Estudio de soluciones AAI, definición de escenarios de prácticas y análisis de seguridad

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Felipe Sánchez González
Tutor/es:
Gabriel López Millán

Facultad Informática Universidad Murcia

4 de Junio de 2023

Estudio de soluciones AAI, definición de escenarios de prácticas y análisis de seguridad

Estudio de soluciones para Infraestructuras de Autenticación y Autorización (AAI), como SAML, OAuth2.0 o OpenID-connect, el despliegue de escenarios de prueba y el análisis de seguridad de los protocolos en los que se basan.

Autor
Felipe Sánchez González

Tutor/es
Gabriel López Millán
Departamento de Ingeniería de la Información y las Comunicaciones

Facultad de Informática

Grado en Ingeniería Informática

UNIVERSIDAD DE MURCIA

Murcia, 4 de Junio de 2023

Agradecimientos

Para comenzar esta ronda de agradecimientos, me gustaría agradecer a la comunidad de alumnos de este grado de Ingeniería Informática por tener siempre un afán de colaboración y cooperación. Me siento orgulloso de haber pertenecido a este grupo que, a diferencia de otros grados, no tiene problemas al compartir sus apuntes, ideas o conocimientos con aquellos que, por alguna razón, carecen de ellos.

También agradecer al profesorado por compartir sus conocimientos y experiencias de manera excepcional y hacerme así avanzar a lo largo del grado.

Concretamente, me gustaría agradecer al profesor *Gabriel López Millán* por despertar en mí una curiosidad cada vez mayor en el mundo de las redes, seguridad y servicios mediante las asignaturas que imparte. Estoy seguro de que esto no hubiese ocurrido sin la pasión y dedicación mostrada en cada clase ayudándose de sus colaboradores *Alice* y *Bob*. También por su propuesta de este TFG y la confianza depositada en mí, además de su posterior paciencia y dedicación inmensa a la hora de exponerme sus ideas y explicaciones. Personalmente, será un honor volver a ser su alumno si acabo entrando en el nuevo Máster de Ciberseguridad.

Por último, me gustaría expresar mi gratitud a aquellos que comenzaron como desconocidos, pero que se convirtieron en grandes amigos a lo largo de este camino. Quiero agradecer especialmente a *Fran*, quien me ha contagiado su actitud de aprendizaje constante; a *Gonzalo*, por su paciencia y dedicación inquebrantable en cada uno de nuestros proyectos; y a *Juanjo*, por transmitirme su meticulosidad y brindarme siempre su ayuda en el aprendizaje de conceptos complejos. Sin su constante e infinita colaboración, tanto académica como personal, habría sido poco probable que pudiese encontrarme en condiciones de completar el grado en cuatro años. Esta experiencia universitaria ha sido mucho más gratificante gracias a los momentos, tanto buenos como difíciles, que hemos compartido juntos.

A Silvia, la fuente de luz que ilumina cada día de mi vida. Sin su apoyo incondicional, este trabajo no habría sido posible.

A Blasa María, Consuelo, Juan Antonio y María Isabel, por su invaluable ayuda y por siempre creer en mí, acompañándome en cada paso de este camino hacia donde estoy hoy.

¿Qué importa como me llame? Se nos conoce por nuestros actos.

Batman (Batman Begins).

Declaración firmada sobre originalidad del trabajo

D./Dña. **Felipe Sánchez González**, con DNI [redacted], estudiante de la titulación de **Grado en Ingeniería Informática** de la Universidad de Murcia y autor del TF titulado "**Estudio de soluciones AAI, definición de escenarios de prácticas y análisis de seguridad**".

De acuerdo con el Reglamento por el que se regulan los Trabajos Fin de Grado y de Fin de Máster en la Universidad de Murcia (aprobado C. de Gob. 30-04-2015, modificado 22-04-2016 y 28-09-2018), así como la normativa interna para la oferta, asignación, elaboración y defensa delos Trabajos Fin de Grado y Fin de Máster de las titulaciones impartidas en la Facultad de Informática de la Universidad de Murcia (aprobada en Junta de Facultad 27-11-2015)

DECLARO:

Que el Trabajo Fin de Grado presentado para su evaluación es original y de elaboración personal. Todas las fuentes utilizadas han sido debidamente citadas. Así mismo, declara que no incumple ningún contrato de confidencialidad, ni viola ningún derecho de propiedad intelectual e industrial

Murcia, a 4 de Junio de 2023

[redacted]

Fdo.: Felipe Sánchez González
Autor del TFG

Extended Abstract

The use of digital services has experienced a massive increment in recent years. Nowadays, any citizen has an account with big service providers like Google, Amazon, Microsoft, etc. This has led to the continuous generation of private and identity data that is being stored and managed by the services people and businesses use daily. From profiles on social networks to banking accounts, digital identity has become even more prevalent and valuable.

Before this success of online services, the use cases for identity management were as simple as an login form with a username and a password. However, as years passed, other cases began to emerge, such as SSO (Single Sign On), federated authentication, login from smartphones and delegated authentication, among others.

In this context, protocols for AAI (Authentication and Authorization Infrastructure) began to be developed to meet those use case demands. To fulfill these needs, protocols such as SAML (Security Assertion Markup Language), OAuth 2.0, and OIDC (OpenID Connect) were created.

One one hand, this AAI are vital to some service provider because, concepts like federated authentication, which is an agreement between two entities with different domains where they establish a trusted relationship for the exchange of identity information, allow them to partially alleviate the cost associated with storing user identities and some legal responsibilities associated with it. As a result, these service providers can focus their efforts on delivering quality services to all users.

On the other hand, identity provider organizations responsible for the storage and registration of user credentials and identities can focus on enhancing the security level for the proper functioning of their system. In addition to this, these identity providers also gain a dominant position on the Internet, becoming significant companies that other organizations turn to for establishing identity federations.

On the side of end-users, these concepts help them avoid the burden of having to remember different credentials for each service they want to use. This results in the user potentially only having to remember one password to authenticate with the identity provider, promoting the use of more secure access credentials. Another benefit for the user is the convenience provided by SSO. This concept states that a user, for various accesses to one or different services, only needs to enter their login credentials once.

This saves time for the user, which can create a positive experience and become a key factor in deciding whether to use one service over another.

However, a clear disadvantage of these centralized authorization and authentication systems is precisely their centralization. If an attacker manages to compromise the security of an identity provider or intercept and decrypt certain messages related to user identity, it can cause significant harm to the confidentiality of information associated with user identities. Considering that these identity providers (such as Google, Facebook, Twitter, etc.) often have massive user bases, an error in any of these protocols can compromise the identities of billions of users.

With this context, this Degree's thesis presents scenarios developed with Docker for testing and security analysis of some of the most notable security considerations of the SAML, OAuth 2.0, and OIDC protocols. These scenarios can serve as a ground for establishing teaching practices that facilitate the analysis and configuration of these solutions at an academic level.

To commence this thesis, an state-of-art research was performed to introduce the three technologies in question.

The first of these protocols, SAML, is one of the most widely adopted solutions for SSO and identity federation globally. Its main advantage is its support for both authentication and authorization, and the use of signed XML payloads, called Assertions, to establish an interoperable protocol for identity federations and SSO. This protocol introduces two actors apart from the user: the Identity Provider (IdP), responsible for registering and authenticating users, and the Service Provider (SP), which offers a service and relies on the IdP to authenticate users. To achieve these objectives, SAML supports two main flows, depending on where the flow is initiated: SP-initiated flow or IdP-initiated flow. While both flows are widely adopted, the explanation of how this protocol work is focused on the SP-initiated flow as it is more commonly used. With this flow in mind, the main messages of the flow and the main components of an Assertion were explained to provide a general understanding of how SAML achieve this secure interdomain exchange of identity information.

The second protocol, OAuth 2.0, is a widely implemented protocol dedicated to delegated authorization, which involves granting permissions to a third-party application to perform certain actions on a user's protected resources. It is worth noting that OAuth 2.0 uses JSON for information exchange, making it lighter than SAML. However, despite many services using OAuth 2.0 for years to authenticate, it is not designed to provide authentication functions. It is primarily designed for authorization purposes only. Furthermore, OAuth 2.0 introduces three actors apart from the user: the OAuth client, which is the third-party app that want to make use of user's resources; the Authorization Server (AS), which is in charge of providing the client with tokens when the user have authorized it; and the Resource Server (RS) that serves the user's

resources. In addition, OAuth 2.0 can be used with different flows depending on the type of client that it is going to be implemented. These flows include the Authorization Code Grant, primarily used by clients that can act as intermediaries between the user and the AS, typically web servers; Implicit Grant, suitable for clients implemented within a browser using JavaScript; Client Credentials Grant, utilized by clients that only require access to their own resources; and Resource Owner Grant, which is less commonly used due to the need for user credentials to obtain the token. Since Authorization Code Grant is the most common flow, we use it as basis for providing a general description of protocol messages and tokens such as the Authorization Code and Access Code. This will help us explain how the OAuth 2.0 protocol enables the use of redirections and tokens to provide a natural way to allow and delegate authorization to a third-party application access certain protected resources on behalf of a user.

The last protocol, OIDC, addresses the authentication limitation of OAuth 2.0. OIDC is a protocol that builds on top of OAuth 2.0 to provide an authentication layer. This protocol is necessary because with OAuth 2.0 a user authenticates in way that is unknown to the protocol. OAuth 2.0 alone does not provide a way to identify the authenticated user, understand how the authentication was performed, or determine the validity period of the authentication. In its state of art section, we explain how the AS, now referred to as OpenID Provider (OP), can generate an ID Token. This ID Token serves the purpose of transporting information about the authentication process to OAuth clients, which are now referred to as Relaying Parties (RP). In addition, we explain how RPs can verify the authenticity of the ID Token using the digital signatures present within it.

With these three protocols in mind, the first step taken was to build and deploy two environments using Docker containers to create a platform for testing and analyzing the studied protocols.

We chose to utilize Docker because it not only provides isolation but also offers enhanced portability. Docker allows us to encapsulate the necessary components and dependencies into a few files, making it easier to transfer the entire environment to another machine. Unlike other alternatives like VirtualBox, there is no need to move the entire system. Additionally, the deployment process with Docker is much more convenient, allowing a fast deployment and configuration of the testing environments.

The first environment was built for SAML. In this environment, the simpleSAMLphp application and Apache were used to deploy three SAML components within Docker containers. Firstly, an IdP was configured to provide login capabilities. Test user credentials were added to test the Assertions generated once users' logins were completed. Secondly, a first SP was developed to provide a test application that utilizes the configured IdP and its identities to consume the generated Assertions, ensuring the correct authentication of the test users. Thirdly, a second SP, which is almost identical to the first one, was developed to test the SSO across the different SPs.

In addition to the SAML components, a user container was defined to provide easy access to this environment. This was achieved by including a noVNC container with Ubuntu Desktop and network analysis tools within the defined Docker internal network, where the other elements communicate. With this container, a tester can access the desktop in their chosen browser using noVNC. Once the tester is working within this container, they can perform test actions within the SAML messages and simultaneously capture the traffic for later use or real-time analysis using Wireshark or SAML Tracer extension.

In this case, we have use this environment to record logs and traces in order to perform a security analysis of SAML. In this analysis, the SAML messages and their main characteristics are described to inform about how real life implementations materialize the directives given by the technical rules of the protocol. In addition, small proof of concepts that could abuse certain protocol flaws are shown in order to inform about these possibilities and how to avoid them.

The second environment was specifically built for OAuth 2.0 and OIDC. To create this environment, we utilized the oauthlib and Flask libraries to establish an OAuth Client that can also function as an RP.

The OAuth Client and RP components were encapsulated within a Docker container, exposing port 80 to access the web application. Within this web application, various functionalities were provided to test the redirection and authorization processes. Additionally, the application was configured to make use of the Google Calendar API, allowing users to view their Google Calendar events list within this OAuth client.

To achieve this, a registration process for the OAuth client was performed to obtain a Client ID and Client Secret. These credentials were necessary to establish the OAuth flows correctly to authenticate the OAuth client with the Google AS.

Furthermore, an OIDC button was incorporated into the web page. As this OAuth client is now an OIDC RP, data related with the authentication process is now sent by the AS, now called OP. This data is carried in an ID Token that allows the RP to query some details about this authentication, such as the user identifier, email, and expiration time, etc. Additionally, the RP can verify the authenticity of this ID Token by validating the signature contained within the token. When the mentioned OIDC button is clicked, the user is redirected to an endpoint that displays the unencoded contents of all ID Token parts and shows the user if this token is verified or not.

To facilitate the testing of this environment, a user container was established to allow a user to make some testing. For this purpose, a noVNC container with Ubuntu Desktop was once again deployed within the Docker internal network of the environment. With this system a user could use this OAuth client and test the different characteristics of this environment. At the same time, this user can generate logs and traces that could be used for later analysis.

With this environment up and running, some traces were generated and analyzed to learn about the precise parameters and flows used by the OAuth 2.0 and OIDC protocols in real implementations as the one built. In addition to that, the major security implications about the OAuth 2.0 protocol were researched in order to perform some basic proof of concepts to test some of this security concepts and to check patches and fixes that could be applied in order to solve those security risks.

In summary, this thesis focuses on the research and analysis of three widely used protocols in the context of AAI: SAML, OAuth 2.0, and OIDC. Throughout this research, some state of the art was written about how these protocols work with the description of the general flows and components that they use. After this, a design process was made in order to build two Docker-based benchmark environments to simulate real-world scenarios and evaluate the protocols' functionalities and configurations. Finally, an analysis of the traces and logs generated by the environments was made to provide a better understanding of the parameters that a real implementation can produce. Furthermore, the benchmark environments has allowed us to test some possible vulnerabilities that this protocols have and some of the patches that can be used to solve them.

Índice general

Índice de figuras

Índice de Códigos

1 Introducción

En los últimos años, el acceso a servicios digitales a través de Internet se ha visto incrementado de forma masiva. Hoy en día, prácticamente cualquier ciudadano dispone de una cuenta en alguno de los grandes proveedores de servicios como puede ser Google, Amazon, Microsoft, etc. Esto ha desembocado en la generación continua de una gran cantidad de datos personales y de identidad que son almacenados y gestionados por muchos de los servicios que las personas y las empresas usan diariamente. Desde perfiles en redes sociales hasta cuentas bancarias en línea, la identidad digital se ha vuelto cada vez más presente y valiosa.

Antes de este éxito de los servicios en línea, los casos de uso para la gestión de la identidad pasaban por algo tan simple como un formulario de acceso con nombre de usuario y contraseña. Sin embargo, conforme pasaron los años se empezaron a formular otros casos como el SSO, autenticación federada, el acceso desde móviles con acceso a internet y autenticación delegada, entre otros.

En este contexto, se empiezan a desarrollar protocolos para construir Infraestructuras de Autenticación y Autorización (IAA) que cumpliesen con las demandas de todos estos casos de uso. Para cubrir estas necesidades nacen protocolos como SAML, OAuth 2.0 y OIDC.

Por un lado, estas IAA son vitales para los proveedores de servicios porque, conceptos como la autenticación federada, que es una acuerdo entre dos entidades con un dominio diferente en el que establecen una relación de confianza para el intercambio de información de identidades, les permiten poder desprenderse parcialmente del coste derivado de tener que realizar la custodia de las identidades de los usuarios y algunas responsabilidades legales asociadas a ello. Como consecuencia, estos proveedores de servicios pueden centrar sus esfuerzos en proporcionar servicios de calidad para todos los usuarios.

Por otro lado, las organizaciones proveedoras de identidad encargadas de la custodia y el registro de las credenciales e identidades de los usuarios pueden centrarse en aumentar el nivel de seguridad para el correcto funcionamiento de su sistema. Además de esto, estos proveedores de identidad también consiguen tener una posición dominante en Internet, convirtiéndose así en empresas importantes a las que otras organizaciones recurren para establecer federaciones de identidad.

En el lado de los usuarios finales, estos conceptos les ayudan a no tener que cargar con la fatiga derivada de tener que recordar credenciales diferentes para cada servicio que quieren usar. Esto deriva en que el usuario tan solo tiene que recordar potencialmente una contraseña para autenticarse con el proveedor de identidad, favoreciendo esto a que el usuario pueda utilizar credenciales de acceso más seguras. Otro de los beneficios para el usuario es la propia conveniencia que proporciona el SSO. Este concepto establece que un usuario, para distintos accesos a uno o diferentes servicios, tan solo tiene que introducir sus credenciales de acceso una sola vez. Esto resulta en un ahorro de tiempo para el usuario que puede generar experiencia positiva y convertirse en elemento clave a la hora de decidirse a utilizar un servicio u otro.

Sin embargo, una clara desventaja de estos sistemas centralizados de autorización y autenticación es, precisamente, esa propia centralización. Si un atacante consigue comprometer la seguridad un proveedor de identidad o interceptar y desencriptar ciertos mensajes relacionados con la identidad del usuario, puede suponer un gran perjuicio para la confidencialidad de información asociada a las identidades de los usuarios. Teniendo en cuenta que estos proveedores de identidad (Google, Facebook, Twitter, etc.) suelen tener cantidades masivas de usuarios, un error en alguno de los protocolos asociados a estos sistemas puede suponer catástrofes a nivel mundial pudiendo verse comprometidas las identidades de miles de millones de usuarios.

Con este contexto, se presenta este Trabajo de Fin de Grado (TFG) en el que se han desarrollado escenarios para la realización de pruebas y análisis de seguridad de algunas de las consideraciones de seguridad más destacables de los protocolos SAML, OAuth 2.0 y OIDC.

Antes de nada, se ha realizado un estudio de estos protocolos y se ha escrito un apartado de estado del arte en el cual se han descrito los objetivos que tratan de cumplir estos protocolos. Además, se han destacado los diferentes flujos, mensajes y parámetros que hacen que estos protocolos cumplan con su cometido.

El primero de estos protocolos, SAML, es una de las soluciones para SSO y federación de identidad más ampliamente adoptadas a nivel global. Su principal ventaja es su soporte tanto de autenticación como autorización y el uso de cargas XML firmadas, llamadas *Assertions*, para establecer un protocolo interoperable con el que poder establecer federaciones de identidad y SSO. Además de explicar la función de los actores que introduce: IdP y SP, se ha elegido su flujo más utilizado (SP-initiated flow) para enumerar los distintos mensajes que se intercambian y los campos más destacados que forman los *Assertions*.

Para este protocolo se ha construido un escenario en el que se han configurado un Identity Provider (IdP) y dos Service Provider (SP). Para el IdP se han generado los certificados autofirmados necesarios para la firma de *Assetions* y se han añadido usuarios con los que poder posteriormente realizar las pruebas de autenticación. Para

los SP se ha configurado una página principal muy simple en la que se pudiese realizar la autenticación mediante el IdP. Se construyen dos SP en vez de uno es para poder comprobar el funcionamiento del SSO entre dos servicios diferentes que comparten el mismo IdP. Además de estos actores de SAML, se introduce un nodo cliente para realizar las distintas pruebas e interactuar con el escenario. Posteriormente al diseño del escenario, se ha realizado un análisis de los mensajes SAML que puede generar este escenario y sus particularidades. Tras esto, se ha llevado a cabo un análisis de las principales consideraciones de seguridad de SAML con pequeñas pruebas de concepto haciendo uso del cliente introducido en el escenario.

