Securización de REST

Álex Durán Macías

Resum—REST es un estilo de arquitectura para el diseño de Servicios Web o Web Services (WS) y, a pesar de no ser considerado un estándar o protocolo, sí que utiliza algunos. Este estilo permite aprovechar al máximo los recursos de la web, consiguiendo así ser más ligero y mejorar tanto eficiencia como rendimiento frente a otras opciones como SOAP. Por este motivo, grandes compañías como Google o Facebook han cambiado las interfaces de sus servicios para hacerlas de forma RESTful. El gran problema que éste presenta es la falta de seguridad, y es que cualquier persona malintencionada puede interceptar los mensajes HTTP y leerlos, lo que provoca que sea necesario añadir algún tipo de seguridad. De aquí nace el objetivo de este proyecto, en el que se han encontrado y clasificado diferentes métodos para la securización de REST según sus características para llevar a cabo su análisis y prueba, seleccionando aquellos que parecen más interesantes. Para poder llevar a cabo este trabajo, ha sido necesario definir previamente qué es un Servicio Web, qué ventajas supone, cómo se puede crear uno (existen varios lenguajes de programación y frameworks), y cuáles son los motivos por los que interesaría añadir seguridad a REST.

Paraules clau—REST, RESTful, servicio, web, desarrollo, diseño, HTTP, XML, JSON, SOAP, W3C, Java, Jersey, Maven, seguridad, autenticación, autorización, validación, criptografía, certificados, firmas, digitales, HTTPS, SSL, TLS, OAuth, OpenID.

Abstract—REST is an architectural style for Web Services (WS) designing and, even though it is neither a standard nor a protocol, it uses a few of them. This style allows us to take advantage of the web resources to its maximum, achieving a more light and efficient service against other options such as SOAP. For that reason, big companies such as Google or Facebook have changed some of their services interfaces to use them in a RESTful way. This one introduces the big problem and that is the lack of security, because any malicious attacker can intercept the HTTP messages and read them, which causes a need for adding any kind of security. This is where the project's goal is born, in which it has been found and classified different security methods for REST classified by their properties to analyse and test them, selecting the ones that seem more interesting. To do this job, it has been necessary to first define what a Web Service is, what benefits can have, how one can be done (there are some programming languages and frameworks), and what the reasons to implement security in REST are.

Index Terms—REST, RESTful, service, web, development, design, HTTP, XML. JSON, SOAP, W3C, Java, Jersey, Maven, security, authentication, authorization, validation, cryptography, certificate, signature, digital, HTTPS, SSL, TLS, OAuth, OpenID.

1 Introducción

En los últimos años, el estilo para el diseño de Servicios Web ha ido cobrando popularidad frente a otras alternativas como el uso de SOAP. En especial, grandes compañías como Google o Facebook han cambiado las interfaces de sus servicios para hacerlas de forma RESTful que las hace más sencillas e intuitivas de usar. A pesar de las grandes ventajas que supone como una mayor flexibilidad, escalabilidad o reusabilidad, debido al uso intrínseco de HTTP y otros protocolos, por su propia naturaleza un servicio web RESTful se vuelve más vulnerable en la red no sólo frente a intercepción de mensajes sino otro tipo de ataques como la suplantación de identidad. Internet no es un lugar seguro y, por ello, será necesario añadir diferentes tipos de seguridad al WS utilizando diferentes métodos que, a su vez, tiene varias formas de implementarse.

- *E-mail de contacte: alex.duma94@gmail.com*
- *Menció realitzada: Tecnologies de la Informació.*
- *Treball tutoritzat per: Jordi Duran Cals (departament)*
- *Curs 2015/16*

1.1 Objetivos

El objetivo principal de este este trabajo es encontrar esos tipos de seguridad en la web para definir cada uno de ellos y que así se facilite su comprensión y se encuentren sus puntos más críticos, de forma que se pueda realizar una clasificación de los métodos a añadir en un WS según sus características para mejorar la seguridad de cada tipo. Con ello, se pretende analizar algunos de los métodos que parezcan interesantes y hacer pruebas sobre un subconjunto de éstos para descubrir cuáles serán más adecuados en la securización de un servicio web con REST. Cabe mencionar que, cada método puede implementarse en multitud de lenguajes de programación y utilizando diferentes frameworks existentes que facilitan su trabajo pero probar cada uno de éstos haría este proyecto interminable, por lo que se recomienda que, en caso de que no haya ninguna limitación o imposición, se emplee la opción que resulte más cómoda al propio desarrollador.

Antes de poder conseguir el objetivo principal, es necesario que se hayan cumplido unos secundarios en los que se aclaren y definan algunos conceptos previos a este trabajo para entender correctamente qué es un servicio web, qué es REST y por qué se recomienda su uso.

El documento se ha dividido en una explicación introductoria sobre el contexto necesario para comprender el propósito del artículo, la metodología que se ha seguido sobre el trabajo realizado, así como los análisis y pruebas contenidos en ésta que han permitido llegar a las conclusiones finales.

2 ESTADO DEL ARTE

2.1 Contexto general de REST

REpresentational State Transfer o REST fue introducido el año 2000 por Roy Fielding en la Universidad de California, en su tesis "Architectural Styles and the Design of Network-based Software Architectures" [1], pero no ha sido hasta estos últimos años que ha ido cobrando popularidad como alternativa al uso de SOAP. Especialmente, grandes compañías como Google o Facebook han cambiado las interfaces de sus servicios para hacerlas de forma RESTful, lo que las hace más sencillas de usar.

En muchas ocasiones se comparan SOAP y REST pero cabe mencionar que, mientras que el primero es un estándar reconocido por la W3C, el segundo no es más que un **estilo de arquitectura** para el diseño de aplicaciones orientadas a la web por lo que no existe ninguna entidad que lo regule. Aunque no sea tratado como un estándar, éste combina diferentes principios de otros estilos de arquitectura, añadiendo otros propios y usando también otros estándares reconocidos.

Entre los estándares y protocolos que se recomiendan con el uso de REST se encuentran algunos como **HTTP** o **XML** [2], utilizado en las peticiones y respuestas REST cuando la seguridad es muy importante gracias a la facilidad para codificar y enviar mensajes (ficheros .xml) con este lenguaje. Hay que tener en cuenta que, dado que REST no está atado a XML como sucede con SOAP, también existen otras alternativas como **CSV** (Comma-Separated Values) o **JSON** (JavaScript Object Notation). Si se habla de una implementación REST mediante Java, cobra gran importancia el estándar abierto JSON capaz de sustituir a XML a pesar del inconveniente de que, para que los datos sean entendibles para humanos y máquinas éstos deben ser parseados antes de ser enviados y después de ser recibidos utilizando otros lenguajes de programación. Para realizar las **pruebas de este trabajo**, se ha usado una implementación Java para crear servicios web de ejemplo que han empleado esta estructura para la transmisión de datos entre cliente y servidor.

2.2 Características principales de REST

Las características intrínsecas de un servicio web construido basándose en REST [3], se derivan en su mayoría de las propiedades y rasgos de los protocolos y estándares que utiliza mencionados anteriormente.

Un aspecto a tener muy en cuenta es que gracias al uso de HTTP se obtiene un muy **bajo acoplamiento** entre cliente y servidor, de manera que el cliente sólo debe conocer la URI principal y escoger las acciones que proporciona el servidor, consiguiendo así una gran **escalabilidad**.

Existen muchos desarrolladores que prefieren usar REST junto a JSON para tener una alternativa más **ligera** y **eficiente** con los recursos de la web debido al uso que hace de HTTP. No sólo esto, sino que con el auge en los últimos años de la computación en la nube (*cloud computing*) y las aplicaciones web (*web hosted applications*), las tecnologías basadas en REST facilitan el uso de interfaces que los clientes pueden llamar desde servidores remotos. Además, dado que REST siempre funciona mejor cuando el cliente que hace la llamada se trata de un humano y, al ser un estilo **sin estado** (*stateless*), situaciones en las que guardar el estado del cliente sea imprescindible y frecuente no será recomendable utilizar REST, pues ello va en contra de su propia definición.

Otros motivos importantes que han popularizado REST corresponden a ser completamente **independiente** de las plataformas del servidor y del cliente, y la posibilidad de utilizar **diversos lenguajes y estándares** para la comunicación, aunque la **elección** de éstos dependerá del lenguaje de programación a usar, el entorno y los requisitos del propio WS. Dado que no existe una "solución universal", en esta elección se habrán de sopesar también los pros y contras de cada uno.

