueden estar en formato JWT.

- **Flujo de autorización**: el cliente solicita un código de autorización al punto final de autorización (/autorizar).Este código se puede usar contra el punto final del token (/simbólico)para solicitar un token de ID (en formato JWT), un token de acceso o un token de actualización.
- **flujo implícito**: el cliente solicita tokens directamente desde el punto final de autorización (/autorizar).Los tokens se especifican en la solicitud. Si se solicita un token de ID, se devuelve en formato JWT.
- **Flujo híbrido**: el cliente solicita un código de autorización y ciertos tokens desde el punto final de autorización (/autorizar).Si se solicita un token de ID, se devuelve en formato JWT. Si no se solicita un token de ID en este paso, se puede solicitar más tarde directamente desde el punto final del token (/simbólico).

2.2.4 Ejemplo

Para este ejemplo usaremos Auth0[13]como servidor de autorización. Auth0 permite configurar dinámicamente diferentes proveedores de identidad. En otras palabras, siempre que un usuario intente iniciar sesión, los cambios realizados en el servidor de autorización pueden permitir que los usuarios inicien sesión con diferentes proveedores de identidad (como Twitter, Facebook, etc.). Las aplicaciones no necesitan comprometerse con proveedores específicos una vez implementadas. Entonces nuestro ejemplo

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

puede ser bastante simple. Configuramos la pantalla de inicio de sesión de Auth0 usando la biblioteca Auth0.js[14]en todos nuestros servidores de muestra. Una vez que un usuario inicia sesión en un servidor, también tendrá acceso a los otros servidores (incluso si no están interconectados).

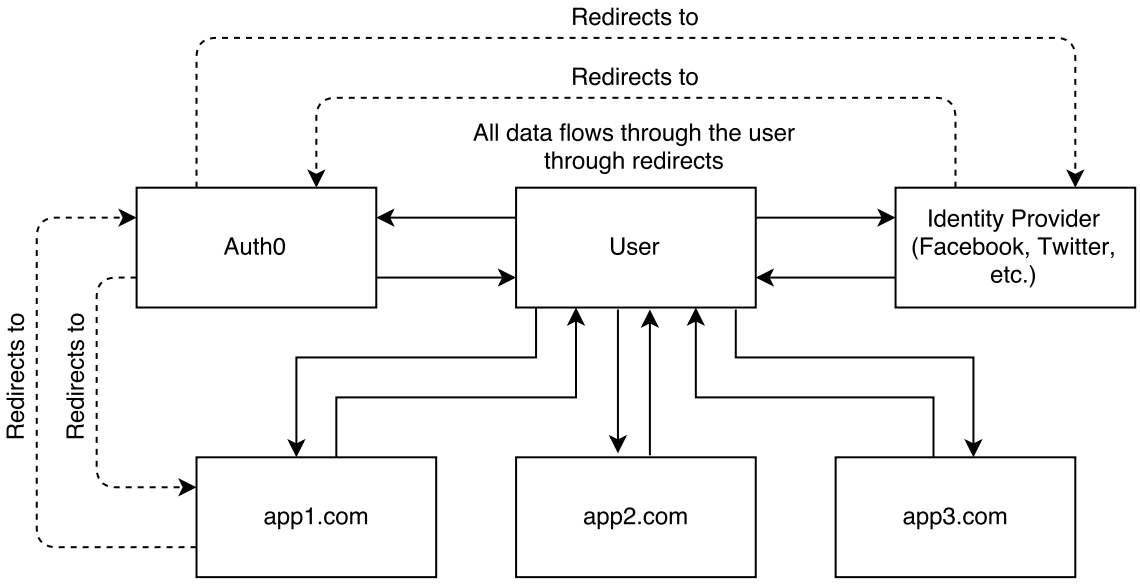

Figura 2.8: Auth0 como servidor de autorizaciones

2.2.4.1 Configurar Auth0 Lock para aplicaciones Node.js

Configuración de la biblioteca Auth0[15]se puede hacer de la siguiente manera. Usaremos el mismo ejemplo usado para el ejemplo de sesiones sin estado:

```
constanteautor0=nuevoventana.autor0.WebAuth({
    dominio:dominio, Identificación del
    cliente:Identificación del cliente,
    audiencia:'app1.com/protegido',
    alcance:'compra de perfil openid', tipo
    de respuesta:'ficha_id_token',
    redirigirUri:'http://aplicación1.com:3000/autorización/', modo
    de respuesta:'formulario_post'
});

// (...)
```

[14]https://github.com/au th0/auth0.js
[15]https://github.com/au th0/auth0.js

```
ps('#botón-iniciar sesión').en('hacer clic',función(evento){
    autor0.autorizar({
        inmediato:'ninguna'
    });
});
```

Tenga en cuenta el uso de laaviso: 'ninguno'parámetro para elautorizarllamar. losautorizarLa llamada redirige al usuario al servidor de autorización. Con elningunaparámetro, si el usuario ya ha dado autorización para que una aplicación use sus credenciales para acceder a un recurso protegido, el servidor de autorización simplemente redirigirá a la aplicación. Esto le parece al usuario como si ya hubiera iniciado sesión en la aplicación.

En nuestro ejemplo, hay dos aplicaciones:app1.comyapp2.com.Una vez que un usuario ha autorizado ambas aplicaciones (lo que sucede solo una vez: la primera vez que el usuario inicia sesión), cualquier inicio de sesión posterior a cualquiera de ambas aplicaciones también permitirá que la otra aplicación inicie sesión sin presentar ninguna pantalla de inicio de sesión.

Para probar esto, vea elLÉAMEarchivo para el ejemplo ubicado en el samples/single-sign-on-federated-identitydirectorio para configurar ambas aplicaciones y ejecutarlas. Una vez que ambos se estén ejecutando, vaya a app1.com:3000dieciséisy app2.com:3001[17]e inicie sesión. Luego cierre la sesión de ambas aplicaciones. Ahora intente iniciar sesión en uno de ellos. Luego regrese al otro e inicie sesión. Notará que la pantalla de inicio de sesión estará ausente en ambas aplicaciones. El servidor de autorización recuerda los inicios de sesión anteriores y puede emitir nuevos tokens de acceso cuando lo solicite cualquiera de esas aplicaciones. Por lo tanto, siempre que el usuario tenga una sesión en el servidor de autorización, ya habrá iniciado sesión en ambas aplicaciones.

La implementación de técnicas de mitigación de CSRF se deja como un ejercicio para el lector.

---

dieciséishttp://app1.com:3000

[17]http://app2.com:3001

Capítulo 3

Tokens web JSON en detalle

Como se describe enCapítulo 1, todos los JWT se construyen a partir de tres elementos diferentes: el encabezado, la carga útil y los datos de firma/cifrado. Los primeros dos elementos son objetos JSON de cierta estructura. El tercero depende del algoritmo utilizado para firmar o cifrar y, en el caso de*sin cifrar*JWT se omite. Los JWT se pueden codificar en un*representación compacta*conocido como *Serialización compacta JWS/JWE*.

> Las especificaciones JWS y JWE definen un tercer formato de serialización conocido como *Serialización JSON*, una representación no compacta que permite varias firmas o destinatarios en el mismo JWT. Se explica en detalle en los capítulos 4 y 5.

La serialización compacta es un Base64[1]Codificación segura para URL de UTF-8[2]bytes de los dos primeros elementos JSON (el encabezado y la carga útil) y los datos, según sea necesario, para la firma o el cifrado (que no es un objeto JSON en sí). Estos datos también están codificados en Base64-URL. Estos tres elementos están separados por puntos (“.”).

> JWT usa una variante de la codificación Base64 que es segura para las URL. Esta codificación básicamente sustituye los caracteres "+" y "/" por los caracteres "-" y "_", respectivamente. El relleno también se elimina. Esta variante se conoce como base64url[3]. Tenga en cuenta que todas las referencias a la codificación Base64 en este documento se refieren a esta variante.

La secuencia resultante es una cadena imprimible como la siguiente (líneas nuevas insertadas para facilitar la lectura):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

Observe los puntos que separan los tres elementos del JWT (en orden: el encabezado, la carga útil y la firma).

En este ejemplo, el encabezado decodificado es:

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

```
{
  "algo":"HS256",
  "tipo":"JWT"
}
```


La carga útil decodificada es:


```
{
  "sub":"1234567890",
  "nombre":"Juan Doe",
  "administración":verdadero
}
```


Y el secreto requerido para verificar la firma essecreto.

> JWT.io[4]es un patio de recreo interactivo para aprender más sobre los JWT. Copie el token de arriba y vea qué sucede cuando lo edita.

3.1 El encabezado

Cada JWT lleva un encabezado (también conocido como el*Cabecera JOSÉ*) con afirmaciones sobre sí mismo. Estas afirmaciones establecen los algoritmos utilizados, si el JWT está firmado o encriptado y, en general, cómo analizar el resto del JWT.

Según el tipo de JWT de que se trate, pueden ser obligatorios más campos en la cabecera. Por ejemplo, los JWT cifrados contienen información sobre los algoritmos criptográficos utilizados para el cifrado de claves y el cifrado de contenido. Estos campos no están presentes para los JWT sin cifrar.

El único reclamo obligatorio para un*sin cifrar*El encabezado JWT es elalgoritmoreclamar:

- **algoritmo**: el algoritmo principal en uso para firmar y/o descifrar este JWT.

Para los JWT sin cifrar, esta reclamación debe establecerse en el valorninguna.

Las reclamaciones de encabezado opcionales incluyen eltipoyciudadreclamación (es:

- **tipo**: el tipo de medio[5]del propio JWT. Este parámetro solo debe usarse como ayuda para usos en los que los JWT se pueden mezclar con otros objetos que llevan un encabezado JOSE. En la práctica, esto rara vez sucede. Cuando esté presente, esta reclamación debe establecerse en el valorJWT.
- **ciudad**: el tipo de contenido. La mayoría de los JWT llevan reclamos específicos más datos arbitrarios como parte de su carga útil. Para este caso, la notificación de tipo de contenido*no debe*ser establecido Para los casos en los que la carga útil es un JWT en sí mismo (un JWT anidado), este reclamo*deber*estar presente y llevar el valorJWT. Esto le dice a la implementación que se requiere un procesamiento adicional del JWT anidado. Los JWT anidados son raros, por lo que el ciudadel reclamo rara vez está presente en los encabezados.

Entonces, para los JWT sin cifrar, el encabezado es simplemente:

---

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

```
{
      "algo":"ninguna"
}
```

que se codifica para:

eyjhbGciOiJub25lIn0

> Es posible agregar reclamos adicionales definidos por el usuario al encabezado. Esto generalmente tiene un uso limitado, a menos que se requieran ciertos metadatos específicos del usuario en el caso de JWT cifrados antes del descifrado.

3.2 La carga útil

```
{
   "sub":"1234567890",
   "nombre":"Juan Doe",
   "administración":verdadero
}
```

El payload es el elemento donde se suelen añadir todos los datos interesantes del usuario. Además, ciertas afirmaciones definidas en la especificación también pueden estar presentes. Al igual que el encabezado, la carga útil es un objeto JSON. Ningún reclamo es obligatorio, aunque los reclamos específicos tienen un significado definido. La especificación JWT especifica que se deben ignorar los reclamos que no son entendidos por una implementación. Las reivindicaciones con significados específicos adjuntos se conocen como *reclamos registrados*.

3.2.1 Reclamos Registrados

- **es**: de la palabra *editor*. Una cadena que distingue entre mayúsculas y minúsculas o URI que identifica de forma única a la parte que emitió el JWT. Su interpretación es específica de la aplicación (no existe una autoridad central que gestione los emisores).
- **sub**: de la palabra *tema*. Una cadena que distingue entre mayúsculas y minúsculas o URI que identifica de forma única a la parte sobre la que este JWT transporta información. En otras palabras, las afirmaciones contenidas en este JWT son declaraciones sobre esta parte. La especificación JWT especifica que esta afirmación debe ser única en el contexto del emisor o, en los casos en que eso no sea posible, única a nivel mundial. El manejo de este reclamo es específico de la aplicación.
- **aud**: de la palabra *audiencia*. Una sola cadena o URI que distingue entre mayúsculas y minúsculas o una matriz de dichos valores que identifican de forma única a los destinatarios previstos de este JWT. En otras palabras, cuando este reclamo está presente, la parte que lee los datos en este JWT debe encontrarse en el *aud* reclamar o ignorar los datos contenidos en el JWT. Como en el caso de la *es* y *sub* reclamos, este reclamo es específico de la aplicación.

- **Exp**: de la palabra *vencimiento* (tiempo). Un número que representa una fecha y hora específicas en el formato "segundos desde época" según lo define POSIX[6]. Esta afirmación establece el momento exacto desde

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

que este JWT se considera *inválido*. Algunas implementaciones pueden permitir cierto sesgo entre los relojes (al considerar que este JWT es válido durante unos minutos después de la fecha de vencimiento).

- **nbf**: de *no antes* (tiempo). Lo contrario de la *Exp* reclamar. Un número que representa una fecha y hora específicas en el formato "segundos desde época" según lo define POSIX[7]. Este reclamo establece el momento exacto a partir del cual se considera este JWT *válido*. La hora y la fecha actuales deben ser iguales o posteriores a esta fecha y hora. Algunas implementaciones pueden permitir un cierto sesgo.

- **Yo en**: de *Emitido en* (tiempo). Un número que representa una fecha y hora específicas (en el mismo formato que *Exp* y *nbf*) en el que se emitió este JWT.
- **jti**: de *ID de JWT*. Una cadena que representa un identificador único para este JWT. Este reclamo se puede usar para diferenciar los JWT con otro contenido similar (evitando las repeticiones, por ejemplo). Depende de la implementación garantizar la unicidad.

Como habrás notado, todos los nombres son cortos. Esto cumple con uno de los requisitos de diseño: mantener los JWT lo más pequeños posible.

> Cadena o URI: de acuerdo con la especificación JWT, una URI se interpreta como cualquier cadena que contenga un carácter :. Depende de la implementación proporcionar valores válidos.

3.2.2 Reclamos Públicos y Privados

Todas las reclamaciones que no forman parte del *reclamos registrados* sección son **privado** o **público** reclamación (es.

- **Privado** reclamaciones: son aquellas que están definidas por *usuarios* (consumidores y productores) de los JWT. En otras palabras, se trata de reclamaciones ad hoc utilizadas para un caso particular. Como tal, se debe tener cuidado para evitar colisiones.
- **Público** reclamaciones: son reclamaciones que son *registrado* con el registro de reclamos de token web JSON de IANA[8] (un registro donde los usuarios pueden registrar sus reclamos y así evitar colisiones), o nombrado usando un nombre resistente a colisiones (por ejemplo, anteponiendo un espacio de nombres a su nombre).

En la práctica, la mayoría de las reclamaciones son reclamaciones registradas o reclamaciones privadas. En general, la mayoría de los JWT se emiten con un propósito específico y un conjunto claro de usuarios potenciales en mente. Esto simplifica la elección de nombres resistentes a colisiones.

Al igual que en las reglas de análisis de JSON, las reclamaciones duplicadas (claves JSON duplicadas) se manejan manteniendo solo la última aparición como válida. La especificación JWT también hace posible que las implementaciones consideren los JWT con reclamos duplicados como *inválido*. En la práctica, si no está seguro acerca de la implementación que manejará sus JWT, tenga cuidado de evitar reclamos duplicados.

---

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

3.3 JWT no protegidos

Con lo que hemos aprendido hasta ahora, es posible construir JWT no seguros. Estos son los JWT más simples, formados por un encabezado simple (generalmente estático):

```
{
      "algo":"ninguna"
}
```

y una carga útil definida por el usuario. Por ejemplo:

```
{
   "sub":"usuario123",
   "sesión":"ch72gsb320000udocl363eofy", "nombre":
   "Bonito nombre", "última página":"/vistas/
   configuraciones"
}
```

Como no hay firma ni cifrado, este JWT se codifica simplemente como dos elementos (nuevas líneas insertadas para facilitar la lectura):

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

Un JWT no seguro como el que se muestra arriba puede ser adecuado para el uso del lado del cliente. Por ejemplo, si el ID de la sesión es un número difícil de adivinar y el cliente solo utiliza el resto de los datos para construir una vista, el uso de una firma es superfluo. Estos datos pueden ser utilizados por una aplicación web de una sola página para construir una vista con el nombre "bonito" para el usuario sin tocar el backend mientras se le redirige a su última página visitada. Incluso si un usuario malintencionado modificara estos datos, no ganaría nada.

> Tenga en cuenta el punto final (.) en la representación compacta. Como no hay firma, es simplemente una cadena vacía. Sin embargo, el punto todavía se agrega.

En la práctica, sin embargo, los JWT no seguros son raros.

3.4 Creación de un JWT no seguro

Para llegar a la representación compacta de las versiones JSON del encabezado y el payload, realice los siguientes pasos:

1. Tome el encabezado como una matriz de bytes de su representación UTF-8. La especificación JWT*no es* requiere que el JSON se minimice o elimine los caracteres sin sentido (como los espacios en blanco) antes de codificar.
2. Codifique la matriz de bytes utilizando el algoritmo Base64-URL, eliminando los signos de igual (=) finales.
3. Tome la carga útil como una matriz de bytes de su representación UTF-8. La especificación JWT*no es*requiere que el JSON se minimice o elimine los caracteres sin sentido (como los espacios en blanco) antes de codificar.

4. Codifique la matriz de bytes utilizando el algoritmo Base64-URL, eliminando los signos de igual (=) finales.
5. Concatenar las cadenas resultantes, poniendo primero el encabezado, seguido de un “.” carácter, seguido de la carga útil.

La validación tanto del encabezado como de la carga útil (con respecto a la presencia de declaraciones requeridas y el uso correcto de cada declaración) debe realizarse antes de la codificación.

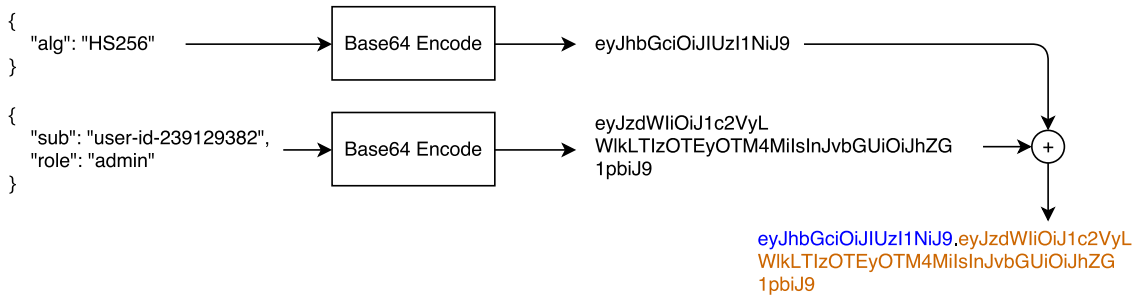

Figura 3.1: Generación de JWT compacto no seguro

3.4.1 Código de muestra

```
// variante segura de URL de Base64
función b64(calle){
    volver nuevo Buffer(calle).('base64')
                                . reemplazar(/=/g,'')
                                . reemplazar(/\+/gramo,'-')
                                . reemplazar(/\/gramo,'_');
}

función codificar(h,pags){
    constante headerEnc=b64(JSON.encadenar(h));
    constante payloadEnc=b64(JSON.encadenar(pags));
    devolver `${headerEncpspayloadEnc};
}
```

El ejemplo completo está en el archivo.codificación.jsdel código de ejemplo adjunto.

3.5 Análisis de un JWT no seguro

Para llegar a la representación JSON desde el formulario de serialización compacto, realice los siguientes pasos:

1. Encuentra el primer período d "." personaje. Tomar la cadena anterior (sin incluirla).

2. Decodifique la cadena utilizando el algoritmo Base64-URL. El resultado es el encabezado JWT.
3. Tome la cuerda después del punto del paso 1.
4. Decodifique la cadena utilizando el algoritmo Base64-URL. El resultado es la carga útil de JWT.

Las cadenas JSON resultantes se pueden "embellecer" agregando espacios en blanco según sea necesario.

3.5.1 Código de muestra

```
función descodificar(jwt){
    constante [encabezadoB64, carga útilB64] = jwt.separar('.');
    // Esto también admite el análisis de la variante segura de URL de Base64.
    constante headerStr = nuevo Buffer(encabezadoB64, 'base64').(); constante
    payloadStr = nuevo Buffer(carga útilB64, 'base64').(); devolver {

        encabezamiento: JSON.analizar gramaticalmente(CadenaCabecera),

        carga útil: JSON.analizar gramaticalmente(carga útilStr)
    };
}
```

El ejemplo completo está en el archivo.codificación.jsdel código de ejemplo adjunto.