El segundo, OAuth 2.0, es un protocolo ampliamente implementado dedicado a la autorización delegada, es decir, la cesión de permisos a una aplicación de terceros para realizar ciertas acciones sobre los recursos protegidos de un usuario. Algo a destacar es que hace uso de JSON para el intercambio de información lo que lo hace más ligero que SAML. Sin embargo, pese a que muchos servicios han hecho esto durante años, OAuth 2.0 no está diseñado para proporcionar funciones de autenticación, solo está diseñado para la autorización. Para explicar el protocolo, se han introducido los actores que este define: cliente OAuth, Authorization Server (AS) y Resource Server (RS). Tras esto, se ha dado una pequeña explicacion de los diferentes flujos que pueden ser utilizados para OAuth 2.0 segun el tipo de cliente: Authorization Code Grant, Implicit Grant, Client Credentials Grant y Resource Owner Credentials Grant. Sin embargo, se ha utilizado el Authorization Code Grant para explicar como funciona el protocolo al ser el flujo mas utilizado.

El último protocolo, OIDC, sirve para solventar esa carencia de OAuth 2.0. OIDC es un protocolo que actúa sobre OAuth 2.0 para proporcionarle una capa de autenticación basada en *tokens* JWT firmados que contienen información sobre el usuario que se ha autenticado. Para este protocolo, primero se ha dado una explicación sobre su necesidad y que problemas de OAuth 2.0 solventa. Además, se introducen los nuevos nombres que asigna este protocolo: OpenID Provider (OP) y Relaying Party (RP). Tras esto, se ha realizado una exposición de los conceptos que introduce OIDC y la manera en la que el ID Token es generado y firmado.

Para estos dos últimos protocolos, debido a su estrecha relación, se ha optado por realizar un escenario práctico en el que se utilizasen los dos. Para ello, se ha construido un cliente OAuth que puede utilizarse como un RP de OIDC. El cliente OAuth representa una aplicación web que muestra los eventos de un usuario de su Google Calendar. Para que esto sea posible, el cliente se apoya en el protocolo de OAuth para obtener una autorización delegada que le permite obtener los eventos del calendario del usuario. Como se ha comentado, este cliente también puede hacer de RP, siendo posible hacer uso de la autenticación ofrecida por OIDC mediante un botón diferente en la página principal. Para poder realizar una interacción sencilla con este escenario, se ha introducido un nodo cliente en el que se puede realizar las pruebas de autorización y

autenticación relacionadas con los protocolos. Gracias a este escenario se ha realizado un análisis de las trazas que este ha generado y posteriormente se han realizado algunas pruebas de concepto analizando posibles vulnerabilidades y mitigaciones del protocolo.

Se ha tratado de diseñar estos escenarios como base para la creación de laboratorios docentes que permitan el análisis y configuración de estas soluciones a nivel académico. En este aspecto, el uso de contenedores permite que estos escenarios contengan una configuración básica y funcional que pueda ser desplegada en segundos, permitiendo a su vez una alta personalización.

Para finalizar, para este Trabajo de Fin de grado se ha realizado un estudio de los protocolos SAML, OAuth y OIDC. Posteriormente, se han diseñado escenarios de prueba con el fin de permitir el análisis y configuración de estos protocolos, así como su uso como base en laboratorios docentes. Por ultimo, se ha realizado un análisis de las trazas generadas y un estudio sobre la seguridad de los protocolos mediante pequeñas pruebas de concepto.

2 Estado del arte

2.1 SAML

Security Assertion Markup Language (SAML) [1] es un estándar abierto de la Organization for the Advancement of Structured Information Standards (OASIS) para la autenticación y autorización en entornos distribuidos. Además, está basado en XML sobre HTTP permitiendo la interoperabilidad entre sistemas con diferentes tecnologías y ofreciendo así la verificación de la estructura de los mensajes intercambiados.

El principal objetivo de SAML es proporcionar un mecanismo de autenticación y autorización para ofrecer Single Sign-On (SSO) a ciertos servicios en la red.

SSO es el mecanismo en el cual una única acción de autenticación y autorización por parte el usuario puede permitirle acceso a todos los sistemas donde tenga permiso sin necesidad de presentar múltiples credenciales de autenticación [2]. En el caso de SAML, SSO permitirá a los usuarios de los servicios web tener que mantener solo unas credenciales que les permitirán el acceso a múltiples servicios y, por tanto, no tener que realizar diferentes acciones de autenticación para cada uno de ellos.

Las ventajas de este mecanismo son numerosas: En primer lugar, SSO permite una mayor rapidez en el acceso a los servicios, ya que el usuario no tiene que volver a introducir sus credenciales de acceso en cada *log-in*.

Por otra parte, el SSO proporciona un incremento en la seguridad del sistema. Esto es debido a que ahorra al usuario el tener que recordar y mantener las credenciales de acceso a cada servicio. Como consecuencia, el usuario tiende a utilizar contraseñas más fuertes y se reduce el riesgo de que haya un compromiso de la seguridad de la autenticación del servicio. Un ejemplo de esto puede ser el guardado de contraseñas en sitios poco seguros como *post-it* o archivos de texto plano por la necesidad del usuario de mantener diferentes credenciales [3].

Desde el punto de un administrador, el uso de una solución de SSO apoyándose en un proveedor de identidad hace que pueda ofrecerle a los usuarios del sistema un método de autenticación robusto sin la necesidad de tener que gestionar políticas de contraseña, sistemas de almacenamiento seguros y toda la infraestructura necesaria para la correcta custodia de las credenciales del usuario. Como consecuencia se incrementa la seguridad del sistema y se reducen los recursos necesarios para el mantenimiento de credenciales.

Para conseguir SSO, en SAML se pueden identificar tres agentes principales que actuarán en los intercambios del protocolo:

- **Identity Provider (IdP)**: Entidad que se encarga de gestionar la autenticación de los usuarios que pueda tener en su sistema.
- **Service Provider (SP)**: Entidad que ofrece un servicio especifico al usuario. Esta entidad hará uso del IdP para la autenticación de usuarios.
- **Subject**: Entidad que accede a cierto servicio de uno o varios SP y que se autenticará frente al IdP. No tiene por que ser una persona, puede ser un equipo o una organización, sin embargo, en este trabajo le llamaremos Usuario ya que es el caso mas típico.

Asimismo, existen diferentes maneras de desarrollar un intercambio SAML. Introducimos así los flujos de SAML, los cuales se diferencian por la entidad donde se haya iniciado el intercambio:

- **SP-initiated**: El proceso de intercambio es iniciado en el servicio al que se quiere acceder.
- **IdP-initiated**: El proceso de intercambio es iniciado en el IdP, normalmente a través de un portal corporativo.

Para explicar las bases de SAML nos centraremos en el flujo iniciado por SP cuyos pasos procedemos a enumerar a continuación y que se aprecian de manera gráfica en la figura 2.1:

1. El Usuario accede al servicio o aplicación proporcionado por el proveedor de servicios (SP1).
2. La aplicación requiere de una autenticación mediante el proveedor de identidad (IdP) por lo que se devuelve una redirección HTTP al portal de autenticación del IdP. La redirección tiene una carga XML llamada *SAML Request* con parámetros como el identificador de la petición, el identificador del SP1 que ha producido la petición o el IdP al que va dirigida entre otros.
3. El Usuario se autentica en el portal del IdP con sus credenciales, normalmente un nombre de usuario o Network Access Identifier (NAI) y contraseña, pero podría ser con certificados, datos biométricos, etc.
4. Tras la autenticación, el IdP genera un *SAML Assertion* que contiene información de quién, cuándo y cómo se desarrolló la autenticación. Esto se incluye dentro de un mensaje XML llamado *SAML Response*, el cual viajará en una redirección al SP1.
5. La aplicación procesa este *SAML Assertion* y le da una respuesta al usuario sobre el acceso al servicio.

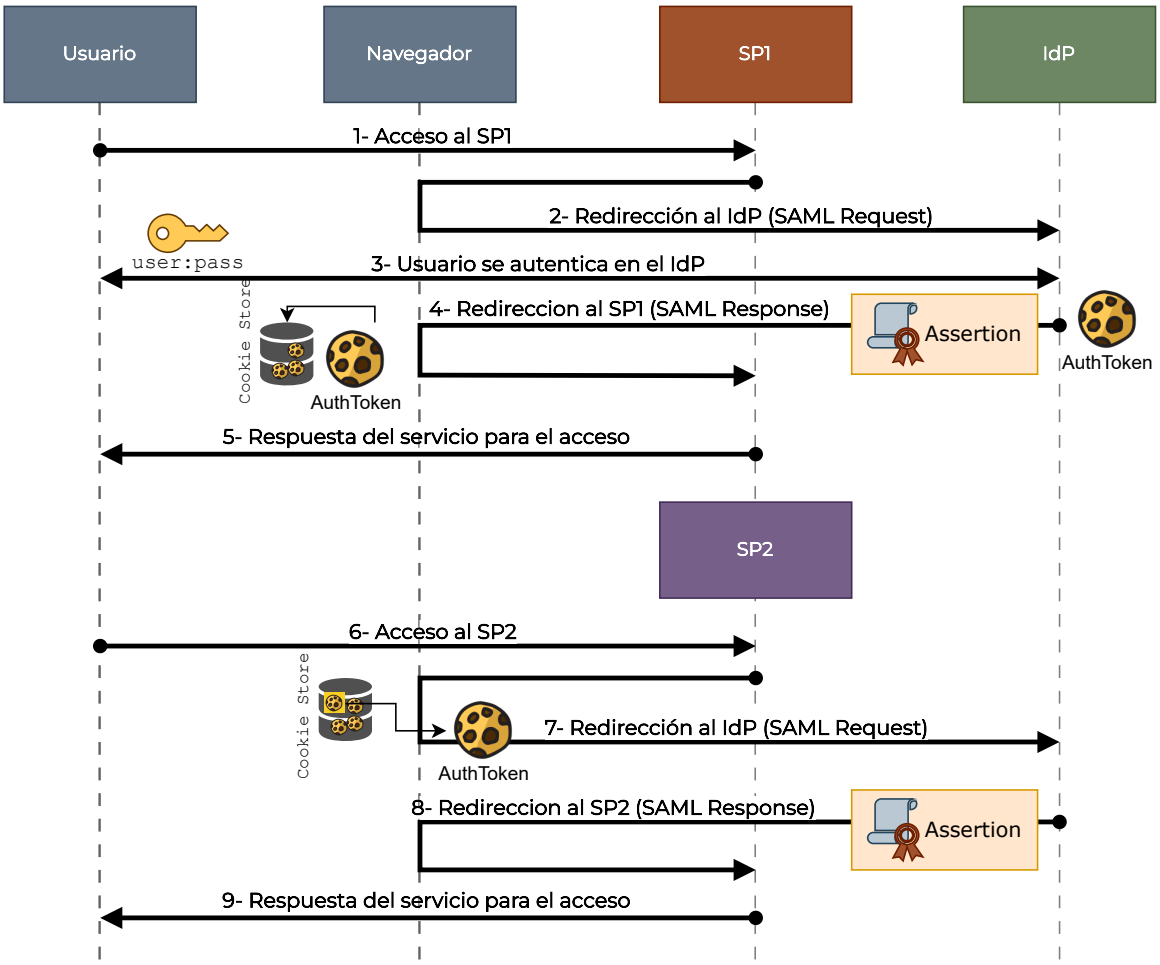

Figura 2.1: SSO con un flujo SAML iniciado en el SP (SP-initiated flow).

Como vemos, con esto se define la primera función principal de SAML que es la autenticación y autorización de usuarios a través de diferentes dominios. Sin embargo, todavía queda describir cómo se gestiona el acceso trasparente entre múltiples servicios sin que haya que volver a autenticar los usuarios.

Para ello, el protocolo se ayuda de *cookies* HTTP [4], concretamente el IdP inserta una *cookie* denominada *token* de autenticación, una vez el usuario se autentica correctamente, en su portal de autenticación.

En los pasos 6-9 de la figura 2.1 se aprecia como SAML hace efectivo el mecanismo de SSO en el acceso al SP2, el cual tiene también una relación de confianza con el mismo IdP. Esto ocurre gracias a que, cuando el usuario se autentica en el paso 3, el IdP añade una *cookie* con la cabecera HTTP Set-Cookie al generar la respuesta *SAML Response*. Mas tarde, el usuario accede al SP2 y en la redirección de esta segunda aplicación en el paso 7, el navegador incluirá la *cookie* en la petición de autenticación

para el IdP. Con esto el IdP comprobará esta *cookie* y enviará la *SAML Assertion* en el paso 8 al SP2. Con esto se consigue autenticar al usuario en dos servicios diferentes sin que el usuario haya tenido que volver a introducir sus credenciales, es decir, con una sola autenticación.

Hasta ahora se ha hablado de manera abstracta de los mensajes los mensajes SAML y sus correspondientes *Assertion*, por lo que a continuación, se va a dar una pequeña definición de los mensajes.

Una *SAML Authentication Request* es la petición XML generada por parte del SP del servicio que al que el usuario pretende acceder. A la forma de transmitir un mensaje se le llama *binding*, nosotros utilizaremos HTTP Redirect para la petición, pero existe otros como HTTP-POST, el cual se utilizará en la respuesta.

En esta redirección, el SP incluye la petición XML comprimida y codificada en Base64 como un parámetro en la URL. Teniendo esto en cuenta, desde el navegador del usuario se produce un GET a la dirección de *login* especificada del IdP con la petición *SAML Authentication Request* como parámetro.

Los campos mas importantes de esta petición *SAML Authentication Request* son los siguientes:

- **ID**: Identificador único asociado a cada una de las peticiones. El valor de este campo deberá coincidir con el valor del campo InResponseTo de la respuesta que sera generada por el IdP.
- **AssertionConsumerServiceURL**: Indica la URL donde el SP espera que se envie la respuesta. Esta URL se suele llamar Assertion Consumer Service (ACS).
- **Destination**: URL que indica cuál es el receptor legitimo de esta petición, que en este caso será el IdP.
- **ProtocolBinding**: Mecanismo que debe ser usado para transportar respuesta de la petición a través de la capa de transporte. En este campo es común encontrarse HTTP-POST.
- **Issuer**: Identificador del SP que realizó la petición. El IdP puede utilizar el valor de este campo para comprobar que este SP tiene una relación de confianza con él.
- **IssueInstant**: Indica cuando se generó esta petición. Según el tiempo que haya transcurrido desde este instante, el IdP puede optar por rechazar la petición.

Cuando la petición es procesada y se realiza la autenticación con el cliente, el IdP produce un mensaje *SAML Authentication Response* con contenido XML. Este mensaje viaja normalmente en un HTTP-POST al SP gracias al navegador del usuario que se autenticó.

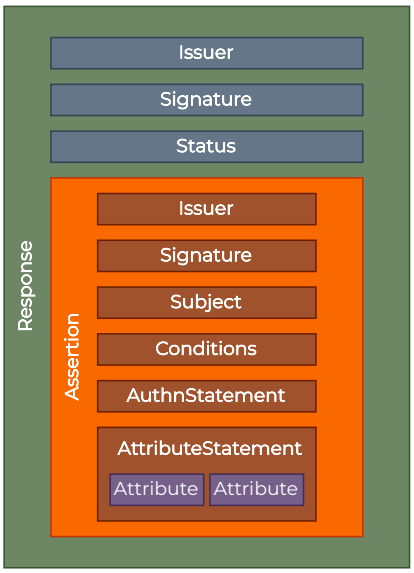

Figura 2.2: Estructura de una SAML Authentication Response.

Este mensaje *SAML Authentication Response* transporta mucha mas información que su antecesor y se divide en las secciones de la figura 2.2.

Como se muestra, tanto el mensaje completo como la parte de *Assertion* van firmados digitalmente por el IdP, normalmente con la clave privada asociada al certificado compartido con el SP en el intercambio de metadatos. En este mensaje, entre otros datos, se indica al SP si la autenticación ha sido satisfactoria (Status). Además, en la *Assertion* se indica información adicional sobre la autenticación, a continuación describimos sus apartados mas importantes:

- **Subject**: Identifica al usuario autenticado por el IdP.
- **Conditions**: Indica el periodo de validez y a quien va dirigido el mensaje. Dentro de este campo, se puede encontrar el apartado Audience, en el cual se indica el SP al cual se dirige esta *Assertion*.
- **AuthnStatement**: Describe cuándo (momento del acto de autenticación) y cómo ocurrió esta autenticación, es decir, cuál fue el método de autenticación utilizado.
- **AttributeStatement**: Aquí se indican otros detalles y atributos sobre el usuario que les puede servir de ayuda a los SP que gestionen la identidad del usuario.

Para que el contenido de estos mensajes se generen correctamente y sean aceptados por SP e IdP, antes debe ocurrir un intercambio de información ente estas dos entidades. A este proceso se conoce como intercambio de metadatos y sirve para que ambos compartan información sobre sus organizaciones (identificador, que nombres utilizar,

URLs, certificados, etc.). Con este proceso, ambas entidades pueden establecer una relación de confianza mutua a la que podemos llamar federación si están en dominios diferentes.

Junto a los mecanismos de *SAML Assertion* para la autenticación y la autorización y el uso de *cookies* para el acceso transparente entre servicios, SAML proporciona un mecanismo de SSO de una manera sencilla y segura. Además, el hecho de que esté basado en documentos XML facilita que tenga una estructura bien definida, resultando esto en un protocolo interoperable entre diferentes sistemas.

2.2 OAuth 2.0

En los esquemas previos a OAuth, cuando un usuario necesitaba delegar sus recursos protegidos existentes en un servicio a una aplicación de terceros, la aplicación se autenticaba con el servicio utilizando las credenciales del propio usuario. Es decir, el usuario tenía que proporcionar a esta aplicación de terceros sus credenciales de acceso del servicio, lo cual creaba problemas como:

- Las aplicaciones de terceros debían guardar estas credenciales para utilizarlas después, normalmente se hacía en texto plano.
- Estas aplicaciones podían acceder a todos los recursos protegidos del usuario.
- Para denegar el acceso de una aplicación a sus recursos, el usuario debía cambiar su contraseña.
- Una brecha en la aplicación puede resultar en un compromiso de la contraseña y recursos protegidos del usuario.

En la figura 2.3 se muestra un ejemplo de como funcionaban este tipo de sistemas previos a OAuth. Como se muestra, la aplicación de terceros pide al usuario directamente el usuario y contraseña de su servicio de correo para poder enviar correos y ver los contactos en su nombre.