Pese a todas estas características uno de sus principales problemas es que, como otros servicios web, **REST no ofrece** ningún mecanismo de **seguridad**, encriptación, gestión de sesiones… por lo que es importante que éstos sean añadidos de forma que no se vulnere la seguridad del WS y sean incorporados encima de HTTP, como el uso de un protocolo de usuario-contraseña o la encriptación del propio HTTP usando REST sobre HTTPS. Otra desventaja que se debería mencionar es que, al no ser un estándar definido, en la práctica no hay ninguna **regulación** establecida y esto depende enteramente de los creadores del WS en cuestión.

2.3 Por qué añadir seguridad a REST

La falta de seguridad es uno de los principales problemas que tiene REST y es que al trabajar REST sobre HTTP y depender de él para el transporte, se utilizan y "heredan" las medidas de seguridad aplicadas a este último (aunque existen medidas adicionales para añadir la seguridad al WS fuera de HTTP). Debido a esto, cualquier persona malintencionada puede **interceptar los mensajes HTTP** que se utilizan en la comunicación de este estilo de arquitectura y leerlos.

Un aspecto muy importante a tener en cuenta es que las funciones que proporciona un servicio web serán utilizadas no sólo internamente (dentro del propio entorno de desarrollo), sino que probablemente también se dará permiso para su uso a terceras personas. Por este motivo es posible que **no** se tenga **control** total sobre todos los clientes que accederán a un servicio que se posea, por lo que añadir seguridad para evitar un mal uso del mismo se convierte en una necesidad. Además,

existen muchos servicios web que se ofrecen como producto de negocio (también de pago) y un cliente no deseará utilizar uno que no garantice su propia seguridad, disminuyendo así el valor competitivo del servicio y sus posibles usuarios.

3 METODOLOGÍA

3.1 Planificación general

Para poder lograr todos los **objetivos** propuestos en los apartados anteriores, se ha dividido el trabajo a realizar en **diferentes tareas** que se desarrollarían en un documento por cada una de ellas, formando así el dossier del trabajo. En la medida de lo posible se ha intentado dividir al máximo las tareas en sub-tareas de forma que sean más sencillas y simples de realizar, facilitando así la posterior predicción del tiempo de realización. Las **primeras de estas tareas** hacen referencia a la introducción del trabajo definiendo los conceptos básicos para la comprensión de la segunda etapa en la que se ha realizado una búsqueda de la seguridad de REST con el objetivo de obtener toda la información posible para que en las **últimas tareas** ya se pudieran desarrollar los escenarios de prueba que llevarían a las conclusiones finales.

A continuación se presenta la **lista de tareas** que se han llevado a cabo a lo largo del trabajo[1]:

1. **Definición de Servicio Web.** Dado que el lector puede no estar familiarizado con los Servicios Web RESTful, para poder comprenderlo ha sido necesario explicar antes qué es un Servicio Web, además de definir las dos maneras de crear un Servicios Web.
    a. Definición de Servicio Web.
    b. Creación de Servicios Web con SOAP.
2. **Definición de REST en el contexto aplicación-servidor.** Se han recopilado los conceptos indispensables para entender qué es REST.
    a. Definición del estilo REST y los componentes necesarios para poder utilizarlo. Se ha comentado qué es REST y los estándares que utiliza como HTTP, XML y/o JSON.
    b. Explicación acerca de cómo construir un Servicio Web RESTful.
    c. Análisis de por qué se habría de utilizar REST. Se han recolectado y ordenado qué ventajas ofrece este método frente al resto.
3. **Analizar por qué es interesante aplicar seguridad a REST.** Se han comentado los motivos por los cuáles puede ser necesario añadir seguridad a REST y qué nuevas posibilidades puede ofrecer esto.
4. **Buscar los tipos de securización principales que existen.** El objetivo principal de esta tarea es mostrar los datos recopilados sobre los tipos de seguridad que existen para poder organizar los métodos de securización que se pueden utilizar junto a REST según cada tipo y hacer una exposición de cada método, algunos ya incluidos en las fuentes de información adjuntadas anteriormente. Puesto que esta tarea es de recopilación de información teórica, no se ha profundizado en los detalles de configuración e implementación de estos métodos como el uso de ciertos lenguajes o frameworks ya que esto se ha hecho en tareas posteriores.
5. **Comprobar los métodos encontrados.** Realizada junto la tarea anterior mientras se iban encontrando todos los métodos posibles, ha sido necesario filtrarlos y evaluarlos ya que no todos pueden ser considerados útiles o eficientes para la securización de REST, y otros no pueden ser probados. En el momento de la clasificación, se dividieron en tres grupos:
    a. Descartar los métodos que no se podrán probar ni analizar.
    b. Seleccionar los que será posible analizar.
    c. Seleccionar los que será posible probar.
6. **Probar los métodos seleccionados.** Se han llevado a cabo pruebas intentado implementar estos métodos de seguridad en pequeños WS a modo de ejemplo según diferentes escenarios diseñados de forma que se puedan extraer más datos no encontrados durante el análisis de los mismos.
    a. Buscar las herramientas y manuales necesarios.
    b. Realizar pruebas (descartar algún método si es necesario).
7. **Análisis de métodos seleccionados y explicar cómo se han implementado al probarlos.** Por cada método encontrado y seleccionado para análisis se han estudiado y recopilado las características principales a partir de todas las pruebas que se han considerado necesarias según diferentes escenarios. Para poder mostrar cómo se ha llegado a las conclusiones de cada método, también se resumen los pasos que se han hecho para ejecutar los ejemplos así como las especificaciones del escenario en cuestión. Se entrará en profundidad en cómo se han llevado a cabo las pruebas y análisis en siguientes apartados de este artículo.

3.2 Proceso para el análisis teórico de los tipos y métodos de seguridad

Para poder llegar a los métodos de seguridad que se pueden utilizar para proteger un servicio web con el estilo de REST, ha sido necesario realizar un trabajo previo de investigación para encontrar primero los tipos de seguridad que se podrían aplicar a un servicio web RESTful.

De aquí surge una de las partes más importantes del trabajo en la que se ha ido **redactando** en un mismo **documento**, por cada **tipo de seguridad** encontrado (como autenticación o criptografía), una definición del concepto correspondiente con una breve introducción, algún ejemplo práctico de su uso junto a los errores más comunes que suelen suceder, y la información de todos los **métodos de seguridad** encontrados que están asociados a ese tipo. Con estos métodos se ha seguido un

[1] La tabla donde se resumen las tareas de la planificación se encuentra en la sección 1 del anexo (A1).

esquema similar en el que se definía el método, se explicaba su uso con ayuda de algún ejemplo (como una consulta HTTP completa con autenticación básica), y la valoración según si sería interesante incorporarlo a un servicio web o no, junto al motivo de esta conclusión.

3.3 Proceso para el diseño y ejecución de los escenarios de prueba

Con todos los métodos hallados gracias a los análisis explicados en el apartado anterior, aparece entonces la parte más importante del trabajo: las pruebas prácticas para el **testeo de los métodos de securización** de REST.

Con el objetivo de tener un mejor organización, el testeo de estos métodos se ha dividido en diferentes **escenarios**, donde cada uno de ellos corresponde a la implementación de un método de los seleccionados para probar. Cada escenario se ha dividido en la **definición del entorno** que se ha empleado para al ejecutar las pruebas (tipo de implementación, especificaciones y configuración del servidor y del cliente), la **explicación del proceso** que se ha seguido para ejecutar las pruebas que incluye una pequeña parte con información necesaria sobre el servidor o cliente para comprender las pruebas y la **demostración de los resultados** obtenidos tras cada prueba. Al final de cada escenario, se ha incluido un resumen **valorando los resultados** sobre el método probado así como la implementación utilizada en el mismo, mediante tablas para tener una representación más visual y facilitar su comprensión.

En general, todas las pruebas han sido realizadas utilizando: **Apache Tomcat 8.0.35** para el software del servidor, el conjunto de herramientas de programación de código abierto (IDE) **Eclipse Mars JEE** junto al **framework Jersey (JAX-RS 2.0)** para la **implementación**, y el **plug-in** del entorno Eclipse **Maven** para la construcción de los proyectos del WS RESTful (junto a las dependencias de los proyectos web). Además, también ha sido muy útil emplear la herramienta plug-in de Google Chome **Postman** para poder **enviar mensajes HTTP** con multitud de cabeceras diferentes (como autenticación básica HTTP, simples mensajes "POST"…) y la implementación del cliente que proporciona JAX-RS.

