Universidad de Buenos Aires
Facultades de Ciencias Económicas, Ciencias Exactas y Naturales e Ingeniería

Carrera de Especialización en Seguridad Informática

Trabajo Final

"Seguridad en implementación y configuración de Servicios Web tipo REST de Java en servidores Glassfish"

Autor:
David Arteaga Grigoriev

Tutor de Trabajo Final:
Dr. Pedro Hecht

Buenos Aires, Nov 2019
Cohorte 2017

[Página intencionalmente en blanco]

Declaración Jurada de origen de los contenidos

Por medio de la presente, el autor manifiesta conocer y aceptar el Reglamento de Trabajos Finales vigente y se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual.

Firmado

David Arteaga Grigoriev

Licencia

Este trabajo está publicado con licencia Creative Commons: Atribución 4.0 Internacional (CC BY 4.0)

Usted es libre para:

Compartir, copiar y redistribuir el material en cualquier medio o formato.

Adaptar, remezclar, transformar y crear a partir del material Para cualquier propósito, incluso comercialmente.

El licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia Bajo los siguientes términos:

- **Atribución:** Usted debe darle crédito a esta obra de manera adecuada, proporcionando un enlace a la licencia, e indicando si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo del licenciante.
- **No Comercial:** la explotación de la obra queda limitada a usos no comerciales.

No hay restricciones adicionales, usted no puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otros hacer cualquier uso permitido por la licencia. Aviso usted no tiene que cumplir con la licencia para los materiales en el dominio público o cuando su uso esté permitido por una excepción o limitación aplicable. No se entregan garantías. La licencia podría no entregarle todos los permisos que necesita para el uso que tenga previsto. Por ejemplo, otros derechos como relativos a publicidad, privacidad, o derechos morales pueden limitar la forma en que utilice el material.

Más información: **https://creativecommons.org/licenses/by/4.0/deed.es**

Tabla de Contenidos

1. Introducción

El presente trabajo final de Especialización en Seguridad Informática, tiene como enfoque mostrar aspectos de seguridad vinculados a la configuración *Servicios Web* en servidores Glassfish desarrollados en el lenguaje orientado a objetos Java con enfoque web en entornos empresariales. Así mismo, mostrar la implementación de estos de manera segura con el fin de ser consumidos por usuarios finales de modo fiable. El objetivo principal es dejar en evidencia las consecuencias que puede conllevar una incorrecta implementación y configuración de un *Servicio Web*, y bajo esta premisa, brindar una guía detallada de cómo realizar una correcta implementación con configuraciones básicas de seguridad. Con esto se brindan múltiples sugerencias y criterios que se deben tener en cuenta para la implementación y configuración de un *Servicio Web* seguro.

Un *Servicio Web* o *Web Service (WS de ahora en adelante),* es un método de comunicación entre dispositivos, en donde uno de ellos hace de servidor, el cual expone servicios a través de la *web* o *world wide web* (www) para que otros dispositivos, llamados clientes, puedan hacer uso de dichos servicios expuestos por el servidor. Por servicio se entiende como implementaciones, en su mayoría en servidores dedicados, que se encuentra a la escucha de peticiones, que al ser recibida una petición, realiza operaciones que son independientes al cliente o dispositivo que ha hecho el llamado del servicio. Por último el servidor que expone el servicio, devuelve una respuesta de la operación realizada por la petición del cliente.

Existen diferentes tipos de *WS* basados en diferentes principios arquitectónicos: Simple Object Access Protocol (*SOAP*), Web Service Description Language (*WSDL*) y Representational State Transfer (*REST*), el presente documento está basado en los *WS* de tipo *REST*, ya que son los más usados en dispositivos móviles por su simplicidad y tamaño de

transmisión en los mensajes entre el servidor y los clientes que hagan uso de los sus servicios expuestos [1].

Dicho esto y teniendo en cuenta los objetivos planteados, se ha realizado una Prueba de Concepto (*PoC* de ahora en adelante por sus siglas de *Proof of Concept*) de implementación de un *WS* con configuraciones básicas de seguridad y otro sin ninguna configuración de seguridad y valores por defecto. El fin de esta prueba es mostrar las repercusiones que tiene una mala o nula configuración al momento de prestar un servicio de este tipo y el cómo esto afecta la confidencialidad e integridad de los datos, los cuales son dos de los pilares básicos de la seguridad informática. Ambas implementaciones presentan los mismos recursos en el *WS* los cuales quedan a disposición de los usuarios finales que harán uso de estos recursos a través de sus clientes. Las principales diferencias entre las implementaciones son las configuraciones de roles, autenticación y permisos de autorización. Por consiguiente, el primer *WS REST* implementado tiene el fin de mostrar las consecuencias de una implementación y configuración de un *WS* sin ninguna medida a nivel de seguridad, el cómo afecta esto la integridad y la confidencialidad de los datos manipulados. Por otra parte la segunda implementación de *WS REST,* muestra cómo con unas recomendaciones básicas de seguridad, a la hora de la implementación y configuración de un *WS,* se pueden proteger estos principios de seguridad informática que conllevan a una capa de protección extra para los datos manipulados por el servicio ante un riesgo de un ataque de terceros.

En el trabajo se muestran, para ambas implementaciones, los flujos de datos a la hora de hacer uso de los recursos en los servicios expuestos y cómo a través de un escaneo de datos en la red, simulando un ataque de un tercero, se puede evidenciar la exposición de información sensible por una mala configuración en un *WS*.

Por último se presentan los conclusiones encontradas para la *PoC* entregada, la comparación de los resultados obtenidos a través de las implementaciones hechas y conclusiones con respecto a los objetivos tratados.

2. Desarrollo Teórico

2.1. Alcance y objetivos

La implementación realizada y descrita en el presente documento brinda una guía de forma detallada para una correcta implementación y un desarrollo seguro de un *WS REST* básico en un servidor Glassfish, su respectiva descripción con sugerencias en el ámbito de la seguridad y cómo hacer un consumo seguro de los recurso expuestos por los *WS* implementados. El objetivo principal prioriza en mostrar cómo afecta la integridad y confidencialidad en los datos manipulados en una incorrecta configuración e implementación de un *WS* que se comunican a través de transacciones realizadas por usuarios finales y los servicios expuestos por dicho *WS.*

Se definieron las siguientes métricas y parámetros necesarios para el desarrollo de los *WS* implementados:

- Implementación insegura y por defecto de un *WS* con dos servicios expuestos en un servidor Glassfish con configuraciones por defecto. Las configuraciones por defecto e inseguras usadas en esta implementación son explicadas en la sección 3.2.
- Implementación segura y con configuraciones básicas de seguridad de un *WS* con dos servicios expuestos en un servidor Glassfish con configuraciones de seguridad básicas. Las configuraciones básicas usadas para esta implementación son explicadas en la sección 3.3.
- La implementación de los *WS* se hizo en Java Enterprise Edition como un proyecto web empresarial, usando el entorno de desarrollo integrado (*IDE*) Netbeans 8.1. El montaje del *WS* al IDE se encuentra en el Anexo adjunto.
- Los dos servicios que serán expuestos para ambas implementaciones serán los mismos servicios, con el fin de ver el manejo de flujo de

datos y su correcta manipulación al ser consumidos por un usuario final. Con esto se deja evidencia de cómo con una correcta configuración no se ven expuestos los principios de confidencialidad e integridad de los datos.

- El servidor hace uso de una Base de datos (*BD*) de modelo relacional el cual está conectado por los recursos expuestos por los servicios del *WS*. Para la *PoC* se realizó en una *BD* en MySQL al contar con una edición de comunidad que es gratuita y de código libre para los desarrolladores y no se requiere la compra de licencias. El montaje de la *BD* en MySQL de los datos que hacen uso los servicios implementados se encuentran en el Anexo adjunto.
- Para consumir los servicios expuestos por ambos *WS* se hizo uso del navegador Chrome y la herramienta de peticiones web Postman. Las pruebas de funcionalidad realizadas se encuentran en la sección 3.2.3 para el *WS* inseguro y en la sección 3.3.5 para el *WS* seguro.
- El escaneo realizado entre el extremo final que hace consumo del *WS* y el servidor en sí que expone el WS es un escaneo a nivel local de la red. Este escaneo permite ver cómo se vulnera la confidencialidad y la integridad de los datos entre las dos partes comunicadas, dejando en evidencia las falencias de un *WS* sin configuraciones de seguridad y como el mismo escaneo no puede vulnerar estos principios para un *WS* con configuraciones básicas de seguridad planteadas en este documento. El detalle del escaneo de datos se encuentra en la sección 4.

2.2. Definición *REST*

Para poder desarrollar un *WS REST* es necesario entender que *REST* es un conjunto de arquitecturas, este conjunto de arquitecturas serán definidas por seis propiedades básicas para considerar una implementación de tipo *REST* [2].

La primera propiedad que define una implementación *REST* es que los servicios no se publican en llamadas a procedimientos remotos (*RPC* de sus siglas en inglés *Remote Procedure Call*). Por *RPC* se entiende como un mecanismo que permite la ejecución de procesos, comúnmente en otras computadoras, bajo un mismo espacio de red compartida o en un mismo sistema [3]. Entonces, *REST* define que los servicios no se publican en un conjunto arbitrario de métodos u operaciones, en *REST* no se publican interfaces con métodos, donde en la programación orientada a objetos se define interfaz como un medio común entre objetos que cumplen con características iguales [4]. Un ejemplo sencillo para entender interfaces se puede declarar una interfaz de tipo auto, donde se definen propiedades del auto, por ejemplo cuántas personas entran dentro del auto, otra propiedad puede ser la velocidad máxima del auto; entonces cuando cree un nuevo objeto de tipo auto, por ejemplo una furgoneta, este nuevo objeto extiende de la interfaz auto, debe cumplir con las propiedades que se definieron en dicha interfaz. Para el ejemplo sería número de personas dentro del auto y velocidad máxima para ese nuevo objeto furgoneta que extiende de la interfaz de auto [4].

La segunda propiedad que define *REST* habla de los recursos y la forma en que se publican. Por recurso se entiende como una entidad que representa un concepto de negocio y puede ser de acceso público [2]. Un ejemplo de recurso puede ser *'verificacionLogin'*, donde un usuario al acceder al recurso puede enviar su usuario y contraseña y el servicio, al realizar operaciones ajenas al cliente, devuelve al usuario un *token* de acceso o una respuesta negativa o positiva de verificación de credenciales.