Para intentar solventar estos problemas nace OAuth 2.0 [5] (Open Authorization),

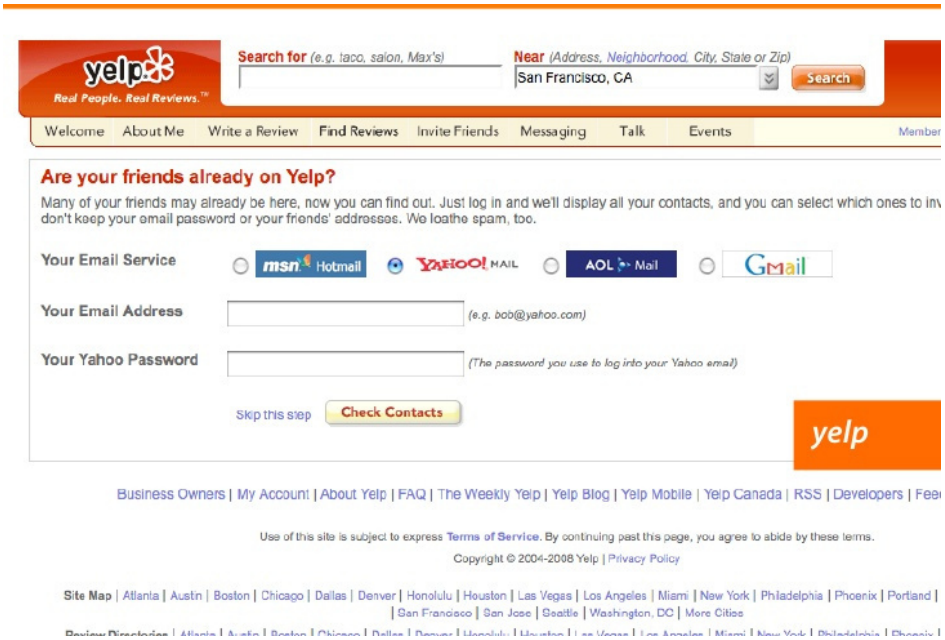

Figura 2.3: Ejemplo de una autorización a una aplicación de terceros sin OAuth.

un *framework* estándar de autorización abierto desarrollado por la Internet Engineering Task Force (IETF) que permite a aplicaciones de terceros obtener acceso limitados a un recurso HTTP, ya sea en nombre del usuario propietario generando una interacción entre el usuario y el servicio HTTP o permitiendo a estas aplicaciones obtener acceso por ellas mismas.

En este documento nos centraremos el *framework* de OAuth 2.0 que identifica cuatro roles que se ven implicados en los intercambios:

- **Resource Owner (RO)** : Entidad capaz de otorgar acceso a un recurso protegido. Normalmente esta entidad será el usuario del que se obtienen los datos.
- **Client (Cliente)**: Es la aplicación de terceros que desea hacer uso de los recursos protegidos del RO.
- **Authorization Server (AS)**: Es el servidor encargado de proveer los tokens de acceso al Cliente una vez autenticado el RO y se ha obtenido autorización.
- **Resource Server (RS)**: Es el servidor que se encarga de servir los datos que son propiedad del RO. Esta es la entidad que recibirá las peticiones del Cliente las cuales deberán incluir un token de acceso válido.

Para facilitar el entendimiento de estos roles, a continuación se detalla un ejemplo: Un usuario final (RO) puede permitir a un servicio de impresión (Cliente) acceso a sus fotos almacenadas en un servicio de almacenamiento (RS) sin cederle su nombre de usuario y contraseña. En vez de esto, el RO se autentica directamente con el servidor (AS) en el cual confía el servicio de almacenamiento de fotos. Este servidor produce unas credenciales (Access Token) que materializan la delegación de autorización por parte del RO.

Se debe destacar que el AS y el RS pueden estar en una misma entidad o una separada.

Además, existen diversas maneras y contextos en los que usar OAuth 2.0. A esto se le denomina *Grant Types* y son diferentes formas por las que el Cliente puede conseguir un Access Token. En concreto, existen cuatro *Grant Types* en OAuth 2.0:

- **Authorization Code**: utilizado principalmente para Clientes que pueden hacer de intermediario entre el Authorization Server y el Usuario y pueden redirigir a su navegador, como puede ser un servidor web. Es el uso mas común.
- **Implicit**: pensado para Clientes que estén implementados en un navegador mediante lenguajes como JavaScript.
- **Client Credentials**: es utilizado para Clientes OAuth que no van a acceder a recursos de un usuario final. En cambio, es el propio Cliente el que va a actuar como RO accediendo sus propios recursos protegidos.

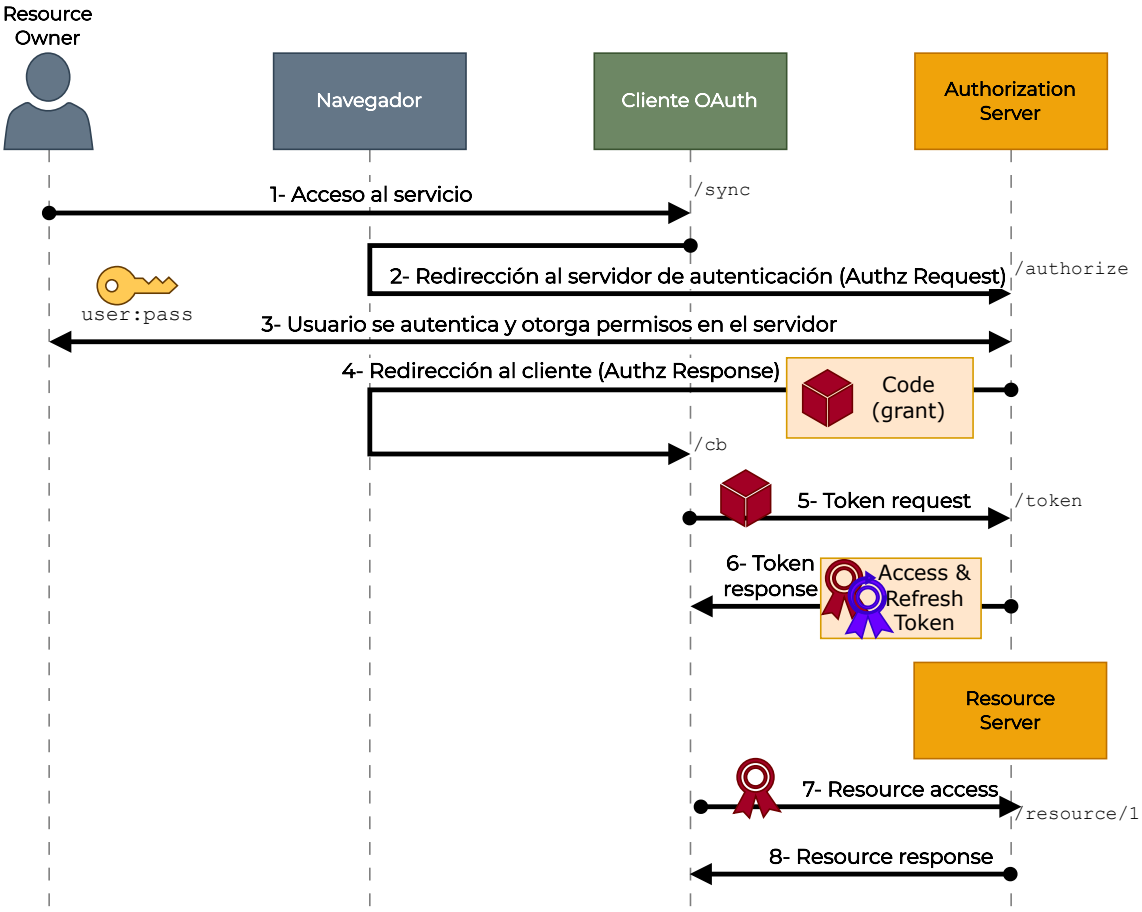

Figura 2.4: Intercambio OAuth 2.0 con Authorization Code Grant.

- **Resource Owner Credentials**: el Cliente hace uso de las credenciales del RO para obtener un Access Token y no tener que guardarlas. Es el menos recomendable de todos ya que implica una gran confianza por parte del RO al Cliente OAuth.

Sin embargo, en nuestro caso nos centraremos en el Authorization Code Grant, ya que es el más ampliamente utilizado.

A continuación, vamos a explicar el flujo de Authorization Code Grant y nos apoyaremos en la figura 2.4 para describir sus pasos:

1. El RO accede a la aplicación Cliente y esta le solicita al RO que la autorice para poder acceder a alguno de sus recursos protegidos.

2. El Cliente OAuth redirige al RO al AS, en este mensaje el Cliente incluye su identificador.

3. El AS se encarga de realizar la autenticación del RO.
4. El AS redirige de vuelta al navegador del RO al Cliente. En este mensaje va incluido el código de autorización (Authorization Code).
5. El Cliente incluye el Authorization Code en un mensaje dirigido al AS para pedir el Access Token.
6. El AS comprueba este Authorization Code y devuelve al Cliente un mensaje que incluye un Access Token y un Refresh Token opcionalmente.
7. Una vez el Cliente ya ha obtenido el Access Token, este es capaz de lanzar peticiones a alguno de los *endpoints* protegidos del RS correspondientes al RO. Para conseguir esto, el Cliente envía una petición al RS al recurso deseado en la cual incluye el Access Token.
8. El RS comprueba este Access Token y devuelve la información asociada al recurso del RO.

Una vez visto el intercambio, puede surgir la duda de por qué es necesario el intercambio Authorization Code y Access Token y no obtener directamente el Access Token. Esto es debido a que, en este tipo de flujo, se pretende hacer uso dos canales: *front-channel* y *back-channel*. En primer lugar, el *front-channel* es el utilizado por el usuario final para interactuar con la aplicación Cliente y el AS, gracias a esto se puede transmitir el Authorization Code. Por otro lado, el *back-channel* es utilizado por el Cliente para que sea transmitido el Access Token. Con esta diferenciación conseguimos que el Access Token no quede expuesto garantizando un canal seguro. Si el Access Token se obtuviese directamente junto al Authorization Code en la redirección HTTP del navegador, posibles terceros podrían obtenerlo de manera más sencilla, por ejemplo mirándolo en la URL del navegador o en el historial. Con esto se busca tener un *back-channel* seguro y protegido para transmitir el Access Token, es por esto por lo que el protocolo obliga a utilizar TLS para este intercambio.

A continuación, se va a explicar el contenido principal de los mensajes de OAuth 2.0:

El primer mensaje que podemos identificar es el *Authorization Request*. Este mensaje es enviado por parte del Cliente OAuth, por medio de una redirección del navegador del RO, al AS para obtener un Authorization Code. Podemos encontrar un ejemplo de este mensaje en el código 2.1.

Código 2.1: Ejemplo de un mensaje *Authorization Request*

```
GET /authorize?response_type=code
       &client_id=s6BhdRkqt
       &state=xyz
       &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
       &scope=mail profile
```

```
Host: server.example.com
```

En él encontramos los diferentes parámetros que se explican a continuación:

- **response_type=code**: Obligatorio. Indica el tipo de flujo de autorización que se va a realizar es un Code Grant.
- **client_id**: Obligatorio. Identificador de la aplicación Cliente que se debe haber registrado previamente.
- **redirect_uri**: Opcional. URI de redirección a la que se debe redirigir al RO una vez que se haya autenticado. Esta URI debe coincidir con la registrada previamente.
- **scope**: Opcional. Lista de permisos que se solicitan al RO. En este caso, correo electrónico y perfil.
- **state**: Recomendado. Para proteger de ataques CSRF.

También podemos identificar el mensaje *Authorization Response*. Este mensaje es generado por el AS y enviado al Cliente OAuth mediante una redirección al navegador del RO. Esta es la vía que utiliza el AS para poder comunicar el Authorization Code al Cliente siempre y cuando la autenticación y autorización del RO haya sido satisfactoria.

Código 2.2: Ejemplo de una redireccion con *Authorization Response*

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
      &state=xyz
```

En el código 2.2 se adjunta un ejemplo de este mensaje del que podemos destacar los siguientes parámetros:

- **code**: Obligatorio. Authorization Code que se intercambiará por un Access Token. Debe tener una expiración corta, recomendado menos de 10 minutos.
- **state**: Debe coincidir con el que se envió en el paso anterior (si se envió).

Con el Authorization Code anterior el Cliente puede solicitar un Access Token al AS. El mensaje encargado de esa solicitud se llama *Token Request* y es una petición HTTP POST generada por el Cliente y enviada al AS obligatoriamente mediante TLS.

Código 2.3: Ejemplo de un mensaje *Token Request*

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
```

```
&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&client_id=exampleid
&client_secret=examplesecretverylong
```

En el listado 2.3 podemos ver un ejemplo de este mensaje del cual podemos destacar los siguientes parámetros:

- **grant_type=authorization_code**: Obligatorio. Indica que esta utilizando el Authorization Code.
- **client_id**: Obligatorio. Identificador de la aplicación Cliente que se debe haber registrado previamente.
- **client_secret**: Clave secreta de la aplicación Cliente que se debe haber registrado previamente.
- **code**: Obligatorio. Authorization Code obtenido en el mensaje *Authorization Response*.
- **redirect_uri**: URI de redirección del Cliente que coincidir con la URI en la *Authorization Request* (si se utilizó).

Como respuesta a la petición anterior, el AS envía al Cliente OAuth el mensaje *Token Response* que contiene el Access Token y el Refresh Token si el servidor verifica correctamente la petición anterior.

Código 2.4: Ejemplo de un mensaje *Token Response*

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
    "access_token":"2YotnFZFEjr1zCsicMWpAA",
    "token_type":"Bearer",
    "expires_in":3600,
    "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
}
```

Un ejemplo de este mensaje con su cuerpo se puede ver en el listado 2.4. A continuación se detallan los parámetros de este mensaje:

- **access_token**: Access Token que se utiliza para acceder a los recursos protegidos.
- **expires_in**: Tiempo de expiración del Access Token.

- **refresh_token**: Refresh Token que se utiliza para obtener un nuevo Access Token.
- **token_type**: Tipo de token. Puede ser Bearer o MAC.

Una vez que el Cliente OAuth tiene un Access Token, puede acceder a los recursos protegidos del RO. Para ello, el Cliente OAuth envía un HTTP GET del recurso al que quiere acceder al RS con el Access Token en la cabecera de autorización.

Código 2.5: Ejemplo de un acceso a un recurso con un Access Token

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer 2YotnFZFEjr1zCsicMWpAA
```

En el listado 2.5 se muestra un ejemplo de una petición al recurso protegido `/resource/1`. Esta petición va acompañada de la cabecera `Authorization` cuyo valor es el token de acceso obtenido anteriormente y que es de tipo `Bearer` .

El código anterior será presentado al RS, el cual consultará su validez comprobando que con ese *token* el Cliente puede acceder al recurso que se ha solicitado. Tras comprobarlo, el RS emitirá una respuesta a esta petición con los recursos solicitados.

Como conclusión, OAuth 2.0 proporciona un *framework* que permite ofrecer un mecanismo de autorización delegada, permitiendo a los usuarios propietarios de recursos protegidos en un servidor otorgar acceso delegado a aplicaciones de terceros que actúan como clientes OAuth. Esto es posible gracias a los diferentes flujos definidos por OAuth 2.0, como el Authorization Code Grant, y al uso de los distintos tipos de códigos como el Authorization Code y el Access Token. En conjunto, estos aspectos hacen de OAuth 2.0 un estándar ampliamente utilizado para la autorización delegada en aplicaciones web y móviles.

2.3 OpenID Connect

En el pasado, muchos servicios empezaron a utilizar OAuth 2.0 también para la autenticación. Sin embargo, este protocolo solo estaba diseñado para la autorización, por lo que muchos de estos servicios estaban utilizando el protocolo erróneamente. En estos casos, se permite al usuario autenticarse de manera externa a OAuth 2.0. De esta manera, no hay ninguna relación entre el proceso de autenticación y autorización. Con OAuth 2.0 la única información que obtiene el cliente OAuth es un *token* que simplemente actúa como un *ticket* que le autoriza y que es presentado a un Resource Server para obtener un recurso protegido. El cliente no tendría ninguna información sobre el contexto, es decir, cómo se ha autenticado el cliente, cuál es el tiempo de expiración de ese proceso de autenticación, etc.

Para solventar esto, se introduce OpenID Connect (OIDC) 1.0 [6], una capa implantada sobre OAuth 2.0 para permitir a los clientes la verificación de la autenticación realizada por el usuario en el servidor de autenticación además de la obtención de información básica sobre el usuario de manera interoperable y respetando el paradigma REST.

OIDC proporciona sus mecanismos de autenticación extendiendo OAuth 2.0 gracias a los siguientes conceptos:

- Los clientes que deseen utilizar OIDC pueden hacer uso del valor `openid` incluyéndolo en el campo `scope` del mensaje *Authorization Request*.
- Los datos asociados a la autenticación realizada por parte del usuario son presentados al cliente mediante un JSON Web Token (JWT) denominado ID Token firmado por el servidor de autenticación.

Además de esto, el protocolo renombra las entidades participantes de OAuth 2.0 de la siguiente manera:

- **End-User (Usuario)**: Es el usuario que se autentica.
- **Relaying Party (RP)**: Es un cliente OAuth 2.0 que esta haciendo uso de OIDC para poder autenticar al Usuario.
- **OpenID Provider (OP)**: Es un AS de OAuth 2.0 que permite el uso de OIDC.

Como OIDC se basa en OAuth 2.0, la autenticación y autorización puede seguir varios flujos. Sin embargo, aquí se explicará el más utilizado, que es el de Authorization Code (figura 2.5) y cuyos pasos se mencionan a continuación:

1. El Usuario accede a un recurso del RP donde es necesaria la autenticación y autorización en el OP.
2. El RP redirecciona el navegador del Usuario hacia el punto de autenticación del OP. La diferencia con el flujo OAuth 2.0 tradicional es que en el campo `scope`

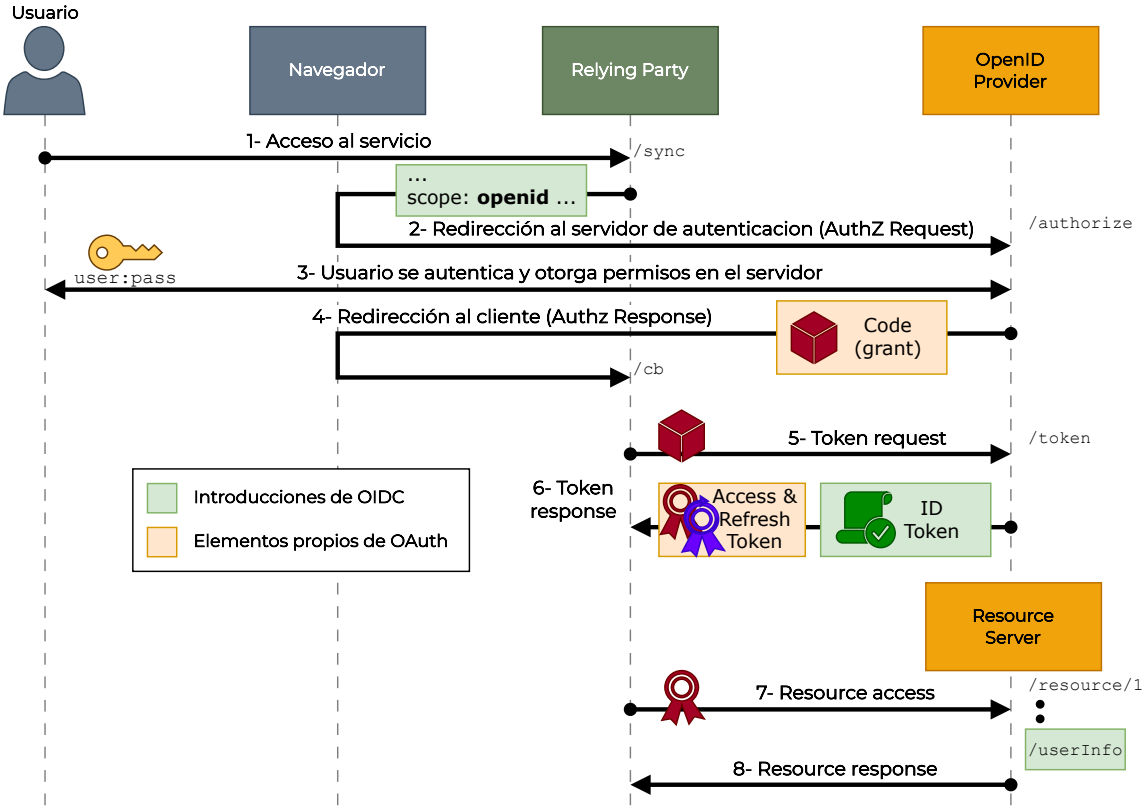

Figura 2.5: Flujo de Authorization Code en una configuración con OIDC

de la petición se introduce el valor `openid`. De esta manera, el OP detecta ese valor y puede tratar el flujo a corde con la especificación.