Como se ha mencionado en apartados anteriores, existen multitud de **lenguajes y frameworks**[2] que facilitan la construcción de servicios web y la adición de seguridad, pero en su momento para la implementación de los servicios web RESTful se escogió JAX-RS 2.0 por preferencia del autor del trabajo debido a la experiencia previa con el lenguaje de programación Java adquirida a lo largo del estudio del Grado.

4 ANÁLISIS DE TIPOS Y MÉTODOS DE SEGURIDAD

A continuación se presentarán los diferentes tipos de seguridad para WS RESTful que se han ido encontrando a lo largo de la investigación de este trabajo y los principales métodos que contiene cada uno.

4.1 Autenticación

4.1.1 Definición

La autenticación se encarga de **validar** si el **cliente** que entra (**log in**) a un sistema software es la entidad que dice ser para asegurar que un atacante no está suplantando la identidad de otro cliente. Esta puede ser basada en **sesiones** (session-based) estableciendo mediante POST un token que identificará la conexión con un cierto cliente, utilizando una clave que se enviará en el cuerpo de un mensaje POST o, sólo si es completamente necesario, utilizar una **cookie**. Debido a que los servidores web guardan logs de las transacciones que se realizan no debería aparecer nunca en sus URLs: nombres de usuario, contraseñas, claves ni tokens de sesión.

4.1.2 Basic HTTP Authentication

Una de las formas más sencillas y fáciles de implementar es **Basic HTTP Authentication** [4] con HTTPS, pues no necesita la ayuda de APIs o frameworks adicionales. Sólo es necesario un nombre de usuario y una contraseña que se codificarán en **Base64** y utilizar siempre encriptación SSL/TLS para evitar que un atacante pueda ver y descifrar fácilmente las credenciales anteriores que se enviarán en un mensaje HTTP. Gracias al uso de HTTPS, éste garantiza que el canal de transmisión es seguro.

```
GET /something.html HTTP/1.1
Host: www.example.com
Authorization: Basic
QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

4.1.3 HTTP Digest Access Authentication

El método anterior exige el uso de SSL/TLS pero, si esto no se desea, existe una alternativa más compleja llamada **HTTP Digest Access Authentication** que utiliza hashes de las claves (es un tipo de **Hash-based Message Authentication** o **HMAC**) en lugar de enviarlas encriptadas. El proceso consiste en que: el servidor envía al cliente un valor aleatorio (***nonce***) y un string (***realm***) que identifica esa autenticación; a continuación el cliente envía un hash (usualmente MD5) que combina la contraseña, el nonce, el método HTTP usado y la URI; y finalmente el servidor generará independientemente su propio hash que deberá ser igual que el enviado por el cliente para que la autenticación sea satisfactoria.

Al no utilizar el protocolo SSL, tiene la ventaja de ser un poco más rápido pero también más vulnerable a ataques que se podrían evitar con él debido a que para poder generar el hash de la contraseña no debe estar cifrada en el servidor. Además, el uso de MD5 puede provocar que la probabilidad de las colisiones para una misma contraseña de cliente sea demasiado alta y un atacante puede aprovecharse de ello.

```
GET /something.html HTTP/1.1
Host: www.example.com
Authorization: [Digest]
```

[2] La tabla donde se muestran los frameworks más conocidos para la creación de WS RESTful está en la sección 2 del anexo (A2).

4.1.4 OAuth 1.0a

Otro protocolo muy conocido, ampliamente usado y probado es **OAuth 1.0a** [5] (versión revisada de OAuth 1.0 que tenía ciertos errores que vulneraban la seguridad). A pesar de ser un estándar abierto para la autorización, también se encarga de la autenticación y suele emplearse para ello [6]. Basado en las firmas digitales (**digital signature**), emplea algoritmos de **hash** para calcularlas (como el conocido HMAC-SHA1), obteniendo así un valor que combina el **token** secreto junto a otra información que ayuda en la autenticación y autorización. En este caso, la autenticación es simple:

1. El cliente obtiene del servidor un "**Request token**" que no ha sido autorizado.
2. A continuación, el cliente firma ese token para garantizar que es quien dice ser y lo envía al servidor.
3. Finalmente el servidor sólo tiene que comprobar si la firma es válida, en cuyo caso enviará al cliente un "**Access token**" que podrá usar para acceder al WS. Este último token no sólo sirve para autenticar al usuario sino también para "**autorizarlo**", indicando los "privilegios" que tendrá dentro del WS.

La mayor ventaja que posee es que no necesita enviar el secreto directamente por lo que añadir SSL no es necesario, pero este alto nivel de seguridad viene al precio de provocar un alto **overhead** de tiempo debido a la complejidad del proceso de generación (cálculos computacionales) de la firma digital [7].

4.1.5 OAuth 2.0

Existen casos en los que el overhead que provoca OAuth 1.0 supone un problema. A causa de esto, se creó una versión diferente del protocolo: el framework llamado **OAuth 2**. Esta versión **elimina** completamente el uso de **firmas digitales** por lo que necesita HTTPS para encargarse de la encriptación, lo que provoca que sea mucho más **simple** que su antecesor pero a la vez sea mucho menos seguro. A pesar de ser menos utilizado y perder parte de seguridad en la autenticación, sigue siendo una buena solución para casos con datos menos sensibles y servidores con una capacidad de cálculo limitada. Cabe mencionar que como su antecesor, incorpora también soporte para la autenticación, aunque su objetivo principal es la **autorización,** lo cual permitiría a aplicaciones de una third-party obtener un acceso limitado por HTTP al servicio web [8].

4.1.6 OpenID Connect (OIDC) + OAuth 2.0

Dado que OAuth 1.0 y 2.0 son estándares para obtener autorización, siendo la autenticación no tan importante, en muchos casos se recomienda **OpenID Connect** con OAuth 2.0. Su uso proporcionará ciertas ventajas como una autenticación más segura o evitar el tiempo de computación adicional que la primera versión de OAuth suponía.

OpenID Connect 1.0 [9] es una capa de autenticación que trabaja sobre el framework de autorización de OAuth 2.0 con un diseño que facilita su uso en plataformas móviles y con mecanismos opcionales para la encriptación y la gestión de sesiones. Utiliza el formato de datos JSON y HTTP de forma RESTful para la comunicación (es una API RESTful), permitiendo así que los clientes que acceden a través de un navegador o aplicación puedan verificar su **identidad** ante un **servidor de autorización**. Esto se consigue fácilmente añadiendo un **ID en el access token** que contiene datos sobre la autenticación [10].

4.1.7 Observaciones

De todos los métodos anteriores, el más común en servicios REST por su simpleza es el Basic Authentication con SSL/TLS u OAuth 1.0 cuando los datos son demasiado sensibles (considerando siempre el *overhead* que éste provoca). Aun así, es necesario repetir que OAuth (en sus dos versiones) **no es un protocolo** para la autenticación sino una consecuencia del mismo: es la llamada "**pseudo-autenticación**". Se pueden observar ciertas diferencias en el proceso de autenticación con el protocolo OpenID y con OAuth: mientras que OpenID permite utilizar las mismas credenciales para acceder a diferentes sitios, OAuth utiliza las credenciales para la autorización obtenidas de un sitio para acceder a otro. Estas diferencias son fáciles de observar en la figura 1 de la sección 3 del anexo (A3, figura 1).

4.2 Autorización

4.2.1 Definición

Mientras que la autenticación se centra en validar si el cliente que intenta acceder un WS es quien dice ser (verificación de la identidad), en la autorización lo que se valida es si el **cliente** que intenta **acceder a ciertos recursos** del WS **debería poder** hacerlo (si es el cliente correcto), es decir, si se tiene el nivel de autorización suficiente como para poder acceder y realizar ciertas operaciones sobre un determinado recurso. Es precisamente por esto que en muchos casos, como ocurre en el protocolo de OAuth 1.0, la autorización depende en gran medida de la autenticación ya que no tiene sentido comprobar que alguien pueda acceder a unos recursos si éste no se ha **identificado** (no se está seguro de que sea quien dice ser).

Además de la identificación de este método de securización, también pretende proteger un servicio web durante el proceso de autorización de diferentes **ataques comunes** en la red.

A diferencia de lo sucedido con la autenticación web donde existe una gran variedad de protocolos con soporte continuo por parte de su respectiva comunidad, **no** hay tanta **variedad** en lo que a autorización se refiere. De esta forma, OAuth 2.0 es el estándar referente para ello y es el que se suele recomendar. A pesar de esto, existen diferentes ataques muy comunes en la red que se pueden solucionar sin la necesidad de emplear este protocolo.

A continuación se mostrarán cómo se desarrollan algunos de estos ataques y formas sencillas de evitarlos que son opcionales al protocolo OAuth para aumentar la

seguridad.

4.2.2 Autorización con OAuth 1 y 2

Ya se ha explicado en anteriores apartados que el objetivo principal de este protocolo es la autorización y cómo se consigue con éste, además de obtener de manera indirecta una "pseudoautenticación".

4.2.3 Problemas con CSRF y XSS

Ataques muy comunes e igual de peligrosos que pueden hacerse si no se protegen correctamente los **métodos/verbos HTTP** sobre un cierto recurso del WS, son **Cross-Site Request Forgery** (CSRF) [11] y **Cross-Site Scripting** (XSS).

El primero (**CSRF**) se produce cuando un atacante envía **peticiones maliciosas** a un usuario para que se valide en la "web maliciosa" y él cree estar accediendo al servicio web en cuestión, de forma que el atacante puede **redirigir** sus mensajes al navegador de la víctima y usar sus credenciales para acceder al servicio web objetivo. Un ejemplo sería estar visitando una página web maliciosa mientras se está logueado en la web de un banco y en la primera web se encuentra la siguiente línea de código:

```
<iframe
src="http://examplebank.com/app/transferFunds?
amount=1500&destinationAccount=...>
```

En este caso, el atacante se ha "saltado" la autenticación y autorización del servicio web con el objetivo de extraer dinero de la cuenta de la víctima. Para prevenir esto, el método más recomendado por su sencillez y eficacia es el **Synchronizer Token Pattern** [12], en el que se añade un **token aleatorio** asociado a la sesión del usuario en cada petición, que se encuentra escondido en un campo de formulario o en la URL (si la petición es GET) y es diferente en cada sesión y/o petición para asegurar que procedan exactamente de la sesión que corresponde.

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"
value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWE
wYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZ
MGYwMGEwOA==">
…
</form>
```

XSS es todavía más común que CSRF, siendo la vulnerabilidad **más común** en el software actual. Éste se lleva a cabo aprovechando vulnerabilidades en los scripts del lado del cliente para **manipular** y/o insertar porciones de **código** (principalmente con HTML y JavaScript), que se cargarán en el navegador de la víctima cuando visite la página (una práctica común es ocultar código JS ejecutable cuando se hace "click" en un botón).

Si **no se comprueban** correctamente los **datos** que se obtienen de un **cliente** víctima de este ataque, se ejecutará el código malicioso que ésta envía sin ser consciente y puede llegar a provocar graves daños a la página. Para prevenir este tipo de ataque es necesario asegurarse de que cada variable que se mostrará en la página del cliente está codificada correctamente (**HTML Entity Encoding**), sustituyendo el formato HTML (**caracteres especiales**) por **representaciones alternativas** (**entities**). De este modo, una porción de código HTML malicioso que contuviera algo como el ejemplo siguiente no se ejecutaría:

```
<div>
This is a text.
<script>          →   &lt;script&gt;  (# entity)
(# Malicious code)
</script>         →   &frasl&lt;script&gt;
</div>
```

El **problema principal** del HTML Entity Encoding es que codificar los datos sólo funciona con código HTML y sería **inútil** dentro de los siguientes contextos: un **script**, en un **event handler** (como "onmouse=()", **CSS** o una **URL**. Así, el ejemplo anterior sería una **vulnerabilidad** si el script no se encontrara dentro de un tag <p>. Para cada contexto que se acaba de mencionar, se requieren codificar los datos de maneras especiales [13].

4.2.4 Observaciones

Además de los métodos que se han explicado, existen otros que no son realmente medidas de seguridad al uso, pues protecciones como la **validación de inputs** o la **codificación de outputs** (como HTML Entity Encoding) son prácticas que se llevan a cabo como "añadidos" para evitar ciertas vulnerabilidades que una página o servicio web pueda tener y, aunque son importantes para tener un sistema lo más protegido posible, son realmente sencillas de implementar (la mayoría son **triviales**). Es debido a este motivo, que se ha decidido **no incluir** tales medidas en el ámbito de este proyecto.

4.3 Criptografía

4.3.1 Definición

La criptografía es ampliamente utilizada para **proteger comunicaciones** por lo que, en este contexto, para añadir seguridad al intercambio de mensajes HTTP entre el servidor web y el cliente es necesaria si se desea garantizar a los usuarios del servicio web **confidencialidad** y privacidad. A pesar de que añade gran protección, también influirá enormemente de manera negativa en el rendimiento del software que la use.

En este contexto, es muy importante proteger los **datos en tránsito (*in transit*)** pero sin olvidar también los datos que se están **almacenados (*at rest*)**, pues si se protegen las comunicaciones perfectamente pero el sistema no regula y almacena los datos como debería el paso anterior se vuelve inútil.

4.3.2 Secure Socket Layer (SSL), Transport Layer Security (TLS) y el certificado digital

Muchos son los servicios web que lo pueden implementar en su funcionamiento, incluso protocolos como el OAuth ya explicado previamente, que aprovecha el cifrado con clave privada (o simétrica) con SSL realizado a nivel de aplicación para ayudar a aumentar la seguridad en las comunicaciones a través de la red (entre un servidor y un navegador web) de forma que un

atacante no pueda entender los mensajes enviados.

Además del cifrado de las comunicaciones, este protocolo hace uso de **certificados digitales** para vincular a una entidad (cliente) con su llave pública gracias a las firmas digitales de estos para **evitar** problemas de **suplantación de identidad** [14]. El funcionamiento básico del protocolo se muestra en la figura 2 (A4).

La ventaja principal que tiene SSL es que todo ese proceso se consigue de forma **transparente al usuario** que realiza el acceso (no es necesario que sepa cómo funciona ni que conozca las claves que se emplean).

4.3.2 HTTPS

Como REST utiliza HTTP, es recomendable utilizar siempre que sea posible SSL con este. Así se obtiene el llamado **HTTPS**, que no es más que la combinación de los protocolos HTTP y SSL usado en cada transacción web: **https**://www.facebook.com.

5 RESULTADOS DE LOS ESCENARIOS DE PRUEBA

Ahora que se han explicado todas las características básicas de los diferentes métodos existentes para añadir seguridad según el tipo al que corresponden, es momento de clasificarlos de realizar las pruebas necesarias. En la tabla 3 (A5), se resumen los métodos que se tratarán en los diferentes escenarios implementados mediante el framework Java Jersey (especificación JAX-RS 2.0).

Para introducirse en el uso del framework, se ha empezado con el método más sencillo para el primer escenario: la **autenticación básica de HTTP y autorización basada en roles** sin ningún tipo de protección adicional. Para el segundo escenario, se incorporó **HTTP S** a las pruebas anteriores. En el tercer escenario se emplearía ya el soporte que incorpora Jersey para implementar **OAuth 1.0**. Finalmente, en el último escenario, se ha usado OAuth 2.0 mediante una implementación diferente: **Apache OLTU**.

5.1. Escenario 1: Autenticación HTTP básica, autorización basada en roles con Jersey

Las pruebas se han realizado con un ordenador doméstico con Apache Tomcat 8.0 como **servidor**, y la aplicación Postman como **cliente** que permite enviar mensajes HTTP con autenticación básica.

Escenario 1 - Conclusiones del método	
Seguridad garantizada: Bajo	La autenticación básica sólo comprueba un nombre y contraseña que se envía en claro (codificado con BASE64), por lo que cualquier atacante que intercepte un mensaje puede leerlo sin problemas. No se recomienda este método a menos que emplee junto a HTTPS, que cifrar el contenido.
Complejidad de construcción: Muy bajo	Construir este método de seguridad es trivial, pues simplemente se realiza una comprobación del nombre, contraseña y rol del usuario que intenta acceder a un recurso del servidor.
Escalabilidad: Muy alto	Al ser una sencilla comparación, es fácilmente realizable para cualquier cantidad de usuarios que ingresen en el servicio web.
Coste en el rendimiento: Inexistente	Por el mismo motivo anterior y al no necesitar de ninguna gestión extra como la de certificados (existente en otros métodos), a penas afecta al rendimiento del programa.
Información disponible: Muy alto	Al ser el método más sencillo, existen multitud de demostraciones y ejemplos para diferentes implementaciones en la web.
Estándar definido: Inexistente	Como ya se ha mencionado, al garantizar tan poca seguridad no se recomienda implementar este método solo, por lo que no existe un estándar que lo regule.