La tercera propiedad dice que cada recurso posee un identificador único y global que lo distingue de los diferentes recursos que se exponen en un *WS,* que

aunque existieran objetos con la misma información se definen como diferentes objetos para el recurso siempre y cuando tengan un identificador diferente.

La cuarta propiedad nos dice que un recurso posee un estado interno, este no puede ser accedido directamente desde el exterior y lo que en realidad es accesible es una o varias representaciones del estado original. La implementación del recurso es lo que define la parte visible para el exterior y la representación del estado que se le va a entregar al haya invocado al recurso que se expone [2]. Un ejemplo de esto es si se define un objeto de tipo Usuario, este usuario consta de atributos o características, que son definidas previamente por el desarrollador. Este objeto puede tener por ejemplo login, contraseña, nombre e imagen de perfil. Si otro usuario accede a un recurso llamado *'darUsuarioPorIdentificador'* donde este recibe un identificador por parámetro, el cual representa al identificador de otro usuario, el recurso devolverá al usuario que esté relacionado con ese identificador. Pero es en el desarrollo donde se define hasta qué punto se le devuelven las propiedades de ese otro usuario por el cual se está preguntando, puede que solo se le devuelvan los atributos de nombre e imagen de perfil, ya que el usuario que realizó la petición en un principio no tendría que poder ver el login y contraseña de otro usuario. Esta información puede ser entregada de distintas formas de documentación por ejemplo *XML*, *HTML* o *JSON*.

La quinta propiedad que define *REST* es que todos los recursos comparten una interfaz única y constante, ya que no se pueden definir operaciones arbitrarias sobre el recurso. Como todas comparten una misma interfaz, y como se explicó anteriormente, todos los recursos tienen las mismas operaciones, estas operaciones nos permiten manipular el estado público del recurso expuesto [2]. Un *REST* típico tiene definidas cuatro operaciones que se utilizan a través de *HTTP* para que sea coherente con la definición del protocolo [1]:

- **POST:** Utilizado para crear nuevos recursos en el servidor según el servicio expuesto.
- **GET:** Utilizado para recuperar recursos en el servidor. Como definia la cuarta propiedad se recupera la una o varias representación del estado original del recurso.

- **PUT:** Para cambiar o actualizar un recurso se hace a través de esta función y el servicio expuesto para dicho fin.
- **DELETE:** Para borrar o eliminar un recurso se usa esta operación.

La sexta y última propiedad define que las operaciones mostradas anteriormente pueden ser prohibidas o permitidas libremente por el servicio, según la implementación que se haga de la misma y según la función que se exponga en éste. Además de la posibilidad de interrelacionar y referenciar entre los recursos según los identificadores globales asignados [2].

2.3. Seguridad en Servicios Web

Puntualmente la seguridad en servicios web busca resguardar y procesar la información que se genera entre el cliente y el *WS*, adicionalmente de mantener la integridad, confidencialidad de datos y una alta disponibilidad al prestar un servicio, por lo que desde este punto de vista, se aplican varios principios y mecanismos para cumplir con este objetivo:

- Inicialmente se debe asegurar la autenticación de los usuarios y el proveedor del servicio o de los *WS* a consumir.
- La autorización de accesos a los servicios expuestos no serán los mismos para los roles que se han definido según las funciones del *WS.* Por ejemplo, un usuario que hace compras sobre una página dedicada a Ecommerce, no tendrá autorización a los servicios que tenga el proveedor de dichos productos, o un usuario y el administrador del *WS*. Los accesos en cada caso serán muy diferentes y estarán determinados por su alcance, rol y funciones que deba tener cada grupo de usuarios. Es esto lo que nos brinda la certeza que un usuario determinado no usará recursos expuestos más allá de donde esté autorizado. Además de negar a un usuario no autenticado el que pueda hacer uso de cualquier servicio expuesto por el *WS*.
- La necesidad de autenticación y autorización para el consumo de varios servicios presentados en un *WS* no se deben generar en cada uno de ellos,

puesto que un *WS* al tener muchos servicios embebidos va a tener diferentes niveles de acceso a cada uno de estos servicios presentados.

- A nivel de comunicación mantener la privacidad e integridad de los datos es un tópico muy importante, ya que la información no viaja cifrada. Es necesario generar una comunicación fiable y segura mediante SSL o TLS. Adicional también se puede implementar firmas XMLDSIG, que permiten la firma de partes específicas del documento XML aunque ésta es una solución aplicada a *SOAP* [5].

Los principios de seguridad y sus métodos recomendados (citando algunos) para cumplir estos principios en los *WS* a grandes rasgos son los siguientes:

Principio de Seguridad	**Recomendación**
Autenticación de Usuarios	Tokens
	Certificados SSL o TLS
	Autenticación HTTP (*BAM*)
Autorización de usuarios	SAML
	OAuth
	Constraints
Integridad	SSL
	TLS
	HTTPS
	WS-Signature
No Repudio	Logs
Confidencialidad	SSL
	TLS
	HTTPS
Manejo de Políticas	OAuth
	WS-Policy

Tabla 1: Principios de Seguridad y Métodos.

Algunas de las organizaciones encargadas de velar por la seguridad de los *WS* son las siguientes:

- W3C (Consorcio World Wide Web)
- OASIS (Organization for the Advancement of Structured Information Standards)
- IBM/Microsoft/Verisign/RSA Security

2.4. Conexión

A pesar de ser un tema de comunicación, se relaciona más la seguridad de conexión con el servidor que aloja el *WS*, como lo puede ser Jboss, Glassfish, Wildfly, entre otros. Se nombra esta parte ya que es un tema fundamental en la implementación de la seguridad del *WS* seguro, puesto que se debe garantizar la integridad y confidencialidad de los datos que se emiten y se reciben a través de un canal seguro de comunicación entre los dispositivos finales y el servidor que expone los servicios. Es por esto que la implementación planteada como segura se hará a través de un canal seguro con un certificado auto firmado por el servidor de Glassfish debido al alcance planteado. Un certificado firmado por una autoridad requiere una inversión extra y un alojamiento que soporte la transferencia de datos por *SSL* o *TLS*, la prueba local con un certificado autofirmado se consideró suficiente para la PoC.

Cuando se ejecuta un *WS,* es de vital importancia, la seguridad en el envío y recepción de datos. Normalmente existen muchos mecanismos y métodos criptográficos que ayudan a velar por la seguridad. La gran mayoría tienen librerías que permiten la integración de estos con los lenguajes usados para el desarrollo de los *WS* y en las plataformas o clientes donde se hacen consumo de los servicios expuestos. Esto permite lograr una correcta transmisión de datos, generando un incremento en las medidas de seguridad de la información y garantizando la integridad de los datos entre los interlocutores. Dicho lo anterior, se genera una capa más de seguridad de la aplicación y de los servicios presentados por el *WS*

siempre y cuando esté implementado de ambos lados y estén coordinados con los métodos que se estén usando para lograr la comunicación segura.

Para la capa de transporte, una forma de brindar seguridad desde la fuente hasta el destino, y que tiene la capacidad de realizar autenticación e identificación de participantes de una comunicación, es la implementación de un protocolo de seguridad como por ejemplo el uso de sockets seguros (*SSL* de sus siglas en inglés de *Secure Sockets Layer*) o su versión actualizada *Transport Layer Security* (*TLS*). El emisor y el receptor, en el caso del *PoC* el cliente y el servidor, se autentican y envían los datos encriptados para garantizar una comunicación cifrada por medio de alguno de los protocolos nombrados.

El protocolo que permite esta comunicación en la capa de aplicación es *HTTPS* siendo el mecanismo por el cual se realiza la autenticación del cliente y del servidor mediante certificados. El protocolo de *SSL* transmite la información por la red en formato cifrado y es así como evita la modificación de los datos a su destino. Es posible aplicar servicios web *HTTPS* a todos los tipos de clientes.

El funcionamiento en su forma básica es presentando su certificado al cliente para determinar la identidad del servidor. El cliente no necesita presentar su certificado al servidor para que este determine la identidad de aquel, aunque de ser necesario, es posible modificarlo y depende de la criticidad de los recursos brindados por el *WS* y la información que este maneje. La validación del cliente con el servidor para el *PoC* se hace directamente con las credenciales enviadas en las cabeceras de la solicitud al *WS.* Para el caso de la implementación segura, consta de un certificado autofirmado, el navegador o aplicación que use el usuario, mostrará una advertencia al usuario de que nuestro servidor posee un certificado que no ha sido firmado por una entidad no reconocida, por lo que el usuario deberá aceptar la advertencia o instalar el certificado para poder acceder a los recursos expuestos.

2.5. Autenticación y Autorización

La distinción entre autenticación y autorización es un factor fundamental para el correcto entendimiento de una implementación de un *WS REST.* Según sea su configuración, este hará la toma de decisiones en cuanto al rechazo o aceptación de peticiones que desean hacer los usuarios finales al tratar de consumir los servicios expuestos en un servidor.

La autenticación es la verificación del cliente usado por el usuario al invocar un servicio del cual éste quiera hacer uso. Para la *PoC* elaborada en el documento, la autenticación se hace por *Método de Autenticación Básica* (*BAM* de sus siglas en inglés para *Basic Authentication Method*), el cual verifica un par de credenciales de usuarios definidas en la configuración de autenticación del servidor. Para este caso, se desarrolló en Glassfish 4 y la configuración de los roles dentro de la implementación del *WS* se hace en los archivos de configuración del proyecto y las credenciales para dichos roles de usuarios son definidas en el servidor donde se hace el despliegue de los servicios. La configuración de los roles y usuarios es explicada en profundidad en la sección 3.3.2.

Por otro lado la autorización define que un usuario una vez ya autenticado, son verificados sus permisos según un rol dado dentro de las configuraciones del servicio y si dicho rol tiene autorizado el consumo del servicio al cual quiere obtener acceso, se le sea permitido el consumo de este. Por otro lado, si su rol no tiene la autorización necesaria para hacer consumo de un servicio expuesto, se le negara el consumo y se le informará que no tiene autorización sobre el recurso al cual se quiere acceder pese a que la autenticación haya sido válida.

2.6. Funcionamiento del Basic Authentication Method (*BAM*)

Mediante el uso de *BAM* se busca brindar una capa de seguridad a los *WS* implementados en el servidor con un desarrollo y configuraciones de seguridad

básicas, por lo cual se brindará el análisis de la autenticación desde esta solución [6].