3. El Usuario realiza la autenticación y la autorización interactuando con el OP.
4. El OP redirige el navegador del Usuario con el Authorization Code.
5. El RP lanza una petición (*Token Request*) al OP con el Authorization Code.
6. El OP envía una respuesta (*Token Response*) que contiene el Access Token, el Refresh Token y, como diferencia frente a OAuth 2.0, añade un ID Token que va firmado por el OP y que contiene información representativa sobre el evento de autenticación y el Usuario como puede ser el OP que lo generó, el identificador del Usuario, tiempo de expiración, etc.
7. En este paso, el RP puede acceder a cualquier recurso permitido por el `scope` que se ha pedido. Sin embargo, OIDC introduce un *endpoint* adicional llamado `UserInfo` el cual puede ser utilizado por el RP para obtener mas información acerca del Usuario como puede ser el ID, correo o foto de perfil.

Como se ha mencionado anteriormente, un ID Token es un JWT [7] que es firmado

por el OP y es incluido en mensaje de *Token Response* en el campo `id_token`, como en el ejemplo del código 2.6. Sin embargo, como se puede ver en el ejemplo, este ID Token no es fácilmente legible, esto es porque ha sido construido con diferentes partes del ID Token en Base64.

Código 2.6: Mensaje *Token Response* con un ID Token

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
    "access_token": "SlAV32hkKG",
    "token_type": "Bearer",
    "refresh_token": "8xLOxBtZp8",
    "expires_in": 3600,
    "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzc
     yI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4Mjg5
     NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsCiAibm9uY2UiOiAibi0wUzZ
     fV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5Nz
     AKfQ.ggW8hZ1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr_P6q
     Jp6IcmD3HP99Obi1PRs-cwh3LO-p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJ
     NqeGpe-gccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiOtX7Tpd
     QyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEoRoS
     K5hoDalrcvRYLSrQAZZKflyuVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4
     XUVrWOLrLl0nx7RkKU8NXNHq-rvKMzqg"
}
```

En concreto, los JWT se forman con la concatenación de tres elementos los cuales se describirán a continuación.

La primera parte es la cabecera. En el código 2.7 se muestra el valor del ejemplo anterior decodificado en Base64. En concreto, en esta cabecera del ejemplo se indica con el campo `alg` que el algoritmo para la firma del JWT es `RS256` (RSA Signature with SHA-256). También se incluye el campo `kid`, que es el identificador de la clave con la que se hizo la firma. En esta cabecera también podrían ser incluidos campos como el de `x5u` indicando la URL donde encontrar el certificado de clave publica asociado a la clave privada utilizada para la firma.

Código 2.7: Cabecera de un JWT

```
{
    alg: "RS256",
    kid: "1e9gdk7"
}
```

La segunda parte contiene el *payload* del JWT. En el ejemplo del código 2.8 se aprecia la carga JSON una vez realizada la decodificación en Base64, donde vemos el uso de algunos de los campos que pueden ser utilizados en este ID Token. A continuación, se describen los campos del código 2.8 que son justamente los que son obligatorios en OIDC:

- **iss**: El identificador del emisor del la respuesta (el OP).
- **sub**: El identificador único del Usuario que puede ser consumido por el RP.
- **aud**: La audiencia para la cual se ha generado el ID Token. Tiene que contener el `client_id` del RP al que se envía la respuesta.
- **exp**: Tiempo de expiración a partir del cual el ID Token no puede ser aceptado. Su valor esta representado como el numero de segundos desde 1970-01-01T0:0:0Z UTC.
- **iat**: Momento en el cual el JWT fue generado. Su valor esta representado como el numero de segundos desde 1970-01-01T0:0:0Z UTC.

Código 2.8: *Payload* de un JWT

```
{
 "iss": "http://server.example.com",
 "sub": "248289761001",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970
}
```

La tercera y última parte es la firma, cuyo contenido se pasa a Base64 antes de concatenarla con el resto de ID Token. Esta se realiza conforme al estándar JSON Web Signature (JWS) [8] cuyo funcionamiento se puede resumir en los siguientes pasos:

1. Se obtiene el método utilizado para securizar la firma. En el ejemplo que estamos tratando vemos en el listado 2.7 que es `RS256`.
2. La cabecera es codificada en Base64, el resultado lo podemos denominar *cabecera*64.
3. El cuerpo es codificado en Base64, el resultado lo podemos llamar *cuerpo*64.
4. Se concatenan los dos elementos anteriores con el carácter ".".
5. Se utiliza una clave compartida o clave privada en el caso de RSA256 para encriptar la cadena formada en el paso anterior, es decir, se realizaría la siguiente

operación:

$$firma = RSA256(cabecera64 \parallel "." \parallel cuerpo64\|, K^{-})$$

Como hemos visto, estos serian los tres elementos que conforman el ID Token y cuya estructura final se formaría de la siguiente manera:

$$id\_token = cabecera64 \parallel "." \parallel cuerpo64 \parallel "." \parallel firma64$$

Como conclusión, las incorporaciones introducidas por OIDC han permitido que OAuth tenga un protocolo en el cual apoyarse para ofrecer la parte de autenticación de manera segura y relativamente simple.

3 Análisis de objetivos y metodología

Uno de los objetivos que se han guiado en este trabajo es la creación de dos escenarios de prácticas relacionados con las tres tecnologías estudiadas. Con esto se pretende dar un contexto para la posible personalización de los escenarios de manera que se puedan practicar las diferentes configuraciones asociadas a estas tecnologías. Así mismo, se ofrecen mecanismos para que se puedan recabar datos o *logs* para un posterior análisis y estudio de las trazas obtenidas, pudiendo así comparar de los estándares asociados a los protocolos con el comportamiento real del escenario proporcionado. De igual manera, se ha querido seguir un enfoque *dockerizado* a la hora de construir estos escenarios, de manera que se pueda tener un entorno virtual ya declarado y funcional a la hora de realizar las modificaciones. Esta virtualización con Docker, al tener un enfoque más declarativo mediante archivos de texto docker-compose, permite que los usuarios de estos escenarios puedan guardar o portar sus modificaciones de manera sencilla con sistemas como Git, subida de archivos a la nube o incluso pen-drives. Esta es una de las principales ventajas frente a otras alternativas de virtualización, ya que solo se necesitan los ficheros de configuración de estos escenarios para poder montarlos y ejecutarlos, a diferencia de sistemas como VirtualBox donde se necesita portar todo el sistema de máquinas ocupando normalmente varios giga-bytes.

Por otra parte, se ha llevado a cabo un análisis de los protocolos tratados en este trabajo. Para ello, se ha trabajado con los escenarios mencionados anteriormente para extraer trazas representativas asociadas a los protocolos. Una vez extraídas, estas trazas se han analizado y se ha elaborado una explicación las mismas.

Finalmente, se ha realizado un análisis de las vulnerabilidades más importantes asociadas a cada uno de los protocolos. Para comprender mejor alguno de ellos se ha realizado, gracias a los escenarios ya montados, pequeñas pruebas de concepto para ver como se comportaba el protocolo ante ciertas acciones. Por último, se han mencionado las posibles medidas de seguridad que pueden ofrecer los distintos actores en el protocolo ante ciertos ataques.

Para cada protocolo se ha seguido la siguiente metodología:

1. Estudio de la documentación del protocolo para entender su funcionamiento.
2. Búsqueda de posibles herramientas y librerías adecuadas para la construcción y análisis de los escenarios realizados para el protocolo.

3. Construcción y puesta en marcha del escenario de prueba del protocolo con las librerías y herramientas seleccionadas.
4. Realización de pruebas asociadas al funcionamiento del protocolo.
5. Análisis y estudio del comportamiento del escenario mediante trazas o *logs*.
6. Realización de pruebas de concepto y análisis de seguridad del protocolo con la ayuda del escenario.

4 Diseño

4.1 Escenario de SAML

Este escenario para SAML se ha basado en la puesta en marcha de un IdP, dos SP y un cliente para probar como funciona el protocolo, sus flujos y las trazas que se generan.

Para este propósito se ha utilizado simpleSAMLphp [9], una aplicación escrita en PHP para la gestión de autenticación. De entre las razones para su elección se pueden destacar su naturaleza *Open Source* y el continuo mantenimiento llevado a cabo por sus principales colaboradores. Además, esta solución se centra principalmente en los componentes de SAML, aunque puede trabajar con otros protocolos como OAuth o OIDC.

4.1.1 Finalidad

Los objetivos principales que han guiado la construcción y el diseño de este escenario han sido los siguientes:

- Se ha buscado tener una prueba de concepto que estuviese basada en contenedores, de manera que se siguiese un enfoque portable y de fácil despliegue.
- Se ha tratado de construir un escenario en el que poder realizar un análisis de los mensajes SAML de manera sencilla.
- Se ha procurado realizar un diseño de un marco en el que poder probar fácilmente las vulnerabilidades de SAML.

4.1.2 Arquitectura

En el apartado de arquitectura, el escenario cuenta con cuatro nodos principales que son los cuatro contenedores de Docker. Estos contenedores están desplegados con docker-compose y agrupados en una red interna de tipo *bridge* en la que todos ellos son alcanzables y tienen una salida a internet.

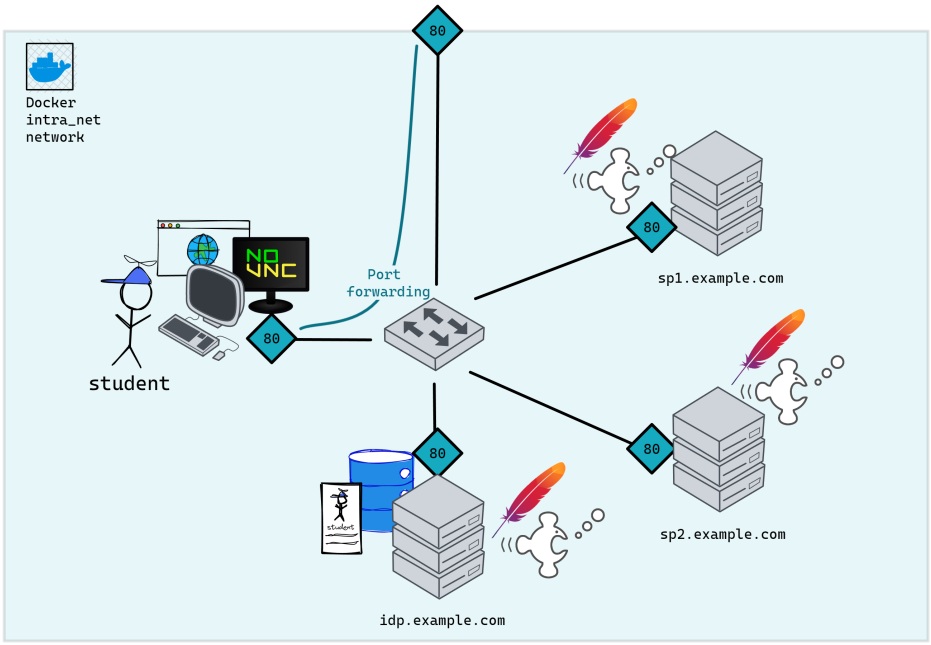

Figura 4.1: Esquema gráfico del escenario montado para SAML.

En la figura 4.1 se puede ver un resumen gráfico de como está desplegado este escenario. Para facilitar la navegación por los SP e IdP, facilitar las configuraciones y simplificar las redirecciones, se ha optado por meter al cliente dentro de esta red interna de Docker. De esta manera, se puede conseguir que todos los contenedores y sus servicios sean accesibles en su puerto original por los demás integrantes de la red interna. Como vemos, tanto el IdP como los dos SP tienen un servicio web con Apache y simpleSAMLphp los cuales son accesibles mediante el puerto 80. Se hace uso de HTTP sin securizar para poder así realizar un análisis de las trazas que se puedan obtener.

Además de los contenedores de los IdP y SP, tenemos el del cliente. Este contenedor representa a un hipotético cliente y será el encargado de realizar las peticiones hacia los SP e IdP. Por esta razón, se va a instalar en el una distribución de Ubuntu a la cual se le han añadido herramientas para poder realizar los análisis y las pruebas, como son Firefox, Wireshark y BurpSuite. Además, se hace uso de noVNC para poder acceder a él y poder usarlo de manera gráfica para realizar las diferentes pruebas. Para que sea posible el acceso a este contenedor, se hace una *forwarding* de su puerto original 80 al puerto 80 fuera de la red interna y desde el cual será accesible desde el *host*.

Con todos estos elementos ya definidos, se puede realizar el despliegue de todo el escenario mediante el comando `docker-compose up`. Una vez hecho esto, ya es posible hacer uso del escenario introduciendo la URL `http://localhost` en la máquina *host* donde hayamos ejecutado el comando. De esta manera se conecta al contenedor cliente y se puede utilizar su escritorio y navegador. En la figura 4.2 se muestra cómo se ha

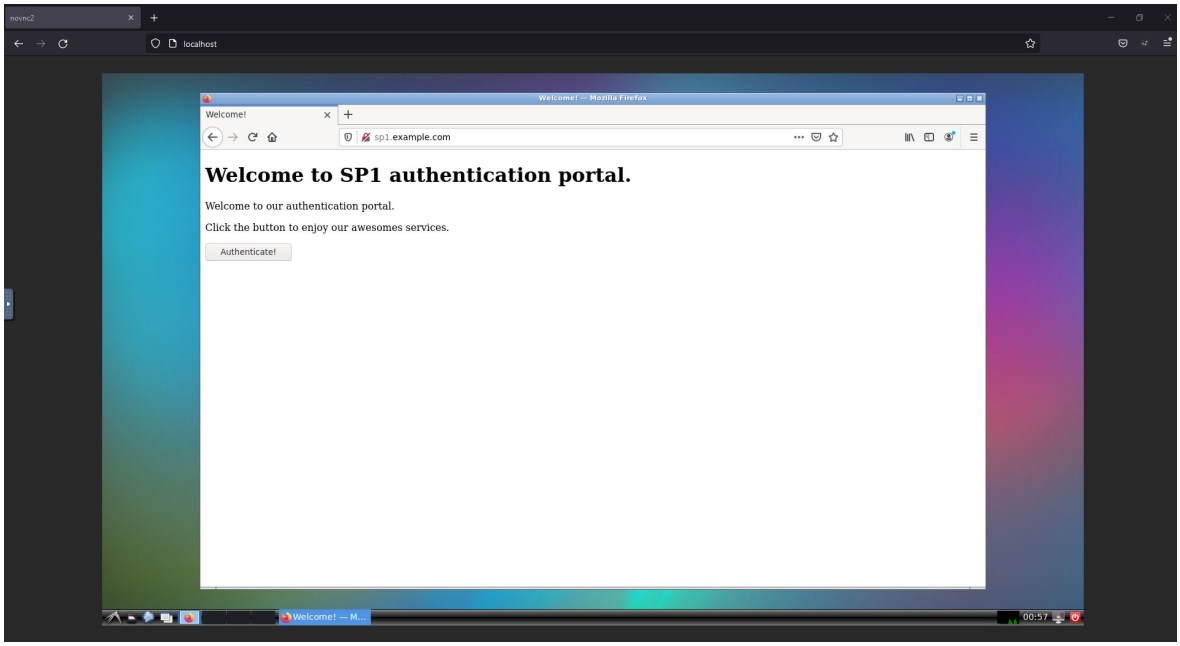

Figura 4.2: Página de inicio del SP1 desde el contenedor cliente.

accedido a la página del SP1 `http://sp1.example.com` desde el contenedor cliente en el navegador de la máquina *host*.

Tras esto se puede realizar la prueba de autenticación en el servicio mediante la autenticación en el IdP. Para ello basta con pulsar el botón con el texto *Autenticate!* que ofrece la página de inicio del SP1. Una vez pulsado, como se puede ver en la figura 4.3, se produce una redirección al IdP en el cual el usuario debe poner su usuario y contraseña (por ejemplo: `silvia` y `silviapass`). Tras comprobar que las credenciales son correctas, el IdP redirige al usuario al SP1 donde se comprobará que el mensaje recibido del IdP es correcto y se le otorgara acceso al usuario al recurso solicitado. Al ser un escenario de pruebas, si el SP1 da por bueno el mensaje del IdP, mostrará al usuario una página donde se mostrará los atributos extraídos del *Assertion*.

Posteriormente, para hacer una prueba del SSO, se accede a la página del SP2 que tiene un botón también para autenticar con el IdP. Como se realizó la autenticación en el IdP anteriormente, esta vez no es necesario que el usuario proporciones sus credenciales de nuevo al IdP, automáticamente generará el *Assertion* con la correspondiente redirección de vuelta al SP2, el cual dará acceso al usuario.

Detalles mas concretos de cómo funciona este flujo de SSO se pueden encontrar en la sección 2.1.

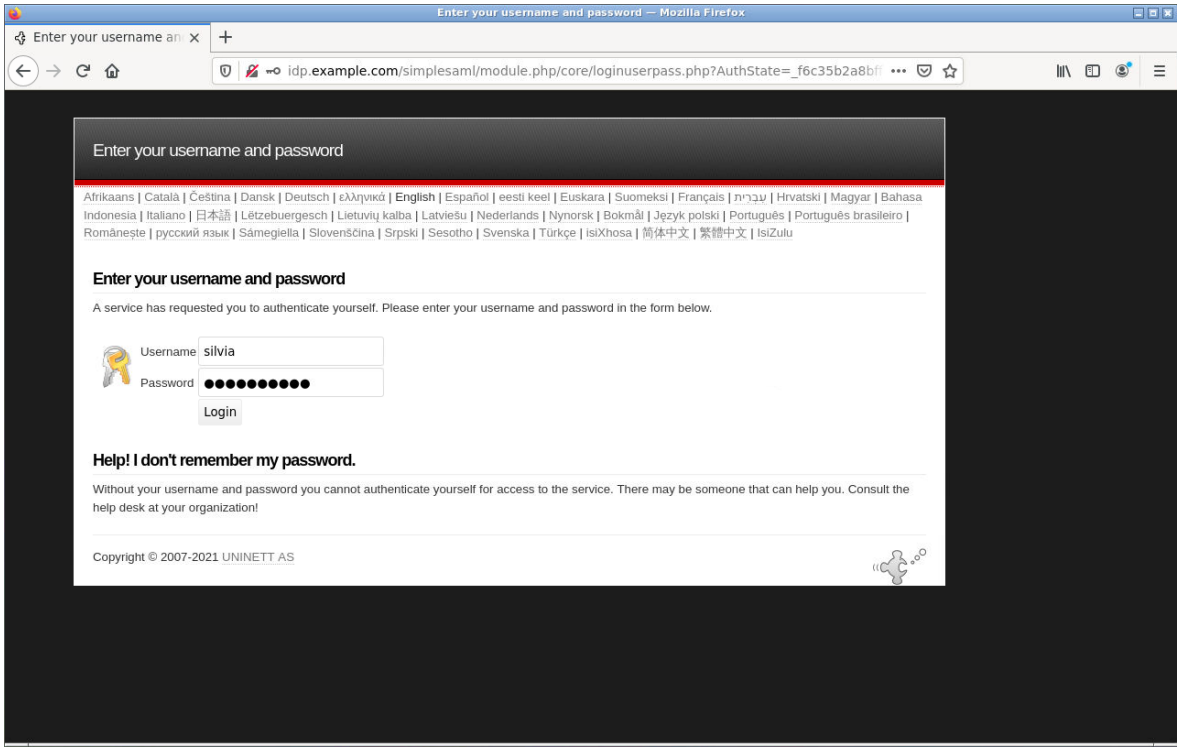

Figura 4.3: Pagina de autenticación del IdP.

4.2 Escenario de OAuth 2.0 y OIDC

Para el escenario de OAuth 2.0 y OIDC de este trabajo se ha realizado la puesta en marcha de un cliente OAuth que funciona también de RP y una máquina usuario que es la que realizará las peticiones y pruebas de este protocolo.

4.2.1 Finalidad

La construcción de este escenario se ha visto guiada por los siguientes objetivos:

- Establecer un escenario portable basado en contenedores en el que poder realizar cambios en el código de manera sencilla.
- Crear un cliente OAuth con una librería bien mantenida y usada actualmente.
- Tener un cliente OAuth que pudiese convertirse en RP sin realizar cambios en el código.
- Establecer un escenario en el que se pudiese generar trazas para posteriormente analizar los mensajes OAuth 2.0 y OIDC.
- Disponer de un escenario en el que poder experimentar con pruebas de concepto relacionadas con las vulnerabilidades de los protocolos.

4.2.2 Arquitectura

Para la creación del cliente OAuth se ha hecho uso de la librería oauthlib [10] para su uso en Python. Esta librería se ha utilizado en conjunto al paquete Flask [11] para poder confeccionar una pequeña interfaz web en la que el usuario pudiese realizar la delegación de la autorización a este cliente. Se ha optado por esta librería por ser *Open Source*, estar relativamente bien mantenida, tener una buena documentación para su uso y ser más genérica que otras librerías similares.

Respecto a la arquitectura de este escenario, se ha seguido un enfoque *dockerizado* en el que se definen dos contenedores: uno para el cliente OAuth y otro para el usuario. Además, el despliegue de estos se realiza con docker-compose y quedan agrupados en una red interna de tipo *bridge* con la que tienen conexión entre ambos y tienen acceso hacia internet.

En la figura 4.4 podemos ver de manera gráfica cómo quedarían estos elementos integrados. Vemos como, por una parte, tenemos el contenedor del cliente OAuth el cual hará uso de oauthlib en una página hecha con Flask. En este servidor se establecerá en el puerto 80, ya que esto nos facilita el análisis de las trazas, y en el se realizarán las redirecciones al servidor de autenticación de Google y se obtendrá el Access Token para poder hacer peticiones a la API de Calendar de Google. De esta manera, esta

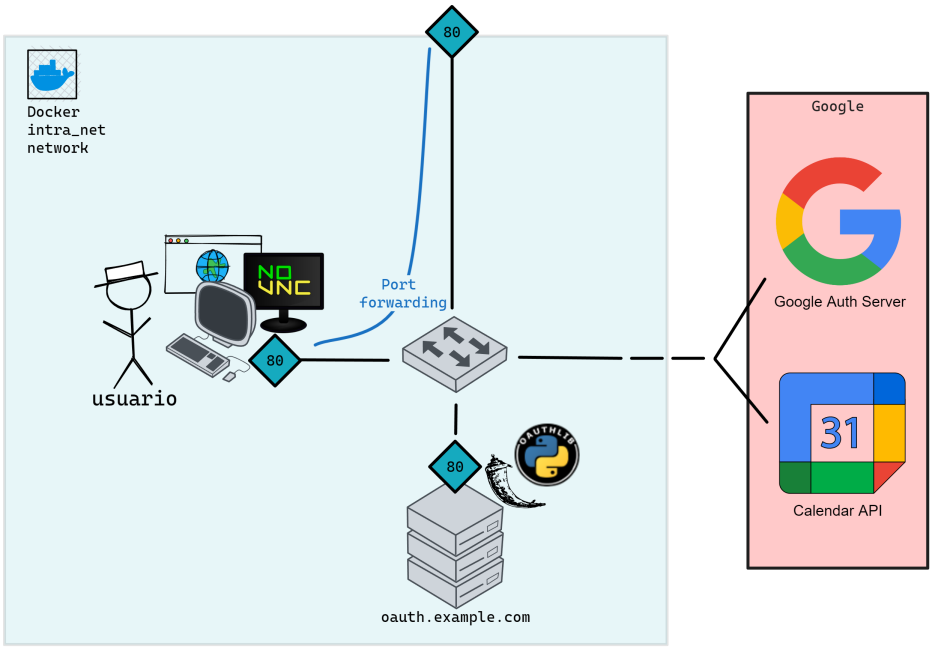

Figura 4.4: Esquema general del escenario para OAuth 2.0 y OIDC

aplicación web va a mostrarle al usuario que haya realizado la autorización una lista de los eventos que tiene en su calendario de Google. Para que este cliente OAuth funcione correctamente, se tuvo que registrar previamente como cliente OAuth en la página que Google habilita para ello (`https://console.cloud.google.com/`). En ella, una vez realizado el registro, es posible obtener el `client_id` y `client_secret` que se escribirán como variables en un fichero `config.py` para que sean utilizadas por el cliente OAuth a la hora de realizar el *Token Request* y que, de esta manera, el Authorization Server pueda autenticar al cliente OAuth, evitando así que otro cliente pueda hacerse pasar por él.

En la parte del contenedor del usuario, se ha instalado Ubuntu con escritorio y noVNC junto a programas como Firefox o Wireshark, aunque mayormente el análisis se realizará gracias a las trazas generadas en los *logs* del servidor ya que es obligatorio el uso de HTTPS con el servidor de Google. Para que sea posible el acceso a este contenedor, se hace una *forwarding* de su puerto original 80 al puerto 80 fuera de la red interna, con el cual será accesible desde el *host*.

Con estos elementos el usuario puede lanzar el contenedor con el comando `docker-compose up`. El usuario que quiera realizar pruebas en el escenario tan solo tendrá que conectarse al contenedor introduciendo en su navegador de la máquina *host* la URL `http://localhost`, entrar en el navegador e introducir la URL del cliente OAuth (`http://oauth.example.com`).

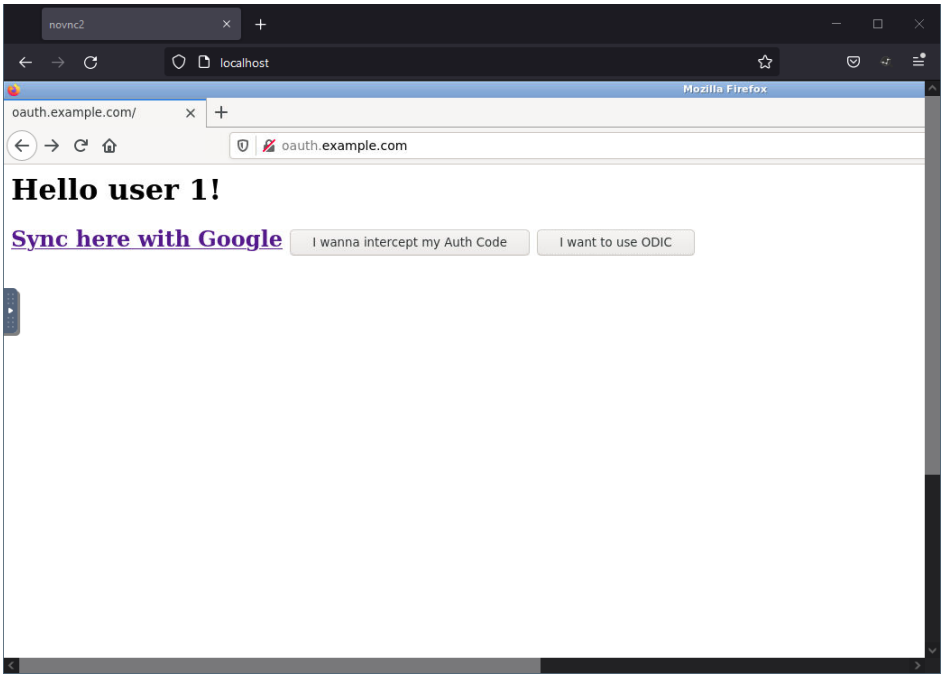

Figura 4.5: Página de inicio del cliente OAuth desde el navegador de la máquina *host*.

Como vemos en la figura 4.5, cuando el usuario accede a la página del cliente OAuth, se muestra un *link* llamado *Sync here with Google* donde el usuario puede clicar para poder empezar la sincronización de sus eventos para que se muestren en la página. Una vez presionado, el usuario es redirigido a la página de autenticación de Google. En este punto el usuario debe utilizar su correo y contraseña de su cuenta de Google para autenticarse y autorizar el acceso al cliente OAuth. Tras llevarse a cabo esto último, se produce una nueva redirección de vuelta a la página del cliente OAuth, el cual, al tener ya el *token* de acceso que necesitaba, mostrará la información de los eventos que tiene el usuario en su calendario. Este es el comportamiento básico del Authorization Code Grant en OAuth 2.0, el cual esta descrito en la sección 2.2.

Para implementar la parte de OIDC, al ser un protocolo complementario a OAuth 2.0, se ha optado por realizar modificaciones mínimas al código de manera que se pudiese activar este protocolo en cualquier momento. Para poder implementar las funciones de una manera más liviana se ha hecho uso de la libreria PyJWT, la cual ha ayudado a la hora de decodificar y comprobar el JWT de OIDC. Con esto y para poder mostrar las propiedades de OIDC, se ha dispuesto un botón llamado *I want to use OIDC* en la página principal del servicio. De esta manera, cuando el usuario pulse este botón, en vez de acceder a sus recursos directamente, se mandara una petición con OIDC y se mostrará parámetros devueltos por el ID Token (JWT) como son su cabecera, *payload* y firma digital realizada por el OpenID Provider de Google. Por último, también se informa de si se ha verificado correctamente el ID Token. En la sección 2.3 se dan mas

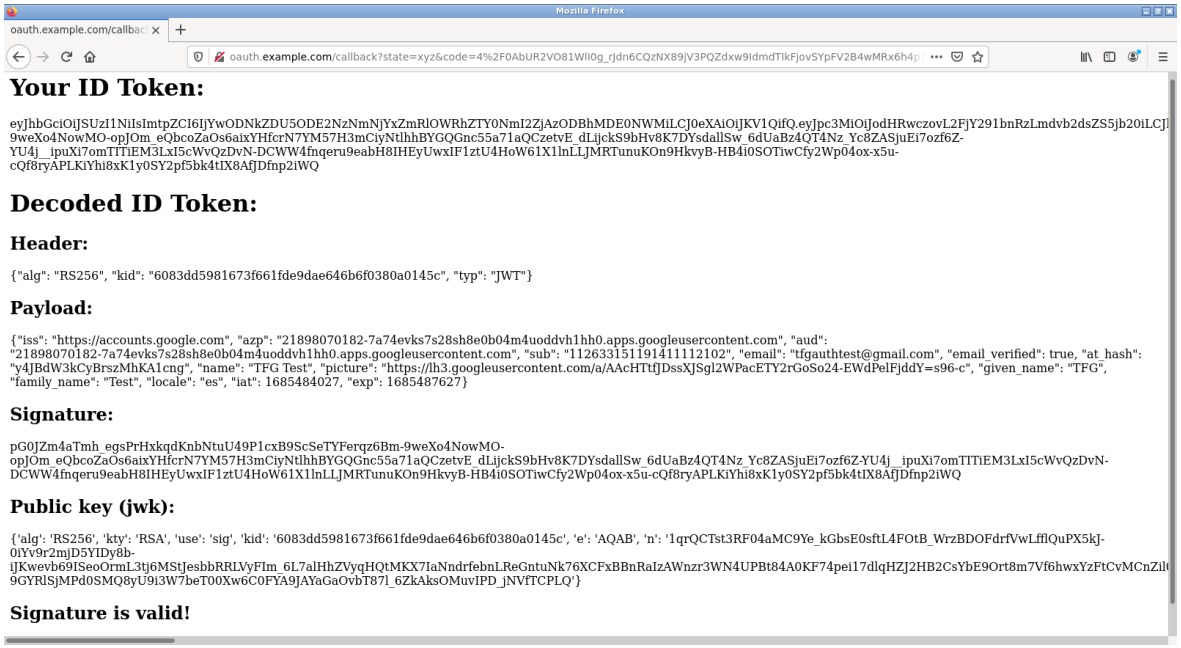

Figura 4.6: Visualización del ID Token en el escenario.

detalles acerca de la construcción de este ID Token y su firma.

5 Análisis de soluciones y pruebas de concepto

5.1 Análisis de seguridad de SAML

Pese a que SAML es un protocolo bien estudiado y establecido en muchos sistemas para dotar de SSO a las organizaciones, este puede presentar algunas cuestiones de seguridad a tener en cuenta a la hora de trabajar con él. Para comprobarlas, se ha hecho uso del escenario previamente descrito para realizar distintas pruebas siguiendo como guía las consideraciones de seguridad ofrecidas por la propia OASIS sobre SAML [12]. Destacar que este escenario hace uso de los *bindings* HTTP Redirect/POST. Para otros tipos de *bindings*, los ataques pueden ser diferentes.

5.1.1 Intercambio básico

Para conocer mejor cómo funcionan los elementos que proporcionan seguridad a SAML, vamos a realizar un análisis de los mensajes que podemos encontrar en un intercambio básico en este escenario.

En primer lugar, al estar usando un flujo iniciado por SP, tras clicar en el botón *Autenticate!* del SP1 se genera el primer mensaje de este flujo que irá dirigido al IdP por medio de una redirección HTTP del navegador del usuario. De hecho, al ser una redirección HTTP, el navegador acaba generando una petición GET a la URL del código 5.1

Código 5.1: Petición de *SAML Request* hacia el IdP