Escenario 1 - Conclusiones de la implementación	
Complejidad de construcción del WS: Medio	La creación de un servicio web no es especialmente difícil, pero sí que se vuelve algo pesada debido a la necesidad de muchos ficheros diferentes para gestionar las dependencias de las funciones internas del propio framework, como por ejemplo añadir SSL a la solución o enlazar los ficheros que gestionan las URIs del WS.
Complejidad de implementación de la seguridad: Muy bajo	Es lo más sencillo que se podría pedir a un framework, ya que sólo es necesario añadir una etiqueta sobre las funciones del recurso con el rol al que se permitirá ejecutarlas.
Trazabilidad de la solución: Medio	Como se ha comentado, es necesario generar muchos ficheros para obtener unas pocas funciones por lo que si no se está acostumbrado a manejarse con el framework, es algo difícil acostumbrarse a la estructura de directorios.
Portabilidad de la solución: Alto	Gracias al uso del lenguaje Java que utiliza el framework, esta solución se puede beneficiar de sus características como la portabilidad.
Coste en el rendimiento: Inexistente	No se ha apreciado una bajada del rendimiento con respecto a consultas de otras soluciones.
Información disponible: Alto	Se pueden encontrar en Internet muchas explicaciones y ejemplos que ilustran con gran cantidad de detalles el uso de esta solución.
Claridad de la documentación oficial: Medio	Explica brevemente los conceptos básicos y flujos de creación de un servicio web con cierta claridad, pero carece de proyectos de ejemplo con los que empezar a trabajar, por lo que tiene un gran margen para mejorar.

5.2. Escenario 2: Autenticación HTTP básica, autorización basada en roles con Jersey y HTTPS

En este escenario el **servidor** utiliza Apache Tomcat con configuración SSL para que se acepten **conexiones seguras con SSL** en el puerto 8444. También ha sido necesaria la herramienta "keytool" para generar las **claves y certificados** que se usan en las conexiones HTTPS. El **cliente** es la aplicación Postman para HTTP y las funciones de Jersey para construir el cliente con los parámetros necesarios para el funcionamiento de HTTPS.

Escenario 2 - Conclusiones del método	
Seguridad garantizada: Medio	A pesar de que todos los datos de la conexión están cifrados y autenticados con el uso de HTTPS, además de la autorización basada en roles, no se ha considerado de nivel alto porque existen mecanismos más complejos que garantizan una mayor seguridad.
Complejidad de construcción: Medio	Crear las claves/certificados, configurar el servidor para aceptar conexiones HTTPS y configurar el cliente para que las realice no es una tarea trivial si es la primera vez que se intenta, pues se deben realizar todos los pasos correctamente y es difícil encontrar errores cuando se están gestionando las claves y certificados.
Escalabilidad: Medio	Establecer un canal de comunicación cifrado implica unos costes de computación adicionales; afortunadamente, el estándar empleado no es SSL sino su sucesor TLS que comporta unos costes menores. Aun así, abrir multitud de conexiones cifradas con diferentes usuarios puede producir un overhead muy alto.
Coste en el rendimiento: Medio	Una conexión HTTPS no debería suponer grandes problemas a menos que el WS deba cargar continuamente en el cliente grandes cantidades de información que se cifrarán (esto puede generar un alto overhead).
Información disponible: Muy alto	Este protocolo es ampliamente conocido y, por lo tanto, encontrar diferentes formas sobre cómo implementarlo es bien sencillo.
Estándar definido: Existente	Dado que es un protocolo ampliamente utilizado, su correcto uso y buenas prácticas está bien definido.

Escenario 2 - Conclusiones de la implementación	
Complejidad de construcción del WS: Medio	Este aspecto es igual que en el escenario anterior.
Complejidad de implementación de la seguridad: Bajo	En cuanto a la implementación con Jersey se refiere, no es especialmente complicado construir el cliente para probar las conexiones HTTPS y, aunque se tienen que utilizar una amplia variedad de objetos de clases diferentes, se configuran de forma bastante intuitiva.
Trazabilidad de la solución: Medio	Este aspecto es igual que en el escenario anterior. Añadir este método de seguridad no supone generar ficheros nuevos, a excepción del cliente que cambiará.
Portabilidad de la solución: Alto	Este aspecto es igual que en el escenario anterior.
Coste en el rendimiento: Inexistente	No se ha visto que el uso de este framework afecte al rendimiento del nuevo cliente.
Información disponible: Medio	No se han encontrado demasiados ejemplos sobre cómo construir correctamente un cliente con SSL y autenticación. De estos pocos ejemplos, la mayoría eran relativamente antiguos y explicaban cómo crearlo con Jersey 1 pero se ha utilizado la versión 2, por lo que éstos no eran útiles.
Claridad de la documentación oficial: Bajo	En este apartado, no sólo la documentación es escasa y poco clara, sino que además no dispone de ningún ejemplo. La documentación para el uso de SSL con Jersey 2 tiene un gran margen de mejora.

5.3. Escenario 3: Jersey OAuth1 support

El servidor sigue siendo el mismo que en escenarios anteriores, pero ahora también existe un **recurso** en el **servidor** al que se desea acceder que, con la **pseudo-autenticación**, comprobará que los secretos contenidos en tokens junto a una firma que pasa el cliente al servidor son los que éste espera. Sólo le proporcionará acceso en caso afirmativo. Existen dos **nuevos recursos** que proporcionarán los **tokens** adecuados al cliente para llevarlo todo a cabo.

Para el cliente se ha utilizado la clase 'Auth1Builder' para construir el cliente con los parámetros adecuados para llevar a cabo OAuth1. Estos estarán compuestos por diferentes peticiones previas a la petición final para intercambiar los tokens necesarios para la autenticación. Con la intención de facilitar la comprensión de este ejemplo, esos parámetros se han definido como constantes en el cliente y el servidor.

Escenario 3 - Conclusiones del método	
Seguridad garantizada: Medio / alto (+HTTPS)	Este método garantiza que sólo los usuarios que tienen un rol determinado pueden acceder a un recurso protegido, lo que garantiza autenticación y autorización. Hay que tener en cuenta que, como en el caso de la autenticación básica los mensajes se envían en claro y un atacante que intercepte los mensajes podría ver su contenido, por lo que sería recomendable usar conexiones HTTPS junto a este método. Aun así, dado que OAuth utiliza firmas digitales para garantizar la autenticación, el atacante debería conocer el secreto con el que el cliente firma sus tokens, así que se dificulta suplantación de identidad.
Complejidad de construcción: Medio / alto (+HTTPS)	El flujo que sigue este método es algo más complejo de entender que los anteriores, pero su dificultad a la hora de construirlo no es mucho más alta. Generar manualmente algunos tokens no es muy difícil (incluso en el caso de Java hay clases "Token" que permiten definirlos), y gestionar los intercambios de éstos, tampoco es complicado. Por lo tanto, a menos que se utilice junto a HTTPS su complejidad no es demasiado alta.
Escalabilidad: Medio / bajo (+HTTPS)	OAuth1 utiliza firmas digitales calculadas mediante algoritmos de hash y, tanto el proceso de crear la firma, como el de verificarla requieren un cierto tiempo adicional debido a los cálculos necesarios que implican estos algoritmos. Por lo tanto, habrá que tener mucho cuidado con el overhead adicional que supondrá utilizar este protocolo, especialmente si se han de gestionar las firmas y tokens de demasiados clientes a la vez. Si además se emplea junto a HTTPS (que es lo recomendable), esta penalización de tiempo todavía se agrava más.
Coste en el rendimiento: Alto / muy alto (+HTTPS)	Como se acaba de comentar, el uso de este protocolo puede afectar mucho al rendimiento del servicio web, por lo que este factor puede ser una limitación muy grande si la capacidad de cómputo del servidor no es lo suficiente alta.
Información disponible: Medio	Este protocolo es ampliamente conocido y existe suficiente información sobre todo su proceso, pero con la aparición de su sucesor se dificulta la búsqueda de información sobre esta versión 1.
Estándar definido: Existente	Dado que es un protocolo ampliamente conocido, su correcto uso y buenas prácticas está bien definido tanto en la documentación oficial del protocolo como por la comunidad del mismo.