El *BAM* es un mecanismo de autenticación que permite que una aplicación cliente, a menudo una aplicación web, actúe en nombre de un usuario final, pero con el permiso del usuario con un rol específico. Las acciones que un usuario final puede realizar son llevadas a cabo por un servidor de recursos según los permisos que tenga su rol y si la autenticación se hace de manera exitosa. Del lado del usuario, éste aprueba la identidad del servidor de recursos al tratar de establecer una conexión segura aceptando el certificado que el servidor posee [6].

Para la *PoC* en la implementación del *WS* seguro, se muestra como Glassfish tiene integrado el manejo de usuarios donde en la configuración se especifica el "reino" al que va a pertenecer un usuario y hace el manejo de los usuarios que se hayan configurado a forma de servidor de autorización. Este punto es explicado con más detalle en la sección 3.3.4.

Esta forma de Autenticación y Autorización, funciona mediante la arquitectura *REST*, que de una forma básica es en donde se hace uso de los métodos *HTTP* para realizar las operaciones de alta (**POST**), baja (**DELETE**), modificación (**PUT**) y consulta (**GET**) de información explicadas previamente en la sección 2.2 de definición *REST*. Recordando, estas operaciones proporcionan un mecanismo de filtro o consulta de selección de resultados y brindan un mecanismo de respuesta parcial, en donde se puede indicar qué campos de información se desean obtener, modificar, eliminar o agregar, normalmente para ahorrar ancho de banda e incrementar velocidad en consultas desde dispositivos con recursos limitados y garantizar una manipulación de datos sensibles desde el punto de vista para el cual se haya implementado el servicio [2].

Una de las maneras de mantener la autenticación del usuario es por medio de *tokens* sin información de estado o *stateless* que da la posibilidad de no almacenar las contraseñas, lo cual evita que las aplicaciones tengan acceso completo a la información del usuario. Además permite revocar el acceso a la cuenta del usuario y de verse comprometida la aplicación por un evento de seguridad, las credenciales de los usuarios no se verán comprometidas. El uso de *tokens* para la implementación segura se encuentra definida dentro del servidor

Glassfish. Este consta de un servicio de creación de *tokens* para los usuarios que se autentiquen con el servicio y les retorna el *token* generado para usar en llamados de recursos posteriores sin la necesidad de autenticarse en cada llamado. Estos *tokens* tienen un tiempo de vida útil y solo pueden ser usados el tiempo para el cual hayan sido configurados. Estas configuraciones son explicadas para la *PoC* en la sección 3.3.4.

Si bien el usuario se puede autenticar en el servicio web con un par usuario/contraseña también lo puede hacer a través de un proveedor de alguna *Interfaz de Programación de Aplicaciones* (*API* de sus siglas en inglés para *Application Programming Interface*) como puede ser ejemplos de proveedores de APIs como Instagram, Google o Facebook. Una vez se realiza la autenticación por medio de un API, cada petición *HTTP* que haga el usuario va acompañada de un *token* en la cabecera. Este *token* no es más que una firma cifrada que permite a una *API REST* identificar al usuario. Este *token* no se almacena en el servidor, si no en el lado del cliente donde es alojado (por ejemplo en *localStorage* o *sessionStorage*) y el *API* es el que se encarga de descifrar ese *token* y redirigir el flujo de la aplicación en un sentido u otro. Como los *tokens* son almacenados en el lado del cliente, no hay información de estado y la aplicación se vuelve totalmente escalable. Se puede usar el mismo *API* para diferentes aplicaciones (*Web*, *Mobile*, *Android*, *iOS*, y demás), donde el envío de datos debe ser en el formato indicado, generar y descifrar *tokens* en la autenticación y posteriores peticiones *HTTP* a través de un *middleware* o una lógica de intercambio de información entre servidores [7].

A nivel de Seguridad, al no utilizar cookies para almacenar la información del usuario, se pueden evitar ataques *CSRF (Cross-Site Request Forgery*) que manipulen la sesión que se envía al servidor o *backend*. Se puede realizar el *token* de tal manera que expire dado un tiempo, brindando una capa extra de seguridad.

Finalmente, a medida que crecen los sistemas y aumenta la cantidad de componentes que necesitan acceso a los servicios de autenticación y a la información manipulada por los recursos, los simples enfoques de seguridad comenzarán a convertirse en un problema de mantenimiento. Lo que se necesita es un enfoque centralizado para la administración de identidades y un servicio para todo el sistema que pueda proporcionar y administrar esos datos de manera segura.

Aquí es donde entran los estándares como *BAM* o *OAuth2* introduciendo una nueva capa en la arquitectura, y una complejidad adicional [8].

2.7. Hash-based Message Authentication (*HMAC)*

Una de las desventajas principales del *BAM,* es que se deben enviar los pares de autenticación, usuario y contraseña, por cada solicitud que el cliente haga de un recurso expuesto por el *WS*. Otra desventaja de *BAM* es la falta de protección en contra de la manipulación de los *encabezados (headers)* y el *cuerpo (body)* de las solicitudes hechas por el usuario. Es por esto que otra medida de seguridad usada para mantener la integridad de los datos y la cual también permite verificar la autenticación de usuarios en un *WS,* es a través del uso de *HMAC* de sus siglas en inglés para *Hash-based Message Authentication Code*.

HMAC es un tipo específico de código de firma de mensajes que está relacionado con una función criptográfica de hash y una llave secreta compartida por los pares que quieren establecer una comunicación, que para el caso de la *PoC* es el cliente y el *WS* seguro que expone los servicios. *HMAC* facilita la autenticación de mensajes porque permite enviar información hacia el *WS* seguro adjuntando un *digest*. El *digest*, que se entiende como el *hash* generado a partir de un texto plano por algún algoritmo de encripción de una sola vía, al ser recibido por el servidor le da la posibilidad de validar la integridad del texto gracias a la función criptográfica y la llave secreta compartida escogidas. Esto permite corroborar la integridad de los datos enviados desde el cliente comparando el *hash* y la información que es enviada por éste [9].

Es por esto que en vez de enviar el par de llaves, usuario y contraseña, cada vez que se haga una solicitud por parte del usuario al servicio, se tendrá una versión del *hash* de la información que se está enviando, esta información es añadida en el encabezado de la solicitud. El *digest* generado con la técnica *HMAC* tendrá la información que desea enviarse al servidor que expone el servicio y solo las partes que conozcan la llave secreta con la que fué generado el *digest*, podrán validar la integridad de los datos que están siendo enviados en los otros campos de la

solicitud. También es aconsejable agregar información de un timestamp o un número aleatorio en la creación del *HMAC* para prevenir aún más la manipulación en el encabezado y cuerpo de la solicitud.

La definición para la creación de una *HMAC* tomada desde *RFC 2104* [10]:

$$\mathrm{HMAC}(K, m) = \mathrm{H}\left((K' \oplus opad) \parallel \mathrm{H}\left((K' \oplus ipad) \parallel m\right)\right)$$

$$K' = \begin{cases} \mathrm{H}(K) & K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

Imagen 1: Definición formal de HMAC.

Para el desarrollo de la *PoC* del *WS* seguro y con el fin de poder demostrar la seguridad que provee *HMAC*, se desarrolló un cliente capaz de enviar solicitudes a los servicios del *WS* seguro. La prueba realizada e información detallada del cliente y uso de *HMAC* se encuentra en Seguridad con HMAC en la sección 4.2.

2.8. OAuth 2.0

El estándar de autorización más común usado por industrias de alta envergadura, es *OAuth 2.0*. El estándar de *OAuth 1.0* contiene varias vulnerabilidades que comprometen el nivel de seguridad y es más difícil de implementar que *OAuth 2.0*, por lo que en esta sección solo se hablara de *OAuth 2.0*. [14]

El funcionamiento de *OAuth* es en esencia que el cliente debe obtener un *token* de acceso previamente para poder hacer uso de los servicios expuestos en el *WS*. Esto hace necesario que se pre-registre con el Servidor de autorización y deba autenticarse en el punto extremo del *token*. El *User Account and Authentication* (*UAA*) usa autenticación básica para este punto final, como lo sugiere la especificación *OAuth*. Si un cliente no está actuando en nombre de un Usuario dentro de un rol específico, el Servidor de Autorización o proveedor de OAuth, puede permitirle obtener *tokens* por derecho propio, directamente desde el punto final del *token* [8].

La otra concesión que se puede realizar es el *código de autorización*. Este es el más común y permite al cliente delegar la autenticación de los usuarios. Los

proveedores de aplicaciones deben informar a los usuarios que nunca deben ingresar sus credenciales en ninguna otra aplicación que no sea el servidor de autorización (o sus delegados de confianza), de modo que una aplicación cliente maliciosa tenga menos posibilidades de engañar al usuario para que entregue sus credenciales. Este caso de uso es una de las razones principales para usar *OAuth* para servicios de gran envergadura y alta escalabilidad. Para la *PoC* al ser una aplicación sencilla y de muestra, no se implementó *OAuth* pero es una de las partes más fundamentales en las aplicaciones de entrega de servicio como los son los *WS.*

La concesión de token de código de autorización para *OAuth* procede de la siguiente manera:

- El *OAuth provider o Authorization Server* autentica al usuario de cualquier manera necesaria. Este paso es obligatorio, pero es específico de la implementación del servidor de autorización y no forma parte de la especificación del despliegue de recursos en el servicio (por ejemplo, es diferente para Google o Twitter, entre otros).
- El cliente inicia el flujo de autorización y obtiene la aprobación del servidor de autorización para actuar en nombre de un usuario con un determinado rol. La aprobación es tratada por el proveedor y según haya sido implementada la parte de verificación de identidad del usuario que quiera ser autorizado ante el servicio.
- Si es autorizado con éxito, el servidor de autorización emite un código de autorización o *token de acceso* para el cliente que usa el usuario el cual invoca el recurso expuesto.
- El cliente intercambia el código de autorización en cada solicitud que le es enviada al recurso o *API* y es validado por el servidor de autorización con cada solicitud que envíe [11].