Capítulo 4

Firmas web JSON

Las firmas web JSON son probablemente la característica más útil de los JWT. Al combinar un formato de datos simple con una serie bien definida de algoritmos de firma, los JWT se están convirtiendo rápidamente en el formato ideal para compartir datos de forma segura entre clientes e intermediarios.

El propósito de una firma es permitir que una o más partes establezcan la *autenticidad* del JWT. La autenticidad en este contexto significa que los datos contenidos en el JWT no han sido alterados. En otras palabras, cualquier parte que pueda realizar una *cheque de firma* puede confiar en los contenidos proporcionados por el JWT. Es importante recalcar que una firma no impide que otras partes *lectura* el contenido dentro del JWT. Esto es lo que se supone que debe hacer el cifrado, y hablaremos de eso más adelante en Capítulo 5.

El proceso de verificar la firma de un JWT se conoce como *validación* o *validando* una ficha. Un token se considera válido cuando se cumplen todas las restricciones especificadas en su encabezado y carga útil. Esto es un *muy importante* Aspecto de los JWT: se requieren implementaciones para verificar un JWT hasta el punto especificado tanto por su encabezado como por su carga útil (y, además, lo que el usuario requiera). Entonces, un JWT puede considerarse válido *aunque no tenga firma* (si el encabezado tiene el *algoritmo* reclamo establecido enninguna).Además, incluso si un JWT tiene una firma válida, puede considerarse inválido por otras razones (por ejemplo, puede haber caducado, según el *Exp* reclamar). Un ataque común contra los JWT firmados se basa en eliminar la firma y luego cambiar el encabezado para convertirlo en un JWT no seguro. Es responsabilidad del usuario asegurarse de que los JWT se validen de acuerdo con sus propios requisitos.

Los JWT firmados se definen en la especificación JSON Web Signature, RFC 7515[1].

4.1 Estructura de un JWT firmado

Hemos cubierto la estructura de un JWT en Capítulo 3. Lo revisaremos aquí y tomaremos nota especial de su componente característico.

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

Un JWT firmado se compone de tres elementos: el encabezado, la carga útil y la firma (líneas nuevas insertadas para facilitar la lectura):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

El proceso para decodificar los dos primeros elementos (el encabezado y la carga útil) es idéntico al caso de los JWT no seguros. El algoritmo y el código de muestra se pueden encontrar al final deCapítulo 3.


```
{
  "algo":"HS256",
  "tipo":"JWT"
}

{
  "sub":"1234567890",
  "nombre":"Juan Doe",
  "administración":verdadero
}
```


Los JWT firmados, sin embargo, llevan un elemento adicional: la firma. Este elemento aparece después del último punto (.) en el formulario de serialización compacto.

Hay varios tipos de algoritmos de firma disponibles según la especificación JWS, por lo que la forma en que se interpretan estos octetos varía. La especificación JWS requiere que todas las implementaciones compatibles admitan un único algoritmo:

- HMAC usando SHA-256, llamadoHS256en la especificación JWA.

La especificación también define una serie de*recomendado*algoritmos:

- RSASSA PKCS1 v1.5 usando SHA-256, llamadoRS256en la especificación JWA.
- ECDSA usando P-256 y SHA-256, llamadoES256en la especificación JWA.

JWA es la especificación de algoritmos web JSON, RFC 7518[2].

Estos algoritmos se explicarán en detalle enCapítulo 7. En este capítulo, nos centraremos en la aspectos prácticos de su uso.

Los otros algoritmos admitidos por la especificación, en capacidad opcional, son:

- HS384, HS512: variaciones SHA-384 y SHA-512 del algoritmo HS256.
- RS384, RS512: SHA-384 y SHA-512 variaciones del algoritmo RS256.
- ES384, ES512: SHA-384 y SHA-512 variaciones del algoritmo ES256.
- PS256, PS384, PS512: RSASSA-PSS + MGF1 con variantes SHA256/384/512.

Estas son, esencialmente, variaciones de los tres algoritmos principales requeridos y recomendados. los significado de estos acrónimos se aclarará enCapítulo 7.

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

4.1.1 Descripción general del algoritmo para la serialización compacta

Para discutir estos algoritmos en general, primero definamos algunas funciones en un entorno de JavaScript 2015:

- **base64:**una función que recibe una matriz de octetos y devuelve una nueva matriz de octetos utilizando el algoritmo Base64-URL.
- **utf8:**una función que recibe texto en cualquier codificación y devuelve una matriz de octetos con codificación UTF-8.
- **JSON.stringify:**una función que toma un objeto de JavaScript y lo serializa en forma de cadena (JSON).
- **sha256:**una función que toma una matriz de octetos y devuelve una nueva matriz de octetos utilizando el algoritmo SHA-256.
- **mac:**una función que toma una función SHA, una matriz de octetos y un secreto y devuelve una nueva matriz de octetos utilizando el algoritmo HMAC.
- **rsassa:**una función que toma una función SHA, una matriz de octetos y la clave privada y devuelve una nueva matriz de octetos utilizando el algoritmo RSASSA.

Para algoritmos de firma basados en HMAC:

```
constanteEncabezado codificado=base64(utf8(JSON.encadenar(encabezamiento)));
constantecodificadoPayload=base64(utf8(JSON.encadenar(carga útil))); constantefirma
=base64(hmac(${Encabezado codificadopscodificadoPayload}`,
                                        secreto,sha256));
constantejwt=${Encabezado codificadopscodificadoPayloadpsfirma};
```

Para algoritmos de firma de clave pública:

```
constanteEncabezado codificado=base64(utf8(JSON.encadenar(encabezamiento)));
constantecodificadoPayload=base64(utf8(JSON.encadenar(carga útil))); constantefirma=
base64(rsassa(psEncabezado codificadopscodificadoPayload},
                                        llave privada,sha256));
constantejwt=${Encabezado codificadopscodificadoPayloadpsfirma};
```

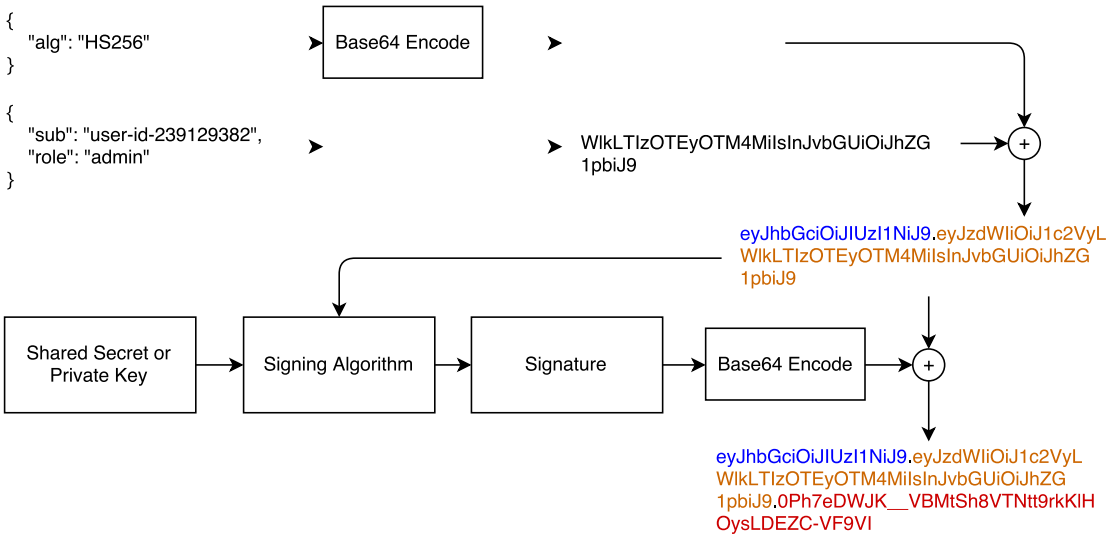

Figura 4.1: Serialización compacta de JWS

Los detalles completos de estos algoritmos se muestran enCapítulo 7.

4.1.2 Aspectos prácticos de los algoritmos de firma

Todos los algoritmos de firma logran lo mismo: proporcionan una forma de establecer la autenticidad de los datos contenidos en el JWT. Cómo lo hacen varía.

El código de autenticación de mensajes hash con clave (HMAC) es un algoritmo que combina una cierta carga útil con un*secreto* usando un desastre criptográfico[3]*función de hash criptográfica* para **permitirque los mensajes sean** enerando y verificando verificarfiestas k ahora el secreto. en otrosr palabras,**HMAC verificado a través de sha secretos rojos**.

El hash criptográfico f unción utilizada en HS256 , el algoritmo de firma más común para JWT, es SHA-256. SHA-256 se explica en detalle enCapítulo 7. Las funciones hash criptográficas toman un mensaje de longitud arbitraria y p producir una salida de fi longitud fija. El mismo mensaje qIllinois siempre produzco h la misma salida.los *cr* no es *yptográfico*pags arte dea funciones de ceniza on se asegura de que sea matemáticamente factible recuperar las mensaje original de th *y* e salida de la función. De esta manera, criptográfico e funciones hash o son*one-wa* *funciones*que puede b solía identificar mensajes sin compartir realmente el mensaje. Una ligera variación en el mensaje (un solo byte, por ejemplo) producirá una salida completamente diferente.

RSASSA es una variación del algoritmo RSA[4](explicado enCapítulo 7) adaptado para firmas. RSA es un algoritmo de clave pública. Los algoritmos de clave pública generan claves divididas: una clave pública y una privada

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

llave. En esta variación específica del algoritmo, la clave privada se puede usar tanto para crear un mensaje firmado como para verificar su autenticidad. La clave pública, en cambio, solo se puede utilizar para verificar la autenticidad de un mensaje. Así, este esquema permite la distribución segura de un**uno a muchos** mensaje. Las partes receptoras pueden verificar la autenticidad de un mensaje guardando una copia de la clave pública asociada con él, pero no pueden crear nuevos mensajes con ella. Esto permite diferentes escenarios de uso que los esquemas de firma de secreto compartido como HMAC. Con HMAC + SHA-256, cualquier parte que pueda verificar un mensaje también puede*crear nuevos mensajes*. Por ejemplo, si un usuario legítimo se vuelve malicioso, él o ella podría modificar los mensajes sin que las otras partes se den cuenta. Con un esquema de clave pública, un usuario que se volviera malicioso solo tendría la clave pública en su poder y, por lo tanto, no podría crear nuevos mensajes firmados con ella.

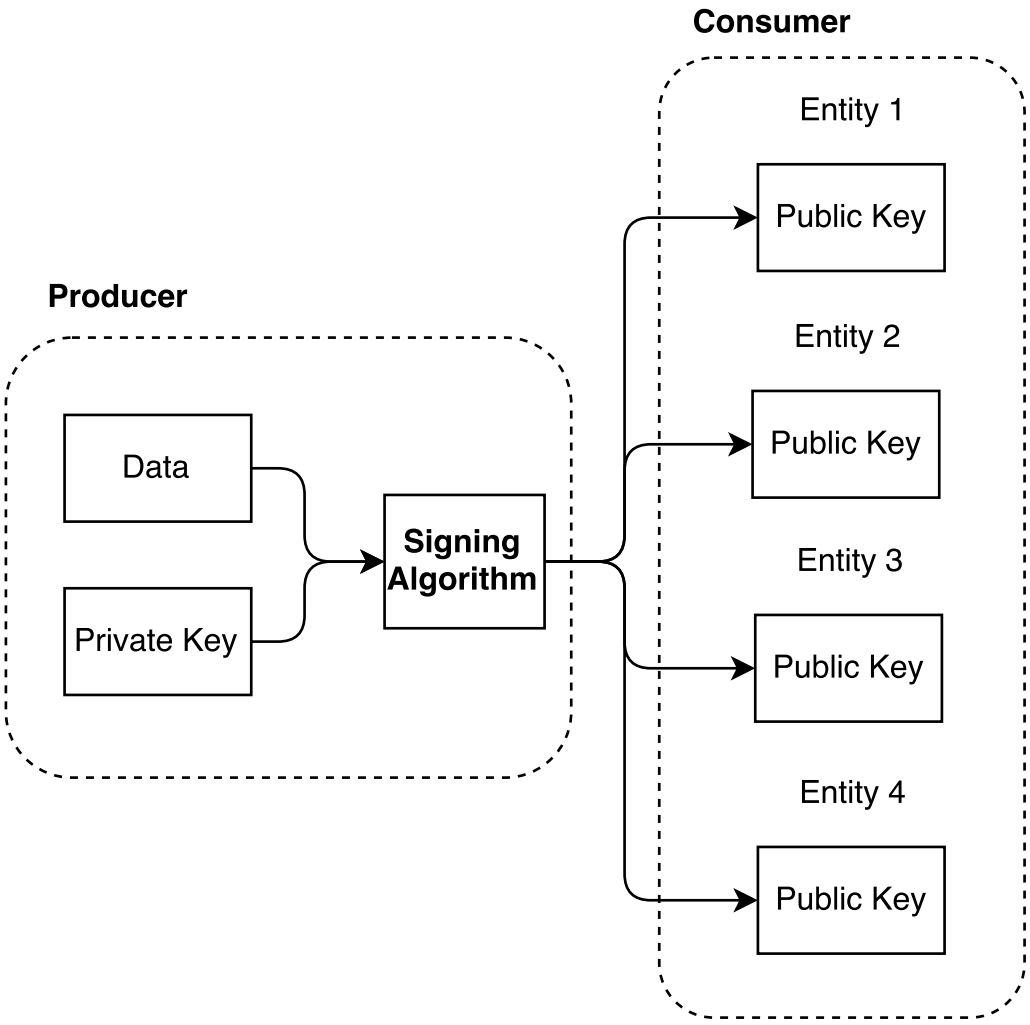

Figura 4.2: Firma de uno a muchos

Criptografía de clave pública[5] permite otros escenarios de uso. Por ejemplo, usando una variación del mismo algoritmo RSA, es posible cifrar mensajes usando la clave pública. Estos mensajes solo se pueden descifrar con la clave privada. Esto permite un **muchos a uno** canal de comunicación seguro que se construirá. Esta variación se usa para JWT encriptados, que se analizan en

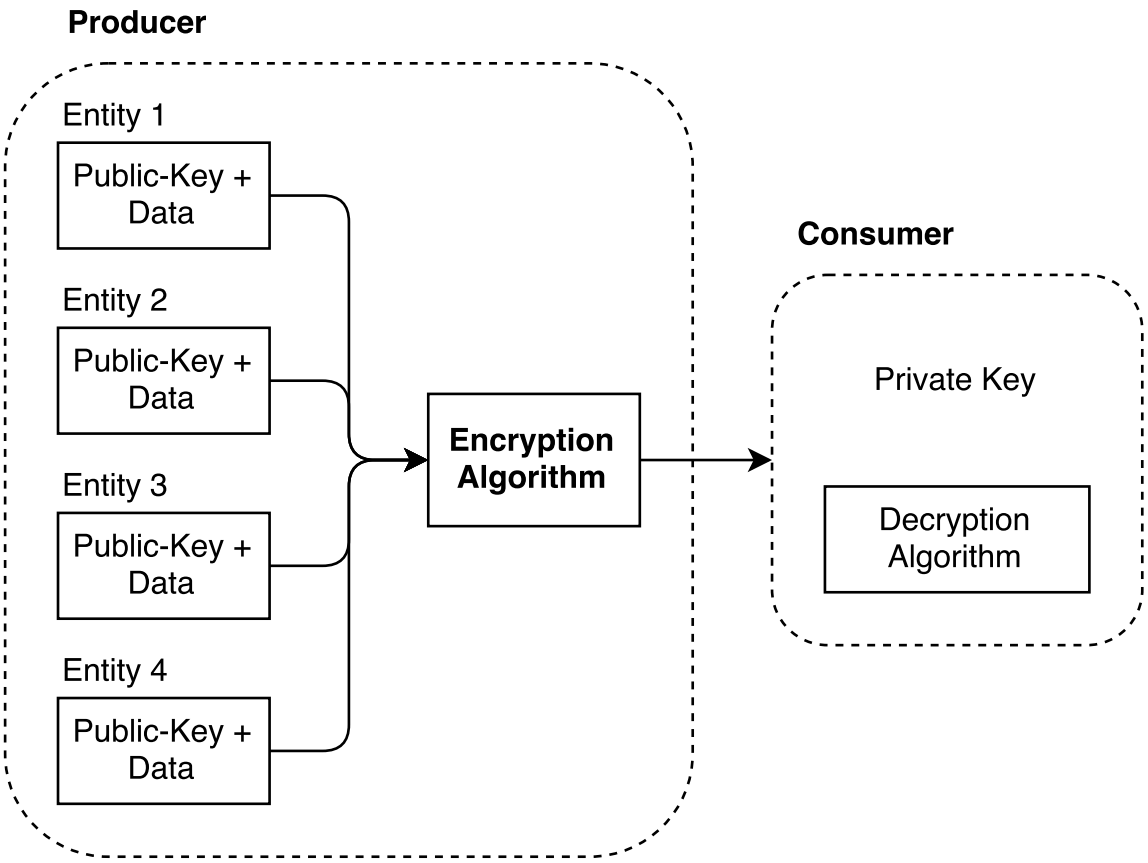

Figura 4.3: Cifrado de muchos a uno

El Algoritmo de Firma Digital de Curva Líptica (ECDSA)[6] es una alternativa a RSA. Este algoritmo también genera un par de claves pública y privada, pero las matemáticas detrás de él son diferentes. Esta diferencia permite requisitos de hardware menores que RSA para s similares garantías de seguridad.

Estudiaremos estos algoritmos con más detalle en C capítulo 7.

[5] https://es.wik ipedia.org/wiki/Publi tecla c_criptógrafo y
[6] https://es.wik ipedia.org/wiki/Ellipt ic_C urve_Digital_ Signature_Algorithm

4.1.3 Reclamos de encabezado JWS

JWS permite casos de uso especiales que obligan al encabezado a llevar más notificaciones. Por ejemplo, para los algoritmos de firma de clave pública, es posible incrustar la URL en la clave pública como un reclamo. Lo que sigue es la lista de reclamos de encabezado registrados disponibles para tokens JWS. Todas estas afirmaciones son *además* a los disponibles para JWT no seguros, y son opcionales según cómo se vaya a utilizar el JWT firmado.

- **jku**: Clave web JSON (JWK) Establecer URL. Un URI que apunta a un conjunto de claves públicas codificadas en JSON utilizadas para firmar este JWT. Se debe utilizar la seguridad de transporte (como TLS para HTTP) para recuperar las claves. El formato de las claves es un JWK Set (ver Capítulo 6).
- **jwk**: clave web JSON. La clave utilizada para firmar este JWT en formato JSON Web Key (ver Capítulo 6).
- **niño**: ID de clave. Una cadena definida por el usuario que representa una sola clave utilizada para firmar este JWT. Este reclamo se usa para señalar cambios de firma de clave a los destinatarios (cuando se usan varias claves).
- **x5u**: URL X.509. Un URI que apunta a un conjunto de certificados públicos X.509 (un estándar de formato de certificado) codificados en formato PEM. El primer certificado del conjunto debe ser el utilizado para firmar este JWT. Cada uno de los certificados subsiguientes firma el anterior, completando así la cadena de certificados. X.509 se define en RFC 5280[7]. Se requiere seguridad de transporte para transferir los certificados.
- **x5c**: Cadena de certificados X.509. Una matriz JSON de certificados X.509 utilizada para firmar este JWS. Cada certificado debe ser el valor codificado en Base64 de su representación DER PKIX. El primer certificado de la matriz debe ser el que se utiliza para firmar este JWT, seguido del resto de certificados de la cadena de certificados.
- **x5t**: Huella digital SHA-1 del certificado X.509. La huella digital SHA-1 del certificado codificado con DER X.509 que se usó para firmar este JWT.
- **x5t#S256**: Idéntico a **x5t**, pero usa SHA-256 en lugar de SHA-1.
- **tipo**: Idéntico al **tipo** valor para JWT sin cifrar, con valores adicionales "JOSE" y "JOSE+JSON" utilizados para indicar serialización compacta y serialización JSON, respectivamente. Esto solo se usa en los casos en que se mezclan objetos portadores de encabezado JOSE similares con este JWT en un solo contenedor.
- **crítico**: de *crítico*. Una matriz de cadenas con los nombres de las notificaciones que están presentes en este mismo encabezado que se usa como extensiones definidas por la implementación que deben manejar los analizadores de este JWT. Debe contener los nombres de las notificaciones o no estar presente (la matriz vacía no es un valor válido).

4.1.4 Serialización JSON JWS

La especificación JWS define un tipo diferente de formato de serialización que no es compacto. Esta representación permite múltiples firmas en el mismo JWT firmado. Es conocido como *Serialización JWS JSON*.