```
http://idp.example.com/simplesaml/saml2/idp/SSOService.php?
SAMLRequest=nVJda9swFP0rRu+O49RLYpEE0obRQNeGOttDX8a1fNsI9OHqXq/
    dv59sd6MbLA8FIYlz7zn36KAVgTWt3HZ8cv
f43CFx8mqNIzkU1qILTnogTdKBRZKsZLX9ciNnk6lsg2evvBHvKOcZQISBtXci2e/
    W4nuJ5fxTCSovVaOa+VQtF/m8qGsoalXX
C4x4fbEoiwWI5BsGisy1iEKRTtTh3hGD4whNZxfpNE/z5TEvZLGM60Eku/
    ga7YAH1om5lVmmm3aCr2BbgxPlbUa6v/bOs36b9Q
1ZVd1VGH5ohZP21Ipk+9v2lXfUWQxv1a/3N3+Eqc3/
    J2x905lBKhsHjecsBUUD2uAjdIZTirMOb6Featdo93Q+z3psInl9PB7S
```

```
w111FJtVry2HfMLmQ+YsMjTA8I+3VfZeeTX+nNvoab87eKPVz+SzDxb4vOUe0U36OLRKDuBIo+
    MYsjH+5SogMK4Fhw5FthlH/v
0/N78A
&RelayState=http://sp1.example.com/auth.php?
```

En el parámetro `SAMLRequest` de la URL viaja comprimido y codificado en Base64 la carga XML del mensaje *SAML Authentication Request*. Si se decodifica y se descomprime, se puede ver el mensaje XML en el código 5.2. En él podemos ver cómo el destinatario es el IdP como se ve en el atributo `Destination`, el ID de la petición en el atributo `ID` (se referenciará luego en la IdP) y la identidad del SP1 que viene establecida en el campo `<saml:Issuer>`

Código 5.2: Mensaje SAML Authentication Request hacia el IdP

```
<samlp:AuthnRequest xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
      xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
      ID="_9e9659ac19cdcd60c87164bba4bcbb7e19cb37947a"
      Version="2.0"
      IssueInstant="2023-01-18T14:48:48Z"
      Destination="http://idp.example.com/simplesaml/saml2/idp/SSOService.php"
      AssertionConsumerServiceURL="http://sp1.example.com/simplesaml/module.php/saml/sp/saml2-acs
          ↪ .php/default-sp"
      ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
      >
   <saml:Issuer>http://sp1.example.com/simplesaml/module.php/saml/sp/metadata.php/default-sp</
       ↪ saml:Issuer>
   <samlp:NameIDPolicy Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"
         AllowCreate="true"
         />
</samlp:AuthnRequest>
```

Tras esto, el IdP comprueba que los datos de esta petición son correctos, es decir, tiene al SP1 registrado (indicado en `<saml:Issuer>`), utiliza un *binding* que soporta, no ha pasado demasiado tiempo desde `IssueInstant`, etc.

Una vez hecha esta verificación, se lleva a cabo la autenticación del usuario. En este caso, el usuario se autentica mediante su nombre de usuario y contraseña. Como la autenticación es correcta, el IdP le responde con `200 OK` en el que incluye la cabecera `Set-Cookies` junto a una *cookie* de autenticación que se almacena en el navegador del usuario y que le servirá al IdP para realizar el SSO posteriormente si se vuelve a realizar una petición desde un SP. Tras esto, el IdP genera un *script* que hace que el navegador del usuario realice un POST HTTP a la URL `AssertionConsumerServiceURL` que se indicó en la petición del código 5.2 del SP1.

En este POST HTTP viaja el mensaje *SAML Authentication Response*, el cual contiene el *Assertion* para poder comprobar la autenticación del usuario en el IdP. Al igual que en la petición del código 5.1, este mensaje va codificado en Base64, sin embargo, en este caso, el parámetro va en el cuerpo del mensaje al ser un POST HTTP y va sin comprimir. En el código 5.3 se muestra el contenido de este mensaje decodificado.

Código 5.3: Mensaje SAML Authentication Response hacia el SP1

```
<samlp:Response xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
      xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
      ID="_c17a870a015717e64252aee831cb1b60d65a6e0412"
      Version="2.0"
      IssueInstant="2023-01-18T14:48:57Z"
      Destination="http://sp1.example.com/simplesaml/module.php/saml/sp/saml2-acs.php/default-sp"
      InResponseTo="_9e9659ac19cdcd60c87164bba4bcbb7e19cb37947a"
      >
   <saml:Issuer>http://idp.example.com/simplesaml/saml2/idp/metadata.php</saml:Issuer>
   <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
   ** certificado del IdP y firma digital del mensaje **
   </ds:Signature>
      <samlp:Status>
      <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success" />
   </samlp:Status>
   <saml:Assertion xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         ID="_2b742ba24395a87b23e4fdb5174ffed22ecf91cb18"
         Version="2.0"
         IssueInstant="2023-01-18T14:48:57Z"
         >
      <saml:Issuer>http://idp.example.com/simplesaml/saml2/idp/metadata.php</saml:Issuer>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      ** certificado del IdP y firma digital del Assertion **
      </ds:Signature>
      <saml:Subject>
         <saml:NameID SPNameQualifier="http://sp1.example.com/simplesaml/module.php/saml/sp/↩
              ↪ metadata.php/default-sp"
             Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"
             >_6ab79b2daf179592bccc67c0c89456042f1dd8d974</saml:NameID>
         <saml:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
            <saml:SubjectConfirmationData NotOnOrAfter="2023-01-18T14:53:57Z"
                 Recipient="http://sp1.example.com/simplesaml/module.php/saml/sp/saml2-acs.php/↩
                      ↪ default-sp"
                 InResponseTo="_9e9659ac19cdcd60c87164bba4bcbb7e19cb37947a"
                 />
         </saml:SubjectConfirmation>
      </saml:Subject>
      <saml:Conditions NotBefore="2023-01-18T14:48:27Z"
          NotOnOrAfter="2023-01-18T14:53:57Z"
          >
         <saml:AudienceRestriction>
            <saml:Audience>http://sp1.example.com/simplesaml/module.php/saml/sp/metadata.php/↩
                 ↪ default-sp</saml:Audience>
         </saml:AudienceRestriction>
      </saml:Conditions>
      <saml:AuthnStatement AuthnInstant="2023-01-18T14:48:57Z"
          SessionNotOnOrAfter="2023-01-18T22:48:57Z"
          SessionIndex="_82dd7a4470a71db400f42d9e0d299547a7785dcaa0"
          >
         <saml:AuthnContext>
            <saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password</↩
                 ↪ saml:AuthnContextClassRef>
         </saml:AuthnContext>
      </saml:AuthnStatement>
      <saml:AttributeStatement>
         <saml:Attribute Name="urn:oid:0.9.2342.19200300.100.1.1"
            NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
            >
            <saml:AttributeValue xsi:type="xs:string">employee</saml:AttributeValue>
         </saml:Attribute>
         <saml:Attribute Name="urn:oid:1.3.6.1.4.1.5923.1.1.1.1"
```