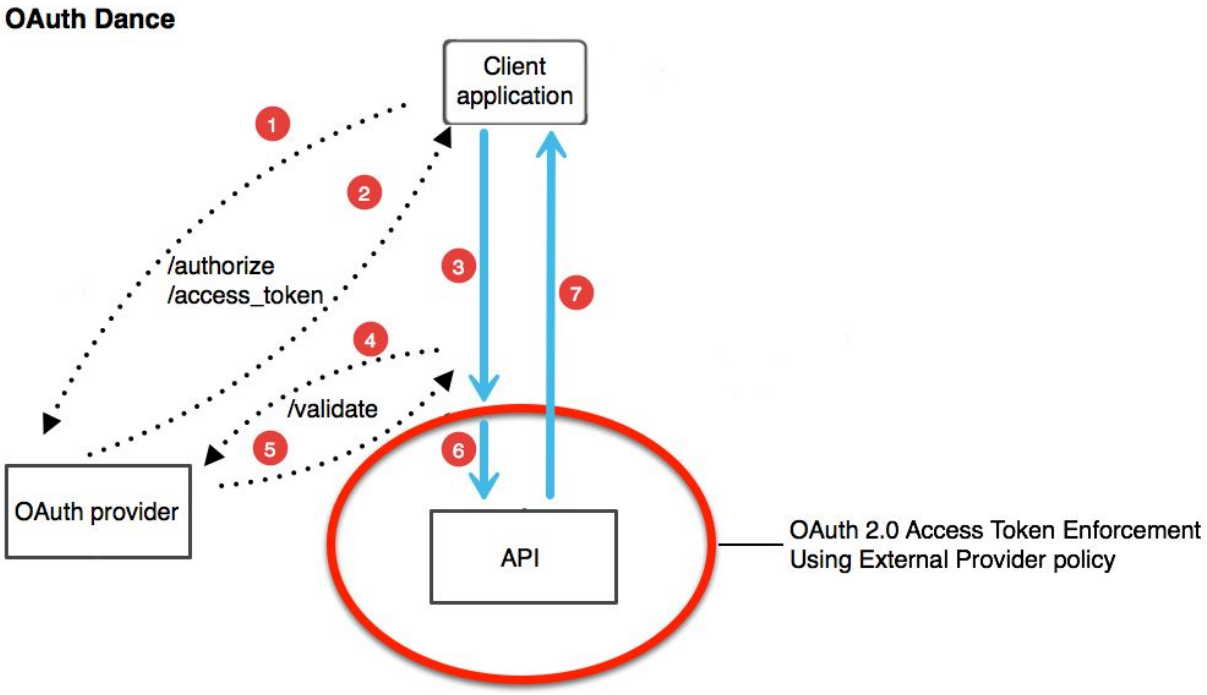


Imagen 2: "Baile OAuth" Flujo de información del estándar OAuth 2.0.

Teniendo en cuenta los pasos anteriores, normalmente lo primero que se requerirá en *OAuth* es una breve conversación entre el usuario final y el proveedor de *OAuth*, el cual debe dirigir el flujo mediante el uso de redireccionamientos *HTTP*. Recordando que como medida principal de seguridad entre la transferencia de datos se haga por un canal seguro aplicando *HTTPS/TLS*. Después de una correcta autorización el proveedor genera un código de autorización utilizado por el cliente para intercambiarlo por un token de acceso, después de esto el usuario solicita específicamente un redireccionamiento al cliente con una redirección de *URL* donde la aplicación toma el código de acceso de la *URL* y lo usa para solicitar el *token de acceso*. Una de las amenazas de seguridad en *OAuth* es el robo de *códigos de acceso* solicitando un redireccionamiento arbitrario, por lo que los Servidores de Autorización se protegen contra esta amenaza al requerir que los clientes registren uno o más *URIs* de redireccionamiento [11].

El servidor de autorización o proveedor *OAuth* se denomina así porque proporciona una interfaz para que los usuarios confirmen que autorizan al cliente, desde donde están accediendo al recurso, a actuar en su nombre y ser debidamente reconocidos como el usuario quien dicen ser. *OAuth* proporcionan una

interfaz simple basada en formularios en el caso general, pero también permiten la aprobación automática de ciertos clientes (por ejemplo, si los propietarios de *Authorization Server* consideran que forman parte de la plataforma).

En cuanto al registro de clientes, la especificación *OAuth* menciona explícitamente el registro como un punto de alta relevancia a nivel de seguridad dado que sirve de blindaje contra potenciales amenazas. De forma ejemplificativa, una condición previa si se desean ver los datos de un usuario usando un *API*, el estar registrado previamente para la aplicación que hace uso del *API* que quiere usar por medio de *OAuth*, como es el caso de los *APIs* de Twitter, Facebook, etc. Hay algunos elementos centrales de un registro de cliente requerido por la especificación (un identificador y un secreto si el cliente es de confianza), y varios que son recomendados (valores de alcance legal y *URI* de redirección registrados). Los servidores de autorización a menudo requieren información adicional, que describe la aplicación y el propietario del registro.

Por otro lado, la autenticación en el servidor de autorización en *OAuth* por sí solo no proporciona un protocolo de autenticación para los usuarios. Sugiere enfáticamente que las aplicaciones cliente deben usar la autenticación básica para acceder al punto final del *token*, pero no dice nada sobre la autenticación de los usuarios cuando se necesita su aprobación para una concesión de *token* (solo que deben estar autenticados). Esto es bueno porque hace que la autenticación sea completamente transversal al proceso de aprobación, y los Servidores de Autorización son libres de implementar la autenticación de la forma que ellos elijan [11].

La autenticación basada por una base de datos de cuentas de usuario es posible y la más común. Pero debido a que la autenticación no es parte de la especificación *OAuth*, es fácil de modificar para admitir otros mecanismos de autenticación o fuentes de datos. Por ejemplo, con algunas líneas de configuración, puede pasar de la autenticación basada en formularios a la autenticación con Google o Yahoo haciendo uso de *Open ID*, o a un servicio de directorio empresarial (por ejemplo, *Active Directory* o *OpenLDAP*) para el almacenamiento de datos.

2.9. Gestión de Contraseñas

El propósito del trabajo no cobija la administración de la base de datos según la información que usen los recursos expuestos, si no la forma en que se autentican y autoriza un usuario final frente a un *WS* para hacer uso de los recursos expuestos. Al ser los recursos los que procesan la información y realizan la interacción de manera asíncrona e independiente del cliente con la *BD,* se deben tener en cuenta la gestión y políticas de almacenamiento en la *BD*. A la hora del manejo y almacenamiento de información como lo son contraseñas, lo ideal es guardar un *hash* de la contraseña del usuario en la base de datos, y no almacenarla en texto plano como se ha realizado en la implementación insegura.

Para una implementación segura desde el lado del *WS,* el almacenamiento de información que se realice desde el recurso expuesto es una parte crucial. Este *hash* puede ser agregado en el digest del encabezado de un *HMAC*, por ejemplo, como se explicó en la sección 2.7. Si un atacante obtiene acceso sobre la *BD* o hace uso de un recurso del *WS* mal implementado, el atacante podría ver las contraseñas almacenadas en texto plano. Es por eso que en la implementación segura las contraseñas son almacenadas con su hash *MD5* y son los hashes los que son comparados a la hora de que otro recurso desee autenticar a los usuarios almacenados en la *BD* a nivel de aplicación.

Otro factor a tener presente en la gestión de contraseñas son las políticas de contraseñas que se tengan, ya que un atacante podría comparar los hashes con palabras sencillas o de diccionario y obtener el texto plano de las claves.

Es importante destacar qué escenarios se deberían evitar para poder gestionar unas buenas políticas de contraseñas a almacenar:

- Misma contraseña que usuario, o que contenga algún dato personal.
- Secuencias conocidas tales como “qwerty” o “123456789”.
- Utilizar únicamente números, mayúsculas o minúsculas.

Idealmente, las contraseñas deberían tener al menos ocho caracteres utilizando caracteres especiales, mayúsculas, minúsculas y números.

2.10. Plataforma

La plataforma que se ha utilizado para crear el *WS* es Glassfish 4. A continuación se hacen varias sugerencias y recomendaciones de seguridad para un funcionamiento seguro a la hora del despliegue de servicios para este tipo de servidor.

En primer lugar, debemos cambiar la contraseña por defecto que tiene el usuario administrador dentro del servidor. El cambio de esta configuración varía según el sistema operativo en el cual se haya instalado el servidor. En caso de que esta configuración no sea cambiada, un atacante podría desplegar aplicaciones web en el servidor con las credenciales que vienen configuradas por defecto. Estas aplicaciones pueden ser dañinas para los usuarios o para el mismo servidor que alberga la aplicación.

Es importante también, realizar una correcta definición de roles para cada reino en particular, definiendo cuáles serán los accesos dependiendo el rol de cada usuario, ya sea proveedor, cliente, o administrador para el caso de aplicaciones que exponen servicios web como se explica en la sección 3.3.4.

Además de esto, continuamente se encuentran vulnerabilidades de una determinada versión de la plataforma, con lo cual, es imprescindible realizar la actualización continua e instalación de parches correspondientes a fin de poseer una infraestructura segura y no vulnerable con las últimas actualizaciones que proveen solución a vulnerabilidades encontradas.

Del mismo modo, debe haber una política de revisión de código ya que los desarrolladores pueden no tener buenas prácticas de desarrollo seguro y exponer datos sensibles. Estos datos pueden ser de una determinada conexión o petición, como también podrían ser datos de usuario o cualquier otro evento de la plataforma se reportada en los logs.

Por último una buena política de backups periódicos para una pronta recuperación de información en caso de que se vea afectada la integridad de los datos.

3. Prueba de Concepto

La prueba de concepto realizada está basadas en la suposición de crear un *WS* para la consulta de usuarios y saldos asociados. El consumo de servicio se hace a través de clientes usados por los usuarios finales, bien podría ser una aplicación web, mobile o de escritorio que haga uso de este servicio. Los datos de los cuales hace uso este servicio son almacenados en una base de datos y el servicio tendrá a su disposición siempre que los requiera.

3.1. Modelo relacional de la base de datos

En primer lugar, se expone la estructura de la base de datos (*BD*) que se creó para la realización de éste trabajo.

La *BD* contiene dos tablas principales, saldo y usuarios. Este modelo se pensó en una base sencilla donde la base consta de usuarios con atributos de ‘email’, ‘nombre’, ‘password’, ‘teléfono’ y un identificador único por cada usuario nombrado ‘ID’. Éstos están relacionados por su identificador con un objeto que representa un dato de saldo, el cual consta de los atributos de ‘saldo’ (cantidad de saldo como cifra), una ‘fecha de vencimiento’ para dicho saldo, un ‘ID’ o identificador para cada saldo y por último, la relación con el identificador del usuario al cual pertenece dicho saldo.