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

En el formulario de serialización JWS JSON, los JWT firmados se representan como texto imprimible con formato JSON (es decir, lo que obtendría al llamarJSON.stringifyen un navegador). Se requiere un objeto JSON superior que lleve los siguientes pares clave-valor:

- **carga útil**: una cadena codificada en Base64 del objeto de carga útil JWT real.
- **firmas**: una matriz de objetos JSON que llevan las firmas. Estos objetos se definen a continuación.

A su vez, cada objeto JSON dentro del**firmas**La matriz debe contener los siguientes pares clave-valor:

- **protegido**: una cadena codificada en Base64 del encabezado JWS. Las reclamaciones contenidas en este encabezado están protegidas por la firma. Este encabezado es obligatorio solo si no hay encabezados sin protección.
  Si hay encabezados desprotegidos, entonces este encabezado puede estar presente o no.
- **encabezamiento**: un objeto JSON que contiene notificaciones de encabezado. Este encabezado no está protegido por la firma. Si no hay ningún encabezado protegido, este elemento es obligatorio. Si hay un encabezado protegido, este elemento es opcional.
- **firma**: una cadena codificada en Base64 de la firma JWS.

A diferencia del formulario de serialización compacto (donde solo está presente un encabezado protegido), la serialización JSON admite dos tipos de encabezados:**protegido**y**desprotegido**. El encabezado protegido es validado por la firma. El encabezado desprotegido no es validado por él. Depende de la implementación o del usuario elegir qué reclamos colocar en cualquiera de ellos. Al menos uno de estos encabezados debe estar presente. Ambos pueden estar presentes al mismo tiempo también.

Cuando hay encabezados protegidos y no protegidos, el encabezado JOSE real se crea a partir de la unión de los elementos de ambos encabezados. No puede haber reclamaciones duplicadas.

El siguiente ejemplo está tomado del JWS RFC[8]:


```
{
  "carga útil":"eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogIm
                h0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
  "firmas":[
    {
      "protegido":"eyJhbGciOiJSUzI1NiJ9",
      "encabezamiento": {"niño":"2010-12-29"},
      "firma":
        "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AA
        uHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAyn
        RFdiuB--f_nZLgrnbyTyWzO5vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB
        _eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqvhJ1phCnvWh6
        IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrBp0igcN_IoypGlU
        PQGe77Rw"
    },
    {
      "protegido":"eyJhbGciOiJFUzI1NiJ9",
      "encabezamiento": {"niño":"e9bc097a-ce51-4036-9562-d2ade882db0d"},
```


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

```
        "firma":"DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDx
                                w5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
      }
    ]
}
```

Este ejemplo codifica dos firmas para la misma carga: firma. una firma RS256 y un ES256

4.1.4.1 Serialización JSON JWS aplanada

La serialización JWS JSON define un formulario simplificado para JWT con una sola firma. Esta forma se conoce como*serialización JWS JSON aplanada*. La serialización aplanada elimina el*firmas*matriz y pone los elementos de una sola firma en el mismo nivel que el*carga útil*elemento.

Por ejemplo, al eliminar una de las firmas del ejemplo anterior, un objeto de serialización JSON aplanado sería:

```
{
  "carga útil":"eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQog
                  Imh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ",
  "protegido":"eyJhbGciOiJFUzI1NiJ9",
  "encabezamiento": {"niño":"e9bc097a-ce51-4036-9562-d2ade882db0d"},
  "firma":"DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFC
                    gfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-Kg6NU1Q"
}
```

4.2 Fichas de firma y validación

Los algoritmos utilizados para firmar y validar tokens se explican en detalle enCapítulo 7. El uso de JWT firmados es lo suficientemente simple en la práctica como para que pueda aplicar los conceptos explicados hasta ahora para usarlos de manera efectiva. Además, hay buenas bibliotecas que puede usar para implementarlas convenientemente. Repasaremos los algoritmos requeridos y recomendados utilizando la más popular de estas bibliotecas para JavaScript. Se pueden encontrar ejemplos de otros lenguajes y bibliotecas populares en el código adjunto.

Los siguientes ejemplos hacen uso del popularjsonwebtokenBiblioteca JavaScript.

importar jwt desde'jsonwebtoken';*//var jwt = require('jsonwebtoken');*

```
constantecarga útil= {
  sub:"1234567890", nombre
  :"Juan Doe",
  administración:verdadero
};
```

4.2.1 HS256: HMAC + SHA-256

Las firmas HMAC requieren un secreto compartido. Cualquier cadena servirá:

```
constantesecreto='mi secreto';

constantefirmado=jwt.señal(carga útil,secreto, {
    algoritmo:'HS256',
    expira en:'5s'//si se omite, el token no caducará
});
```

Verificar el token es igual de fácil:

```
constantedescifrado=jwt.verificar(firmado,secreto, {
    // Nunca olvides hacer esto explícito para prevenir //
    ataques de eliminación de firmas
    algoritmos:['HS256'],
});
```

losjsonwebtokenbiblioteca comprueba la validez del token en función de la firma y la fecha de caducidad. En este caso, si el token se verificara después de 5 segundos de haber sido creado, se consideraría no válido y se generaría una excepción.

4.2.2 RS256: RSASSA + SHA256

Firmar y verificar tokens firmados RS256 es igual de fácil. La única diferencia radica en el uso de un par de claves pública/privada en lugar de un secreto compartido. Hay muchas formas de crear claves RSA. OpenSSL es una de las bibliotecas más populares para la creación y gestión de claves:

```
# Generar una clave privada
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits:2048
# Derivar la clave pública de la clave privada
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Ambos archivos PEM son archivos de texto simples. Sus contenidos pueden copiarse y pegarse en sus archivos fuente de JavaScript y pasarse aljsonwebtokenbiblioteca.

```
// Puede obtener esto de private_key.pem arriba.
constanteprivateRsaKey=`<SU-CLAVE-RSA-PRIVADA>`;

constantefirmado=jwt.señal(carga útil,privateRsaKey, {
    algoritmo:'RS256',
    expira en:'5s'
});

// Puede obtener esto de public_key.pem arriba.
constantepublicRsaKey=`<SU-CLAVE-RSA-PÚBLICA>`;

constantedescifrado=jwt.verificar(firmado,publicRsaKey, {
```

```
    // Nunca olvide hacer esto explícito para prevenir //
    ataques de eliminación de firmas.
    algoritmos:['RS256'],
});
```

4.2.3 ES256: ECDSA utilizando P-256 y SHA-256

Los algoritmos ECDSA también hacen uso de claves públicas. Sin embargo, la matemática detrás del algoritmo es diferente, por lo que los pasos para generar las claves también son diferentes. El "P-256" en el nombre de este algoritmo nos dice exactamente qué versión del algoritmo usar (más detalles sobre esto enCapítulo 7). También podemos usar OpenSSL para generar la clave:

```
# Generar una clave privada (prime256v1 es el nombre de los parámetros utilizados
# para generar la clave, esto es lo mismo que P-256 en la especificación JWA). openssl
ecparam -name prime256v1 -genkey -noout -out ecdsa_private_key.pem
# Derivar la clave pública de la clave privada
openssl ec -in ecdsa_private_key.pem -pubout -out ecdsa_public_key.pem
```

Si abre estos archivos, notará que hay muchos menos datos en ellos. Este es uno de los beneficios de ECDSA sobre RSA (más sobre esto enCapítulo 7). Los archivos generados también están en formato PEM, por lo que simplemente pegarlos en su fuente será suficiente.

```
// Puede obtener esto de private_key.pem arriba. constante
privateEcdsaKey=`<SU-CLAVE-ECDSA-PRIVADA>`;

constantefirmado=jwt.señal(carga útil,privateEcdsaKey, {
    algoritmo:'ES256',
    expira en:'5s'
});

// Puede obtener esto de public_key.pem arriba.
constantepublicEcdsaKey=`<SU-CLAVE-ECDSA-PÚBLICA>`;

constantedescifrado=jwt.verificar(firmado,publicEcdsaKey, {
    // Nunca olvide hacer esto explícito para prevenir //
    ataques de eliminación de firmas.
    algoritmos:['ES256'],
});
```

Referirse aCapitulo 2para aplicaciones prácticas de estos algoritmos en el contexto de JWT.

Capítulo 5

Cifrado web JSON (JWE)

Si bien JSON Web Signature (JWS) proporciona un medio para *validar* datos, JSON Web Encryption (JWE) proporciona una manera de mantener los datos *opaco* a terceros. Opaco en este caso significa *ilegible*. Los tokens encriptados no pueden ser inspeccionados por terceros. Esto permite otros casos de uso interesantes.

Aunque parecería que el cifrado proporciona las mismas garantías que la validación, con la característica adicional de hacer que los datos sean ilegibles, no siempre es así. Para entender por qué, primero es importante tener en cuenta que, al igual que en JWS, JWE proporciona esencialmente dos esquemas: un esquema de secreto compartido y un esquema de clave pública/privada.

El esquema de secreto compartido funciona haciendo que todas las partes conozcan un secreto compartido. Cada parte que posee el secreto compartido puede **encriptar** y **descifrar** información. Esto es análogo al caso de un secreto compartido en JWS: las partes que poseen el secreto pueden verificar y generar tokens firmados.

Sin embargo, el esquema de clave pública/privada funciona de manera diferente. Mientras que en JWS la parte que tiene la clave privada puede firmar y verificar tokens, y las partes que tienen la clave pública solo pueden verificar esos tokens, en JWE la parte que tiene la clave privada es la única que puede **descifrar** la ficha En otras palabras, los titulares de claves públicas pueden **encriptar** datos, pero solo la parte que posee la clave privada puede **descifrar** (y cifrar) esos datos. En la práctica, esto significa que en JWE, las partes que tienen el *público* key puede introducir nuevos datos en un intercambio. Por el contrario, en JWS, las partes que tienen la clave pública solo pueden *verificar* datos pero no introducir nuevos datos. En términos sencillos, JWE no brinda las mismas garantías que JWS y, por lo tanto, no reemplaza el rol de JWS en un intercambio de tokens. JWS y JWE son complementarios cuando se utilizan esquemas de clave pública/privada.

Una forma más sencilla de entender esto es pensar en términos de productores y consumidores. El productor firma o cifra los datos para que los consumidores puedan validarlos o descifrarlos. En el caso de las firmas JWT, la clave privada se usa para firmar JWT, mientras que la clave pública se puede usar para validarla. El productor tiene la clave privada y los consumidores tienen la clave pública. Los datos pueden *solamente* flujo de los poseedores de claves privadas a los poseedores de claves públicas. Por el contrario, para el cifrado JWT, la clave pública se utiliza para cifrar los datos y la clave privada para descifrarlos. En este caso, los datos pueden *solamente* flujo de titulares de claves públicas a titulares de claves privadas: los titulares de claves públicas son los productores y los titulares de claves privadas son los consumidores:

	JWS	JWE
Productor	Llave privada	Llave pública
Consumidor	Llave pública	Llave privada

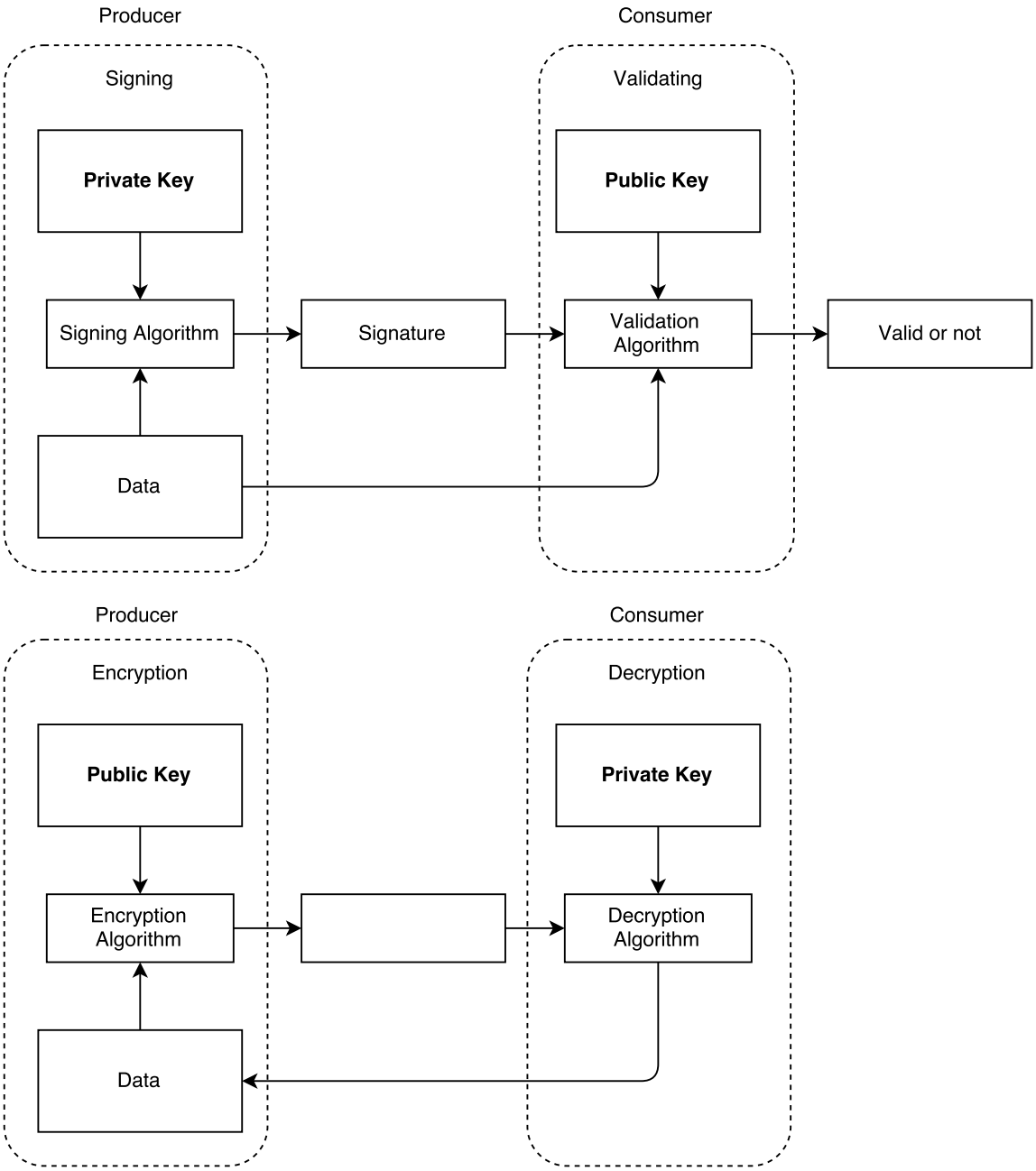

Figura 5.1: Firma frente a cifrado mediante criptografía de clave pública

En este po en algunos gentevo Podemos preguntar:

> En el caso de JWE, ¿no podríamos distribuir la clave privada a todas las partes que quieran enviar datos a un consumidor? Por lo tanto, si un consumidor puede descifrar los datos, puede estar seguro de que también son válidos (porque no se pueden cambiar los datos que no se pueden descifrar).

Técnicamente, sería posible, pero no tendría sentido. Compartir la clave privada es*equivalente* para compartir el secreto. Entonces, compartir la clave privada en esencia convierte el esquema en un esquema secreto compartido, sin los beneficios reales de las claves públicas (recuerde que las claves públicas pueden derivarse de las claves privadas).

Por esta razón, los JWT encriptados a veces son*anidado*: un JWT encriptado sirve como contenedor para un JWT firmado. De esta manera obtienes los beneficios de ambos.

> Tenga en cuenta que todo esto se aplica en situaciones en las que los consumidores son entidades diferentes de los productores. Si el productor es la misma entidad que consume los datos, entonces un JWT encriptado con secreto compartido ofrece las mismas garantías que un encriptado.*y*JWT firmado.
>
> Los JWT encriptados por JWE, independientemente de que tengan o no un JWT firmado anidado, llevan una etiqueta de autenticación. Esta etiqueta permite validar JWE JWT. Sin embargo, debido a los problemas mencionados anteriormente, esta firma no se aplica a los mismos casos de uso que las firmas JWS. El propósito de esta etiqueta es evitar ataques de oráculo de relleno.[1]o manipulación de texto cifrado.

5.1 Estructura de un JWT cifrado

A diferencia de los JWT firmados y no seguros, los JWT cifrados tienen una representación compacta diferente (líneas nuevas insertadas para facilitar la lectura):

eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-kFm1NJn8LE9XShH59_
i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKxGHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-
YxzZIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8Otv
zlV7elprCbuPhcCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTPcFPgwCp6X-
nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGljb3RoZQ.
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.
9hH0vgRfYgPnAHOd8stkvw

Aunque puede ser difícil de ver en el ejemplo anterior, JWE Compact Serialization tiene cinco elementos. Como en el caso de JWS, estos elementos están separados por puntos y los datos contenidos en ellos están codificados en Base64.

Los cinco elementos de la representación compacta son, en orden:

1. **El encabezado protegido:**un encabezado análogo al encabezado JWS.
2. **La clave encriptada:**una clave simétrica utilizada para cifrar el texto cifrado y otros datos cifrados. Esta clave se deriva de la clave de cifrado real especificada por el usuario y, por lo tanto, está cifrada por él.

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

3. **El vector de inicialización:** algunos algoritmos de cifrado requieren datos adicionales (generalmente aleatorios).
4. **Los datos cifrados (texto cifrado):** los datos reales que se cifran.
5. **La etiqueta de autenticación:** datos adicionales producidos por los algoritmos que se pueden utilizar para *validar* el contenido del texto cifrado contra la manipulación.

Como en el caso de JWS y firmas únicas en la serialización compacta, JWE admite una sola clave de cifrado en su forma compacta.

> El uso de una clave simétrica para realizar el proceso de cifrado real es una práctica común cuando se utiliza el cifrado asimétrico (cifrado de clave pública/privada). Los algoritmos de cifrado asimétrico suelen tener una gran complejidad computacional y, por lo tanto, cifrar largas secuencias de datos (el texto cifrado) no es óptimo. Una forma de aprovechar los beneficios del cifrado simétrico (más rápido) y asimétrico es generar una clave aleatoria para un algoritmo de cifrado simétrico y luego cifrar esa clave con el algoritmo asimétrico. Este es el segundo elemento que se muestra arriba, la clave cifrada.
>
> Algunos algoritmos de cifrado pueden procesar cualquier dato que se les pase. Si se modifica el texto cifrado (incluso sin descifrarlo), los algoritmos pueden procesarlo de todos modos. La etiqueta de autenticación se puede usar para evitar esto, actuando esencialmente como una firma. Sin embargo, esto no elimina la necesidad de los JWT anidados explicados anteriormente.

5.1.1 Algoritmos de cifrado de claves

Tener una clave de cifrado cifrada significa que hay dos algoritmos de cifrado en juego en el mismo JWT. Los siguientes son los algoritmos de cifrado disponibles para el cifrado de claves:

- **variantes RSA**: RSAES PKCS #1 v1.5 (RSAES-PKCS1-v1_5), RSAES OAEP y 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**: Acuerdo de clave Elliptic Curve Diffie-Hellman Ephemeral Static con concat KDF y variantes que envuelven previamente la clave con cualquiera de las variantes AES no GCM anteriores.
- **Variantes de PKCS #5**: PBES2 (cifrado basado en contraseña) + HMAC (SHA-256 a 512) + variantes AES sin GCM de 128 a 256 bits.
- **Directo**: sin cifrado para la clave de cifrado (uso directo de CEK).

Ninguno de estos algoritmos es realmente requerido por la especificación JWA. Los siguientes son los algoritmos recomendados (a implementar) por la especificación:

- **RSAES-PKCS1-v1_5** (marcado para la eliminación de la recomendación en el futuro)
- **RSAES-OAEP** con valores predeterminados (marcados para ser obligatorios en el futuro)
- **Envoltura de clave AES-128**
- **Envoltura de clave AES-256**
- **Curva elíptica Diffie-Hellman Estática efímera (ECDH-ES)** utilizando Concat KDF (marcado como obligatorio en el futuro)
- **Envoltura de clave ECDH-ES + AES-128**

- **Envoltura de clave ECDH-ES + AES-256**

Algunos de estos algoritmos requieren parámetros de encabezado adicionales.

5.1.1.1 Modos de administración de claves

La especificación JWE define diferentes modos de gestión de claves. Estas son, en esencia, formas en las que se determina la clave utilizada para cifrar la carga útil. En particular, la especificación JWE describe estos modos de administración de claves:

- **Envoltura de llaves**: la clave de cifrado de contenido (CEK) se cifra para el destinatario mediante un*simétrico*algoritmo de cifrado.

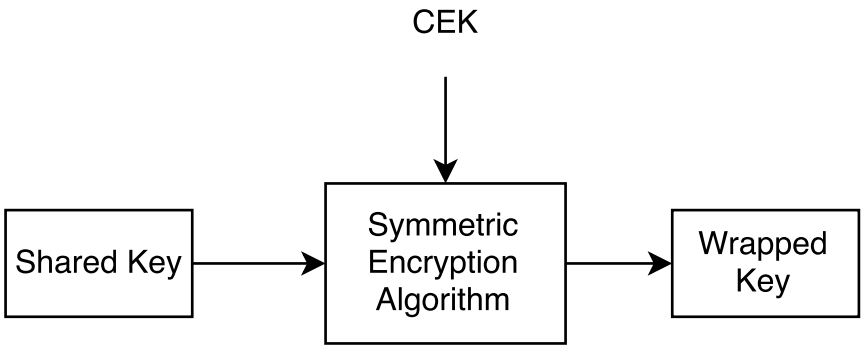

Figura 5.2: Envoltura de llaves

- **Cifrado de claves**: la CEK se cifra para el destinatario mediante un*asimétrico* algoritmo de cifrado.

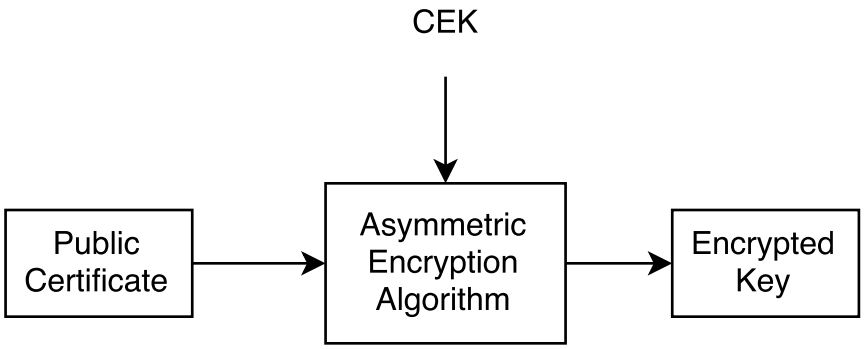

Figura 5.3: Cifrado de claves

- **Dirección<u>ect clave de acuerdo</u> elemento**: se utiliza un algoritmo de concordancia clave para elegir el CEK.

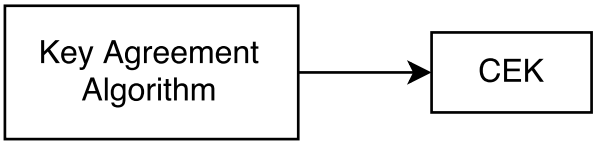

Figura 5.4: Acuerdo de clave directa

- **Acuerdo clave con envoltura clave**: se utiliza un algoritmo de concordancia de claves para seleccionar una CEK simétrica utilizando un algoritmo de cifrado simétrico.

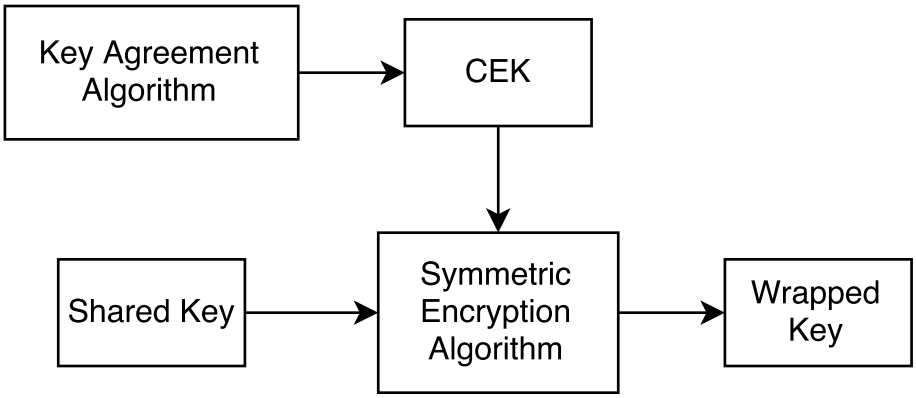

Figura 5.5: Acuerdo de clave directa

- **Cifrado directo**: se utiliza una clave compartida simétrica definida por el usuario como CEK (sin derivación ni generación de claves).

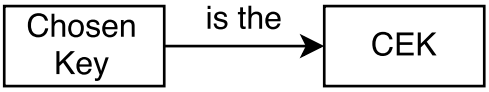

Figura 5.6: Acuerdo de clave directa

Si bien esto constituye una cuestión de terminología, es importante entender las diferencias entre cada modo de gestión y darle a cada uno de ellos un nombre conveniente.

5.1.1.2 Clave de cifrado de contenido (CEK) y clave de cifrado JWE

También es importante comprender la diferencia entre la clave de cifrado CEK y JWE. La CEK es la clave real utilizada para cifrar la carga útil: un algoritmo de cifrado toma la CEK y el texto sin formato para producir el texto cifrado. Por el contrario, la clave de cifrado JWE es la forma cifrada de la CEK o una secuencia de octetos vacíos (según lo requiera el algoritmo elegido). Un

La clave de cifrado JWE vacía significa que el algoritmo utiliza una clave proporcionada externamente para descifrar directamente los datos (cifrado directo) o calcular el CEK real (acuerdo de clave directa).

5.1.2 Algoritmos de cifrado de contenido

Los siguientes son los algoritmos de cifrado de contenido, es decir, los que se utilizan para cifrar realmente la carga útil:

- **AES CBC + HMAC SHA**: AES de 128 a 256 bits con Cipher Block Chaining y HMAC + SHA-256 a 512 para validación.
- **AES GCM**: AES 128 a 256 utilizando el modo de contador de Galois.

De estos, solo se requieren dos: AES-128 CBC + HMAC SHA-256 y AES-256 CBC + HMAC SHA-512. Se recomiendan las variantes AES-128 y AES-256 que utilizan GCM.

Estos algoritmos se explican en detalle enCapítulo 7.

5.1.3 El encabezado

Al igual que el encabezado para JWS y JWT no seguros, el encabezado contiene toda la información necesaria para que las bibliotecas procesen correctamente el JWT. La especificación JWE adapta los significados de los reclamos registrados definidos en JWS a su propio uso y agrega algunos reclamos propios. Estas son las reivindicaciones nuevas y modificadas:

- **algoritmo**: idéntico a JWS, excepto que define el algoritmo que se utilizará para cifrar y descifrar la clave de cifrado de contenido (CEK). En otras palabras, este algoritmo se usa para cifrar la clave real que luego se usa para cifrar el contenido.
- **enc**: el nombre del algoritmo utilizado para cifrar el contenido mediante el CEK.
- **Código Postal**: un algoritmo de compresión que se aplicará a los datos cifrados antes del cifrado. Este parámetro es opcional. Cuando está ausente, no se realiza ninguna compresión. Un valor habitual para esto esDEF,el algoritmo de desinflado común[2].
- **jku**: idéntico a JWS, excepto que en este caso el reclamo apunta a la clave pública utilizada para cifrar la CEK.
- **jkw**: idéntico a JWS, excepto que en este caso el reclamo apunta a la clave pública utilizada para cifrar la CEK.
- **niño**: idéntico a JWS, excepto que en este caso el reclamo apunta a la clave pública utilizada para cifrar la CEK.
- **x5u**: idéntico a JWS, excepto que en este caso el reclamo apunta a la clave pública utilizada para cifrar la CEK.
- **x5c**: idéntico a JWS, excepto que en este caso el reclamo apunta a la clave pública utilizada para cifrar la CEK.

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

- **x5t**: idéntico a JWS, excepto que en este caso el reclamo apunta a la clave pública utilizada para cifrar la CEK.
- **x5t#S256**: idéntico a JWS, excepto que en este caso el reclamo apunta a la clave pública utilizada para cifrar la CEK.
- **tipo**: idéntico a JWS.
- **ciudad**: idéntico a JWS, excepto que este es el tipo de contenido cifrado.
- **crítico**: idéntico a JWS, excepto que se refiere a los parámetros de este encabezado.

Es posible que se requieran parámetros adicionales, según los algoritmos de cifrado en uso. Los encontrará explicados en la sección que analiza cada algoritmo.

5.1.4 Descripción general del algoritmo para la serialización compacta

Al comienzo de este capítulo, se mencionó brevemente la serialización compacta de JWE. Se compone básicamente de cinco elementos codificados en forma de texto imprimible y separados por puntos (.). El algoritmo básico para construir una serialización compacta JWE JWT es:

1. Si lo requiere el algoritmo elegido (algoritmoreclamación), generar una*aleatorio*número del tamaño requerido. Es fundamental cumplir con ciertos requisitos criptográficos de aleatoriedad a la hora de generar este valor. Consulte RFC 4086[3]o utilice un generador de números aleatorios validado criptográficamente.
2. Determine la clave de cifrado de contenido de acuerdo con el modo de administración de claves[4]:
   - Para**Acuerdo de clave directa**: utilice el algoritmo de acuerdo de clave y el número aleatorio para calcular la clave de cifrado de contenido (CEK).
   - Para**Acuerdo clave con envoltura clave**: use el algoritmo de concordancia de clave con el número aleatorio para calcular la clave que se usará para envolver el CEK.
   - Para**Cifrado directo**: la CEK es la clave simétrica.
3. Determine la clave cifrada JWE de acuerdo con el modo de administración de claves:
   - Para**Acuerdo de clave directa y cifrado directo**: la clave cifrada JWE está vacía.
   - Para**Encapsulado de claves, cifrado de claves y acuerdo de claves con encapsulado de claves**: cifrar la CEK para el destinatario. El resultado es la clave cifrada JWE.
4. Calcular un Vector de Inicialización (IV) del tamaño requerido por el algoritmo elegido. Si no es necesario, omita este paso.
5. Comprima el texto sin formato del contenido, si es necesario (Código Postalreclamo de encabezado).
6. Cifre los datos utilizando el CEK, el IV y los datos autenticados adicionales (AAD). El resultado es el contenido cifrado (texto cifrado JWE) y la etiqueta de autenticación. El AAD solo se usa para serializaciones no compactas.
7. Construya la representación compacta como:

```
base64(encabezamiento)+'.'+
base64(Clave cifrada)+'.'+                    // Pasos 2 y 3
```

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

```
base64(Vector de inicialización)+'.'+//Paso 4 base64
(texto cifrado)+'.'+ base64(Etiqueta de autenticación)
                                                    //Paso 5
                                                    // Paso 6
```

5.1.5 Serialización JWE JSON

Además de la serialización compacta, JWE también define una representación JSON no compacta. Esta representación cambia tamaño por flexibilidad, permitiendo, entre otras cosas, el cifrado del contenido para múltiples destinatarios mediante el uso de varias claves públicas al mismo tiempo. Esto es análogo a las múltiples firmas permitidas por JWS JSON Serialization.

JWE JSON Serialization es la codificación de texto imprimible de un objeto JSON con los siguientes miembros:

- **protegido**: el objeto JSON codificado en Base64 del encabezado afirma estar protegido (validado, no cifrado) por este JWE JWT. Opcional. Al menos este elemento o el encabezado desprotegido debe estar presente.
- **desprotegido**: reclamaciones de encabezado que no están protegidas (validadas) como un objeto JSON (no codificado en Base64). Opcional. Al menos este elemento o el encabezado protegido debe estar presente.
- **IV**: Cadena Base64 del vector de inicialización. Opcional (solo presente cuando lo requiere el algoritmo).
- **anuncio**: Datos autenticados adicionales. Cadena Base64 de los datos adicionales que están protegidos (validados) por el algoritmo de cifrado. Si no se proporciona AAD en el paso de cifrado, este miembro debe estar ausente.
- **texto cifrado**: Cadena codificada en Base64 de los datos cifrados.
- **etiqueta**: cadena Base64 de la etiqueta de autenticación generada por el algoritmo de cifrado.
- **destinatarios**: una matriz JSON de objetos JSON, cada uno de los cuales contiene la información necesaria para el descifrado por parte de cada destinatario.

Los siguientes son los miembros de los objetos en eldestinatariosformación:

- **encabezamiento**: un objeto JSON de reclamos de encabezado desprotegidos. Opcional.
- **clave_cifrada**: clave cifrada JWE codificada en Base64. Solo presente cuando se utiliza una clave cifrada JWE.

El encabezado real utilizado para descifrar un JWE JWT para un destinatario se construye a partir de la unión de cada encabezado presente. No se permiten reclamos repetidos.

El formato de las claves cifradas se describe enCapítulo 6(Claves Web JSON).

El siguiente ejemplo está tomado de RFC 7516 (JWE):

```
{
      "protegido":"eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0", "desprotegido": {"jku":
      "https://servidor.ejemplo.com/claves.jwks"}, "destinatarios":[
```

```
        {
          "encabezamiento": {"algo":"RSA1_5","niño":"2011-04-29"
          }, "clave_cifrada":
             "UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-
             kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKx
             GHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRirb6Y5Cl_p-ko3
             YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPh
             cCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-cFPg wCp6X-
             nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A"
          },
          {
            "encabezamiento": {"algo":"A128KW","niño":"7"},
            "clave_cifrada":"6KB707dM9YTIgHtLvtgWQ8mKwboJW3of9locizkDTHzBC2IlrT1oOQ"
          }
        ],
        "iv":"AxY8DCtDaGlsbGljb3RoZQ",
        "texto cifrado":"KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY",
        "etiqueta":"Mz-VPPyU4RlcuYv1IwIvzw"
}
```


Este JSON serializado JWE JWT transporta una sola carga útil para dos destinatarios. El algoritmo de cifrado es AES-128 CBC + SHA-256, que puede obtener del encabezado protegido:


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


Al realizar la unión de todos los reclamos para cada destinatario, se construye el encabezado final para cada destinatario:

Primer destinatario:


```
{
   "algo":"RSA1_5", "niño":
   "2011-04-29", "enc":
   "A128CBC-HS256",
   "jku":"https://servidor.ejemplo.com/claves.jwks"
}
```


Segundo destinatario:


```
{
    "algo":"A128KW",
    "niño":"7",
    "enc":"A128CBC-HS256",
    "jku":"https://servidor.ejemplo.com/claves.jwks"
}
```

5.1.5.1 Serialización JSON JWE aplanada

Al igual que con JWS, JWE define un*plano*Serialización JSON. Este formulario de serialización solo se puede utilizar para un único destinatario. De esta forma, eldestinatariosmatriz es reemplazada por unaencabezamientoyclave_cifrada par o elementos (es decir, las claves de un solo objeto de la matriz de destinatarios toman su lugar).

Esta es la representación aplanada del ejemplo de la sección anterior resultante de incluir solo el primer destinatario:


```
{
      "protegido":"eyJlbmMiOiJBMTI4Q0JDLUhTMjU2In0", "desprotegido": {"jku":
      "https://servidor.ejemplo.com/claves.jwks"}, "encabezamiento": {"algo":
      "RSA1_5","niño":"2011-04-29"}, "clave_cifrada":

         "UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-
         kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKx
         GHZ7PcHALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRirb6Y5Cl_p-ko3
         YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPh
         cCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-cFPg wCp6X-
         nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A", "iv":"AxY8DCtDaGlsbGljb3RoZQ",

      "texto cifrado":"KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY",
      "etiqueta":"Mz-VPPyU4RlcuYv1IwIvzw"
}
```


5.2 Cifrado y descifrado de tokens

Los siguientes ejemplos muestran cómo realizar el cifrado utilizando el popular nodo-jose[5]biblioteca. Esta biblioteca es un poco más compleja quejsonwebtoken (usado para los ejemplos de JWS), ya que cubre mucho más terreno.

5.2.1 Introducción: Gestión de claves con node-jose

A los efectos de los siguientes ejemplos, necesitaremos utilizar claves de cifrado en varias formas. Esto es administrado pornodo-josea través de unalmacén de claves.Aalmacén de claveses un objeto que gestiona claves. Generaremos y agregaremos algunas claves a nuestro almacén de claves para que podamos usarlas más adelante en los ejemplos. Puede recordar de los ejemplos de JWS que no se requería tal abstracción para el jsonwebtoken biblioteca. losalmacén de clavesla abstracción es un detalle de implementación denodo-jose.Puede encontrar otras abstracciones similares en otros idiomas y bibliotecas.

Para crear un almacén de claves vacío y agregar algunas claves de diferentes tipos:

```
// Crear un almacén de claves vacío
constantealmacén de claves=José.JWK.crearAlmacénDeClaves();
```

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

```
// Genera algunas claves. También puede importar claves generadas a partir de
fuentes // externas.
constantepromesas=[
      almacén de claves.generar('oct',128, {niño:'Ejemplo 1'}), almacén
      de claves.generar('RSA',2048, {niño:'ejemplo-2'}), almacén de
      claves.generar('CE','P-256', {niño:'ejemplo-3'}),
];
```

Connodo-jose,la generación de claves es un asunto bastante simple. Todos los tipos de claves utilizables con JWE y JWS son compatibles. En este ejemplo, creamos tres claves diferentes: una clave AES simple de 128 bits, una clave RSA de 2048 bits y una clave de curva elíptica usando la curva P-256. Estas claves se pueden utilizar tanto para el cifrado como para las firmas. En el caso de claves que admiten pares de claves pública/privada, la clave generada es la*privado*llave. Para obtener las claves públicas, simplemente llame a:

```
variableLlave pública=llave.aJSON();
```

La clave pública se almacenará en formato JWK.

También es posible importar claves preexistentes:

```
// donde entrada es un: // *
        jose.JWK.Instancia clave
// *     Representación de objeto JSON de un JWK
José.JWK.asKey(aporte).
            después(función(resultado){
               // {resultado} es un jose.JWK.Key
               // {result.keystore} es un único jose.JWK.KeyStore });


// donde entrada es un: // *
        Serialización de cadenas de un JSON JWK/(codificado en base64) PEM/
//      (codificado en binario) DER
// *     Búfer de un JSON JWK/(codificado en base64) PEM/(codificado en binario) DER
// el formulario es un:
// * "json" para un JWK en cadena JSON
// * "pkcs8" para una clave privada PKCS8 codificada en DER (¡sin cifrar!) // *
"spki" para una clave pública SPKI codificada en DER
// * "pkix" para un certificado PKIX X.509 codificado en DER // *
"x509" para un certificado PKIX X.509 codificado en DER // *
"pem" para un certificado PEM codificado de PKCS8 / SPKI / PKIX
José.JWK.asKey(aporte,forma).
            después(función(resultado){
               // {resultado} es un jose.JWK.Key
               // {result.keystore} es un único jose.JWK.KeyStore });
```

5.2.2 AES-128 Key Wrap (Clave) + AES-128 GCM (Contenido)

AES-128 Key Wrap y AES-128 GCM son algoritmos de clave simétrica. Esto significa que se requiere la misma clave tanto para el cifrado como para el descifrado. La clave para el "ejemplo-1" que generamos antes es una de esas claves. En AES-128 Key Wrap, esta clave se usa para envolver una clave generada aleatoriamente, que luego se usa para encriptar el contenido usando el algoritmo AES-128 GCM. También sería posible utilizar esta clave directamente (modo Cifrado Directo).

```
función encriptar(llave,opciones,Texto sin formato){
    devolver José.JWE.crearCifrar(opciones,llave)
                        . actualizar(Texto sin formato)
                        . final();
}

función a128gcm(compacto){
    constante llave=almacén de claves.obtener('Ejemplo 1');
    constante opciones= {
        formato:compacto?'compacto':'general',
        contentAlg:'A128GCM'
    };

    devolver encriptar(llave,opciones,JSON.encadenar(carga útil));
}
```

losnodo-josebiblioteca trabaja principalmente con promesas[6]. El objeto devuelto pora128gcmes una promesa. loscrearCifrarLa función puede cifrar cualquier contenido que se le pase. En otras palabras, no es necesario que el contenido sea un JWT (aunque la mayoría de las veces lo será). Es por esta razón queJSON.stringifydebe llamarse antes de pasar los datos a esa función.

5.2.3 RSAES-OAEP (Clave) + AES-128 CBC + SHA-256 (Contenido)

Lo único que cambia entre las invocaciones delcrearCifrarson las opciones que se le pasan. Por lo tanto, es igual de fácil usar un par de claves pública/privada. En lugar de pasar la clave simétrica a crear, cifrar,uno simplemente pasa la clave pública o privada (para el cifrado solo se requiere la clave pública, aunque esta puede derivarse de la clave privada). Para fines de legibilidad, simplemente usamos la clave privada, pero en la práctica, lo más probable es que se use la clave pública en este paso.