Escenario 3 - Conclusiones de la implementación	
Complejidad de construcción del WS: Medio	Este aspecto no ha variado con respecto a escenarios anteriores, aunque sí se debe añadir configuración adicional para que se pueda implementar correctamente la seguridad, por lo que hay que incluir en el proyecto nuevas dependencias con Maven como las clases para el cliente o servidor de OAuth1 de Jersey.
Complejidad de implementación de la seguridad: Medio / alto (+HTTPS)	La configuración básica de este método no requiere demasiadas líneas de código por lo que no es difícil, pero es necesario tener muy claro el funcionamiento del protocolo para definir correctamente en el servidor todas las claves, secretos y roles necesarios, así como construir todo de manera equivalente para que las credenciales del servidor coincidan con las del cliente. Separar correctamente los recursos que proporcionan los tokens, el que autoriza, y al que se quiere proporcionar acceso es algo difícil la primera vez que se intenta implementar este método.
Trazabilidad de la solución: Medio / bajo (+HTTPS)	Este aspecto es muy similar a los escenarios anteriores a pesar de la necesidad de crear más recursos en el servidor para proteger uno solo. Es recomendable dividir el proyecto en distintos "paquetes" según si pertenecen al servidor, cliente, configuración…
Portabilidad de la solución: Medio / bajo (+HTTPS)	Por características inherentes al propio protocolo, habrá que tener cuidado con la capacidad de cómputo del servidor que lo implemente, pues puede no ser suficiente del todo y afectar negativamente al rendimiento del servicio web.
Coste en el rendimiento: Alto / muy alto (+HTTPS)	Este framework no supone una penalización en el rendimiento pero, como ya se ha mencionado, este protocolo sí que la provoca.
Información disponible: Medio	A pesar de haber mucha información y ejemplos sobre cómo crear un cliente para acceder a servidores que trabajan con OAuth1, ha sido bastante complicado encontrar ejemplos sobre la implementación de la parte del servidor del protocolo y es que, ni siquiera parece haber una forma "universalmente" aceptada para la generación de los tokens (creación de los recursos que proporcionan el request y access token).
Claridad de la documentación oficial: Bajo	En este apartado, no sólo la documentación es escasa y poco clara, sino que además no dispone de ningún ejemplo. Todo el trabajo relacionado con la creación del servidor se ha basado en los tests encontrados en la web del proyecto (no se ha encontrado ningún ejemplo "real").

5.4. Escenario 4: Jersey OAuth2 support

Para construir este escenario se han tenido algunos **problemas** con la implementación de la parte de OAuth2 en el servidor debido a que **Jersey sólo tiene soporte** para el lado del **cliente** [15]. Es por este motivo que se ha optado por no mantener tampoco JAX-RS 2.0 para el lado del cliente, por lo que ha sido necesario usar **Apache OLTU** para poder llevar a cabo el proceso tanto en el **servidor** como en el cliente. Es necesario mencionar que esta API de la familia de protocolos OAuth también dispone de otras **implementaciones** Java relacionadas como JWT (tokens necesarios para llevar a cabo el proceso de OAuth con estructura JSON) y **OpenID Connect**, utilizada en otro escenario para garantizar una **autenticación completa** ya que sin esta capa adicional y de forma similar a la versión 1, OAuth2 sólo proporciona una **pseudo-autenticación**.

Escenario 4 - Conclusiones del método	
Seguridad garantizada: Medio	De igual forma que sucede con el escenario anterior, es recomendable utilizar este método junto a una conexión HTTPS no sólo para establecer los parámetros de la autenticación sino también el resto de mensajes. Además, no usar firmas digitales provoca que, aunque el rendimiento general del WS mejore la seguridad no sea tan alta. En este caso, al no haber la autenticación completa que se conseguiría añadiendo OpenID, la seguridad que se alcanza no es la máxima posible. En otras palabras, se obtendría autorización según un token con un secreto compartido y pseudoautenticación.
Complejidad de construcción: Medio	El flujo que se sigue en OAuth2 es algo más simple que el de la versión anterior, por lo que en cuanto a la generación de tokens el request token se pierde (ahora es un código) y sólo resta hacer el access token. La ausencia de firmas digitales que se sustituye por el uso de cuentas de usuario también simplifica en gran medida el proceso de autorización, pues el único cálculo pesado a procesar es la validación del token. Como se acaba de comentar, la ausencia de firmas evita que se produzcan los overheads tan pronunciados como sucede con la primera de este protocolo. De esta manera, el mayor problema con el que se tenía que lidiar antes se elimina y sólo se tendría que tener en cuenta el intercambio y verificación de los tokens de cada cliente, así como los datos de cuentas almacenadas en el servidor.
Escalabilidad: Alto / medio (+HTTPS)	
Coste en el rendimiento: Bajo / medio (+HTTPS)	El rendimiento no se debería ver demasiado afectado por su uso. Aun así, añadir HTTPS siempre provocará un overhead adicional que nunca debería menospreciarse.
Información disponible: Alto	Este protocolo es ampliamente conocido y existe mucha información sobre todo su proceso. No sólo esto, sino que al ser un estándar reconocido y relativamente actual, su implementación se cubre en multitud de lenguajes de programación y abundan los ejemplos para todos los casos.
Estándar definido: Existente	Dado que es un protocolo ampliamente conocido, su correcto uso y buenas prácticas está bien definido tanto en la documentación oficial del protocolo como por la comunidad del mismo.

Escenario 4 - Conclusiones de la implementación	
Complejidad de construcción del WS: Medio	Dado que en este escenario no se ha utilizado una implementación Jersey, ha sido necesario añadir las dependencias Maven de OAuth2 propias de Apache OLTU, aunque la construcción del WS no se ha visto afectada. Como sucede con su versión anterior, la parte más difícil de implementar este protocolo no es la construcción del proceso sino la gestión de los datos necesarios para la ejecución de éste. Registrar los datos del cliente en el servidor *(client_ID, client_secret, username, password)* y asegurarse de que terceras personas no descubren los más sensibles mientras se establecen, es la parte más vulnerable debido a la falta de las firmas digitales para asegurar la identidad del cliente. Por ello, aunque la creación de los recursos que gestionan la peticiones y autentican/autorizan a los cliente es muy similar a la de OAuth1 con Jersey éstas deben desarrollarse un poco más, gestionando todos los mensajes de error (como mínimo los que se deseen) y las acciones para validar las credenciales que con Jersey se hacían a través de una sola clase.
Complejidad de implementación de la seguridad: Medio	
Trazabilidad de la solución: Medio / bajo (+HTTPS)	Este aspecto es muy similar a los escenarios anteriores a pesar de la necesidad de crear más recursos en el servidor para proteger uno solo. Es recomendable dividir el proyecto en distintos "paquetes" según si pertenecen al servidor, cliente, configuración…
Portabilidad de la solución: Alto / medio (+HTTPS)	Por características inherentes al propio protocolo y lenguaje utilizados la portabilidad de la solución será muy alta.
Coste en el rendimiento: Bajo / medio (+HTTPS)	Este framework no supone una penalización en el rendimiento y, como ya se ha mencionado, el protocolo tampoco la provoca.
Información disponible: Alto	El protocolo OAuth2 es conocido y usado por gran cantidad de gente y Apache OLTU es una de las soluciones más utilizadas para implementarlo en Java. Por este motivo la cantidad de información disponible abunda y no ha sido difícil encontrar ejemplos en la red.
Claridad de la documentación oficial: Medio	En cuanto a la definición de los métodos y clases la documentación es algo más clara con respecto a la de Jersey pero, a diferencia de éste, no dispone de explicaciones ni ejemplos sobre cómo se implementa el flujo básico del proceso. Para esto deberá comprarse un libro titulado OAuth 2 in Action (publicado por manning).

6 CONCLUSIONES

6.1. Resultados obtenidos

En este trabajo se ha intentado ofrecer una **definición** objetiva de los distintos tipos de seguridad existentes y de los métodos para aplicarlos, así como mostrar una **comparación** de algunos de ellos a través de diferentes escenarios de **pruebas** utilizando el framework de **Jersey** (con su especificación de **JAX-RS 2.0**) no sólo centrándose en los propios métodos sino también dando una opinión sobre el uso de esta implementación. Muchos de estos **métodos** que se han ido explicando se **complementan** los unos a otros incluso entre tipos de seguridad diferentes, de manera similar a cómo la pseudo-autenticación de OAuth está incluida dentro de su propia autorización, aunque también hay otros que no pueden emplearse al mismo tiempo, como las dos versiones de OAuth que, por cómo están definidas, se **excluyen** mutuamente.

De entre todos ellos, destaca OAuth 2.0 con OpenID y HTTPS, que parece ser la forma más robusta y económica en términos de rendimiento conocida y utilizada por multitud de personas y organizaciones para garantizar autenticación (OpenID y los certificados digitales de HTTPS), autorización (OAuth 2.0) y proteger los datos con criptografía (confidencialidad de HTTPS). Añadir esta capa de autenticación a OAuth2 no supone un esfuerzo adicional especialmente grande, pues sólo hay que modificar ligeramente las peticiones que ya se envían y, en consecuencia, parte de las respuestas. Si se desea se pueden ver los resultados de los test fallidos en anexo A6.