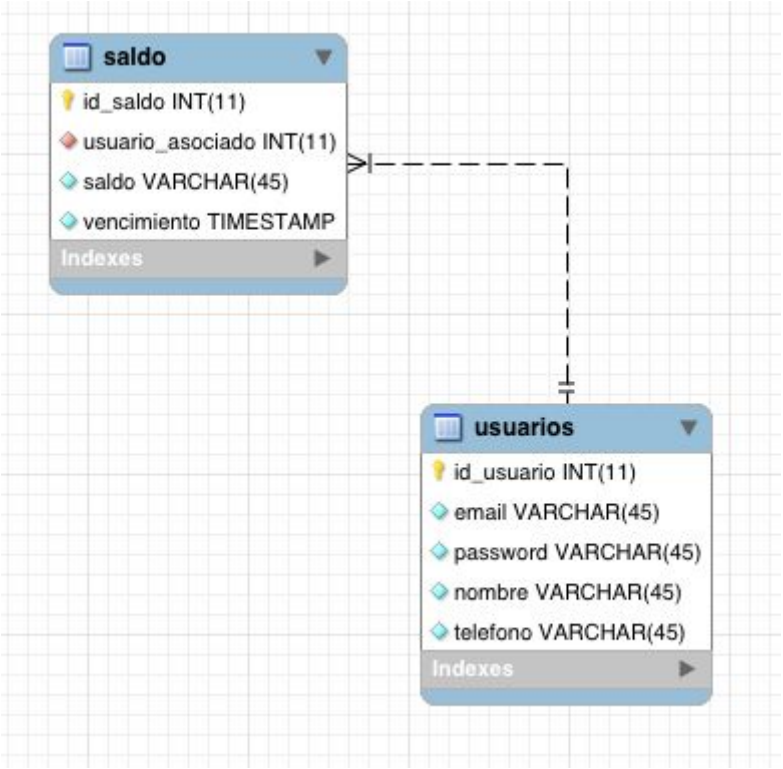


Imagen 3: Modelo relacional de la BD.

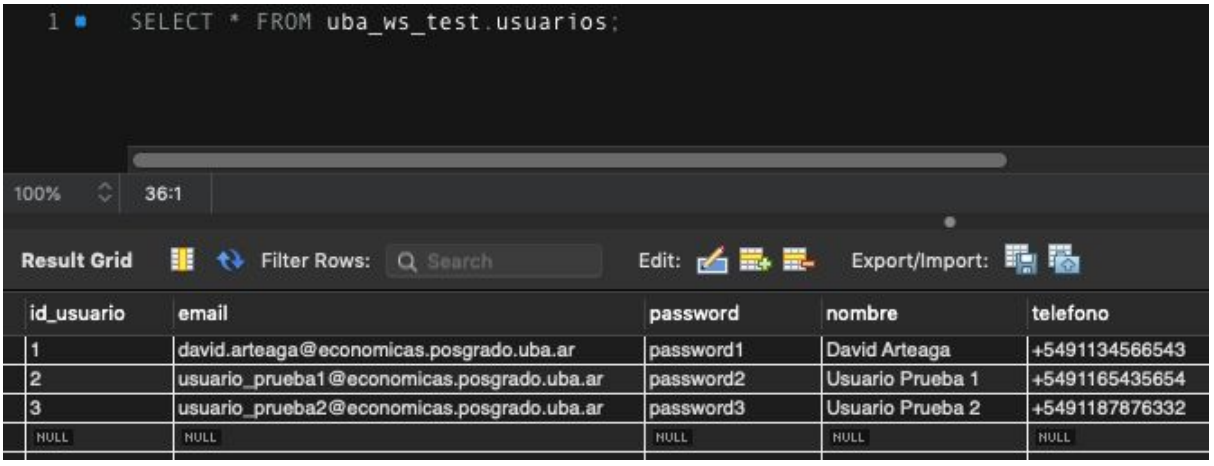
La información es almacenada para ambos casos, según el modelo mostrado, en una *BD MySQL Community Server* en su versión 5.6.38. A la fecha la última versión de *MySQL Community Server* es la 8.0.15, aunque no esté relacionado con la seguridad en los *WS*, una sugerencia de seguridad es tener todos los servicios que se estén prestando en el servidor a configurar, como en este caso es el servicio de la *BD*, actualizados y con los parches de seguridad al día, ya que esto puede representarse como otro vector de ataque para llegar a vulnerar el servidor que presta el conjunto de servicios.

3.2. Configuración, implementación y prueba del *WS* inseguro

En esta sección se muestra como se llevó a cabo la configuración del *WS* con configuración por defecto y la información que hacen uso los servicios implementados para este *WS*.

3.2.1. Configuración de Información almacenada en la *BD* usada por el *WS* inseguro.

A continuación vemos la información almacenada por la tabla de usuarios, donde cabe recordar que en la sección de gestión de contraseñas, se hace énfasis en el almacenamiento de información para una implementación segura, y una de las recomendaciones es no almacenar los datos en texto plano. Como buena medida de seguridad, deberíamos guardar el *hash* de la contraseña como se explica en la sección 2.9 Gestión de contraseñas. Dado que esta es la *BD* que será usada por las implementaciones del *WS* inseguro, la información almacenada será guardada en texto plano.



id_usuario	email	password	nombre	telefono
1	david.arteaga@economicas.posgrado.uba.ar	password1	David Arteaga	+5491134566543
2	usuario_prueba1@economicas.posgrado.uba.ar	password2	Usuario Prueba 1	+5491165435654
3	usuario_prueba2@economicas.posgrado.uba.ar	password3	Usuario Prueba 2	+5491187876332
NULL	NULL	NULL	NULL	NULL

Imagen 4: Información almacenada de la tabla usuarios en la BD insegura.

La información que contiene la tabla “saldo” no requiere ser modificada, así que para ambas implementaciones se usa la misma información mostrada. Como se mostró en el caso de la tabla “usuarios”, la información guardada en la tabla “saldo” también consta de un identificador único por saldo creado, el identificador del usuario al que está asociado el saldo, el valor del saldo y el vencimiento para la representación del objeto saldo.

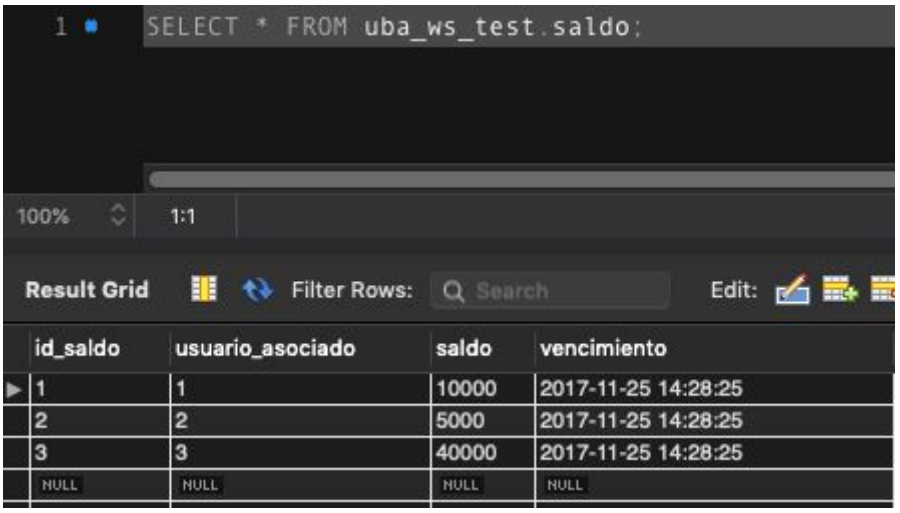


Imagen 5: Información almacenada en la BD de la tabla saldo.

3.2.2. Configuración de archivo "web.xml" *WS* inseguro

Todos los archivos de configuración, tanto los del *WS* inseguro y el seguro de las implementaciones desarrolladas para este documento, se encuentran en la carpeta raíz del proyecto. El link de descarga del proyecto se encuentran en la sección 6 de repositorios.

El archivo "web.xml" es el archivo descriptor de la implementación a desplegar en el servicio, este archivo descriptor sirve para determinar cómo se asignan las *URLs* a los servlets, qué *URL* requieren autenticación, entre otros [12]. Para el caso del *WS* inseguro no se especifican las restricciones de seguridad para los servlets implementados que son los de "Usuarios" y "Saldo" pero se especifica su mapeo respectivo de *URL*.

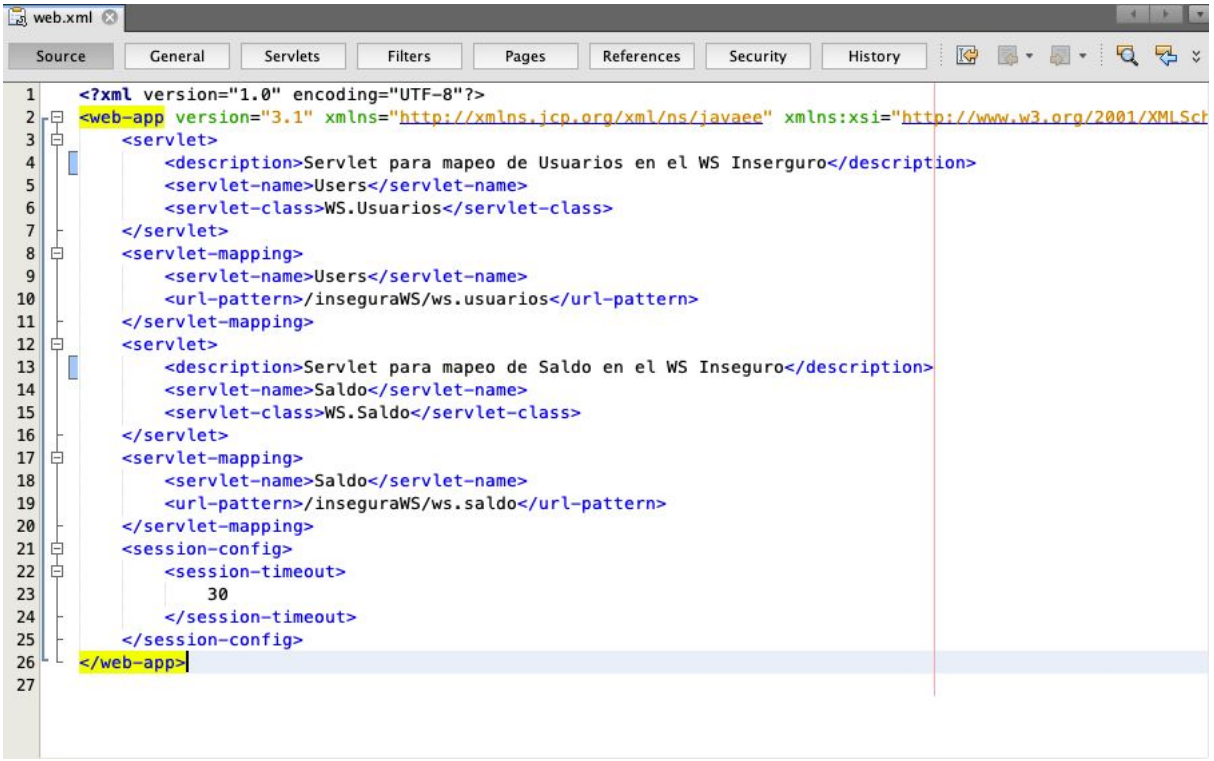


Imagen 6: Configuración archivo "web.xml" WS inseguro.