```
función encriptar(llave,opciones,Texto sin formato){
    devolver José.JWE.crearCifrar(opciones,llave)
                        . actualizar(Texto sin formato)
                        . final();
}

función rsa(compacto){
```

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

```
    constantellave=almacén de claves.obtener('ejemplo-2');
    constanteopciones= {
        formato:compacto?'compacto':'general',
        contentAlg:'A128CBC-HS256'
    };

    devolverencriptar(llave,opciones,JSON.encadenar(carga útil));
}
```

contentAlgselecciona el algoritmo de cifrado real. Recuerda que solo existen dos variantes (con diferentes tamaños de clave): AES CBC + HMAC SHA y AES GCM.

5.2.4 ECDH-ES P-256 (Clave) + AES-128 GCM (Contenido)

La API para curvas elípticas es idéntica a la de RSA:

```
funciónencriptar(llave,opciones,Texto sin formato){
    devolverJosé.JWE.crearCifrar(opciones,llave)
                    . actualizar(Texto sin formato)
                    . final();
}

funciónecdhes(compacto){
    constantellave=almacén de claves.obtener('ejemplo-3');
    constanteopciones= {
        formato:compacto?'compacto':'general',
        contentAlg:'A128GCM'
    };

    devolverencriptar(llave,opciones,JSON.encadenar(carga útil));
}
```

5.2.5 JWT anidado: ECDSA usando P-256 y SHA-256 (Firma) + RSAES-OAEP (Clave cifrada) + AES-128 CBC + SHA-256 (Contenido cifrado)

Los JWT anidados requieren un poco de malabarismo para pasar el JWT firmado a la función de cifrado. En concreto, los pasos de firma + cifrado deben realizarse manualmente. Recuerde que estos pasos se realizan en ese orden: primero firmar, luego cifrar. Aunque técnicamente nada impide que se invierta el orden, firmar el JWT primero evita que el token resultante sea vulnerable a los ataques de eliminación de firmas.

```
funciónanidado(compacto){
    constantefirma de clave=almacén de claves.obtener('ejemplo-3'); constante
    Clave de encriptación=almacén de claves.obtener('ejemplo-2');
```

```
    constantefirmaPromesa=José.JWS.CrearSigno(clave de firma)
                                        . actualizar(JSON.encadenar(carga útil))
                                        . final();

    constantepromesa=nuevoPromesa((resolver,rechazar)=> {

        firmaPromesa.después(resultado=> {
            constanteopciones= {
                formato:compacto?'compacto':'general',
                contentAlg:'A128CBC-HS256'
            };
            resolver(encriptar(Clave de encriptación,opciones,JSON.encadenar(resultado))); },
        error=> {
            rechazar(error);
        });

    });

    devolverpromesa;
}
```

Como se puede ver en el ejemplo anterior,nodo-josetambién puede servir para firmar. No hay nada que impida el uso de otras bibliotecas (comojsonwebtoken)para ese propósito. Sin embargo, dada la necesidad denodo-jose,no tiene sentido agregar dependencias y usar API inconsistentes.

> Realizar el paso de firma primero solo es posible porque JWE exige el cifrado autenticado. En otras palabras, el algoritmo de cifrado también debe realizar el paso de firma. Las razones por las que JWS y JWE se pueden combinar de manera útil, a pesar de la autenticación de JWE, se describieron al comienzo deCapítulo 5. Para otros esquemas (es decir, para cifrado general + firma), la norma es primero cifrar y luego firmar. Esto es para evitar la manipulación del texto cifrado que puede dar lugar a ataques de cifrado. También es la razón por la que JWE exige la presencia de una etiqueta de autenticación.

5.2.6 Descifrado

El descifrado es tan simple como el cifrado. Al igual que con el cifrado, la carga útil debe convertirse explícitamente entre diferentes formatos de datos.

```
// Prueba de descifrado
a128gcm(verdadero).después(resultado=> {
    José.JWE.crearDescifrar(almacén de claves.obtener('Ejemplo 1'))
                . descifrar(resultado)
                . después(descifrado=> {
                    descifrado.carga útil=JSON.analizar gramaticalmente(descifrado.carga útil);
                    consola.Iniciar sesión(`Resultado descifrado: ${JSON.encadenar(descifrado)}`)
                ; },error=> {
                    consola.Iniciar sesión(error);
```

```
                });
},error=> {
    consola.Iniciar sesión(error);
});
```

El descifrado de los algoritmos RSA y Elliptic Curve es análogo, utilizando la clave privada en lugar de la clave simétrica. Si tiene un almacén de claves con el derechoniñoreclamos, es posible simplemente pasar el almacén de claves alcrearDescifrary haga que busque la tecla correcta. Entonces, cualquiera de los ejemplos anteriores se puede descifrar usando exactamente el mismo código:

```
José.JWE.crearDescifrar(almacén de claves)//simplemente pase el almacén de claves aquí
            . descifrar(resultado)
            . después(descifrado=> {
                descifrado.carga útil=JSON.analizar gramaticalmente(descifrado.carga útil);
                consola.Iniciar sesión(`Resultado descifrado: ${JSON.encadenar(descifrado)}`)
            ; },error=> {
                consola.Iniciar sesión(error);
            });
```

Capítulo 6

Claves web JSON (JWK)

Para completar la imagen de JWT, JWS y JWE, ahora llegamos a la especificación JSON Web Key (JWK). Esta especificación se ocupa de las diferentes representaciones de las claves utilizadas para las firmas y el cifrado. Aunque existen representaciones establecidas para todas las claves, la especificación JWK tiene como objetivo proporcionar una representación unificada para todas las claves admitidas en la especificación JSON Web Algorithms (JWA). Un formato de representación unificado para claves permite compartir fácilmente y mantiene las claves independientes de las complejidades de otros formatos de intercambio de claves.

JWS y JWE admiten un tipo diferente de formato de clave: certificados X.509. Estos son bastante comunes y pueden contener más información que un JWK. Los certificados X.509 se pueden incrustar en JWK y los JWK se pueden construir a partir de ellos.

Las claves se especifican en diferentes notificaciones de encabezado. Los JWK literales se colocan bajo eljwkreclamar. losjku afirmación, por otro lado, puede apuntar a una*establecer*de claves almacenadas bajo una URL. Ambas afirmaciones están en formato JWK.

Una muestra de JWK:


```
{
      "kty":"CE",
      "crv":"P-256",
      "X":"MKBCTNICKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4", "y":
      "4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM", "d":
      "870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE", "usar"
      :"enc",
      "niño":"1"
}
```

6.1 Estructura de una clave web JSON

Las claves web JSON son simplemente objetos JSON con una serie de valores que describen los parámetros requeridos por la clave. Estos parámetros varían según el tipo de llave. Los parámetros comunes son:

- **kty**: “tipo de llave”. Esta afirmación diferencia los tipos de claves. Los tipos admitidos sonCE,para claves de curva elíptica;RSApara claves RSA; yoctpara claves simétricas. Este reclamo es obligatorio.
- **usar**: esta afirmación especifica el uso previsto de la clave. Hay dos usos posibles:firma (para la firma) yenc (para el cifrado). Este reclamo es opcional. Se puede utilizar la misma clave para el cifrado y las firmas, en cuyo caso este miembro no debería estar presente.
- **operaciones_de_teclado**: una matriz de valores de cadena que especifica usos detallados para la clave. Los valores posibles son: firmar, verificar, cifrar, descifrar, envolver clave, desenvolver clave, derivar clave, derivar bits.Ciertas operaciones no deben usarse juntas. Por ejemplo,señalyverificarson apropiados para la misma clave, mientras queseñalyencriptarno son. Este reclamo es opcional y no debe usarse al mismo tiempo que elusarreclamar. En los casos en que ambos estén presentes, su contenido debe ser consistente.
- **algoritmo**: “algoritmo”. El algoritmo destinado a ser utilizado con esta clave. Puede ser cualquiera de los algoritmos admitidos para operaciones JWE o JWS. Este reclamo es opcional.
- **niño**: “identificador de clave”. Un identificador único para esta clave. Se puede utilizar para hacer coincidir una clave con unaniño reclamar en el encabezado JWE o JWS, o elegir una clave de un conjunto de claves de acuerdo con la lógica de la aplicación. Este reclamo es opcional. Dos llaves en el mismo juego de llaves pueden llevar el mismoniñosolo si tienen diferentesktyreivindicaciones y están destinados al mismo uso.
- **x5u**: una URL que apunta a un certificado de clave pública X.509 o cadena de certificados en forma codificada PEM. Si existen otras declaraciones opcionales, deben ser consistentes con el contenido del certificado. Este reclamo es opcional.
- **x5c**: un certificado DER X.509 codificado en Base64-URL o una cadena de certificados. Una cadena de certificados se representa como una matriz de dichos certificados. El primer certificado debe ser el certificado al que se refiere este JWK. Todos los demás reclamos presentes en este JWK deben ser consistentes con los valores del primer certificado. Este reclamo es opcional.
- **x5t**: una huella digital/huella digital SHA-1 codificada en Base64-URL de la codificación DER de un certificado X.509. El certificado al que apunta esta huella digital debe ser coherente con las afirmaciones de este JWK. Este reclamo es opcional.
- **x5t#S256**: idéntico alx5treclamo, pero con la huella digital SHA-256 del certificado.

Otros parámetros, comox, y,od (del ejemplo al comienzo de este capítulo) son específicos del algoritmo clave. Las claves RSA, por otro lado, llevan parámetros comon, e, dp,etc. El significado de estos parámetros quedará claro enCapítulo 7, donde se explica en detalle cada algoritmo clave.

6.1.1 Conjunto de claves web JSON

La especificación JWK admite grupos de claves. Estos se conocen como “Conjuntos JWK”. Estos juegos llevan más de una llave. El significado de las teclas como grupo y el significado del orden de estas teclas lo define el usuario.

Un conjunto de claves web JSON es simplemente un objeto JSON con unllavesmiembro. Este miembro es una matriz JSON de JWK.

Conjunto JWK de muestra:


```
{
      "llaves":[
            {
                  "kty":"CE",
                  "crv":"P-256",
                  "X":"MKBCTNICKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4", "y":
                  "4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM", "usar"
                  :"enc",
                  "niño":"1"
            },

            {
                  "kty":"RSA",
                  "norte":"0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
                        4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMs
                        tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
                        QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbI
                        SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
                        w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw", "mi":"AQAB",

                  "algo":"RS256",
                  "niño":"2011-04-29"
            }
      ]
}
```


En este ejemplo, hay dos claves públicas disponibles. El primero es de tipo curva elíptica y se limita a*cifrado* operaciones por parte delusarreclamar. El segundo es de tipo RSA y está asociado a un algoritmo específico (RS256)por elalgoritmoreclamar. Esto significa que esta segunda clave está destinada a ser utilizada para *firmas*.

Capítulo 7

Algoritmos web JSON

Probablemente haya notado que hay muchas referencias a este capítulo a lo largo de este manual. La razón es que una gran parte de la magia detrás de JWT radica en los algoritmos empleados con él. La estructura es importante, pero los muchos usos interesantes descritos hasta ahora solo son posibles gracias a los algoritmos en juego. Este capítulo cubrirá los algoritmos más importantes en uso con los JWT en la actualidad. No es necesario comprenderlos en profundidad para usar los JWT de manera efectiva, por lo que este capítulo está dirigido a mentes curiosas que desean comprender la última pieza del rompecabezas.

7.1 Algoritmos generales

Los siguientes algoritmos tienen muchas aplicaciones diferentes dentro de las especificaciones JWT, JWS y JWE. Algunos algoritmos, como Base64-URL, se utilizan para formularios de serialización compactos y no compactos. Otros, como SHA-256, se utilizan para firmas, cifrado y huellas dactilares clave.

7.1.1 Base64

Base64 es un algoritmo de codificación de binario a texto. Su objetivo principal es convertir una secuencia de octetos en una secuencia de caracteres imprimibles, a costa de un tamaño adicional. En términos matemáticos, Base64 convierte una secuencia de números radix-256 en una secuencia de números radix-64. La palabra *base* se puede utilizar en lugar de *base*, de ahí el nombre del algoritmo.

> Nota: la especificación JWT no utiliza Base64. Es el *Base64-URL* variante descrita más adelante en este capítulo, que es utilizada por JWT.

Para comprender cómo Base64 puede convertir una serie de números arbitrarios en texto, primero es necesario estar familiarizado con los sistemas de codificación de texto. Los sistemas de codificación de texto asignan números a caracteres. Aunque esta asignación es arbitraria y, en el caso de Base64, se puede definir la implementación, el estándar de facto para la codificación Base64 es RFC 4648.[1].

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

0 un	17 R	34 yo	51 z
1B	18 S	35j	52 0
2C	19 T	36k	53 1
3D	20U	37l	54 2
4 mi	21V	38 metros	55 3
5F	22W	39 norte	56 4
6G	23X	40 o	57 5
7 horas	24 años	41 pags	58 6
8 yo	25Z	42 q	59 7
9J	26 un	43r	60 8
10K	27b	44 segundos	61 9
11L	28c	45 toneladas	62 +
12M	29 días	46 tu	63 /
13 norte	30 e	47v	
14 O	31 f	48w	(almohadilla) =
15P	32 gramos	49x	
16 q	33 horas	50 años	

En la codificación Base64, cada carácter representa 6 bits de los datos originales. La codificación se realiza en grupos de cuatro caracteres codificados. Entonces, 24 bits de datos originales se toman juntos y se codifican como cuatro caracteres Base64. Dado que se espera que los datos originales sean una secuencia de valores de 8 bits, los 24 bits se forman concatenando tres valores de 8 bits de izquierda a derecha.

Codificación Base64:

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

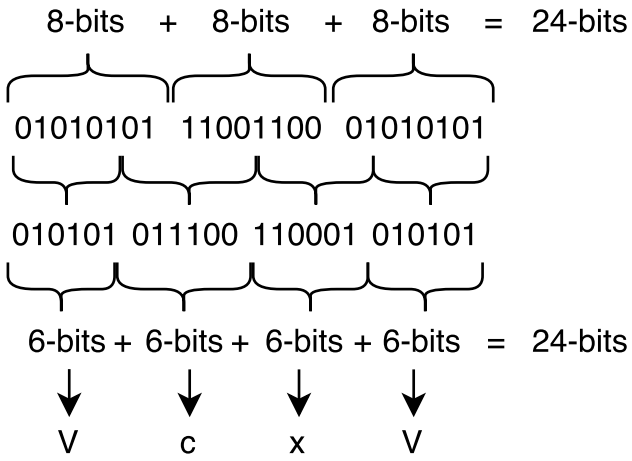

Figura 7.1: Codificación Base64

Si el número de octetos en los datos de entrada no es divisible por tres, entonces la última porción de datos a codificar tendrá menos de 24 bits de datos. Cuando este es el caso, se agregan ceros a los datos de entrada concatenados para formar un número entero de grupos de 6 bits. Hay tres posibilidades:

1. Los 24 bits completos están disponibles como entrada; no se realiza ningún procesamiento especial.
2. Hay 16 bits de entrada disponibles, se forman tres valores de 6 bits y al último valor de 6 bits se le agregan ceros adicionales a la derecha. La cadena codificada resultante se rellena con un carácter = extra para que quede explícito que faltaban 8 bits de entrada.
3. Hay 8 bits de entrada disponibles, se forman dos valores de 6 bits y al último valor de 6 bits se le agregan ceros adicionales a la derecha. La cadena codificada resultante se rellena con dos caracteres = adicionales para que quede explícito que faltaban 16 bits de entrada.

Algunas implementaciones consideran que el carácter de relleno (=) es opcional. Si realiza los pasos en el orden opuesto, obtendrá los datos originales, independientemente de la presencia de los caracteres de relleno.

7.1.1.1 Base64-URL

Ciertos caracteres de la tabla de conversión Base64 estándar no son seguros para URL. Base64 es una codificación conveniente para pasar datos arbitrarios en campos de texto. Dado que solo dos caracteres de Base64 son problemáticos como parte de la URL, una variante segura para URL es fácil de implementar. El carácter + y el carácter / se sustituyen por el carácter - y el carácter _.

7.1.1.2 Código de muestra

El siguiente ejemplo implementa un codificador Base64-URL tonto. El ejemplo está escrito pensando en la simplicidad, en lugar de la velocidad.

```
constantemesa=[
      'A','B','C','D','MI','F','GRAMO','H','YO','J', 'K','L','METRO',
      'NORTE','O','PAGS','Q','R','S','T', 'tú','V','W','X','Y','Z','a','b','C',
      'd', 'mi','F','gramo','h','i','j','k','yo','metro','norte', 'o','pags',
      'q','r','s','t','tú','v','w','X', 'y','z','0','1','2','3','4','5','6','7', '8','9','-'
      ,'_'


];

/**
 * @entrada de parámetrosun búfer, Uint8Array o Int8Array, matriz
 * @devolucionesuna cadena con los valores codificados
 */
exportarfunccodificar(aporte){
      dejarresultado="";

      por(dejari=0;i<aporte.longitud;i+=3){
            constanterestante=aporte.longitud-i;
```

```
            dejarconcat=entrada[yo]<<dieciséis;
            resultado+=(mesa[concat>>>(24-6)]);

            si(restante>1){
                  concat|=entrada[yo+1]<<8;
                  resultado+=tabla[(concat>>>(24-12))&0x3F];

                  si(restante>2){
                        concat|=entrada[yo+2];
                        resultado+=tabla[(concat>>>(24-18))&0x3F]+
                                    mesa[concat&0x3F];
                  }más{
                        resultado+=tabla[(concat>>>(24-18))&0x3F]+"=";
                  }
            }más{
                  resultado+=tabla[(concat>>>(24-12))&0x3F]+"==";
            }
      }

      devolverresultado;
}
```

7.1.2 SHA

El algoritmo hash seguro (SHA) utilizado en las especificaciones JWT se define en FIPS-180[2]. No debe confundirse con el SHA-1[3]familia de algoritmos, que han quedado en desuso desde 2010. Para diferenciar esta familia de la anterior, esta familia a veces se denomina*SHA-2*.

Los algoritmos en RFC 4634 son SHA-224, SHA-256, SHA-384 y SHA-512. De importancia para JWT son SHA-256 y SHA-512. Nos centraremos en la variante SHA-256 y explicaremos sus diferencias con respecto al resto de variantes.

Al igual que muchos algoritmos hash, SHA funciona procesando la entrada en fragmentos de tamaño fijo, aplicando una serie de operaciones matemáticas y luego acumulando el resultado realizando una operación con los resultados de la iteración anterior. Una vez que se procesan todos los fragmentos de entrada de tamaño fijo, se dice que se calcula el resumen.

La familia de algoritmos SHA se diseñó para evitar colisiones y producir una salida radicalmente diferente, incluso cuando la entrada solo cambia ligeramente. Es por ello que son considerados*seguro*: es computacionalmente inviable encontrar colisiones para diferentes entradas, o calcular la entrada original del resumen producido.

El algoritmo requiere una serie de funciones predefinidas:

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

```
función rotr(X,norte){
    devolver(X>>>norte)|(X<<(32-norte));
}

función ch(X,y,z){
    devolver(X&y)^((~X)&z);
}

función comandante(X,y,z){
    devolver(X&y)^(X&z)^(y&z);
}

función bsig0(X){
    devolver rotr(X,2)^rotr(X,13)^rotr(X,22);
}

función bsig1(X){
    devolver rotr(X,6)^rotr(X,11)^rotr(X,25);
}

función ssig0(X){
    devolver rotr(X,7)^rotr(X,18)^(X>>>3);
}

función ssig1(X){
    devolver rotr(X,17)^rotr(X,19)^(X>>>10);
}
```

Estas funciones se definen en la especificación. losrotrLa función realiza una rotación bit a bit (hacia la derecha).

Además, el algoritmo requiere que el mensaje tenga una longitud predefinida (un múltiplo de 64); por lo tanto, se requiere relleno. El algoritmo de relleno funciona de la siguiente manera:

1. Un solo binario1se adjunta al final del mensaje original. Por ejemplo:

   Mensaje original:
   01011111 01010101 10101010 00111100
   Extra 1 al final:
   01011111 01010101 10101010 00111100 1

2. Se añade un número N de ceros para que la longitud resultante del mensaje sea la solución a esta ecuación:

   L = Longitud del mensaje en
   bits 0 = (65 + N + L) mod 512

3. Luego, el número de bits en el mensaje original se agrega como un número entero de 64 bits:

   Mensaje original:

01011111 01010101 10101010 00111100
Extra 1 al final:
01011111 01010101 10101010 00111100 1
N ceros:
01011111 01010101 10101010 00111100 10000000 ...0...
Mensaje rellenado:
01011111 01010101 10101010 00111100 10000000 ...0... 00000000 00100000

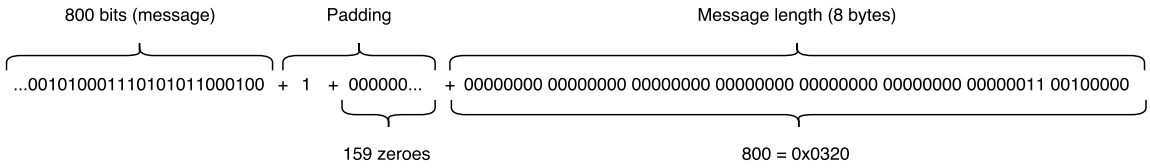

Figura 7.2: relleno SHA

Una implementación simple en JavaScript podría ser:

```
función padMensaje(mensaje){
    si(!(mensaje en vez de Uint8Array)&&!(mensaje en vez de Int8Array)){
        tirar nuevo Error("contenedor de mensajes no compatible");
    }

    constante bitLongitud=mensaje.longitud*8;
    constante longitud total=bitLongitud+sesenta y cinco;//Extra 1 + tamaño del mensaje.
    dejar longitud acolchada=(longitud total+(512-longitud total%512)) /32; dejar acolchado=
    nuevo Uint32Array(Longitud acolchada);

    por(dejar i=0;i<mensaje.longitud; ++i){
        acolchado[Matemáticas.piso(i /4)]|=(mensaje[yo]<<(24-(i%4)*8));
    }

    acolchado[Matemáticas.piso(mensaje.longitud/4)]|=(0x80<<(24-(mensaje.longitud%4)*8)); //QUE HACER
    : admite mensajes con longitud de bits superior a 2^32 acolchado[acolchado.longitud-1]=bitLongitud;


    devolver acolchado;
}
```

A continuación, el mensaje rellenado resultante se procesa en bloques de 512 bits. La implementación a continuación sigue el algoritmo descrito en la especificación paso a paso. Todas las operaciones se realizan en enteros de 32 bits.