```
59              NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
60              >
61              <saml:AttributeValue xsi:type="xs:string">member</saml:AttributeValue>
62              <saml:AttributeValue xsi:type="xs:string">employee</saml:AttributeValue>
63          </saml:Attribute>
64      </saml:AttributeStatement>
65   </saml:Assertion>
66</samlp:Response>
```

En primer lugar se puede destacar el atributo `InResponseTo`, que tiene el mismo valor que el `ID` de la petición del código 5.2, indicando que se está respondiendo a esa petición.

El campo `<saml:Issuer>` indica la identidad del IdP, que es quien ha generado este mensaje.

A continuación, en el campo `<ds:Signature>` (cuyo contenido se ha omitido debido a su longitud) se indica la firma digital del mensaje realizada con la clave privada del IdP y el certificado X.509 asociado a ella codificado en Base64. Esta firma dota de integridad a todo el mensaje y autentica al IdP que es quien se espera que lo haya producido.

Justo después, se puede identificar el campo `<samlp:Status>`, en el cual, el IdP indica si la autenticación ha sido satisfactoria. En este caso vemos como la propiedad `StatusCode` toma el valor de `Success` por lo que es una autenticación exitosa por parte del usuario.

Tras esto, se entra en la parte del *Assertion*, indicado con el campo `<saml:Assertion>`. En esta parte del XML podemos encontrar otra firma. Esta es similar a la anterior, sin embargo, esta vez es una firma que asegura la integridad del *Assertion* y no el mensaje entero como la anterior.

En el campo `<saml:Subject>` podemos encontrar información que identifica al usuario que se ha autenticado. En concreto, el campo `<saml:NameID>` se puede observar el ID con el que el IdP identifica al usuario en su sistema.

Dentro del *Assertion* también encontramos el campo `<saml:Conditions>` el cual indica en el atributo `NotBefore` y `NotOnOrAfter` que el plazo de validez del *Assertion* es del `2023-01-18T14:48:27Z` a `2023-01-18T14:53:57Z` y en su campo `<saml:Audience>` se indica que esta *Assertion* va dirigida al SP1.

En el campo `<saml:AuthnContextClassRef>`, dentro del campo `<saml:AuthnStatement>` podemos ver que el tipo de autenticación que se ha realizado ha sido mediante contraseña.

Por otra parte, en el campo `<saml:AttributeStatement>` podemos encontrar los distintos atributos que el IdP proporciona sobre el usuario autenticado al SP1. El primero de ellos representa al atributo `Userid` que el IdP utiliza y que toma como

valor `employee`, el nombre de usuario que tiene este usuario. El segundo corresponde con `eduPersonAffiliation`, con él se está indicando que el usuario tiene los cargos de `member` y `employee` en el registro del IdP.

Este mensaje llega al SP1 gracias al navegador del usuario. En este momento, SP1 comprueba la firma digital del mensaje y de la propia *Assertion* con el certificado que tiene del IdP en los metadatos, comprueba si el IdP ha indicado que ha sido una autenticación exitosa y realiza comprobaciones adicionales como puede ser el tiempo de validez del mensaje y el *Assertion*.

Tras hacer todas las comprobaciones, el SP1 otorga acceso al usuario y le muestra el recurso al que quiere acceder.

Como vemos, esta respuesta puede llegar a contener bastante información acerca del usuario por eso se deben tener en cuenta las consideraciones de seguridad que se mencionan a continuación.

5.1.2 Autenticación, confidencialidad e integridad

En el apartado de la autenticación, un actor SAML no puede saber por sí solo si el servidor con el que se está comunicando es quién dice ser. En este sentido, el protocolo recomienda apoyarse en protocolos como TLS [13] e IPSec [14] para proporcionar esta autenticación entre los actores a nivel de sesión. Sin embargo, a nivel de mensaje, el protocolo recomienda hacer uso del XML Signature [15] para poder firmar los mensajes transmitidos de manera que se pueda autenticar a las diferentes partes mediante el uso de una firma digital.

En la parte de la confidencialidad de los mensajes, el protocolo SAML no proporciona un método para hacer que estos no sean entendibles por otras entidades a las que no van destinados. En este aspecto, se vuelve a recomendar el uso de TLS o de IPSec para proporcionar confidencialidad a nivel de sesión. A nivel de mensaje, se recomienda hacer uso de XML Encription [16] para asegurar la confidencialidad de los mensajes XML.

Respecto a la integridad de los mensajes, se recomienda el uso de TLS o IPSec a nivel de conexión y el uso de XML Signature para asegurar la integridad a nivel de mensaje.

Mencionar que en nuestro escenario no se hace uso de ningún mecanismo de protección a nivel de conexión como puede ser TLS o IPSec. Debido a esto, todos los mensajes no tienen ningún tipo de autenticación, confidencialidad ni integridad a nivel de conexión. Sin embargo, a nivel de mensaje si se hará uso de XML Signature cuando se realiza la firma de *Assertions* por parte del IdP, dotándolos de autenticación e integridad.

5.1.3 Robo de *Assertion*

Si un agente malicioso es capaz de hacer una copia de la SAML Response generada por el IdP para cierto usuario, entonces, este agente puede hacer uso de esta *Assertion* y acceder al SP al que se ha accedido el usuario con esa *Assertion*.

Este caso podría darse en sistemas como el que tenemos montado en nuestro escenario, en los que no se hace uso de TLS u otros métodos para asegurar la confidencialidad de los mensajes entre el servidor y el navegador del usuario. En este caso, si por ejemplo, un agente malicioso es capaz de escuchar los mensajes que se están transmitiendo por la red podría materializar un ataque.

A continuación, se describe una pequeña prueba de concepto de este ataque: En primer lugar, se ha configurado el *proxy* de BurpSuite en el navegador Firefox que va a utilizar el usuario para realizar la autenticación. Una vez se ha autenticado el usuario, gracias al historial de BurpSuite llevamos la respuesta que contiene la *Assertion* a la herramienta *Repeater* donde podremos realizar la petición de nuevo.

Tras esto, se activa en esta herramienta las opciones de procesar *cookies* y seguir redirecciones.

Por último, se realiza el envío al SP1 del POST interceptado por parte del IdP que contiene la *Assertion*. Como vemos en la figura 5.1, el resultado es que el SP1 nos asigna ya su propia *cookie* de autenticación y nos muestra la página de autenticación correcta.

Para solventar este problema, aparte de hacer uso de TLS, IPSec o XML Encryption como herramientas para dotar de confidencialidad al mensaje, se proponen estas medidas:

- Se debe asegurar que el IdP y el SP tengan un reloj bien sincronizado.
- Se debe utilizar los valores `NotBefore` y `NotOnOrAfter` mas ajustados posibles para que haya una ventana de tiempo pequeña para usar la *Assertion*.
- El SP debe comprobar el periodo de validez de todos los *Assertions* obtenidos.
- Opcionalmente se propone que el SP pueda utilizar el *statement* `SubjectLocality` (si se incluye) para comprobar la IP del navegador con la IP de contenida en el *Assertion*.

Para constatar que la comprobación del periodo de validez solventa este problema, se ha dejado un tiempo y se ha vuelto a enviar el POST robado. En este caso, el SP1 nos informa de que se ha producido un error, denegando así el *Assertion* robado por no cumplirse el periodo de validez que estaba establecido en él. Concretamente SimpleSAML indica el siguiente error: `Error validating SubjectConfirmation in Assertion: NotOnOrAfter in SubjectConfirmationData is in the past: 1685303769`

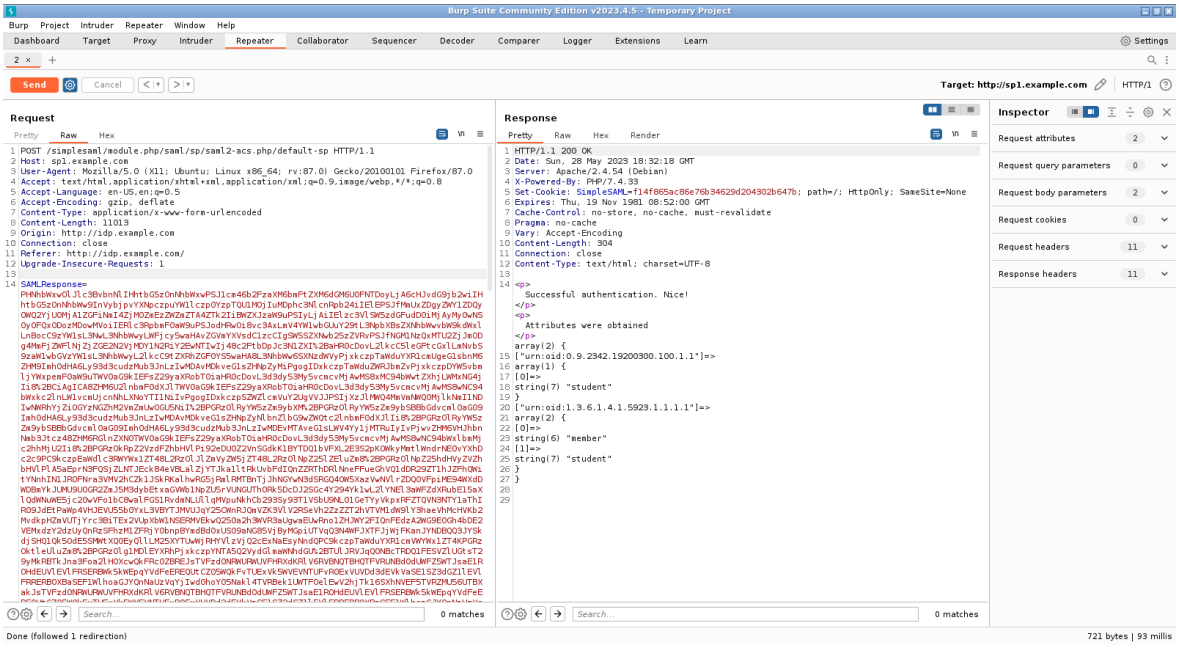

Figura 5.1: Envío de la copia del SAML Response interceptado y la respuesta afirmativa del SP1.

5.1.4 *Assertion* falsificado

Se puede dar el caso de que un agente malicioso quiera generar o modificar una *Assertion* y enviarla al SP para obtener acceso al servicio. Por ejemplo: autenticarse con una identidad que no es suya, cambiar sus atributos, etc.

A continuación vamos a describir una pequeña prueba de concepto en la que un hipotético usuario malicioso, que tiene ya una identidad registrada en el IdP, toma una *Assertion* generada de manera correcta, pero la modifica para intentar tener más privilegios en el SP1.

Lo primero que se hace es obtener el mensaje *SAML Response* que se ha generado en el IdP con la ayuda de BurpSuite. Posteriormente se lleva este POST a la herramienta *Repeater* donde se modifica, gracias al *plugin* de SAML Raider, uno de los atributos que están contenidos en la *Assertion*. En concreto, se cambia el atributo `student` por `admin`, en un intento de ganar más privilegios con este cambio.

En la figura 5.2 se muestra esta petición modificada y la respuesta que devuelve el SP1. Como vemos, el SP1 se da cuenta de la modificación y devuelve el mensaje de error `Reference validation failed` que indica que la validación de la firma no se ha realizado correctamente.

Con este ejemplo, vemos la utilización del método propuesto por SAML para evitar este tipo de ataque, que consiste en el deber de los SP de comprobar la validez de la

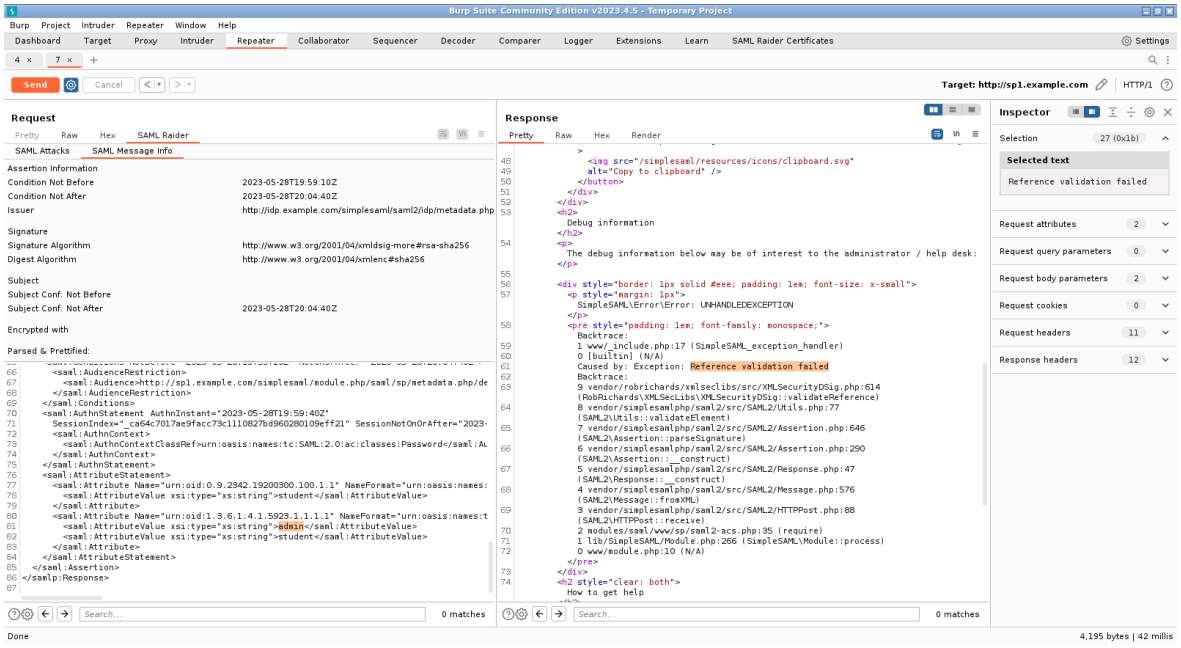

Figura 5.2: Modificación del *SAML Response* generado y la respuesta del SP1

firma y autenticar al emisor.

5.1.5 Conclusión

Como se ha visto, SAML presenta algunos detalles que deben ser tomados en cuenta por parte de los implementadores y los desarrolladores que hacen uso de él. La mayoría de estos ataques pueden ser solventados haciendo uso de soluciones como TLS, IPSec, XML Signature o XML Encryption. Sin embargo, se han de tener en cuenta las indicaciones de seguridad dadas por el protocolo. También se deben destacar los problemas encontrados para el uso del perfil Browser/Artifact [17] a los cuales OASIS dio una respuesta [18] aclarando dichos problemas.

5.2 Análisis de seguridad de OAuth 2.0

Anteriormente se ha explicado como OAuth 2.0 permite que los usuarios den acceso a sus recursos en línea de forma segura y controlada. Sin embargo, como cualquier protocolo, presenta ciertos riesgos de seguridad que deben ser considerados y abordados adecuadamente. En esta sección, exploraremos algunas de las consideraciones de seguridad más importantes para OAuth 2.0.

5.2.1 Intercambio básico

En esta sección se van a mostrar las trazas de lo que representa un intercambio básico con el flujo Authorization Code que se usa en el escenario a fin de identificar los distintos elementos que introduce OAuth 2.0 y que dotan de seguridad al protocolo.

En primer lugar, el flujo comienza con el RO accediendo a un recurso para el cual el cliente OAuth necesita una autorización delegada. Para este caso, el cliente OAuth representa una aplicación de visualización de eventos de Google Calendar del RO. En este momento, el cliente genera una redirección HTTP y envía al RO al Authorization Server. Este mensaje del código 5.4 se llama *Authorization Request* y, como vemos, incluye el parámetro `client_id` que identifica a este cliente OAuth, el `redirect_uri` que incluye la URI en la que el cliente va a recibir la posterior respuesta del Authorization Server y el `scope` en el que podemos identificar los permisos de información del usuario y la lectura de eventos.

Código 5.4: *Authorization Request* en el escenario

```
https://accounts.google.com/o/oauth2/v2/auth?response_type=code
&client_id=21898070182-7a74evks7s28sh8e0b04m4uoddvh1hh0.apps.↩
    ↪ googleusercontent.com
&redirect_uri=http://oauth.example.com/callback
&scope=https://www.googleapis.com/auth/userinfo.email https://www.↩
    ↪ googleapis.com/auth/userinfo.profile https://www.googleapis.com/↩
    ↪ auth/calendar.events.readonly https://www.googleapis.com/auth/↩
    ↪ calendar.readonly
&state=xyz
```

Tras esto, el Authorization Server de Google realiza ciertas comprobaciones para asegurar que el cliente existe y la URI de redirección es correcta. Una vez comprobadas, el Authorization Server le indica al RO que introduzca sus credenciales de acceso de Google y que autorice al cliente a utilizar los recursos que se indican con el campo `scope`.

Tras la autorización del RO, el Authorization Server genera una redirección HTTP al navegador del RO enviándolo a la URI de redirección del cliente (que se puede verse

en el código 5.4). Este mensaje se llama *Authorization Response* y, como vemos en el código 5.5 permite que el cliente pueda obtener el Authorization Code enviado por el Authorization Server el cual está incluido en el parámetro `code`.

Código 5.5: *Authorization Response* en el escenario

```
http://oauth.example.com/callback?state=xyz
&code=4/0AVHEtk5DuMo-↩
    ↪ df2W_TVrHTuavD7k76MwqPjFXDBB_MjHjNsiJyjW9GLDgTb_VFF0KEB0MQ
&scope=email profile https://www.googleapis.com/auth/calendar.events.↩
    ↪ readonly openid https://www.googleapis.com/auth/userinfo.email ↩
    ↪ https://www.googleapis.com/auth/userinfo.profile https://www.↩
    ↪ googleapis.com/auth/calendar.readonly
&authuser=0
&prompt=none
```

Con este Authorization Code, ahora el cliente generará un POST HTTP para poder pedir un Access Token. Este mensaje llamado *Token Request* se envía directamente al Authorization Server sin interacción del navegador del RO. En el código 5.6 vemos como en el parámetro `code` se envía el código que se ha obtenido anteriormente. Además, vemos como el cliente se autentica gracias al parámetro `client_secret`.

Código 5.6: *Token Request* en el escenario