3.2.3. Pruebas funcionales *WS* inseguro

Las pruebas realizadas para corroborar la funcionalidad del *WS* se realizaron a través del navegador Chrome y la herramienta de envío de peticiones *HTTP* Postman. Se realizaron pruebas de **GET** y **PUT** para los servicios expuestos de "Usuarios" y "Saldo". Dado que no se hizo ninguna configuración previa para los servicios expuestos cualquier persona con conocimiento de las *URLs* del servicio podría realizar este tipo de peticiones al *WS* a través de un cliente.

3.2.3.1. Funciones GET

La primera prueba realizada es un **GET** para los servicios "Usuarios" y "Saldo", que según se implementó en el proyecto retorna los valores referenciados en la *BD* de todos los usuarios y saldos almacenados respectivamente.

En situaciones reales y dado el ejemplo planteado en el trabajo, se pueden llegar a pensar en usar estas funciones para requerir información de los usuarios o de un usuario en específico. También se puede considerar un caso de uso en donde

el usuario quiera ver a través de un cliente su cantidad de saldos disponibles y fechas a vencer de los mismos.

Imagen 7: Función GET a Usuarios WS inseguro a través de Navegador.

Esta misma petición se puede realizar en la herramienta Postman, donde se especifica la función a realizar en la *URL* del servicio dada. Postman también permite la edición de los encabezados y cuerpo de la solicitud a realizar, además de poder seleccionar el método de autorización en el servicio. Para este caso al no tener configuraciones de autorización establecidas no es necesario configurar ningún parámetro al momento de realizar la petición.

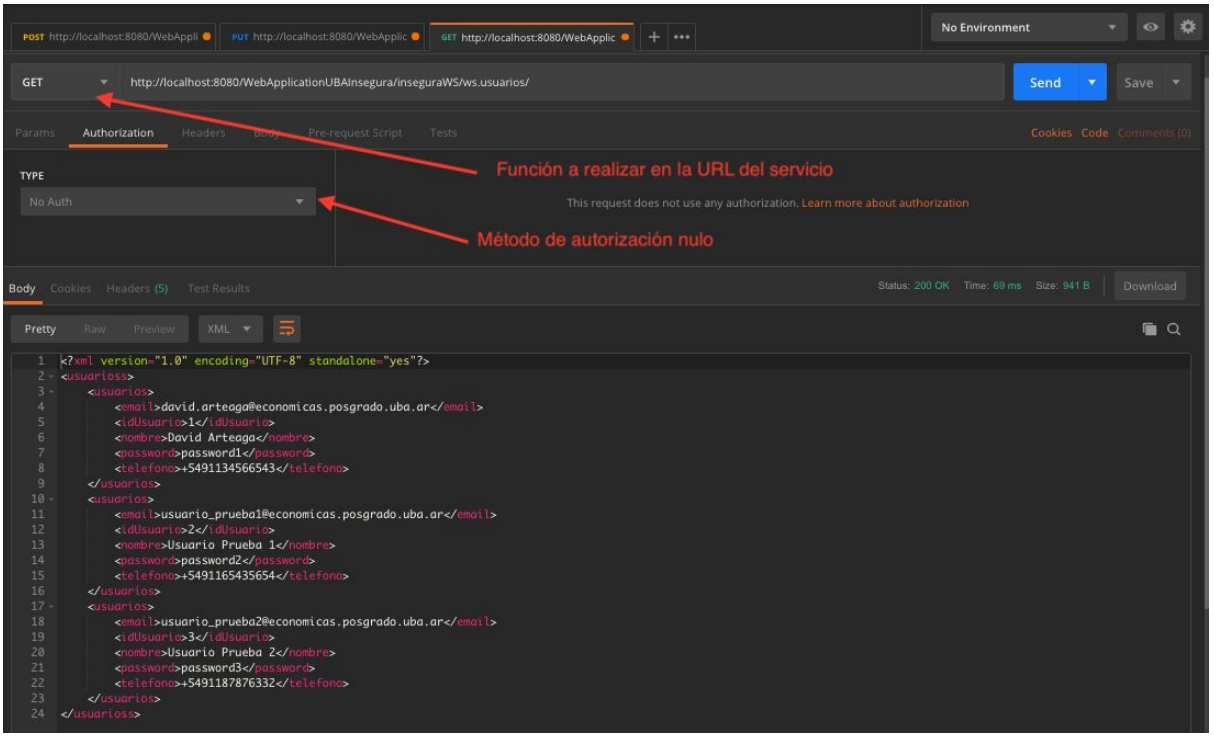


Imagen 8: Función GET a Usuarios WS inseguro a través de Postman.

Las misma función fue probada para el servicio de “Saldo” con las mismas especificaciones realizadas para la consulta de “Usuarios”

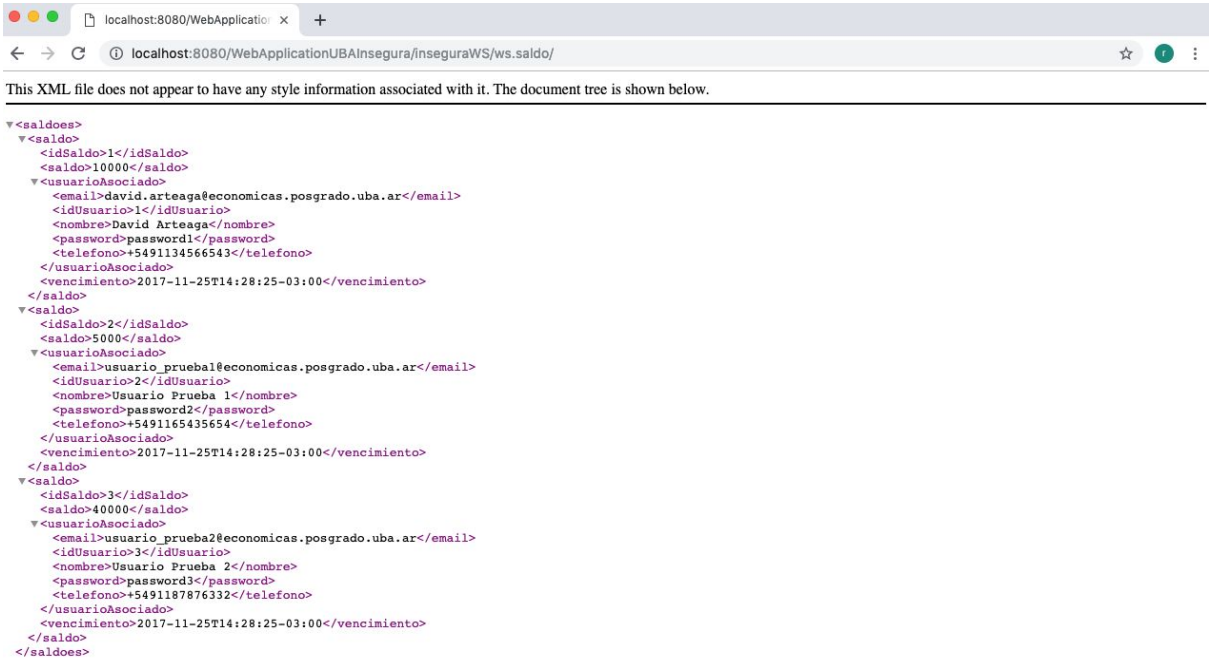


Imagen 9: Función GET a Saldo WS inseguro a través de Navegador.

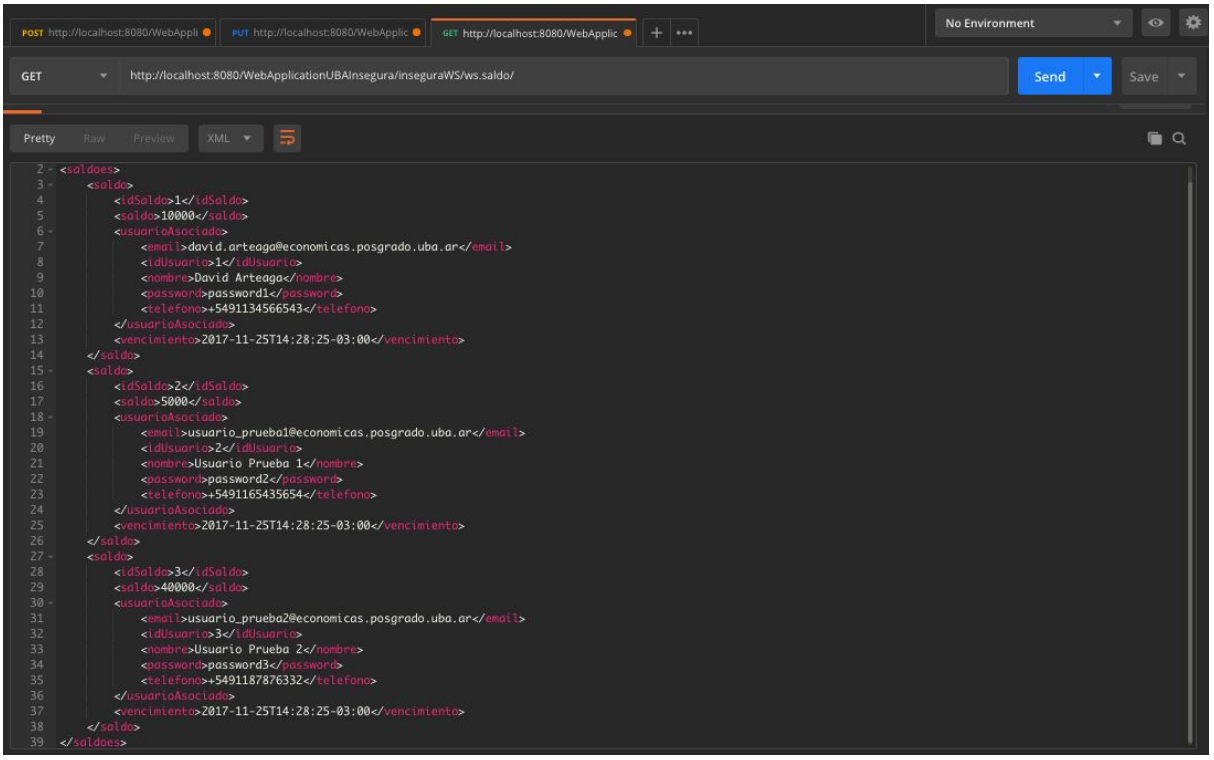


Imagen 10: Función GET a Saldo WS inseguro a través de Postman.