```
exportación predeterminada función sha256(mensaje,bytes de retorno){
    // Valores hash iniciales
    constante h_=Uint32Array.de(
```

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

constanteacolchado=padMensaje(mensaje);
constantew=nuevoUint32Array(64);
por(dejari=0;i<acolchado.longitud;i+=dieciséis){
        por(dejart=0;t<dieciséis; ++t){
                w[t]=acolchado+t];
        }
        por(dejart=dieciséis;t<64; ++t){
                w[t]=ssig1(w[t-2])+w[t-7]+ssig0(w[t-15])+w[t-dieciséis];
        }

        dejara=h_[0]>>>0;
        dejarb=h_[1]>>>0;
        dejarC=h_[2]>>>0;
        dejard=h_[3]>>>0;
        dejarmi=h_[4]>>>0;
        dejarF=h_[5]>>>0;
        dejargramo=h_[6]>>>0;
        dejarh=h_[7]>>>0;

        por(dejart=0;t<64; ++t){
                dejart1=h+bsig1(mi)+ch(mi,F,gramo)+k[t]+w[t]; dejart2=
                bsig0(a)+comandante(a,b,C); h=gramo;

                gramo=F;
                F=mi;
                mi=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]=(mi+h_[4])>>>0; h_[5]=(F
            +h_[5])>>>0; h_[6]=(gramo+h_[
            6])>>>0; h_[7]=(h+h_[7])>>>0;

        }

        //(...)
}
```

La variablekcontiene una serie de constantes, que se definen en la especificación.

El resultado final está en la variableh_[0..7].El único paso que falta es presentarlo en forma legible:

```
si(returnBytes){
        constanteresultado=nuevoUint8Array(h_.longitud*4
        ); h_.para cada((valor,índice)=> {
            constantei=índice*4; resultado[yo]=(valor
            >>>24)&0xFF; resultado [yo+1]=(valor>>>
            dieciséis)&0xFF; resultado [yo+2]=(valor>>>8)
            resultado[yo+3]=(valor>>>0)          &0xFF;
                                                 &0xFF;
        });

        devolverresultado;
}más{
        funciónaHex(norte){
            dejarcalle=(norte>>>0).(dieciséis);
            dejarresultado="";
            por(dejari=calle.longitud;i<8; ++i){
                resultado+="0";
            }
            devolverresultado+calle;
        }
        dejarresultado=""; h_.
        para cada(norte=> {
            resultado+=aHex(norte);
        });
        devolverresultado;
}
```

Aunque funciona, tenga en cuenta que la implementación anterior no es óptima (y no admite mensajes de más de 2^{32}).

Otras variantes de la familia SHA-2 (como SHA-512) simplemente cambian el tamaño del bloque procesado en cada iteración y alteran las constantes y su tamaño. En particular, SHA-512 requiere que las matemáticas de 64 bits estén disponibles. En otras palabras, para convertir la implementación de muestra anterior en SHA-512, se requiere una biblioteca separada para operaciones matemáticas de 64 bits (ya que JavaScript solo admite operaciones bit a bit de 32 bits y operaciones matemáticas de punto flotante de 64 bits).

7.2 Algoritmos de firma

7.2.1 HMAC

Códigos de autenticación de mensajes basados en hash (HMAC)[4] hacer uso de una función hash criptográfica (como la familia SHA discutida anteriormente) y una clave para crear un *Código de autenticación* para un mensaje específico. En otras palabras, un esquema de autenticación basado en HMAC toma una función hash, un mensaje y una clave secreta como entradas y produce un código de autenticación como salida. La fuerza de la función hash criptográfica garantiza que el mensaje no se pueda modificar sin la clave secreta. Por lo tanto, los HMAC cumplen ambos propósitos de *autenticación* y *integridad de los datos*.

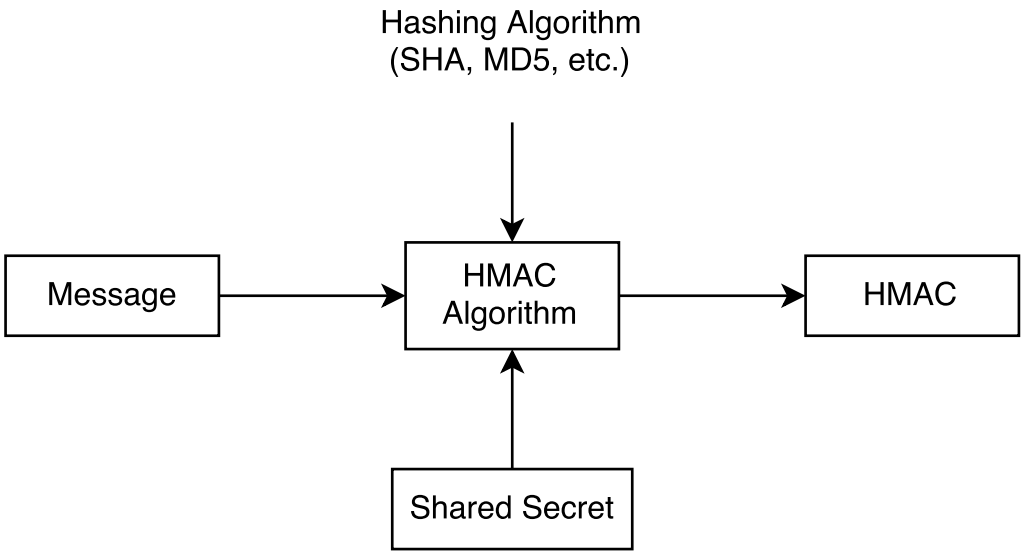

Figura 7.3: HMAC

Las funciones hash débiles pueden permitir que los usuarios maliciosos comprometan la validez del código de autenticación. Por lo tanto, para que los HMAC sean útiles, se debe elegir una función hash fuerte. La familia de funciones SHA-2 sigue siendo lo suficientemente fuerte para los estándares actuales, pero esto puede cambiar en el futuro. MD5, una función hash criptográfica diferente utilizada ampliamente en el pasado, se puede usar para HMAC. Sin embargo, puede ser vulnerable a colisiones y ataques de prefijos. Aunque estos ataques no necesitanpor lo general, MD5 no es adecuado e para usar con HMAC, los algoritmos más fuertes están fácilmente disponibles y deben
Sería considerado.

el algoritmo el ritmo es lo suficientemente simple como para fi t en una sola línea:

Sea H la criptografia tiene funcion h
ser b la longitud del bloque de t él función hash

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

(cuántos bits se procesan por iteración) K sea la
clave secreta
K' sea la clave real utilizada por el algoritmo HMAC L sea la
longitud de la salida de la función hash ipad sea el byte
0x36 repetido B veces
opad sea el byte 0x5C repetido B veces el
mensaje sea el mensaje de entrada
|| sea la función de concatenación

HMAC(mensaje) = H(K' XOR opad || H(K' XOR ipad || mensaje))

K'se calcula a partir de la clave secretakcomo sigue:

Sikes más corto queB,se añaden ceros hastakes deBlongitud. El resultado esK'.Sikes más largo queB, Hes aplicado akEl resultado esK'.Sikes exactamenteBbytes, se utiliza tal cual (kesK').

Aquí hay una implementación de muestra en JavaScript:

```
exportación predeterminadafunciónhmac(hashFn,bits de tamaño de bloque,secreto,mensaje,bytes de retorno){
    si(!(mensajeen vez deUint8Array)){
        tirar nuevoError('el mensaje debe ser de Uint8Array');
    }

    constanteblockSizeBytes=bits de tamaño de bloque /8;

    constanteipad=nuevoUint8Array(bytes de tamaño de
    bloque); constanteopad=nuevoUint8Array(bytes de tamaño
    de bloque); ipad.llenar(0x36); opad.llenar(0x5c);


    constantesecretoBytes=cadenaToUtf8(secreto); dejar
    acolchadosecreto;
    si(secretoBytes.longitud<=bytes de tamaño de bloque){
        constantediferencia=blockSizeBytes-secretoBytes.longitud;
        acolchadosecreto=nuevoUint8Array(bytes de tamaño de bloque);
        acolchadosecreto.establecer(bytes secretos); }más{

        acolchadosecreto=hashFn(bytes secretos);
    }

    constanteipadsecreto=ipad.mapa((valor,índice)=> {
        devolvervalor^paddedSecret[índice];
    });
    constanteopadsecreto=opad.mapa((valor,índice)=> {
        devolvervalor^paddedSecret[índice];
    });

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

```
    constante resultado=hashFn(
        adjuntar(opadSecreto,
                uint32ArrayToUint8Array(hashFn(adjuntar(ipadSecreto,
                                                        mensaje),verdadero))),
        bytes de retorno);

    devolver resultado;
}
```

Para verificar un mensaje contra un HMAC, uno simplemente calcula el HMAC y compara el resultado con el HMAC que vino con el mensaje. Esto requiere el conocimiento de la clave secreta por todas las partes: aquellos que producen el mensaje y aquellos que solo quieren verificarlo.

7.2.1.1 HMAC + SHA256 (HS256)

Comprender Base64-URL, SHA-256 y HMAC es todo lo que se necesita para implementar el algoritmo de firma HS256 de la especificación JWS. Con esto en mente, ahora podemos combinar todo el código de muestra desarrollado hasta ahora y construir un JWT completamente firmado.

```
exportación predeterminada función jwtCodificar(encabezamiento,carga útil,secreto){
    si(tipo de encabezamiento!=='objeto'||tipo de carga útil!=='objeto'){
        tirar nuevo Error('el encabezado y la carga útil deben ser objetos');
    }
    si(tipo de secreto!=='cuerda'){
        tirar nuevo Error("el secreto debe ser una cadena");
    }

    encabezamiento.algoritmo='HS256';

    constante encabezado=b64(JSON.encadenar(encabezamiento));
    constante encCarga útil=b64(JSON.encadenar(carga útil));
    constante jwtDesprotegido=`${encabezado}.${encCarga útil}`;
    constante firma=b64(uint32ArrayToUint8Array(
        hmac(sha256,512,secreto,cadenaToUtf8(jwtDesprotegido),verdadero)));

    devolver `${jwtDesprotegido}.${firma}`;
}
```

Tenga en cuenta que esta función no realiza ninguna validación del encabezado o la carga útil (aparte de verificar si son objetos). Puedes llamar a esta función así:

```
consola.Iniciar sesión(jwtCodificar({}, {sub:" prueba@prueba.com "},'secreto'));
```

Pegue el JWT generado en el depurador de JWT.io[5] y vea cómo se decodifica y valida.

Esta función es muy similar a la utilizada en Capítulo 4 como demostración del algoritmo de firma. De Capítulo 4:

[5] https://jwt.io

```
constante Encabezado codificado=base64(utf8(JSON.encadenar(encabezamiento)));
constante codificadoPayload=base64(utf8(JSON.encadenar(carga útil)));
constante firma=base64(hmac($ps{Encabezado codificadopscodificadoPayload}`,secreto,sha256));
constante jwt=$ps{Encabezado codificadopscodificadoPayloadpsfirma};
```

La verificación es igual de fácil:

```
exportar función jwtVerifyAndDecode(jwt,secreto){
    si(!esCadena(jwt)|| !esCadena(secreto)){
        tirar nuevo Error de tecleado('jwt y secret deben ser cadenas');
    }

    constante separar=jwt.separar('.');
    si(separar.longitud!==3){
        tirar nuevo Error('Formato JWT no válido');
    }

    constante encabezamiento=JSON.analizar gramaticalmente(unb64(separar[0]));
    si(encabezamiento.algoritmo!=='HS256'){
        tirar nuevo Error(`Algoritmo incorrecto: ${encabezamiento.algoritmo}`);
    }

    constante jwtDesprotegido=`${separar[0]}psseparar[1]};
    constante firma=
        b64(hmac(sha256,512,secreto,cadenaToUtf8(jwtDesprotegido),verdadero));

    devolver{
        encabezamiento:encabezamiento,
        carga útil:JSON.analizar gramaticalmente(unb64
        (separar[1])), válido:firma==separar[2]
    };
}
```

La firma se separa del JWT y se calcula una nueva firma. Si la nueva firma coincide con la incluida en el JWT, entonces la firma es válida.

Puede utilizar la función anterior de la siguiente manera:

```
constante secreto='secreto';
constante codificado=jwtCodificar({}, {sub:" prueba@prueba.com "},secreto);
constante descifrado=jwtVerifyAndDecode(codificado,secreto);
```

Este código está disponible en elhs256.jsarchivo de las muestras incluidas[6]con este manual.

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

7.2.2 RSA

RSA es uno de los criptosistemas más utilizados en la actualidad. Fue desarrollado en 1977 por Ron Rivest, Adi Shamir y Leonard Adleman, cuyas iniciales se utilizaron para nombrar el algoritmo. El aspecto clave de RSA radica en su asimetría: la clave utilizada para cifrar algo es*no*la clave utilizada para descifrarlo. Este esquema se conoce como cifrado de clave pública (PKI), donde la clave pública es la clave de cifrado y la clave privada es la clave de descifrado.

Cuando se trata de firmas, la clave privada se utiliza para*señal*una pieza de información y la clave pública se utiliza para*verificar*que fue firmado por una clave privada específica (sin saberlo realmente).

Existen variaciones del algoritmo RSA tanto para la firma como para el cifrado. Primero nos centraremos en el algoritmo general y luego veremos las diferentes variaciones utilizadas con los JWT.

Muchos algoritmos criptográficos, y en particular RSA, se basan en la dificultad relativa de realizar ciertas operaciones matemáticas. RSA elige la factorización de enteros[7]como su principal herramienta matemática. La factorización de enteros es el problema matemático que intenta encontrar números que multiplicados entre sí den como resultado el número original. En otras palabras, los factores de un número entero son un conjunto de pares de números enteros que, cuando se multiplican, dan como resultado el número entero original.

entero = factor_1 x factor_2

Este problema puede parecer fácil al principio. Y para números pequeños, lo es. Tomemos por ejemplo el número 35:

35 = 7x5

Al conocer las tablas de multiplicar del 7 o del 5, es fácil encontrar dos números que produzcan35 cuando se multiplica. Un algoritmo ingenuo para encontrar factores para un número entero podría ser:

1. Dejanortesea el número que queremos factorizar.
2. DejaXser un numero entre2 (inclusive) yn/2 (inclusivo).
3. DividirnorteporXy comprueba si el resto es0.Si es así, has encontrado un par de factores (Xy el cociente).
4. Continúe realizando el paso 3 aumentandoXpor1en cada iteración hastaXalcanza su límite superior n/2 Cuando lo hace, ha encontrado todos los factores posibles denorte.

Este es esencialmente el enfoque de fuerza bruta para encontrar factores. Como puedes imaginar, este algoritmo es terriblemente ineficiente.

> Una mejor versión de este algoritmo se llamadivisión de juicioy establece condiciones más estrictas paraX.En particular, defineX's límite superior comosqrt(n),y, en lugar de aumentar Xpor1en cada iteración, haceXtomar el valor de números primos cada vez más grandes. Es trivial probar por qué estas condiciones hacen que el algoritmo sea más eficiente mientras lo mantienen correcto (aunque fuera del alcance de este texto).

Existen algoritmos más eficientes, pero, a pesar de lo eficientes que son, incluso con las computadoras actuales, ciertos números son computacionalmente inviables para factorizar. El problema se complica cuando ciertas

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

Los números se eligen comonorte.Por ejemplo, sinortees el resultado de multiplicar dos numeros primos[8], es mucho más difícil encontrar sus factores (de los cuales esos números primos son los únicos factores posibles).

> Sinortefuera el resultado de multiplicar dos números no primos, sería mucho más fácil encontrar sus factores. ¿Por qué? Porque los números no primos tienen divisores distintos a ellos mismos (por definición), y estos divisores son a su vez divisores de cualquier número multiplicado por ellos. En otras palabras, cualquier divisor de un factor de un número también es un factor del número. O, en otros términos, sinortetiene factores no primos, tiene más de dos factores. Así que si nortees el resultado de multiplicar dos números primos, tiene exactamente dos factores (el menor número posible de factores sin ser un número primo). Cuanto menor sea el número de factores, más difícil será encontrarlos.

Cuando se eligen dos números primos diferentes y grandes y luego se multiplican, el resultado es otro número grande (llamado semiprimo). Pero este gran número tiene una propiedad especial adicional: es realmente difícil de factorizar. Incluso los algoritmos de factorización más eficientes, como el tamiz de campo numérico general[9], no puede factorizar números grandes que son el resultado de multiplicar números primos grandes en períodos de tiempo razonables. Para dar una idea de la escala, en 2009 se factorizó un número de 768 bits (232 dígitos decimales)[10]después de 2 años de trabajo por un grupo de computadoras. Las aplicaciones típicas de RSA utilizan números de 2048 bits o más.

> Algoritmo de Shor[11]es un tipo especial de algoritmo de factorización que podría cambiar las cosas drásticamente en el futuro. Si bien la mayoría de los algoritmos de factorización son de naturaleza clásica (es decir, operan en computadoras clásicas), el algoritmo de Shor se basa en computadoras cuánticas.[12]. Las computadoras cuánticas aprovechan la naturaleza de ciertos fenómenos cuánticos para acelerar varias operaciones clásicas. En particular, el algoritmo de Shor podría acelerar la factorización, llevando su complejidad al ámbito de la complejidad del tiempo polinomial (en lugar de exponencial). Esto es mucho más eficiente que cualquiera de los algoritmos clásicos actuales. Se especula que si dicho algoritmo pudiera ejecutarse en una computadora cuántica, las claves RSA actuales se volverían inútiles. Todavía no se ha desarrollado una computadora cuántica práctica como lo requiere el algoritmo de Shor, pero esta es un área activa de investigación en este momento.

Aunque actualmente la factorización de enteros es computacionalmente inviable para semiprimos grandes, no hay prueba matemática de que este sea el caso. En otras palabras, en el futuro podrían aparecer algoritmos que resuelvan la factorización de enteros en plazos razonables. Lo mismo puede decirse de RSA.

Dicho esto, ahora podemos centrarnos en el algoritmo real. El principio básico se captura en esta expresión:

$$metro^{m)}_{3} \equiv metro(modelo)$$

Figura 7.4: Expresión básica RSA

[8] https://en.wikipedia.org/wiki/Prime_number
[9] https://en.wikipedia.org/wiki/General_number_field_sieve
[10] http://eprint.iacr.org/2010/006
[11] https://en.wikipedia.org/wiki/Shor%27s_algorithm
[12] https://en.wikipedia.org/wiki/Quantum_computing

Es computacionalmente factible encontrar tres enteros muy grandese, dynorteque satisfacen la ecuación anterior. El algoritmo se basa en la dificultad de encontrardcuando se conocen todos los demás números. En otras palabras, esta expresión se puede convertir en una función unidireccional.dentonces puede considerarse la clave privada, mientras quenorteymison la clave pública.

7.2.2.1 Elegir e, d y n

1. Elige dos números primos distintospagsyq.
   - Se debe usar un generador de números aleatorios criptográficamente seguro para elegir candidatos parapagsyq.Un RNG inseguro puede hacer que un atacante encuentre uno de estos números.
   - Como no hay forma de generar números primos aleatoriamente, después de elegir dos números aleatorios, deben pasar una prueba de primalidad. Las comprobaciones de primalidad deterministas pueden ser costosas, por lo que algunas implementaciones se basan en métodos probabilísticos. Debe considerarse la probabilidad de encontrar un número primo falso.
   - pagsyqdeben ser similares en magnitud pero no idénticos, y deben diferir en longitud por unos pocos dígitos.
2. nortees el resultado depagsvecesq.Este es el módulo de la ecuación anterior. Su número de bits es la longitud de la clave del algoritmo.
3. Calcule la función totient de Euler[13]pornorte.Ya quenortees un número semiprimo, esto es tan simple como: n - (p + q - 1).Llamaremos a este valorfi(n).
4. Elija unmique cumple con los siguientes criterios:
   - 1 < e < fi(n)
   - miyfi(n)debería ser coprimo
5. Elige undque satisface la siguiente expresión:

$$\equiv mi^{-1}(\mathrm{mod}\varphi(norte))$$

Figura 7.5: Expresión básica RSA

La clave pública se compone de valores.norteymi.La clave privada se compone de valores.norteyd. Valores pag qyfi(n)deben desecharse o mantenerse en secreto, ya que pueden usarse para ayudar a encontrard.

De las ecuaciones anteriores, es evidente quemiydson matemáticamente simétricos. Podemos reescribir la ecuación del paso 5 como:

$$Delaware\equiv 1(\mathrm{mod}\varphi(norte))$$

Figura 7.6: Simetría entre e y d

Así que ahora probablemente se esté preguntando qué tan seguro es RSA si publicamos los valoresmiynorte;¿No podríamos usar esos valores para encontrar¿d?Lo que pasa con la aritmética modular es que hay múltiples soluciones posibles. Siempre y cuando eldelegimos satisface la ecuación anterior, cualquier valor es válido. Cuanto mayor sea el valor, más difícil será encontrarlo. Entonces, RSA funciona siempre que solo uno de los valoresmiodes conocido por

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

fiestas públicas. Esta es también la razón por la que se elige uno de esos valores: el valor público se puede elegir para que sea lo más pequeño posible. Esto acelera el cálculo sin comprometer la seguridad del algoritmo.

7.2.2.2 Firma básica

El inicio de sesión en RSA se realiza de la siguiente manera:

1. Se produce un resumen del mensaje a partir del mensaje que se firmará mediante una función hash.
2. Este compendio se eleva luego a la potencia dedmódulon (que es parte de la clave privada).
3. El resultado se adjunta al mensaje como firma.

Cuando un destinatario que posee la clave pública desea verificar la autenticidad del mensaje, puede revertir la operación de la siguiente manera:

1. La firma se eleva a la potencia demimódulonorte.El valor resultante es el valor de resumen de referencia.
2. Se genera un resumen del mensaje a partir del mensaje utilizando la misma función hash que en el paso de firma.
3. Se comparan los resultados de los pasos 1 y 2. Si coinciden, la parte firmante debe estar en posesión de la clave privada.

Este esquema de firma/verificación se conoce como “esquema de firma con apéndice” (SSA). Este esquema requiere que el mensaje original esté disponible para verificar el mensaje. En otras palabras, no permiten la recuperación de mensajes a partir de la firma (el mensaje y la firma permanecen separados).

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

Ahora que tenemos una noción básica de cómo funciona RSA, podemos centrarnos en una variante específica: PKCS#1 RSASSA v1.5 usando SHA-256, también conocido como RS256 en la especificación JWA.

El estándar de criptografía de clave pública n.º 1 (PKCS n.º 1)[14]especificación define una serie de primitivas, formatos y esquemas de cifrado basados en el algoritmo RSA. Estos elementos trabajan juntos para proporcionar una implementación detallada de RSA utilizable en plataformas informáticas modernas. RSASSA es uno de los esquemas definidos en él, y permite el uso de RSA para firmas.

7.2.2.3.1 Algoritmo

Para producir una firma:

1. Aplicar el**EMSA-PKCS1-V1_5-ENCODE**primitiva al mensaje (una matriz de octetos). El resultado es el**mensaje codificado**. Esta primitiva hace uso de una función hash (generalmente una función hash de la familia SHA como SHA-256). Esta primitiva acepta una longitud esperada de mensaje codificado. En este caso será la longitud en octetos del número RSAn (la longitud de la clave).

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

2. Aplicar el**OS2IP**primitiva al mensaje codificado. El resultado es el**representante de mensaje entero**. OS2IP es el acrónimo de "Octet-String to Integer Primitive".
3. Aplicar el**RSASP1**primitiva al representante del mensaje entero utilizando la clave privada. El resultado es el**representante de firma entera**.
4. Aplicar el**I2OSP**primitiva para convertir el representante de la firma entera en una matriz de octetos (el**firma**). I2OSP es el acrónimo de "Integer to Octet-String Primitive".

Una posible implementación en JavaScript, dadas las primitivas mencionadas anteriormente, podría verse así:

```
/**
 * Produce una firma para un mensaje usando el algoritmo RSA como se define
 * en PKCS#1.
 * @parámetro {clave privada}Clave privada RSA, un objeto con
 *                           tres miembros: tamaño (tamaño en bits), n (el módulo) y d (el
 *                           exponente privado), ambos bigInts
 *                           (biblioteca de enteros grandes).
 * @parámetro {hashFn}la función hash requerida por PKCS#1,
 *                    debe tomar un Uint8Array y devolver un Uint8Array
 * @parámetro {tipo hash}Un símbolo que identifica el tipo de función hash pasada.
 *                       Por ahora, solo se admite "SHA-256". Consulte el objeto "hashTypes" para
 *                       conocer los posibles valores.
 * @parámetro {mensaje}Un String o Uint8Array con datos arbitrarios para firmar
 * @devolver{Uint8Array} La firma como Uint8Array
 */
exportarfunciónseñal(llave privada,hashFn,tipo hash,mensaje){
    constantemensaje codificado=
        emsaPkcs1v1_5(hashFn,tipo hash,llave privada.Talla/8,mensaje);
    constanteintMensaje=os2ip(mensaje codificado); constanteintFirma=rsasp1
    (llave privada,intMensaje); constantefirma=i2osp(intFirma,llave privada.Talla/
    8); devolverfirma;

}
```

Para verificar una firma:

1. Aplicar el**OS2IP**primitiva a la firma (una matriz de octetos). Este es el**representante de firma entera**.
2. Aplicar el**RSAVP1**primitiva al resultado anterior. Esta primitiva también toma la clave pública como entrada. Este es el**representante de mensaje entero**.
3. Aplicar el**I2OSP**primitiva al resultado anterior. Esta primitiva toma un tamaño esperado como entrada. Este tamaño debe coincidir con la longitud del módulo de la clave en número de octetos. El resultado es el **mensaje codificado**.
4. Aplicar el**EMSA-PKCS1-V1_5-ENCODE**primitiva al mensaje que se va a verificar. el resultado es otro**mensaje codificado**. Esta primitiva hace uso de una función hash (generalmente una función hash de la familia SHA como SHA-256). Esta primitiva acepta una longitud esperada de mensaje codificado. En este caso será la longitud en octetos del número RSAn (la longitud de la clave).
5. Compare ambos mensajes codificados (de los pasos 3 y 4). Si coinciden, la firma es válida,

de lo contrario no lo es.

En JavaScript:

```
/**
 * Verifica una firma para un mensaje usando el algoritmo RSASSA como se define
 * en PKCS#1.
 * @parámetro {clave pública}Clave privada RSA, un objeto con
 *                              tres miembros: tamaño (tamaño en bits), n (el módulo) y e (el
 *                              exponente público), ambos bigInts
 *                              (biblioteca de enteros grandes).
 * @parámetro {hashFn}la función hash requerida por PKCS#1,
 *                          debe tomar un Uint8Array y devolver un Uint8Array
 * @parámetro {tipo hash}Un símbolo que identifica el tipo de función hash pasada.
 *                              Por ahora, solo se admite "SHA-256". Consulte el objeto "hashTypes" para
 *                              conocer los posibles valores.
 * @parámetro {mensaje}Una cadena o Uint8Array con datos arbitrarios para verificar
 * @parámetro {firma}Un Uint8Array con la firma
 * @devolver{Boolean} true si la firma es válida, false en caso contrario.
 */
exportarfunciónverificarPkcs1v1_5(Llave pública,
                                                hashFn,
                                                tipo hash,
                                                mensaje,
                                                firma){
    si(firma.longitud!==Llave pública.Talla/8){
        tirar nuevoError('longitud de firma no válida');
    }

    constanteintFirma=os2ip(firma);
    constanteintVerification=rsavp1(Llave pública,intFirma);
    constantemensaje de verificación=i2osp(intVerificación,Llave pública.Talla/8);

    constantemensaje codificado=
        emsaPkcs1v1_5(hashFn,tipo hash,Llave pública.Talla/8,mensaje);

    devolveruint8ArrayEquals(mensaje codificado,mensaje de verificación);
}
```

7.2.2.3.1.1 Primitiva EMSA-PKCS1-v1_5 Esta

primitiva toma tres elementos:

- El mensaje
- La longitud prevista del resultado
- Y la función hash a utilizar (que debe ser una de las opciones del paso 2)

1. Aplique la función hash seleccionada al mensaje.

2. Produzca la codificación DER para la siguiente estructura ASN.1:

```
ResumenInfo ::=SECUENCIA{
      digestAlgorithm DigestAlgorithm,
      resumen OCTET STRING
}
```

Dóndedigerires el resultado del paso 1 yAlgoritmo de resumenes uno de:

```
Algoritmo de resumen ::=
      Identificador de algoritmo { {PKCS1-v1-5DigestAlgorithms} }

PKCS1-v1-5DigestAlgoritmos          ALGORITMO-IDENTIFICADOR ::= {
      { PARÁMETROS OID id-md2NULO { OID id-
      md5 PARÁMETROSNULO { PARÁMETROS
      OID id-sha1NULO { OID id-sha256
      PARÁMETROSNULO}| { OID id-sha384
      PARÁMETROSNULO}| { OID id-sha512
      PARÁMETROSNULO}
}
```

3. Si la longitud solicitada del resultado es menor que el resultado del paso 3 más11 (reqLength < step2Length + 11), entonces la primitiva no produce el resultado y genera un mensaje de error ("longitud del mensaje codificado previsto demasiado corta").
4. Repita el octeto0xFFel siguiente número de veces:longitud solicitada + step2Length - 3.Este arreglo de octetos se llamaPD.
5. Produzca el mensaje codificado final (EM) como (|| es el operador de concatenación):

ME = 0x00 || 0x01 || PD || 0x00 || paso2Resultado

Los OID ASN.1 generalmente se definen en sus propias especificaciones. En otras palabras, no encontrará el OID SHA-256 en la especificación PKCS#1. Los OID SHA-1 y SHA-2 se definen en RFC 3560[15].

7.2.2.3.1.2 Primitiva OS2IP

La primitiva OS2IP toma una matriz de octetos y genera un entero representativo.

- Sea X_1, X_2, ..., X_{norte}ser los octetos del primero al último de la entrada.
- Calcular el resultado como:

$$resultado = X_1 \cdot 256^{norte-1} + X_2 \cdot 256^{norte-2} + ... + X_{norte-1} \cdot 256 + X_{norte}$$

Figura 7.7: Resultado OS2IP

7.2.2.3.1.3 Primitiva RSASP1

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

La primitiva RSASP1 toma la clave privada y un mensaje representativo y produce un representante de firma.

- Dejarnorteydser los números RSA para la clave privada.
- Dejarmetroser el representante del mensaje.

1. Verifique que el representante del mensaje esté dentro del rango: entre 0 yn - 1.
2. Calcule el resultado de la siguiente manera:

$$s = metro_{d} modificación(norte)$$

Figura 7.8: Resultado RSASP1

PKCS#1 define una forma alternativa y computacionalmente conveniente de almacenar la clave privada: en lugar de mantenernorteyden él se almacenan una combinación de diferentes valores precalculados para determinadas operaciones. Estos valores se pueden usar directamente en ciertas operaciones y pueden acelerar significativamente los cálculos. La mayoría de las claves privadas se almacenan de esta manera. Almacenar la clave privada comonorteydes válido, sin embargo.

7.2.2.3.1.4 Primitiva RSAVP1

La primitiva RSAVP1 toma una clave pública y un representante de firma entero y produce un representante de mensaje entero.

- Dejarnorteymiser los números RSA para la clave pública.
- Dejarsser el representante de la firma entera.

1. Verifique que el representante del mensaje esté dentro del rango: entre 0 yn - 1.
2. Calcule el resultado de la siguiente manera:

$$= s_{mi} modificación(norte)$$

Figura 7.9: Resultado RSAVP1

7.2.2.3.1.5 Primitiva I2OSP

La primitiva I2OSP toma un entero representativo y produce una matriz de octetos.

- DejarLenSea la longitud esperada de la matriz de octetos.
- DejarXSea el representante entero.

1. Si x > 256_{Len}entonces el número entero es demasiado grande y los argumentos son incorrectos.
2. Calcule la representación en base 256 del entero:

$$X = X_1 \cdot 256^{Len-1} + X_2 \cdot 256^{Len-2} + ... + X_{Len-1} \cdot 256 + X_{Len}$$

Figura 7.10: descomposición I2OSP

3. Toma cada x_{len-yo}factor de cada término en orden. Estos son los octetos para el resultado.

7.2.2.3.2 Código de muestra

Dado que RSA requiere aritmética de precisión arbitraria, usaremos el entero grandedieciséisBiblioteca JavaScript.

losOS2IPyI2OSPLas primitivas son bastante simples:

```
función os2ip(bytes){
    dejar resultado=Empezando();

    bytes.para cada((b,i)=> {
        // resultado += b * Math.pow(256, bytes.length - 1 - i);
        resultado=resultado.agregar(
            Empezando(b).multiplicar(
                Empezando(256).pow(bytes.longitud-i-1)
            )
        );
    });

    devolver resultado;
}

función i2osp(intRepr,longitud esperada){
    si(intRepr.mayor o igual(Empezando(256).pow(longitud esperada))){
        tirar nuevo Error('entero demasiado grande');
    }

    constante resultado=nuevo Uint8Array(longitud
    esperada); dejar resto=Empezando(intRepr);
    por(dejar i=longitud esperada-1;i>=0; --i){
        constante posición=Empezando(256).pow(i);
        constante quotrem=resto.divmod(posición); resto
        =quotrem.resto;
        resultado[resultado.longitud-1-i]=quotrem.cociente.();
    }

    devolver resultado;
}
```

losI2OSPprimitiva esencialmente descompone un número en su base 256[17]componentes

losRSASP1primitiva es esencialmente una sola operación y forma la base del algoritmo:

---

dieciséishttps://www.npmjs.com/package/big-integer

[17]https://en.wikipedia.org/wiki/Notación_posicional

```
función rsasp1(llave privada,intMensaje){
    si(intMensaje.es negativo()||
        intMensaje.mayor o igual(llave privada.norte)){
          tirar nuevo Error("representante de mensaje fuera de rango");
    }

    // resultado = intMessage ^ d (mod n)
    devolver intMensaje.modpow(llave privada.d,llave privada.norte);
}
```

Para las verificaciones, elRSAVP1en su lugar se usa primitivo:

```
exportar función rsavp1(Llave pública,intFirma){
    si(intFirma.es negativo()||
        intFirma.mayor o igual(Llave pública.norte)){
          tirar nuevo Error("representante de mensaje fuera de rango");
    }

    // resultado = intSignature ^ e (mod n)
    devolver intFirma.modpow(Llave pública.mi,Llave pública.norte);
}
```

Finalmente, elEMSA-PKCS1-v1_5primitiva realiza la mayor parte del trabajo duro al transformar el mensaje en su representación codificada y rellenada.

```
función emsaPkcs1v1_5(hashFn,tipo hash,longitud esperada,mensaje){
    si(tipo hash!==tipos hash.sha256){
          tirar nuevo Error("Tipo de hash no admitido");
    }

    constante digerir=hashFn(mensaje,verdadero);

    // DER es un conjunto más estricto de BER, esto (afortunadamente) funciona:
    constante berEscritor=nuevo Ber.Escritor(); berEscritor.secuencia de inicio();

          berEscritor.secuencia de inicio();
          // OID SHA-256
          berEscritor.escribir OID("2.16.840.1.101.3.4.2.1");
          berEscritor.escribirnull(); berEscritor.endSequence();

    berEscritor.Buffer de escritura(Buffer.de(digerir),ASN1.Cadena de octetos);
    berEscritor.endSequence();

    // T es el nombre de este elemento en RFC 3447
    constante t=berEscritor.buffer;

    si(longitud esperada<(t.longitud+11)){
          tirar nuevo Error('la longitud prevista del mensaje codificado es demasiado corta');
```

```
    }

    constantePD=nuevoUint8Array(longitud esperada-t.longitud-3);
    PD.llenar(0xff);
    afirmar.OK(PD.longitud>=8);

    devolverUint8Array.de(0x00,0x01,...PD,0x00,...t);
}
```

Para simplificar, solo se admite SHA-256. Agregar otras funciones hash es tan simple como agregar los OID correctos.

losfirmarPkcs1v1_5La función pone todas las primitivas juntas para realizar la firma:

```
/**
 * Produce una firma para un mensaje usando el algoritmo RSA como se define
 * en PKCS#1.
 * @parámetro {clave privada}Clave privada RSA, un objeto con
 *                           tres miembros: tamaño (tamaño en bits), n (el módulo) y d (el
 *                           exponente privado), ambos bigInts
 *                           (biblioteca de enteros grandes).
 * @parámetro {hashFn}la función hash requerida por PKCS#1,
 *                    debe tomar un Uint8Array y devolver un Uint8Array
 * @parámetro {tipo hash}Un símbolo que identifica el tipo de función hash pasada.
 *                       Por ahora, solo se admite "SHA-256". Consulte el objeto "hashTypes" para
 *                       conocer los posibles valores.
 * @parámetro {mensaje}Un String o Uint8Array con datos arbitrarios para firmar
 * @devolver{Uint8Array} La firma como Uint8Array
 */
exportarfunciónfirmarPkcs1v1_5(llave privada,hashFn,tipo hash,mensaje){
    constantemensaje codificado=
        emsaPkcs1v1_5(hashFn,tipo hash,llave privada.Talla/8,mensaje);
    constanteintMensaje=os2ip(mensaje codificado); constanteintFirma=rsasp1
    (llave privada,intMensaje); constantefirma=i2osp(intFirma,llave privada.Talla/
    8); devolverfirma;

}
```

Para usar esto para firmar JWT, se necesita un contenedor simple:

```
exportación predeterminadafunciónjwtCodificar(encabezamiento,carga útil,llave privada){
    si(tipo deencabezamiento!=='objeto'||tipo decarga útil!=='objeto'){
        tirar nuevoError('el encabezado y la carga útil deben ser objetos');
    }

    encabezamiento.algoritmo='RS256';

    constanteencabezado=b64(JSON.encadenar(encabezamiento));
    constanteencCarga útil=b64(JSON.encadenar(carga útil));
```

```
    constantejwtDesprotegido=`${encabezadopsencCarga útil};
    constantefirma=b64(
        pkcs1v1_5.señal(llave privada,
                        mensaje=>sha256(mensaje,verdadero),
                        tipos hash.sha256,cadenaToUtf8(jwtDesprotegido)));

    devolver`${jwtDesprotegidopsfirma};
}
```

Esta función es muy similar a lajwtCodificarfunción paraHS256se muestra en la sección HMAC. La verificación es igual de simple:

```
/**
 * Verifica una firma para un mensaje usando el algoritmo RSASSA como se define
 * en PKCS#1.
 * @parámetro {clave pública}Clave privada RSA, un objeto con
 *                           tres miembros: tamaño (tamaño en bits), n (el módulo) y e (el
 *                           exponente público), ambos bigInts
 *                           (biblioteca de enteros grandes).
 * @parámetro {hashFn}la función hash requerida por PKCS#1,
 *                    debe tomar un Uint8Array y devolver un Uint8Array
 * @parámetro {tipo hash}Un símbolo que identifica el tipo de función hash pasada.
 *                       Por ahora, solo se admite "SHA-256". Consulte el objeto "hashTypes" para
 *                       conocer los posibles valores.
 * @parámetro {mensaje}Una cadena o Uint8Array con datos arbitrarios para verificar
 * @parámetro {firma}Un Uint8Array con la firma
 * @devolver{Boolean} true si la firma es válida, false en caso contrario.
 */
exportarfunciónverificarPkcs1v1_5(Llave pública,
                                  hashFn,
                                  tipo hash,
                                  mensaje,
                                  firma){
    si(firma.longitud!==Llave pública.Talla/8){
        tirar nuevoError('longitud de firma no válida');
    }

    constanteintFirma=os2ip(firma);
    constanteintVerification=rsavp1(Llave pública,intFirma);
    constantemensaje de verificación=i2osp(intVerificación,Llave pública.Talla/8);

    constantemensaje codificado=
        emsaPkcs1v1_5(hashFn,tipo hash,Llave pública.Talla/8,mensaje);

    devolveruint8ArrayEquals(mensaje codificado,mensaje de verificación);
}
```

Para usar esto para verificar los JWT, se necesita un contenedor simple:

```
exportarfunciónjwtVerifyAndDecode(jwt,Llave pública){
    si(!esCadena(jwt)){
        tirar nuevoError de tecleado('jwt debe ser una cadena');
    }

    constanteseparar=jwt.separar('.');
    si(separar.longitud!==3){
        tirar nuevoError('Formato JWT no válido');
    }

    constanteencabezamiento=JSON.analizar gramaticalmente(unb64(separar[0]));
    si(encabezamiento.algoritmo!=='RS256'){
        tirar nuevoError(`Algoritmo incorrecto: ${encabezamiento.algoritmo}`);
    }

    constantejwtDesprotegido=cadenaToUtf8(psseparar[0]psseparar[1]});
    constanteválido=verificarPkcs1v1_5(Llave pública,
                                       mensaje=>sha256(mensaje,
                                       verdadero), tipos hash.sha256,
                                       jwtDesprotegido,
                                       base64.descodificar(separar[2]));

    devolver{
        encabezamiento:encabezamiento,
        carga útil:JSON.analizar gramaticalmente(unb64(separar[1
        ])), válido:válido
    };
}
```

Para simplificar, las claves pública y privada deben pasarse como objetos JavaScript con dos números separados: el módulo (norte)y el exponente privado (d)para la clave privada, y el módulo (norte) y el público exponente (mi)para la clave pública. Esto contrasta con el codificado PEM habitual[18] formato. Ver elrs256.jsarchivo para más detalles.

Es posible usar OpenSSL para exportar estos números desde una clave PEM.

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

OpenSSL también se puede utilizar para generar una clave RSA desde cero:

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

Luego puede exportar los números del formato PEM usando el comando que se muestra arriba.

Los números de clave privada incrustados en elclave de prueba.jsel archivo es delclave de prueba.pemen el directorio de ejemplos que acompaña a este manual. La clave pública correspondiente se encuentra en el pubtestkey.pemexpediente.

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

Copie el resultado de ejecutar el ejemplo rs256.js[19] en el área de JWT en JWT.io[20]. Luego copie el contenido depubtestkey.pemal área de clave pública en la misma página y el JWT se validará con éxito.

7.2.2.4 PS256: RSASSA-PSS usando SHA-256 y MGF1 con SHA-256

RSASSA-PSS es otro esquema de firma con apéndice basado en RSA. "PSS" significa Probabilistic Signature Scheme, en contraste con el habitual *determinista* Acercarse. Este esquema hace uso de un generador de números aleatorios criptográficamente seguro. Si no se dispone de un RNG seguro, las operaciones de firma y verificación resultantes proporcionan un nivel de seguridad comparable a los enfoques deterministas. De esta manera, RSASSA-PSS da como resultado una mejora neta sobre las firmas PKCS v1.5 para los mejores escenarios. Sin embargo, en la naturaleza, los esquemas de PSS y PKCS v1.5 permanecen intactos.

RSASSA-PSS se define en el estándar de criptografía de clave pública n.º 1 (PKCS n.º 1)[21] y no está disponible en versiones anteriores del estándar.

7.2.2.4.1 Algoritmo

Para producir una firma:

1. Aplicar el **EMSA-PSS-ENCODE** primitiva al mensaje. La primitiva toma un parámetro que debe ser el número de bits del módulo de la clave menos 1. El resultado es el **mensaje codificado**.
2. Aplicar el **OS2IP** primitiva al mensaje codificado. El resultado es el **representante de mensaje entero**. OS2IP es el acrónimo de "Octet-String to Integer Primitive".
3. Aplicar el **RSASP1** primitiva al representante del mensaje entero utilizando la clave privada. El resultado es el **representante de firma entera**.
4. Aplicar el **I2OSP** primitiva para convertir el representante de la firma entera en una matriz de octetos (el **firma**). I2OSP es el acrónimo de "Integer to Octet-String Primitive".

Una posible implementación en JavaScript, dadas las primitivas mencionadas anteriormente, podría verse así:

```
exportar función firmarPss(llave privada,hashFn,tipo hash,mensaje){
    si(tipo hash!==tipos hash.sha256){
        tirar nuevo Error('tipo de hash no compatible');
    }

    constante mensaje codificado=emsaPssEncode(hashFn,
                                                tipo hash,
                                                mgf1.unir(nulo,hashFn), 256
                                                /8,//tamaño de hachís llave
                                                privada.Talla-1, mensaje);

    constante intMensaje=os2ip(mensaje codificado);
```

[19] https://github.com/auth0/jwt-handbook-samples/blob/master/rs256.js
[20] https://jwt.io
[21] https://www.ietf.org/rfc/rfc3447.txt

```
    constante intFirma=rsasp1(llave privada,intMensaje); constante firma
    =i2osp(intFirma,llave privada.Talla/8); devolver firma;

}
```

Para verificar una firma:

1. Aplicar el **OS2IP** primitiva a la firma (una matriz de octetos). Este es el **representante de firma entera**.
2. Aplicar el **RSAVP1** primitiva al resultado anterior. Esta primitiva también toma la clave pública como entrada. Este es el **representante de mensaje entero**.
3. Aplicar el **I2OSP** primitiva al resultado anterior. Esta primitiva toma un tamaño esperado como entrada. Este tamaño debe coincidir con la longitud del módulo de la clave en número de octetos. El resultado es el **mensaje codificado**.
4. Aplicar el **EMSA-PSS-VERIFICAR** primitiva al mensaje que se va a verificar y el resultado del paso anterior. Esta primitiva muestra si la firma es válida o no. Esta primitiva hace uso de una función hash (generalmente una función hash de la familia SHA como SHA-256). La primitiva toma un parámetro que debe ser el número de bits del módulo de la clave menos 1.

```
exportar función verificarPss(Llave pública,hashFn,tipo hash,mensaje,firma){
    si(firma.longitud!==Llave pública.Talla/8){
        tirar nuevo Error('longitud de firma no válida');
    }

    constante intFirma=os2ip(firma);
    constante intVerification=rsavp1(Llave pública,intFirma);
    constante mensaje de verificación=
        i2osp(intVerificación,Matemáticas.hacer techo( (Llave pública.Talla-1) /8));

    devolver emsaPssVerificar(hashFn,
                              tipo hash,
                              mgf1.unir(nulo,hashFn), 256
                              /8,
                              Llave pública.Talla-1,
                              mensaje,
                              mensaje de verificación);
}
```

7.2.2.4.1.1 MGF1: la función de generación de máscaras

Las funciones de generación de máscaras toman entradas de cualquier longitud y producen salidas de longitud variable. Al igual que las funciones hash, son deterministas: producen la misma salida para la misma entrada. Sin embargo, a diferencia de las funciones hash, la longitud de la salida es variable. La función de generación de máscaras 1 (MGF1)el algoritmo se define en el estándar de criptografía de clave pública n.º 1 (PKCS n.º 1)[22].

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

MGF1toma un valor inicial y la longitud prevista de la salida como entradas. La longitud máxima de la salida se define como 2^{32}.MGF1utiliza internamente una función hash configurable.PS256especifica esta función hash comoSHA-256.

1. Si la longitud prevista es mayor que 2^{32}, deténgase con el error "máscara demasiado larga".
2. Iterar desde 0 hasta el techo de la longitud deseada dividida la longitud de la salida de la función hash menos 1 (techo (longitud prevista / longitud hash) - 1)haciendo las siguientes operaciones:

    1. Dejac = i2osp(contador, 4)dóndeencimeraes el valor actual del contador de iteraciones.
    2. Dejat = t.concat(hash(seed.concat(c)))dóndetse conserva entre iteraciones, picadilloes la función hash seleccionada (SHA-256)ysemillaes el valor semilla de entrada.
3. Emita los octetos de longitud previstos más a la izquierda del último valor detcomo resultado de la función.

7.2.2.4.1.2 Primitiva EMSA-PSS-CODIFICACIÓN

La primitiva toma dos elementos:

- El mensaje a codificar como una secuencia de octetos.
- La longitud máxima prevista del resultado en bits.

Esta primitiva puede ser parametrizada por los siguientes elementos:

- Una función hash. En el caso dePS256esteSHA-256.
- Una función de generación de máscaras. En el caso dePS256esto esMGF1.
- Una longitud prevista para la sal utilizada internamente.

Todos estos parámetros están especificados porPS256,por lo que no son configurables y para efectos de esta descripción se consideran constantes.

Tenga en cuenta que la longitud prevista utilizada como entrada se expresa en bits. Para los siguientes ejemplos, considere:

```
constanteLongitud prevista=Matemáticas.hacer techo(bits de longitud prevista /8);
```

1. Si la entrada es mayor que la longitud máxima de la función hash, deténgase. De lo contrario, aplique la función hash al mensaje.

```
constantehash1=sha256(mensaje de entrada);
```

2. Si la longitud prevista del mensaje es menor que la longitud del hash más la longitud del salt más 2, deténgase con un error.

```
si(longitud prevista<(hash1.longitud+intencionadaSaltLength+2)){
      tirar nuevoError('Error de codificación');
}
```

3. Genere una secuencia de octetos aleatoria de la longitud de la sal.
4. Concatenar ocho octetos de valor cero con el hash del mensaje y la sal.

```
constantemetro=[0,0,0,0,0,0,0,0,...hash1,...sal];
```

5. Aplicar la función hash al resultado del paso anterior.

```
constantehash2=sha256(metro);
```

6. Genere una secuencia de octetos de valor cero de longitud: la longitud máxima prevista del resultado menos la longitud de sal menos la longitud de hash menos 2.

```
constantePD=nuevoFormación(longitud prevista-intencionadaSaltLength-2).llenar(0);
```

7. Concatenar el resultado del paso anterior con el octeto 0x01 y la sal.

```
constantebase de datos=[...PD,0x01,...sal];
```

8. Aplique la función de generación de máscara al resultado del paso 5 y establezca la longitud prevista de esta función como la longitud del resultado del paso 7 (la función de generación de máscara acepta un parámetro de longitud prevista).

```
constanteMáscara db=mgf1(hash2,base de datos.longitud);
```

9. Calcule el resultado de aplicar la operación XOR a los resultados de los pasos 7 y 8.

```
constanteenmascaradoDb=base de datos.mapa((valor,índice)=> {
    devolvervalor^dbMask[índice];
});
```

10. Si la longitud del resultado de la operación anterior no es un múltiplo de 8, encuentre la diferencia en el número de bits para convertirlo en un múltiplo de 8 restando bits, luego establezca este número de bits en 0 comenzando desde la izquierda.

```
constanteceroBits=8*Longitud prevista-Bits de longitud prevista;
constantezeroBitsMáscara=0xFF>>>ceroBits; enmascaradoDb[0]&=
zeroBitsMáscara;
```

11. Concatenar el resultado del paso anterior con el resultado del paso 5 y el octeto 0xBC. Este es el resultado.

```
constanteresultado=[...enmascaradoDb,...hash2,0xBC];
```

7.2.2.4.1.3 Primitiva EMSA-PSS-VERIFICAR

La primitiva toma tres elementos:

- El mensaje a verificar.
- La firma como un mensaje entero codificado.
- La longitud máxima prevista del mensaje entero codificado.

Esta primitiva puede ser parametrizada por los siguientes elementos:

- Una función hash. En el caso dePS256esteSHA-256.
- Una función de generación de máscaras. En el caso dePS256esto esMGF1.
- Una longitud prevista para la sal utilizada internamente.

Todos estos parámetros están especificados porPS256,por lo que no son configurables y para efectos de esta descripción se consideran constantes.

Tenga en cuenta que la longitud prevista utilizada como entrada se expresa en bits. Para los siguientes ejemplos, considere:

```
constantelongitud esperada=Matemáticas.hacer techo(bits de longitud esperada /8);
```

1. Haga un hash del mensaje que se va a verificar usando la función hash seleccionada.

```
constanteresumen1=hashFn(mensaje,verdadero);
```

2. Si la longitud esperada es menor que la longitud hash más la longitud salt más 2, se considera que la firma no es válida.

```
si(longitud esperada<(resumen1.longitud+salLongitud+2)){
    falso retorno;
}
```

3. Comprobar que el último byte del mensaje codificado de la firma tiene el valor de0xBC

```
si(mensaje de verificación[mensaje de verificación.longitud-1]!==0xBC){
    falso retorno;
}
```

4. Divida el mensaje codificado en dos elementos. El primer elemento tiene una longitud de longitudesperada - hashLength - 1. El segundo elemento comienza al final del primero y tiene una longitud dehashLength.

```
constantelongitud enmascarada=longitud esperada-resumen1.longitud-1; constante
enmascarado=mensaje de verificación.subarreglo(0,longitud enmascarada); constante
resumen2=mensaje de verificación.subarreglo(longitud enmascarada,
                                                    longitud enmascarada+resumen1.longitud);
```

5. Compruebe que el extremo izquierdo8 * longitud esperada - bits de longitud esperada (la longitud esperada en bits menos la longitud solicitada en bits) bits deenmascaradoson 0

```
constanteceroBits=8*longitud esperada-bits de longitud esperada;
constantezeroBitsMáscara=0xFF>>>ceroBits; si((enmascarado[0]&(~
zeroBitsMask))!==0){
    falso retorno;
}
```

6. Pase el segundo elemento extraído del paso 4 (el resumen) a la función MGF seleccionada. Solicite que el resultado tenga una longitud delongitudesperada - hashLength - 1.

```
constanteMáscara db=mgf(longitud enmascarada,resumen2);
```

7. Para cada byte del primer elemento extraído en el paso 4 (enmascarado)aplicar la función XOR usando el byte correspondiente del elemento calculado en el último paso (Máscara db).

```
constantebase de datos=nuevoUint8Array(enmascarado.
longitud); por(dejari=0;i<base de datos.longitud; ++i){
    db[yo]=enmascarado[yo]^Máscara db[i];
}
```

8. Establecer el más a la izquierda8 * longitud esperada - bits de longitud esperadabits desde el primer byte en el elemento calculado en el último paso a 0.

```
constanteceroBits=8*longitud esperada-bits de longitud esperada;
constantezeroBitsMáscara=0xFF>>>ceroBits; base de datos[0]&=
zeroBitsMáscara;
```

9. Compruebe que el extremo izquierdoExpectedLength - hashLength - saltLength - 2bytes del elemento calculado en el último paso son 0. También verifique que el primer elemento después del grupo de ceros sea0x01.

```
constantezeroCheckLength=longitud esperada-(resumen1.longitud+salLongitud+2); si(!base de
datos.subarreglo(0,longitud de verificación cero).cada(v=>v===0)||
    db[longitud de verificación cero]!==0x01){
    falso retorno;
}
```

10. Sacar la sal de la últimasalLongitudoctetos del elemento calculado en el último paso (db).

```
constantesal=base de datos.subarreglo(base de datos.longitud-longitud de sal);
```

11. Calcule un nuevo mensaje codificado concatenando octetos octetos de valor cero, el hash calculado en el paso 1 y la sal extraída en el último paso.

```
constantemetro=Uint8Array.de(0,0,0,0,0,0,0,0,...resumen1,...sal);
```

12. Calcule el hash del elemento calculado en el último paso.

```
constanteresumen esperado=hashFn(metro,verdadero);
```

13. Compare el elemento calculado en el último paso con el segundo elemento extraído en el paso 4. Si coinciden, la firma es válida; de lo contrario, no lo es.

```
devolveruint8ArrayEquals(resumen2,resumen esperado);
```

7.2.2.4.2 Código de muestra

Como era de esperar de una variante de RSASSA, la mayor parte del código requerido para este algoritmo ya está presente en la implementación de RS256. Las únicas diferencias son las adiciones de losEMSA-PSS-CODIFICAR, EMSA-PSS-VERIFICAR,yMGF1primitivos.

```
exportarfunciónmgf1(hashFn,longitud esperada,semilla){
    si(longitud esperada>Matemáticas.pow(2,32)){
        tirar nuevoError('máscara demasiado larga');
    }

    constantetamaño hash=hashFn(Uint8Array.de(0),verdadero).byteLength;
    constantecontar=Matemáticas.hacer techo(longitud esperada/tamaño hash);
    constanteresultado=nuevoUint8Array(tamaño hash*contar); por(dejari=0;i<
    contar; ++i){
        constanteC=i2osp(Empezando(i),4);
```

```
            constantevalor=hashFn(Uint8Array.de(...semilla,...C),verdadero); resultado
            .establecer(valor,i*tamaño hash);
        }
        devolverresultado.subarreglo(0,longitud esperada);
}
exportarfunciónemsaPssEncode(hashFn,
                                        tipo hash,
                                        mgf,
                                        salLongitud,
                                        bits de longitud esperada,
                                        mensaje){
        constantelongitud esperada=Matemáticas.hacer techo(bits de longitud esperada /8);

        constanteresumen1=hashFn(mensaje,verdadero);
        si(longitud esperada<(resumen1.longitud+salLongitud+2)){
            tirar nuevoError('error de codificación');
        }

        constantesal=cripto.bytes aleatorios(salLongitud);
        constantemetro=Uint8Array.de(...(nuevoUint8Array(8)),
                                        . . . resumen1,
                                        . . . sal);
        constanteresumen2=hashFn(metro,verdadero);
        constantePD=nuevoUint8Array(longitud esperada-salLongitud-resumen2.longitud-2); constante
        base de datos=Uint8Array.de(...PD,0x01,...sal); constanteMáscara db=mgf(base de datos.longitud,
        resumen2);
        constanteenmascarado=base de datos.mapa((valor,índice)=>valor^dbMask[índice]);

        constanteceroBits=8*longitud esperada-bits de longitud esperada;
        constantezeroBitsMáscara=0xFF>>>ceroBits; enmascarado[0]&=
        zeroBitsMáscara;

        devolverUint8Array.de(...enmascarado,...resumen2,0xbc);
}
exportarfunciónemsaPssVerificar(hashFn,
                                        tipo hash,
                                        mgf,
                                        salLongitud,
                                        bits de longitud esperada,
                                        mensaje,
                                        mensaje de verificación){
        constantelongitud esperada=Matemáticas.hacer techo(bits de longitud esperada /8);

        constanteresumen1=hashFn(mensaje,verdadero);
        si(longitud esperada<(resumen1.longitud+salLongitud+2)){
```

```
        falso retorno;
    }

    si(mensaje de verificación.longitud===0){
        falso retorno;
    }

    si(mensaje de verificación[mensaje de verificación.longitud-1]!==0xBC){
        falso retorno;
    }

    constantelongitud enmascarada=longitud esperada-resumen1.longitud-1; constante
    enmascarado=mensaje de verificación.subarreglo(0,longitud enmascarada); constante
    resumen2=mensaje de verificación.subarreglo(longitud enmascarada,
                                                longitud enmascarada+resumen1.longitud);

    constanteceroBits=8*longitud esperada-bits de longitud esperada;
    constantezeroBitsMáscara=0xFF>>>ceroBits; si((enmascarado[0]&(~
    zeroBitsMask))!==0){
        falso retorno;
    }

    constanteMáscara db=mgf(longitud enmascarada,resumen2);
    constantebase de datos=enmascarado.mapa((valor,índice)=>valor^dbMask[índice]); base de
    datos[0]&=zeroBitsMáscara;

    constantezeroCheckLength=longitud esperada-(resumen1.longitud+salLongitud+2); si(!base de
    datos.subarreglo(0,longitud de verificación cero).cada(v=>v===0)||
        db[longitud de verificación cero]!==0x01){
        falso retorno;
    }

    constantesal=base de datos.subarreglo(base de datos.longitud-longitud de sal);
    constantemetro=Uint8Array.de(0,0,0,0,0,0,0,0,...resumen1,...sal); constante
    resumen esperado=hashFn(metro,verdadero);

    devolveruint8ArrayEquals(resumen2,resumen esperado);
}
```

el ejemplo completo[23]está disponible en los archivosps256.js, rsassa.js,ypkcs.js.Los números de clave privada incrustados en elclave de prueba.jsel archivo es delclave de prueba.pemen el directorio de ejemplos que acompaña a este manual. La clave pública correspondiente se encuentra en elpubtestkey.pemexpediente. Para obtener ayuda en la creación de claves, consulte elRS256ejemplo.

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

7.3 Actualizaciones futuras

La especificación JWA tiene muchos más algoritmos. En versiones futuras de este manual repasaremos los algoritmos restantes.