```
https://www.googleapis.com/oauth2/v4/token
{'Content-Type': 'application/x-www-form-urlencoded'}
grant_type=authorization_code
&client_id=21898070182-7a74evks7s28sh8e0b04m4uoddvh1hh0.apps.↩
    ↪ googleusercontent.com
&client_secret=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
&code=4/0AVHEtk5DuMo-↩
    ↪ df2W_TVrHTuavD7k76MwqPjFXDBB_MjHjNsiJyjW9GLDgTb_VFF0KEB0MQ
&redirect_uri=http://oauth.example.com/callback
```

Este último mensaje llega al Authorization Server y comprueba que el `client_secret` es correcto para ese cliente. Además, comprueba que el Authorization Code no haya sido utilizado previamente. Tras realizar todas las comprobaciones, el Authorization Server responde al cliente. En el código 5.7 podemos ver la respuesta afirmativa del Authorization Server con un JSON que incluye el Access Token en el campo `access_token`, que es de tipo `Bearer` según el campo `token_type` y su tiempo de expiración, en este caso `3599` segundos (una hora). Este tipo Bearer nos indica que el portador del Authorization Code ha conseguido el `access_token` mediante el `client_id` y `client_secret` y no es necesaria más autenticación. No obstante, se podría estar utilizándose el tipo MAC, que consiste en que el cliente OAuth tiene que autenticarse frente al Resource Server con la respuesta a un reto antes de poder ob-

tener el Access Token. Como vemos, en la respuesta se ha incluido también el campo `id_token` y la cadena `openid` en el campo `scope`. Esto es debido a que el Authorization Server de Google fuerza el uso de OIDC, los omitiremos por ahora.

Código 5.7: *Token Response* en el escenario

```
200 OK
{** cabeceras **}
{
  "access_token": "ya29.a0Ael9sCOFa3sKW6PU5gQjUCdiud9gu
  VskQz5Ao73vsxwVsRyecHOYYkkSoS_pBdQ6d3kK2dSiRw-Ikg2UNfOGJr
  12z7UA1BD3eAOj1pIi_W8wgvXiGWMJgXKfwYYW3WEn2SJAuXGLlPzEQoEh-
  m87e0M5mulJOK0aCgYKAQ0SARISFQF4udJhKPxTXKpHjgwPpKHsGvAmQA0166",
  "expires_in": 3599,
  "scope": "openid https://www.googleapis.com/auth/calendar.readonly ↩
     ↪ https://www.googleapis.com/auth/calendar.events.readonly https↩
     ↪ ://www.googleapis.com/auth/userinfo.email https://www.↩
     ↪ googleapis.com/auth/userinfo.profile",
  "token_type": "Bearer",
  "id_token": "eyJhb..."
}
```

Tras esta respuesta, el cliente es capaz de realizar peticiones para acceder a los recursos que permiten los *scopes* que se han pedido. En el código 5.8 vemos como se hace uso del Access Token obtenido anteriormente para obtener el recurso `/calendar/v3/calendars/primary/events` del RO en el Resource Server de Google. Destacar cómo el Access Token va incluido en esta petición en la cabecera `Authorization` presentando así el cliente la prueba de que ha sido autorizado por el RO para acceder a este recurso.

Código 5.8: Petición de un recurso del RO en el escenario

```
https://www.googleapis.com/calendar/v3/calendars/primary/events
{'Authorization': 'Bearer ya29.a0Ael9sCOFa3sKW6PU5gQjUCdiud9guV
skQz5Ao73vsxwVsRyecHOYYkkSoS_pBdQ6d3kK2dSiRw-Ikg2UNfOGJr12z7UA
1BD3eAOj1pIi_W8wgvXiGWMJgXKfwYYW3WEn2SJAuXGLlPzEQoEh-m87e0M5mul
JOK0aCgYKAQ0SARISFQF4udJhKPxTXKpHjgwPpKHsGvAmQA0166'}
```

Finalmente, el Resource Server comprueba si el cliente puede acceder a ese recurso con ese Access Token y responde a la petición con el recurso solicitado.

5.2.2 Suplantación de identidad del cliente

Un cliente OAuth malicioso podría querer hacerse pasar por otro legítimo y obtener acceso a recursos protegidos. Por ejemplo, un posible atacante de un sitio web podría

crear un cliente T el cual utilizase la identidad de un cliente A en el cual el usuario ya confía y le ha dado permisos. Mediante una redirección, T podría obtener un Authorization Code para obtener un Access Token y obtener acceso haciéndose pasar por A.

Este riesgo es mitigado gracias a que normalmente existe un Client Secret el cual le sirve al cliente para autenticarse frente al Authorization Server a la hora de solicitar el Access Token. Además, el Authorization Server debe hacer que el cliente OAuth tenga que registrar una URL de redirección la cual tendrá que coincidir con la de la petición *Authorization Request*. La seguridad de este sistema pasa porque se da por hecho que el cliente OAuth es capaz de almacenar sus credenciales de manera segura.

En el código 5.9, se muestra un ejemplo de la petición con un `client_id` y un código de autorización correcto que el Authorization Server rechaza por un secreto de cliente erróneo:

Código 5.9: *Token Request con un Client Secret erróneo*

```
[CLIENT -> AUTH SERVER] TOKEN REQUEST:
https://www.googleapis.com/oauth2/v4/token
{'Content-Type': 'application/x-www-form-urlencoded'}
grant_type=authorization_code
&client_id=21898070182-7a74ev28sh8e0b04m4uovh1hh0.apps.googleusercontent.com
&client_secret=erroneo
&code=4/0AVHEtk63J6iogTeOOqWF6yxfMew_lKjQTQwtn958S4hXkNY7q1-AwYj9kWyQw
&redirect_uri=http://oauth.example.com/callback

[AUTH SERVER -> CLIENT] TOKEN RESPONSE:
401
https://www.googleapis.com/oauth2/v4/token
{**cabeceras**}
{
  "error": "invalid_client",
  "error_description": "Unauthorized"
}
```

En siguiente código 5.10 se muestra un ejemplo de la respuesta dada por el Authorization Server ante una petición `Token Request` sin un `client_secret` :

Código 5.10: *Token Request sin un Client Secret*

```
[CLIENT -> AUTH SERVER] TOKEN REQUEST:
https://www.googleapis.com/oauth2/v4/token
{'Content-Type': 'application/x-www-form-urlencoded'}
grant_type=authorization_code
&client_id=21898070182-7a74ev28sh8e0b04m4uovh1hh0.apps.
```

```
    ↪ googleusercontent.com
&code=4/0AVHEtk6BCoR4N_mKcWiHfuDQTN4VGV-bMml7fARRhXtzR12_4Bp8XSMR-↩
    ↪ QICkJnWQBnSow
&redirect_uri=http://oauth.example.com/callback

[AUTH SERVER -> CLIENT] TOKEN RESPONSE:
400
https://www.googleapis.com/oauth2/v4/token
{**cabeceras**}
{
  "error": "invalid_request",
  "error_description": "client_secret is missing."
}
```

Vemos como en este caso se devuelve un error 400 indicando que la petición no es válida e indicando en el cuerpo del error que el causante es la falta de un `client_secret`.

Con este mecanismo se asegura que no se puede impersonar a un cliente OAuth.

5.2.3 Manipulación de la URI de redirección con el Authorization Code

Imaginemos que el cliente OAuth es una aplicación web (`http://amzingedition.com/`) para la edición de fotos *online* que permite que se importen fotos desde la cuenta de Google Drive del usuario y un atacante controla un sitio web (`http://attack.you/`).

El atacante podría registrar una cuenta en la aplicación cliente y empezar el flujo de autorización generando una *Authorization Request*. Después, el atacante coge el *link* generado por parte del cliente y sustituye la URI de redirección hacia el *endpoint* del *callback* del cliente (`http://amzingedition.com/callback`) por con otra URI la cual apunta al sitio web bajo su poder (`http://attack.you/trap`). Un ejemplo de como podría verse este *link* generado por el atacante sería `https://accounts.google.com`↩
↪ `/oauth2?response_type=code&client_id=amazingeditclientid&redirect_uri=`↩
↪ `http://attack.you/trap&scope=photos&state=safestate`

Ahora, si el atacante es capaz de hacer que la víctima utilice este *link* malicioso que ha generado con su URI de redirección, entonces la víctima será redirigida a `http://attack.you/trap` con el Authorization Code como parámetro, luego el atacante obtendría este código en su propio sitio web. Con este código, ahora el atacante puede enviar una petición a la URI de redirección del cliente de edición de fotos (`http://amzingedition.com/callback`) con el código obtenido. En este punto, el editor de fotos intercambiará este Authorization Code robado por el atacante pero

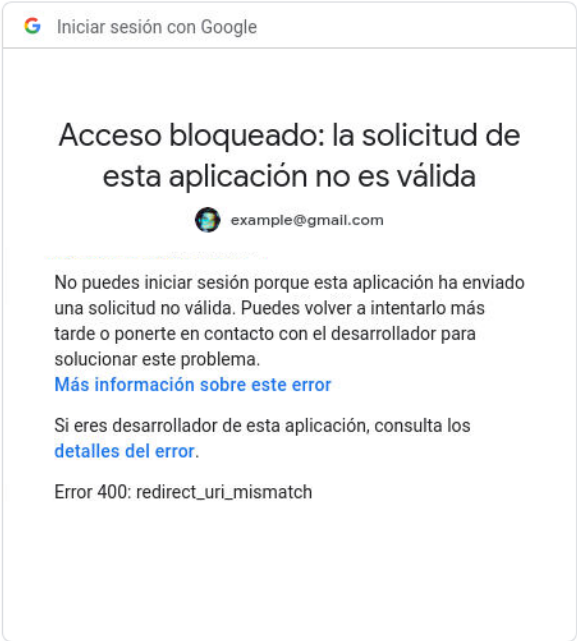

Figura 5.3: Respuesta del Authorization Server ante una URI de redirección no registrada.

generado por la víctima por un Access Code. Con este código, el editor empezará a cargar las fotos del Google Drive de la víctima en la aplicación web y el atacante será capaz de acceder a todas ellas a través de este editor.

La prevención de este ataque pasa porque los Authorization Server tienen que obligar a los clientes a presentar en la *Authorization Request* una URI de redirección idéntica a la que van a usar para intercambiar el Access Token. Además, se recomienda para clientes confidenciales (ej. servidores web) que se registre la URI de redirección del cliente en el momento de su registro. Cuando se incluye esta URI en las peticiones, estas se tienen que validar con las URI previamente registradas.

Cuando el Authorization Server no consigue verificar que la URI de redirección es una de las registradas, cancela el flujo de autorización y, como consecuencia, informa de esto al usuario con un error como el que se ve en la figura 5.3.

5.2.4 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) es un ataque que consiste en que un atacante hace que el navegador de una víctima siga una URI maliciosa a un servicio en el cual la víctima confía.

El atacante sustituye su Authorization Code o Access Code en la URI de redirección al cliente. En este momento el atacante intentará por algún medio que el usuario clique

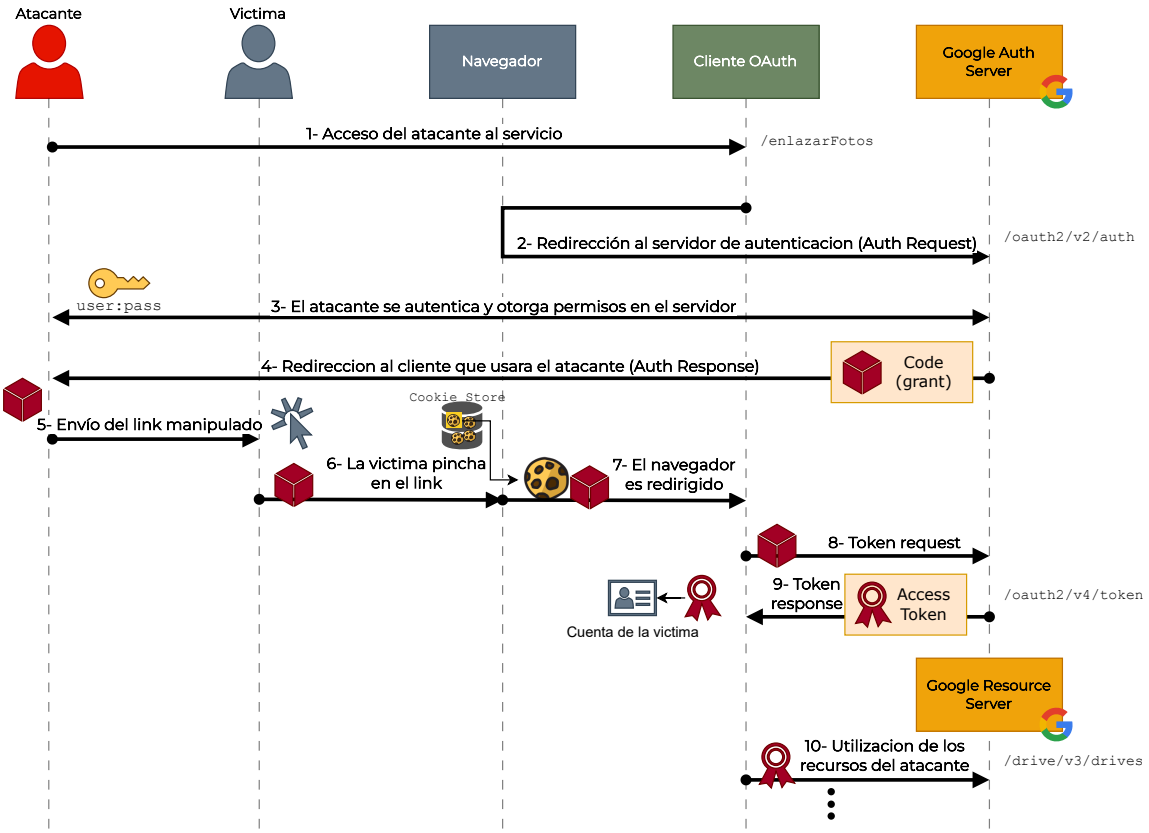

Figura 5.4: Diagrama de secuencia de un posible ataque de CSRF

en *link* que ha generado él y que apunta a la URI de redirección con el Authorization Code del atacante.

Con este proceso, y normalmente con una *cookie* de sesión previamente establecida por parte de la víctima en el cliente, el navegador de esta víctima seguiría la URI del atacante. En este momento, el cliente OAuth continuará con el flujo de OAuth y acabará asociando los recursos del atacante a la cuenta de la víctima en el cliente OAuth.

Por ejemplo, imaginemos que hubiese un servicio que guardase tus fotos (este sería el cliente OAuth) y tuviese un botón de conectar con Google para enviar las fotos a tu Google Drive. En este caso, si la víctima utilizase la URI proporcionada para el atacante, entonces estaría utilizando los recursos de Google Drive del atacante enviando todas sus fotos privadas guardadas en la aplicación del cliente OAuth.

En la figura 5.4 se muestra un diagrama de secuencia que ilustra el proceso de este ataque en OAuth 2.0.

Para poder realizar este ataque, el atacante debe obtener el Authorization Code antes

de que se mande la petición con este al cliente OAuth, es decir, debe interceptar la redirección. Esto es necesario porque si ocurriese de la otra manera y el cliente OAuth canjease el Authorization Code por un Access Code entonces no se podría volver a utilizar ese Authorization Code, ya que en el estándar se indica que el Authorization Code solo puede ser usado una vez y el Authorization Server debe aplicar esta regla.

La protección frente a este ataque consiste en que los clientes OAuth 2.0 tienen que usar el parámetro `state` y establecer en él un valor que asocie las peticiones al navegador que se este utilizando (p.e: el *hash* de una *cookie* de sesion). De esta manera el cliente OAuth debe utilizará el parámetro `state` para comprobar si el valor que le ha llegado en la petición coincide con el valor que está asociado al navegador.

En este caso se ha realizado una prueba de concepto para ver como se comporta el protocolo cuando no se está utilizando estos mecanismos de protección con el parámetro `state`. En primer lugar, tras autenticar al atacante, se ha interceptado la URI con la

Figura 5.5: URL de redirección al cliente generada por el atacante.

redirección al cliente OAuth que contiene su Authorizacion Code. Para que esto fuese más fácil, desde el servidor se ha programado que cuando es un usuario especial (el atacante) el que genera la petición, no se enviará directamente el Authorization Code, sino que se mostrara al usuario la URI que ha utilizado (figura 5.5). Con esto se simula un atacante interceptando la petición y se obtiene la URI de manera más cómoda.

Figura 5.6: Datos del atacante asociados a la víctima una vez accede al link.

Posteriormente, una vez generada la URI del atacante, se accede a la URI desde el navegador de la víctima. Como se ve en la figura 5.6, la aplicación cliente OAuth acaba asociando los datos del atacante a la cuenta de la víctima en la aplicación. En este caso, los datos del `User 3` (la víctima) acaban asociados a la cuenta de Google del atacante (`tfgauthrest@gmail.com`).

5.2.5 *Clickjacking*

El *clickjacking* es la técnica con la cual un atacante hace que un usuario haga una acción la cual no pretendía realizar. Esto se suele conseguir colocando un elemento invisible encima de la acción que el usuario va a cometer. En el caso de OAuth, esto puede desembocar en que un usuario clique en un botón de autorizar creyendo que esta clicando otro elemento como podría ser una imagen o un botón de reproducir. Es decir, un atacante estaría consiguiendo que el RO le este dando acceso al cliente OAuth malicioso sin saberlo.

Para repeler estos ataques, los navegadores modernos realizan un análisis de la cabecera `x-frame-options` que debe ser proporcionada por el Authorization Server al devolver las páginas. Si tiene el valor `deny` o `same-origin`, hará que el navegador no cargue la página pedida si está en un elemento `iframe`. Esto hace que el navegador no cargue la página para autorizar un acceso por lo que, aunque el usuario clique en la imagen, no surgirá ningún efecto.

Para comprobar esto, se ha realizado una pequeña prueba de concepto en la que se trata de realizar el ataque. Para ello, se ha creado un *endpoint* llamado `/clickjacking`, en el cual se cargase un `iframe` que hiciese referencia a la pagina del Authorization Server de Google a la que se redirige al usuario para dar permisos al cliente OAuh maligno. Esta página quedaría colocada tras una imagen que incitase a hacer clic en un botón falso, cuya posición coincidiría con el botón de permitir autorización que se cargaría por debajo. Esta URL para el `iframe` podría ser obtenida por el atacante fácilmente tras hacer el proceso de autorización en su propio cliente OAuth (en este caso estamos utilizando el mismo). Tras construir esta pagina con una imagen engañosa, simulamos que el usuario se ha autenticado previamente en el Authorization Server (ej. en la pagina de autenticación de Google) para que, de esta manera, se obtengan las *cookies* de autenticación por parte del servidor de Google y en la próxima petición de autenticación no tenga que volver a iniciar sesión. Después, se accede al *endpoint* donde esta situada la imagen engañosa con el `iframe` de autorización por debajo.

Como vemos en la figura 5.7, Firefox previene esto y nos muestra un mensaje de error indicando que no se pudo cargar. Esto es debido a que el Authorization Server de Google con el que se ha realizado la prueba, devuelve en su respuesta la cabecera `x-frame-options: deny`, por lo que Firefox analiza esa petición y no la carga debido a que está en un elemento `iframe`.

Si no estuviese esto, el `iframe` se habría cargado correctamente. Al presentar el navegador del usuario `cookies` de autenticación validas, el `iframe` pasaría a cargar automáticamente la pagina de autorización al cliente OAuth maligno para los permisos pedidos. Tras esto, el usuario podría verse tentado a clicar en el botón falso. Un ejemplo de esto podría ser una imagen diciendo que es el ganador de un premio y que debe clicar para obtenerlo. Una vez hecho clic, al seleccionar en realidad el botón de autorización

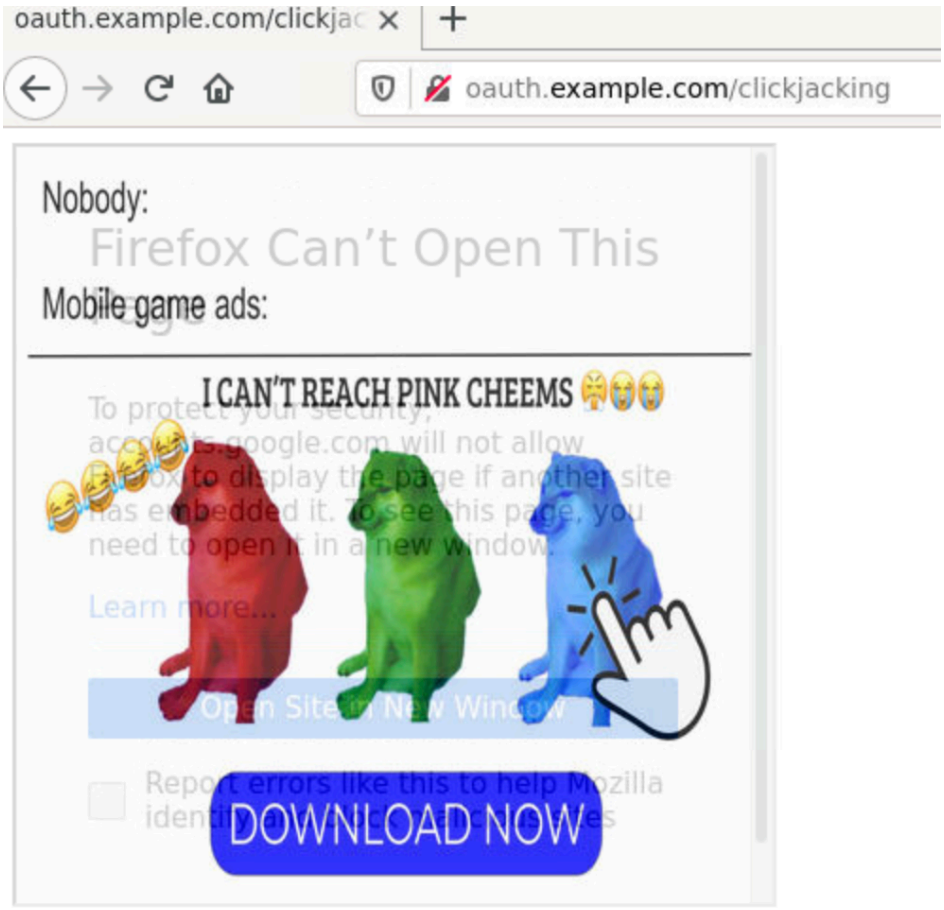

Figura 5.7: Intento de *clickjacking* detectado por Firefox.

al cliente fraudulento, el usuario estaría dándole permisos para utilizar sus recursos protegidos en el Resource Server de Google.

5.2.6 Conclusión

Como hemos analizado, OAuth 2.0 tiene diferentes mensajes y directrices que ayudan a realizar la labor de autorización delegada de una manera eficaz y segura. Concretamente, el Access Token es un concepto bastante potente para dar autorización sin tener que compartir credenciales con el cliente OAuth. Sin embargo, hemos comprobado como estos flujos son susceptibles a ciertos ataques que pueden ser mitigados gracias a la correcta utilización de parámetros como el `state` o cabeceras como `x-frame-options`. Se ha de destacar como los mensajes de OAuth en ningún momento indican información sobre la autenticación del usuario al cliente OAuth por lo que no se debe confiar en este protocolo para proporcionar un mecanismo de autenticación seguro.

5.3 Análisis de seguridad de OIDC

OIDC se construye sobre OAuth 2.0 y proporciona una capa adicional de autenticación. Sin esta capa, OAuth 2.0 solo permite la autorización delegada de los recursos del usuario. El cliente OAuth no tiene información sobre cómo se realizó la autenticación ni quién la llevó a cabo.

La introducción de OIDC resuelve esta limitación al proporcionar ID Tokens, que son pruebas de la autenticación de un usuario y están firmadas por el OpenID Provider (OP). Estos *tokens* permiten al cliente OAuth tener conocimiento de la autenticación del usuario y proporcionar una capa adicional de seguridad en el intercambio de información.

En esta sección, se enfocará en analizar las trazas generadas por OIDC en nuestro escenario de prueba. Aunque no pudimos realizar pruebas de concepto de ataques debido a restricciones de tiempo, analizaremos las trazas relacionadas con OIDC para ver el intercambio la información relacionada con la autenticación y el usuario en una implementación real.

5.3.1 Componentes básicos

En esta sección se va a mostrar los componentes básicos que ayudan a construir la capa de autenticación que ofrece OIDC y la cual se hace uso en el escenario construido. Mediante estas trazas se pueden mostrar los mensajes concretos que es capaz de generar este escenario.

En primer lugar, el protocolo OIDC comienza a aplicarse al incluir la cadena `openid` en el parámetro `scope` del mensaje *Authorization Request* de OAuth 2.0. Esta petición puede verse en el código 5.11 como resultado de pulsar el botón *I want to use OIDC* en el escenario.

Código 5.11: *Authorization Request* con OIDC en el escenario