Por último, dentro de las implementación de cada uno de los dos servicios expuestos, se desarrolló la posibilidad de realizar peticiones **GET** especificando el identificador que indica cada representación de objeto en la *BD*, para consultar los valores asociados al identificador que pide el cliente.

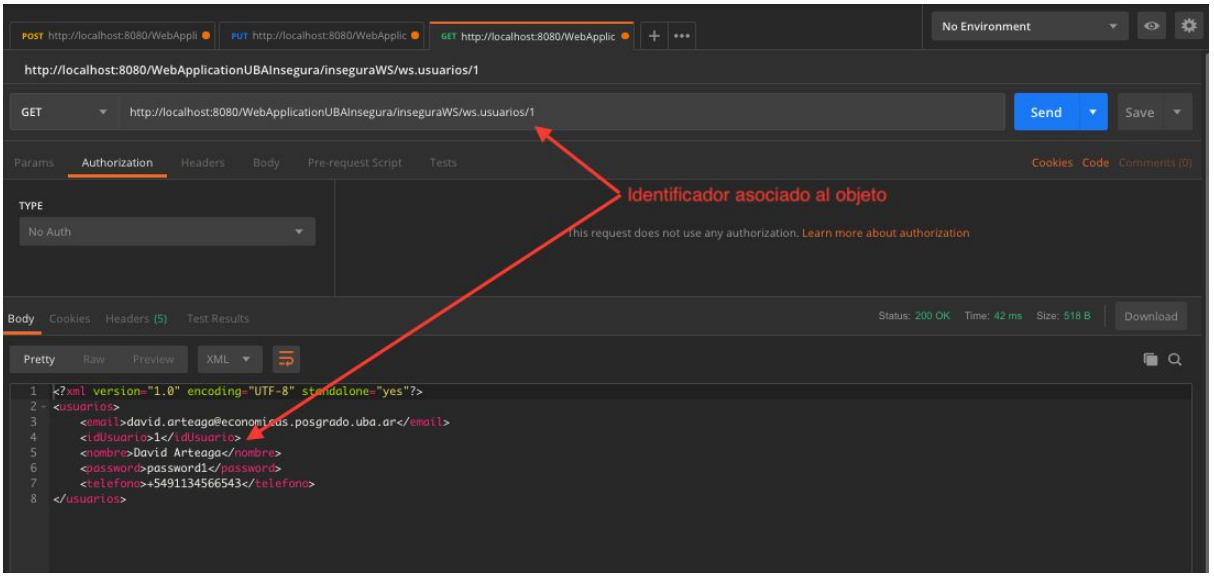


Imagen 11: Función GET a Usuario por id WS inseguro a través de Postman.

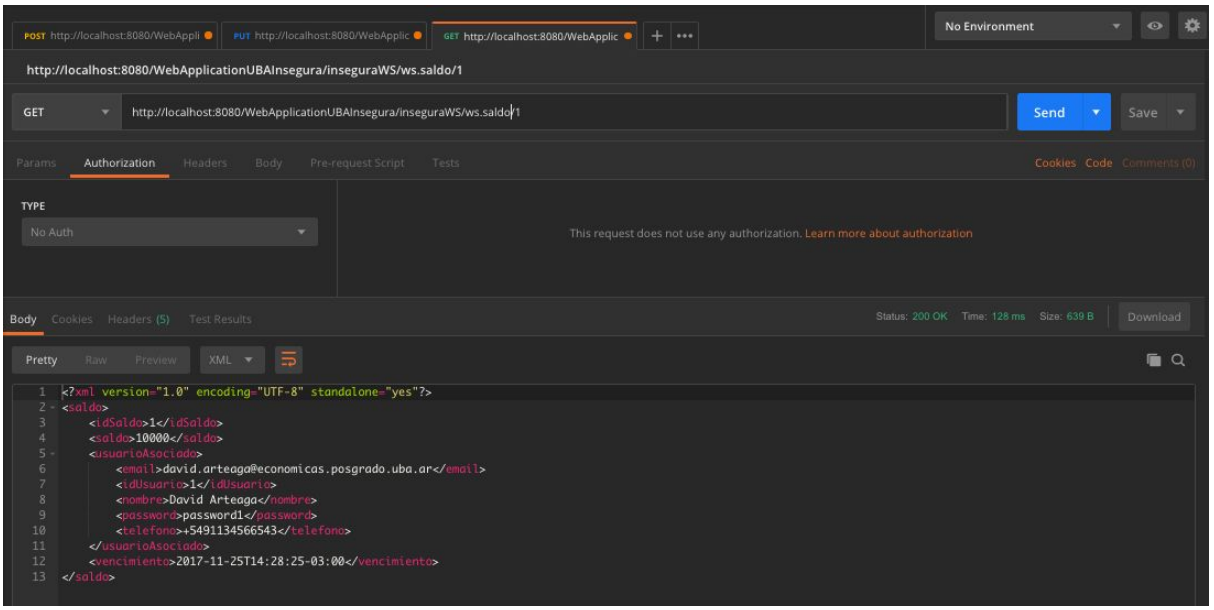


Imagen 12: Función GET a Saldo por id WS inseguro a través de Postman.

3.2.3.2. Función PUT

La segunda prueba realizada es con el método **PUT**, con el cual se realizó una prueba de crear un nuevo saldo a través del servicio expuesto llamado “Saldo”, especificando el identificador del nuevo saldo, donde se podría pensar en una situación real como el usuario final agregando a través de un cliente, un nuevo saldo con un monto nuevo a su registro de saldos.

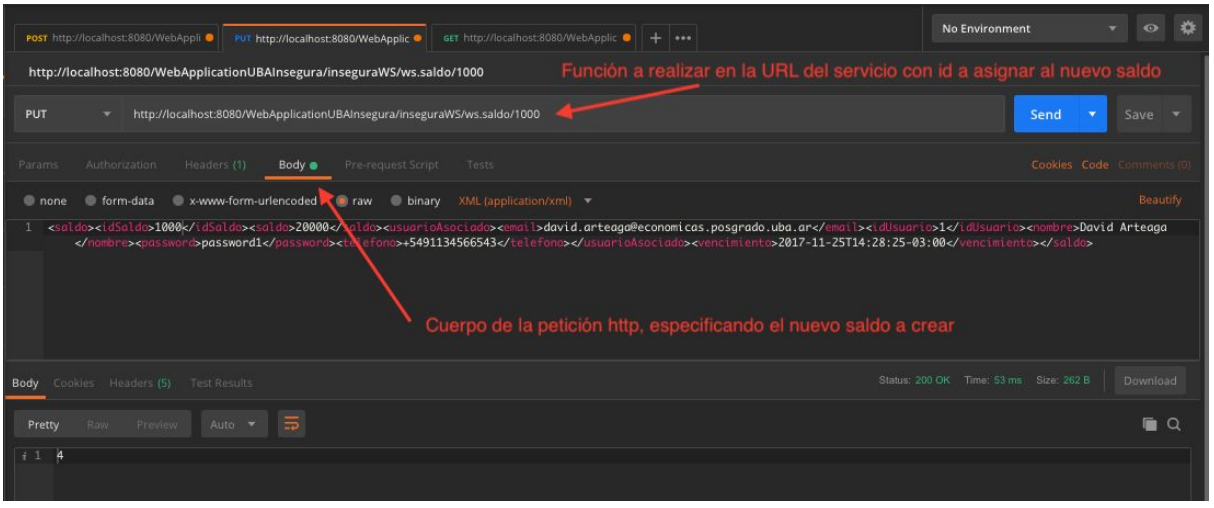


Imagen 13: Función PUT a Saldo por id WS inseguro a través de Postman.

Se verifica que el servicio haya realizado el cambio en la *BD* y se haya generado un nuevo saldo en la tabla correspondiente con los valores dados en el cuerpo de la petición.

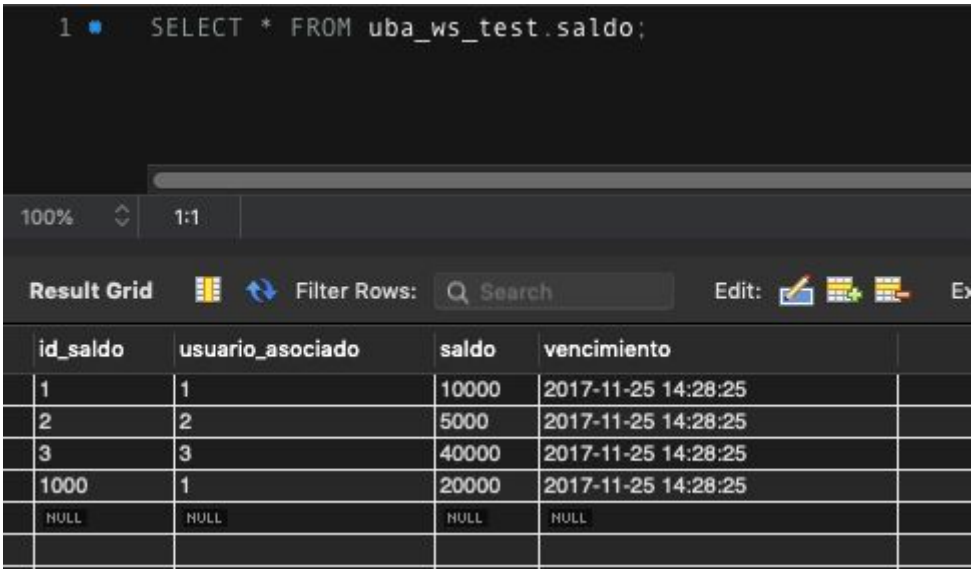


Imagen 14: Información almacenada en la BD de la tabla saldo después de petición PUT con WS inseguro.