6.2 Problemas hallados en la planificación

A lo largo del desarrollo del proyecto, se han producido algunos **cambios** en la **planificación** y, con ello, algunos objetivos de las tareas han ido cambiado pudiendo llegar incluso a unir ciertas tareas o eliminarlas.

Entre algunos de los cambios efectuados al principio de este trabajo, destacan la **separación** de las dos primeras **tareas**, pues en su momento se pensó que resultaría en una mejor organización del dossier si la definición de servicio web y la de REST se separaban. También sería interesante comentar la decisión de añadir a la búsqueda la seguridad el cifrado y conexiones seguras con HTTPS por recomendación del profesor, comparando esta alternativa junto al HTTP corriente.

Incluso cuando el proyecto ya se encontraba más avanzado y, a pesar de que no afectó al tiempo de la planificación (una vez más, sólo fue un cambio de disposición de la documentación del dossier), se decidió que para tener una mejor organización de los métodos de seguridad se **realizarían en paralelo** las **tareas** 4 y 5, de forma que debajo de la definición de cada tipo de seguridad de la tarea 4 (i.e. autenticación) se explicarían además los distintos métodos para su aplicación junto a la decisión correspondiente (descartar o seleccionar), a pesar de que en la tarea 5 se resumirían todas estas decisiones.

Un último cambio que sería necesario mencionar provocó una curiosa consecuencia y es que, no se esperaba que se **intercambiara el orden** de ejecución de las tareas de analizar los métodos de securización y de probarlos. Para los métodos de seguridad que se habían seleccionado para analizar y probar, se decidió invertir el orden y probarlos primero para tener toda la información posible en el momento de realizar sus respectivos análisis. Se pensó que sería mejor hacer primero todos los test de un método para tener toda la información posible en el momento de efectuar el análisis.

6.3. Problemas encontrados en el desarrollo

Como ya se ha comentado anteriormente en este artículo, la **planificación** no ha sido estática desde el inicio del proyecto y ha ido sufriendo ciertas **modificaciones** a medida que se avanzaba, especialmente en cuanto a la organización de la documentación de las tareas se refiere. El **alcance** del trabajo tampoco ha sido la excepción, puesto que en un principio se deseaba probar **diferentes implementaciones** para los métodos de seguridad existentes pero se pensó en aumentar el número de métodos a probar ya que el número de posibles implementaciones (en lenguajes y frameworks) es demasiado grande. Uno de los aspectos que más ha influido y ha **restringido este alcance** ha sido precisamente el centrarse al inicio e invertir **demasiada cantidad de tiempo** en la primera parte acumulando y ordenando **información** teórica para su posterior análisis, que se podría haber usado en desarrollar más escenarios de prueba. Aun así, el haber documentado tanta información no ha sido complemente negativo, pues ha **facilitado** en gran medida **la comprensión** del tema que se trataba, el funcionamiento de los procesos que se usan en los métodos de securización y el análisis de los resultados obtenidos en las pruebas.

En lo relacionado al proceso de desarrollo del trabajo en sí mismo, en la parte de **búsqueda de métodos de securización** fue necesario obtener gran cantidad de información teórica sobre diferentes ámbitos como los servicios web, el estilo de REST y refrescar algunos conceptos de seguridad web que, a pesar de tener algo de experiencia previa en este campo gracias a ciertas asignaturas realizadas en años anteriores, no fue suficiente para crear un servicio web securizado desde cero, por lo que también se aprendió a partir de diferentes tutoriales encontrados en Internet como [16]. También hubo ciertos **problemas con las dos versiones de OAuth** (1 y 2) debido a la **confusión** que existe entre éstas dos sobre el servicio de seguridad que proporcionan, y es que se pueden encontrar en la web numerosas discusiones sobre el uso de OAuth para obtener sólo autenticación cuando, como ya se ha dicho anteriormente, éste no es su objetivo principal sino una consecuencia de proporcionar **autorización**.

Para la **implementación** de los servicios web de ejemplo empleados en las **pruebas**, la primera dificultad que surgió fue construir todo el entorno de desarrollo. Si bien configurar el servidor no es una tarea larga, al no haberlo hecho anteriormente, se convirtió en algo complicado cuando en realidad sólo era necesario añadir unas pocas líneas para configurar el servidor Apache. Especialmente con la configuración para que aceptara conexiones con SSL, a pesar de que crear los certificados y

claves para su funcionamiento se hace de forma casi automática con una herramienta que ya proporciona el propio servidor, conseguir que todo funcionara y "coincidiera" la configuración de seguridad del servidor con la del servicio web llevó algo **más de tiempo de lo esperado**. Asimismo, la **documentación** e información en la red sobre la gestión del servidor y de algunas de las herramientas utilizadas se antoja algo **escasa** e, incluso en algunos casos, confusa. Un ejemplo de esto podría ser la **falta de ejemplos** de OAuth1 con JAX-RS 2.0 (la versión empleada) ya que la mayoría de los encontrados son algo antiguos y aparecen implementaciones de la versión 1.0 que es ligeramente diferente mientras que para OAuth2, al ser más reciente, sí que abundan.

6.4. Futuras líneas de trabajo

El **alcance** de este proyecto se ha **limitado** a los tipos y métodos de seguridad que se han considerado más interesantes para proteger un servicio web RESTful pero esto no significa que sean los únicos, y es que existen todavía muchas maneras de hacer un servicio web lo más seguro posible para evitar multitud de ataques diferentes. La **validación de inputs** o la **codificación de outputs** son algunos ejemplos de aspectos que no se deberían menospreciar, pues también son una gran fuente de vulnerabilidades y hay formas muy sencillas de evitar las más comunes.

Otro factor a tener en cuenta y sobre el que se podría haber trabajado más son otras **implementaciones** de un servicio web con REST a parte de Jersey como otros frameworks o lenguajes en que la su creación podría ser más simple o intuitiva. Estudiar el uso del framework Java Spring con su plug-in "**Spring Security**" [17] o el framework disponible en varios idiomas **RESTfulie** [18] serían unos buenos puntos de partida para extender este trabajo.

Si se desea, también podría investigarse la implementación de **OAuth2 delegando** parte de la responsabilidad de la seguridad del servidor a un proveedor externo, que usualmente proporcionan sus servicios a través de su propia nube. En este caso, podría utilizarse una cuenta de Google en la **Google Cloud Platform** para acceder a un recurso web protegido [19] u otras alternativas como **Authlete** [20]. Aun así, habría que prestar mucha atención a la principal desventaja que podría suponer securizar un servicio web de esta manera dado que se estaría confiando un importante aspecto de la protección a una entidad ajena. Además, aunque poco probable, habría que considerar que, si el servidor que proporciona el WS se encuentra en una VPN donde no tuviera acceso a Internet (o una situación similar), no podría contactar con el proveedor externo y este tipo de mecanismo no se podría emplear para securizarlo.

BIBLIOGRAFIA

[1] "Roy Thomas Fielding (University of California, Irvine)", "Architectural Styles and the Design of Network-based Software Architectures", Chapter 5", 18/02/16, disponible en: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[2] IBM, "New to SOA and web services", 26/02/16, disponible en: http://www.ibm.com/developerworks/webservices/newto/service.html

[3] "Roger L. Costello", "Building Web Services the REST Way", 18/02/16, disponible en: http://www.xfront.com/REST-Web-Services.html

[4] "Erlend Oftedal, Andrew van der Stock", "REST Security Cheat Sheet", 10/03/16, disponible en: https://www.owasp.org/index.php/REST_Security_Cheat_Sheet

[5] "Internet Engineering Task Force (IETF)", "The OAuth 1.0 Protocol", 27/03/16, disponible en: https://tools.ietf.org/html/rfc5849

[6] "IETF OAuth", "User Authentication with OAuth 2.0", revisado 14/04/16, disponible en: http://oauth.net/articles/authentication/

[7] "Eran Hammer", "Beginner's Guide to OAuth – Part IV: Signing Requests", 27/03/16, disponible en: http://nouncer.com/oauth/signature.html

[8] "Internet Engineering Task Force (IETF)", "The OAuth 2.0 Authorization Framework", 27/03/16, disponible en: https://tools.ietf.org/html/rfc6749

[9] "OpenID Foundation", "OpenID Connect FAQ and Q&As", revisado 14/04/16, disponible en: http://openid.net/connect/faq/

[10] "Nat Sakimura", "Write an OpenID Connect server in three simple steps", disponible en: https://nat.sakimura.org/2013/07/28/write-openid-connect-server-in-three-simple-steps/

[11] "VeraCode", "Cross-Site Request Forgery Guide", 20/03/16, disponible en: http://www.veracode.com/security/csrf