```
https://accounts.google.com/o/oauth2/v2/auth?response_type=code
&client_id=21898070182-7a74evks7s28sh8e0b04m4uoddvh1hh0.apps.↩
    ↪ googleusercontent.com
&redirect_uri=http://oauth.example.com/callback
&scope=https://www.googleapis.com/auth/userinfo.email https://www.↩
    ↪ googleapis.com/auth/userinfo.profile https://www.googleapis.com/↩
    ↪ auth/calendar.events.readonly https://www.googleapis.com/auth/↩
    ↪ calendar.readonly openid
&state=xyz
```

Otra de las introducciones de OIDC puede mostrarse en el mensaje *Token Response* del código 5.12 enviado por el OP al RP. En este se puede observar el parámetro

`id_token` que contiene un JWT firmado por el OP que da información sobre el usuario que se ha autenticado.

Código 5.12: ID Token de un *Token Response* del escenario

```
200 OK
{** cabeceras **}
{
  "access_token": "ya29.a0AWY7CknR2RWs5XsdVMx0S2hERngSeFLviuLpI
  2zfcLn2_BqC6eOKbPHA_OARHGmlFwPpuJre1DJml_Etw-h1bo5biM_gK2uDI9
  cU4aIoOCQEgy5e_KY-QdxGKn2ztGlr0upto66r236mCtbwj5aeKIAlf38mbga
  CgYKAQASARASFQG1tDrpa4Mq96ISfF4U2P2tvv3rUA0165",
    ... mas contenido de OAuth 2.0 ...
  "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjYwODNkZDU5ODE2NzNmNj
  YxZmRlOWRhZTY0NmI2ZjAzODBhMDE0NWMiLCJ0eXAiOiJKV1QifQ.eyJpc3MiO
  iJodHRwczovL2FjY291bnRzLmdvb2dsZS5jb20iLCJhenAiOiIyMTg5ODA3MDE
  4Mi03YTc0ZXZrczdzMjhzaDhlMGIwNG00dW9kZHZoMWhoMC5hcHBzLmdvb2dsZ
  XVzZXJjb250ZW50LmNvbSIsImF1ZCI6IjIxODk4MDcwMTgyLTdhNzRldmtzN3M
  yOHNoOGUwYjA0bTR1b2RkdmgxaGgwLmFwcHMuZ29vZ2xldXNlcmNvbnRlbnQuY
  29tIiwic3ViIjoiMTEyNjMzMTUxMTkxNDExMTEyMTAyIiwiZW1haWwiOiJ0Zmd
  hdXRodGVzdEBnbWFpbC5jb20iLCJlbWFpbF92ZXJpZmllZCI6dHJ1ZSwiYXRfa
  GFzaCI6InZYMmFHb3h2Z3FWcHk2NO9JRlR6SlEiLCJuYW1lIjoiVEZHIFRlc3Q
  iLCJwaWN0dXJlIjoiaHR0cHM6Ly9saDMuZ29vZ2xldXNlcmNvbnRlbnQuY29tL
  2EvQUFjSFR0ZkpEc3NYSlNnbDJXUGFjRVRZMnJHb1NvMjQtRVdkUGVsRmpkZFk
  9czk2LWMiLCJnaXZlbl9uYW1lIjoiVEZHIiwiZmFtaWx5X25hbWUiOiJUZXN0I
  iwibG9jYWxlIjoiZXMiLCJpYXQiOjE2ODU0ODg3NDgsImV4cCI6MTY4NTQ5MjM
  0OH0.Rzqx_r31HDJW6LL4btNYfOdq_1MY1lV4SISVbNuc8m0zdwMMFVVilgSHo
  cRf5zp7-bCaf6CP3vmcq28trcVsyfij9R7rElOXd5KWRpD1m0h8gS3WEs9K2KB
  JyfkL9BG3ZIT4v91L7_bGCMJjrUq29sN3sqKgM7k1MnRIlMU9GiMgelkEwT5M2
  3PLBrjWz4eu_SOltavB22aa8gMMDM9fbESwRYwnxfFnNi3x3Wr4WELcGk85GHC
  -unCb-vA5BWUu5I0uL_onE0D5acNElFjVhqig"
}
```

Para comenzar a desmenuzar este JWT, debemos empezar por decodificar en Base64 la cabecera que es el contenido del ID Token hasta el primer ”.”. El resultado de la decodificación se muestra en el código 5.13. Como podemos ver, aquí se indica que el algoritmo seguido para la firma es RSA con SHA256 (`RS256`), y que el ID de la clave utilizada es el indicado por el campo `kid`.

Código 5.13: Cabecera JWT del ID Token del escenaro

```
{
    alg: "RS256",
    kid: "6083dd5981673f661fde9dae646b6f0380a0145c",
    typ: "JWT"
}
```

El segundo elemento de este JWT es la parte del *payload*. Esta parte está entre el primer y segundo '.' y está codificada en Base64. El resultado de su decodificación se muestra en el código 5.14, en el que podemos encontrar los siguientes campos:

- `iss`: Identificador del OP de Google que es quien emite este ID Token.
- `aud`: Identificador del RP que hemos construido que es a quien va dirigido este ID Token.
- `sub`: Identificador único del usuario que se ha autenticado.
- `iat` y `exp`: tiempo de emisión (2023-05-30T23:19:08.000Z) y caducidad (2023-05-31T00:00:47.000Z) del JWT.

Además de los campos propios que introduce Google, podemos destacar el campo `at_hash` que puede ser introducido opcionalmente en OIDC en los flujos Authorization Code como es el utilizado en el escenario. Este campo contiene el valor codificado en Base64URL de la mitad izquierda del *hash* de los octetos de la representación ASCII del Access Token. El algoritmo de *hash* que se usa es el de la cabecera del JWT. En este caso, la cabecera indica el algoritmo `RS256`, luego el *hash* está realizado con SHA-256.

Código 5.14: *Payload* JWT del ID Token del escenaro

```
{
    iss: https://accounts.google.com,
    azp: "21898070182-7a74evks7s28sh8e0b04m4uoddvh1hh0.apps.↩
        ↪ googleusercontent.com",
    aud: "21898070182-7a74evks7s28sh8e0b04m4uoddvh1hh0.apps.↩
        ↪ googleusercontent.com",
    sub: "112633151191411112102",
    email: "tfgauthtest@gmail.com",
    email_verified: true,
    at_hash: "vX2aGoxvgqVpy67OIFTzJQ",
    name: "TFG Test",
    picture: https://lh3.googleusercontent.com/a/↩
        ↪ AAcHTtfJDssXJSgl2WPacETY2rGoSo24-EWdPelFjddY=s96-c,
    given_name: "TFG",
    family_name: "Test",
    locale: "es",
    iat: 1685488748,
    exp: 1685492348
}
```

La firma digital de este JWT es la última parte de este ID Token. Su valor se encuentra a partir del último '.' del ID Token hasta el final. La URI de las claves que utiliza Google para realizar la firma se pueden encontrar en la URL `https://accounts.google.com/.well-known/openid-configuration` en campo `jwsk_uri`,

con esa URI se consigue desde el código Python obtener los certificados que se han utilizado para firmar este JWT. Como estas claves públicas pueden rotar, se debe elegir la que tiene el ID indicado en la cabecera del JWT en el campo `kid`.

5.3.2 Conclusión

Como se ha visto gracias al escenario construido, OIDC es capaz de establecer su capa de autenticación gracias al Token ID generado y transmitido al RP en el momento de la autenticación del usuario. Esto hace que el RP pueda informarse sobre ciertos parámetros del usuario autenticado y que pueda compra bar la veracidad de esta autenticación comprobando la firma digital de este *token*.

6 Conclusiones y vías futuras

En conclusión, hemos abordado y explorado tres importantes protocolos de autenticación y autorización en el contexto de la gestión de identidad digital. El análisis detallado de los protocolos SAML, OAuth 2.0 y OIDC nos ha permitido comprender sus características, fortalezas y limitaciones.

Para llevar a cabo este trabajo, hemos construido y desplegado dos entornos utilizando contenedores Docker, lo que nos ha brindado beneficios significativos, como la portabilidad y la capacidad de aislamiento de cada entorno de prueba. Esta elección nos ha permitido trabajar de manera más eficiente y cómoda al no requerir el traslado de sistemas completos, sino solo unos pocos archivos.

En el primer entorno, nos hemos centrado en SAML, utilizando la aplicación simpleSAMLphp y Apache para configurar un IdP y dos SP. Estos componentes nos han permitido probar y analizar la funcionalidad de SSO y federación de identidad, así como realizar pruebas de autenticación de usuarios y capturar y analizar el tráfico de red utilizando herramientas como Wireshark.

En el segundo entorno, nos hemos enfocado en OAuth 2.0 y OIDC. Hemos desarrollado un cliente OAuth que también actúa como un RP, utilizando las bibliotecas oauthlib y Flask. Mediante este entorno, hemos probado y verificado los flujos de autorización delegada, así como el acceso a recursos protegidos de un usuario, como Google Calendar. También hemos agregado funcionalidad para probar y verificar la implementación de OIDC, examinando el contenido de los ID Tokens generados y verificando su validez.

El diseño de estos escenarios pueden servir de base para crear laboratorios docentes que permitan analizar y configurar estas soluciones a nivel académico. Estos escenarios ya contienen una configuración básica de las librerías que permite un funcionamiento básico, por lo que se deja un entorno que es muy personalizable y que permite un despliegue bastante rápido.

A lo largo de este trabajo, hemos identificado y explorado algunas vulnerabilidades y posibles ataques relacionados con estos protocolos. Sin embargo, hay muchas otras tecnologías y pruebas que podrían explorarse en el futuro. Por ejemplo, podríamos analizar más a fondo diferentes tipos de ataques y explicar sus posibles defensas. Además, podríamos ampliar nuestro enfoque para incluir otras soluciones de gestión de iden-

tidad, como Kerberos, y comparar sus características y ventajas en relación con los protocolos estudiados.

En resumen, este Trabajo de Fin de Grado ha brindado una visión completa y práctica sobre los protocolos SAML, OAuth 2.0 y OIDC, permitiéndonos comprender su funcionamiento, implementación y posibles vulnerabilidades gracias al diseño de escenarios personalizados para cada protocolo.

Bibliografía

[1] Security assertion markup language (saml) v2.0 technical overview. http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html, 2008. (Accedido en el 30/03/2023).

[2] Single sign-on. http://www.opengroup.org/security/sso/. (Accedido en el 17/03/2023).

[3] Jan De Clercq. Single sign-on architectures. In George Davida, Yair Frankel, and Owen Rees, editors, *Infrastructure Security*, pages 40–58, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45831-9.

[4] Adam Barth. HTTP State Management Mechanism. RFC 6265, April 2011. URL https://www.rfc-editor.org/info/rfc6265.

[5] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. URL https://www.rfc-editor.org/info/rfc6749.

[6] OpenID Foundation. Final: Openid connect core 1.0 incorporating errata set 1, Nov 2014. URL https://openid.net/specs/openid-connect-core-1_0.html.

[7] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015. URL https://www.rfc-editor.org/info/rfc7519.

[8] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Signature (JWS). RFC 7515, May 2015. URL https://www.rfc-editor.org/info/rfc7515.

[9] Simplesamlphp home. https://simplesamlphp.org/. (Accedido en el 11/05/2023).

[10] oauthlib/oauthlib: A generic, spec-compliant, thorough implementation of the oauth request-signing logic. https://github.com/oauthlib/oauthlib. (Accedido en el 28/05/2023).

[11] Flask documentation (2.3.x). https://flask.palletsprojects.com/en/2.3.x/. (Accedido en el 28/05/2023).

[12] Security and privacy considerations for the oasis security assertion markup language (saml) v2.0. https://docs.oasis-open.org/security/saml/v2.0/saml-sec-consider-2.0-os.pdf, 2005. (Accedido en el 28/05/2023).

[13] Christopher Allen and Tim Dierks. The TLS Protocol Version 1.0. RFC 2246, January 1999. URL https://www.rfc-editor.org/info/rfc2246.

[14] Sheila Frankel and Suresh Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071, February 2011. URL https://www.rfc-editor.org/info/rfc6071.

[15] Donald Eastlake, Thomas Roessler, David Solo, Magnus Nyström, Joseph Reagle, Frederick Hirsch, and Kelvin Yiu. XML signature syntax and processing version 1.1. W3C recommendation, W3C, April 2013. https://www.w3.org/TR/2013/REC-xmldsig-core1-20130411/.

[16] Thomas Roessler, Frederick Hirsch, Donald Eastlake, and Joseph Reagle. XML encryption syntax and processing version 1.1. W3C recommendation, W3C, April 2013. https://www.w3.org/TR/2013/REC-xmlenc-core1-20130411/.

[17] T. Gross. Security analysis of the saml single sign-on browser/artifact profile. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pages 298–307, 2003. doi: 10.1109/CSAC.2003.1254334.

[18] Sstc response to "security analysis of the saml single sign-on browser/artifact profile". https://www.oasis-open.org/committees/download.php/11191/sstc-gross-sec-analysis-response-01.pdf, 2005. (Accedido en el 29/05/2023).

Lista de Acrónimos y Abreviaturas

ACS	Assertion Consumer Service.
AS	Authorization Server.
CSRF	Cross-Site Request Forgery.
HTTP	Hypertext Transfer Protocol.
IAA	Infraestructuras de Autenticación y Autorización.
IdP	Identity Provider.
IETF	Internet Engineering Task Force.
JWS	JSON Web Signature.
JWT	JSON Web Token.
NAI	Network Access Identifier.
OASIS	Organization for the Advancement of Structured Information Standards.
OIDC	OpenID Connect.
OP	OpenID Provider.
RO	Resource Owner.
RP	Relaying Party.
RS	Resource Server.
SAML	Security Assertion Markup Language.
SP	Service Provider.
SSO	Single Sign-On.
TFG	Trabajo de Fin de Grado.
URI	Uniform Resource Identifier.
XML	eXtensible Markup Language.