[12] "Dave Wichers, Paul Petefish, Eric Sheridan", "Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet", 20/03/16, disponible en: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

[13] "Jeff Williams, Jim Manico, Neil Mattatall", "XSS (Cross Site Scripting) Prevention Cheat Sheet", 27/03/16, disponible en: https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#XSS_Prevention_Rules_Summary

[14] "Universidad Nacional Autónoma de México (UNAM)", "El cifrado web", revisado 27/03/16, disponible en: http://revista.seguridad.unam.mx/numero-10/el-cifrado-web-ssltls

[15] "Jersey" "Jersey documentation – Chapter 16: Security, OAuth2 support", 20/04/16, disponible en: https://jersey.java.net/documentation/latest/security.html#d0e12831

[16] "Lokesh Gupta", "Jersey REST API Security Example", 25/04/16, disponible en: http://howtodoinjava.com/jersey/jersey-rest-security/

[17] "SpringSource", "Spring Projects", 19/03/16, disponible en: https://projects.spring.io/spring-framework/

[18] "Caelum", "RESTfulie – REST from scratch", 19/03/16, disponible en: http://restfulie.caelum.com.br/

[19] "Carmine DiMascio", "Google OAuth2 and JAX-RS", 20/04/16, disponible en: http://carminedimascio.com/2014/02/google-oauth2-and-jax-rs/

[20] "Takahiko Kawasaki", "Authorization Server Implementation in Java", 20/04/16, disponible en: https://github.com/authlete/java-oauth-server

ANEXO

A1. TAREAS DEL PROYECTO

TABLA 1
TAREAS DEL PROYECTO

ID. Nombre de la tarea	Depend.
1. Definición de Servicio Web o Web Service	--
1a. Definición de Servicio Web	--
1b. Creación de Servicios Web con SOAP	1a
2. Definición de REST en el contexto aplicación-servidor	1b, 1c
2a. Definición del estilo REST	1
2b. Guías a seguir para construir un WS RESTful	2a
2c. Análisis de por qué se habría de utilizar REST	2b
3. Analizar por qué es interesante aplicar seguridad a REST	2b
4. Buscar los tipos de securización principales que existen	--
5. Comprobar los métodos encontrados	4
5a. Descartar los que no se podrán probar ni analizar	4
5b. Seleccionar los que será posible analizar	4
5c. Seleccionar los que será probar	4
6. Probar los métodos seleccionados	5a, 5b, 5c
6a. Buscar las herramientas y manuales necesarios	5a, 5b, 5c
6b. Realizar pruebas (descartar alguno si es necesario)	6a
7. Análisis de métodos seleccionados y explicar cómo se han implementado al probarlos	6b

A2. FRAMEWORKS PARA WS CON REST

Frameworks más conocidos para crear servicios web con el estilo de REST:

TABLA 2
FRAMEWORKS PARA CREAR WS CON REST

Nombre	Descripción
Jersey. Java (y JavaEE)	Este framework implementa la API JAX-RS 2.0 que proporciona soporte para la creación de servicios web RESTful, utilizando anotaciones para simplificar algunos procesos.
Spring. Java	Framework de código abierto que ayuda al desarrollo de aplicaciones y configuración de aplicaciones Java. Posee asistencia en la creación de aplicaciones web con el paradigma Modelo-Vista-Controlador o MVC y servicios web RESTful. También proporciona diferentes frameworks con los que se puede combinar, siendo uno de ellos para añadir seguridad.
RESTfulie. Java, Ruby, JavaScript, C#, Python	Ofrece algunas características poco comunes que lo diferencian de frameworks similares como Jersey. Múltiples lenguajes.
Ruby on Rails. Ruby	Framework de aplicaciones web de código abierto que sigue el paradigma Modelo-Vista-Controlador e intenta minimizar la cantidad de código a escribir y el tiempo a emplear en configuración.
Slim. PHP	Uno de los microframework más rápidos para hacer RESTful WS. Ocupa poco espacio y sólo incluye las funcionalidades mínimas, lo que implica que sea rápido pero puede que carezca de alguna función aunque existen muchos recursos extras tanto de terceras personas como de su equipo de desarrollo.
Django. Python	Framework web gratuito y de código libre que sigue el paradigma MVC, desarrollado con el objetivo de facilitar la creación de aplicaciones web poniendo gran énfasis en la reusabilidad e interacción de los componentes. El lenguaje Python es usado para gestionar la configuración, los ficheros y modelos de datos. Es utilizado por sitios web muy conocidos como Mozilla o The Washington Times.
Nancy. ASP.NET	Framework reciente y poco conocido pero ligero y rápido para construir servicios basados en HTTP con .NET. Construido por su comunidad (es de código libre), pretende simplificar la gestión de las peticiones y respuestas HTTP proporcionando un lenguaje específico para ello.

A3. AUTENTICACIÓN OPENID VS OAUTH

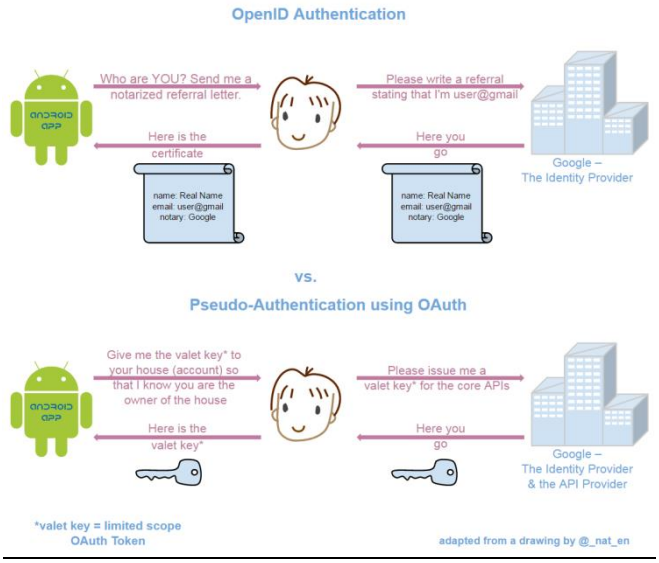

Fig. 1. Comparación entre la autenticación de OpenID y la pseudoautenticación de OAuth.

A4. EJEMPLO DE ACCESO CON SSL

Fig. 2. Proceso básico enumerado para iniciar una conversación cifrada cliente-servidor, utilizado en el acceso a Facebook [14]

A5. MÉTODOS DE SEGURIDAD ANALIZADOS

TABLA 3
MÉTODOS DE SEGURIDAD ANALIZADOS

Autenticación	Autorización	Criptografía
Basic HTTP Authentication	HTML Entity Encoding	SSL (cifrado) / HTTPS
HTTP Digest Access Authentication		Certificados digitales
OAuth 1.0a	OAuth 1.0a	
OAuth 2.0	OAuth 2.0	
OpenID + OAuth 2.0	OpenID + OAuth 2.0	

A6. RESULTADOS DE PRUEBAS FALLIDAS

Durante la última etapa de este trabajo, después de haber acabado las **pruebas** con OAuth2, se **tenía intención de incorporar** también en el trabajo principal los resultados de las pruebas con OAuth2 y OpenID. A pesar de esto, no fue posible obtener resultados satisfactorios con una seguridad funcional debido a falta de tiempo para conseguir que funcionaran, por lo que se decidió no añadirlos.

A continuación se presentará un resumen de los escenarios que no se han incluido porque no ha sido posible implementarlos correctamente.

El primero de estos escenarios no contiene autenticación con OpenID, pero se pensó que podría ser interesante obtener la seguridad de **OAuth2** a través de un **proveedor externo**. El motivo principal para su elección fue no sólo porque simplificaba en gran medida la implementación de la parte del servidor, sino porque permitía encargar este proceso a una entidad con más conocimiento en la labor, de forma que se redujera la probabilidad de que se explotara una vulnerabilidad debido a un error en la propia implementación del WS. Aun así, como es de esperar, se debe confiar plenamente en esta entidad, así como poder acceder a ella en cualquier momento ya que, si no se pudiese obtener el servicio de seguridad que ésta proporciona, los clientes no podrían acceder al WS. Para este caso, se intentó utilizar el servicio de Google [19].

El segundo escenario del cual fallo su ejecución es similar al anterior, pues también se pretendía obtener la seguridad a través de una entidad externa: fue la implementación de OpenID junto a OAuth2 usando a la aplicación **Authlete** como servidor para la **autenticación y autorización** [20]. Con las ventajas e inconvenientes que depender de una entidad externa supone, este debería haber sido (gracias a la adición de un conexión HTTPS) una de las mejores maneras de obtener la mayor seguridad posible.