3.3. Configuración, implementación y prueba del *WS* seguro

En esta sección se muestra como se llevó a cabo la configuración del *WS* con configuración segura, donde los grande aspectos fueron la definición de roles de acceso a los servicios, método de autorización por *BAM* y uso del certificado autofirmado generado por Glassfish para la comunicación entre clientes y servicio por un canal cifrado y seguro de comunicación.

3.3.1. Configuración de Información almacenada en la *BD* usada por el *WS* de configuración segura.

Para la implementación segura el manejo de datos realizado se almacenaron los *hashes* con la función *MD5*, que aunque es atacable por su rapidez al generar los *hashes* de una secuencia de caracteres, es bastante eficiente si se tienen buenas políticas de contraseñas [13]. El *WS* es el encargado en su implementación

de comparar los *hashes* para que el servicio haga uso de la información que está disponible en la *BD*.

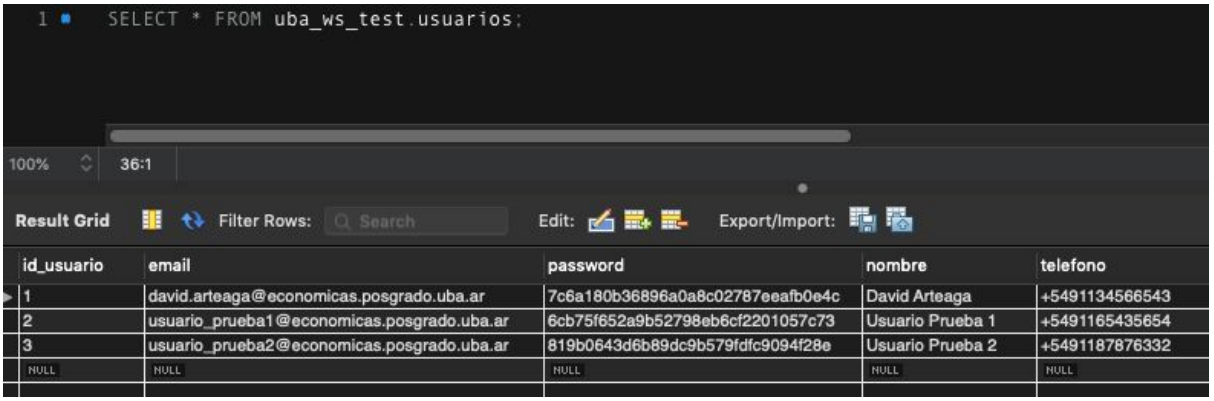


Imagen 15: Información almacenada de la tabla usuarios en la BD segura.

3.3.2. Configuración de usuarios y roles en el servidor para el *WS* seguro.

Una de las consideraciones más importantes que se deben tener en cuenta dentro de la configuración segura de un *WS* que implemente la autorización de usuarios por *BAM*, es la configuración de roles de grupo dentro de la plataforma de administración de *GlassFish*. Esta configuración se hace en la página local de administración de *GlassFish* que a través del navegador, el administrador o personal encargado de la configuración del servidor donde se va a desplegar el *WS*, debe agregar los grupos de roles en la ruta “<Security/Realms/[dominio]>” la cual se encuentra dentro de la opciones de configuración. Para el desarrollo de este *PoC* se usó el dominio “file”. Cabe recordar que una buena medida de seguridad de configuración en el servidor de *Glassfish* es cambiar la contraseña de acceso al panel de administrador, ya que la cuenta de administrador del panel no posee una clave determinada y si un atacante tiene acceso al panel de administración podría remover los servicios que estén desplegados en el servidor o desplegar nuevos servicios que haya desarrollado el atacante. Esta medida de seguridad es una de las mayores sugerencias de configuración del servidor que se debe tener en cuenta para el servidor donde será desplegado el *WS*.

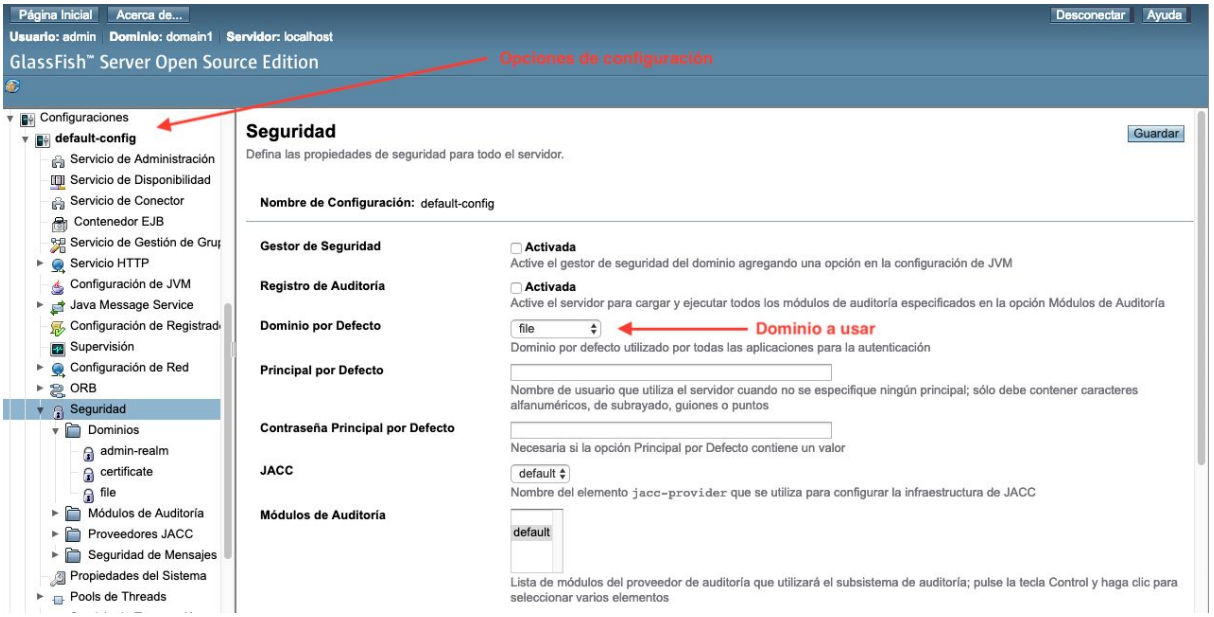


Imagen 16: Configuración de dominio en el servidor Glassfish.

En las configuraciones de dominio se selecciona la opción de gestión de usuarios que será el lugar donde el administrador configure los usuarios y los roles que le asignará a cada usuario que se quiera crear. Para esta prueba se configuraron dos usuarios, uno llamado “administrador” perteneciente al grupo “admins” con una contraseña de prueba “adminp4ss”; otro usuario llamado “usuario” perteneciente al grupo de “users” con una contraseña “userp4ss”. Éste representa a los usuarios que harán comúnmente consumo de los recursos expuestos por el *WS* a través de sus clientes. Para cada rol se definieron permisos diferentes en los métodos *HTTP* que pueden realizar explicados en la siguiente sección.

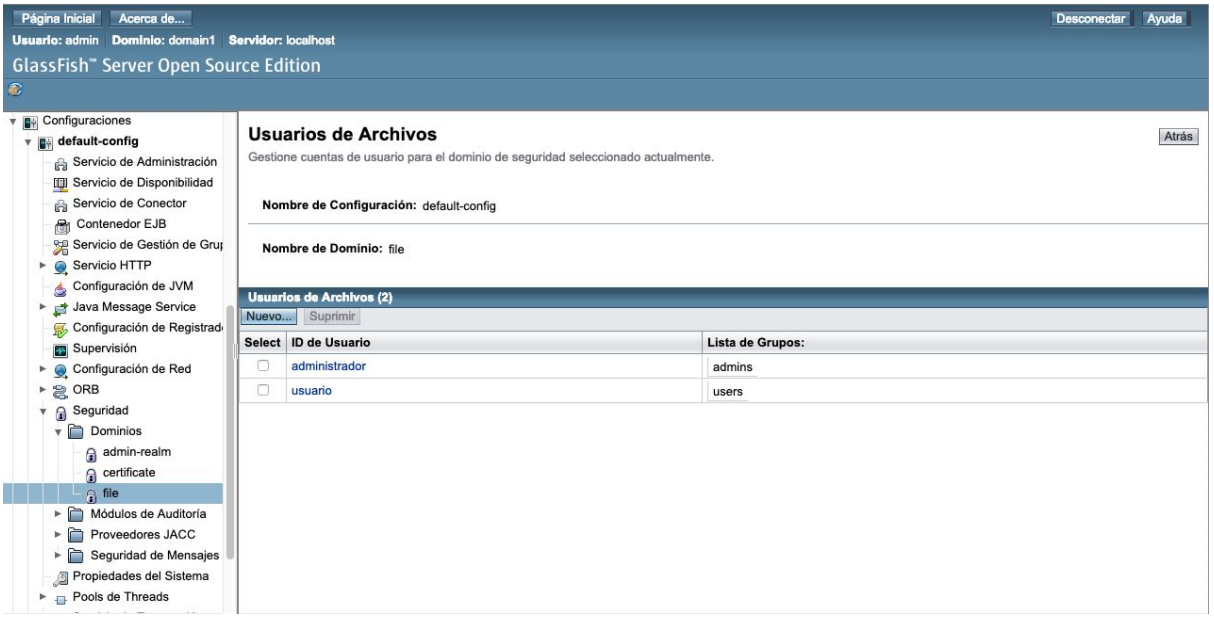


Imagen 17: Configuración de usuarios y grupos del dominio usado del servidor Glassfish.

Después se deben agregar los roles de los usuarios que se añadieron previamente en la parte de asignación de grupos en la configuración del dominio

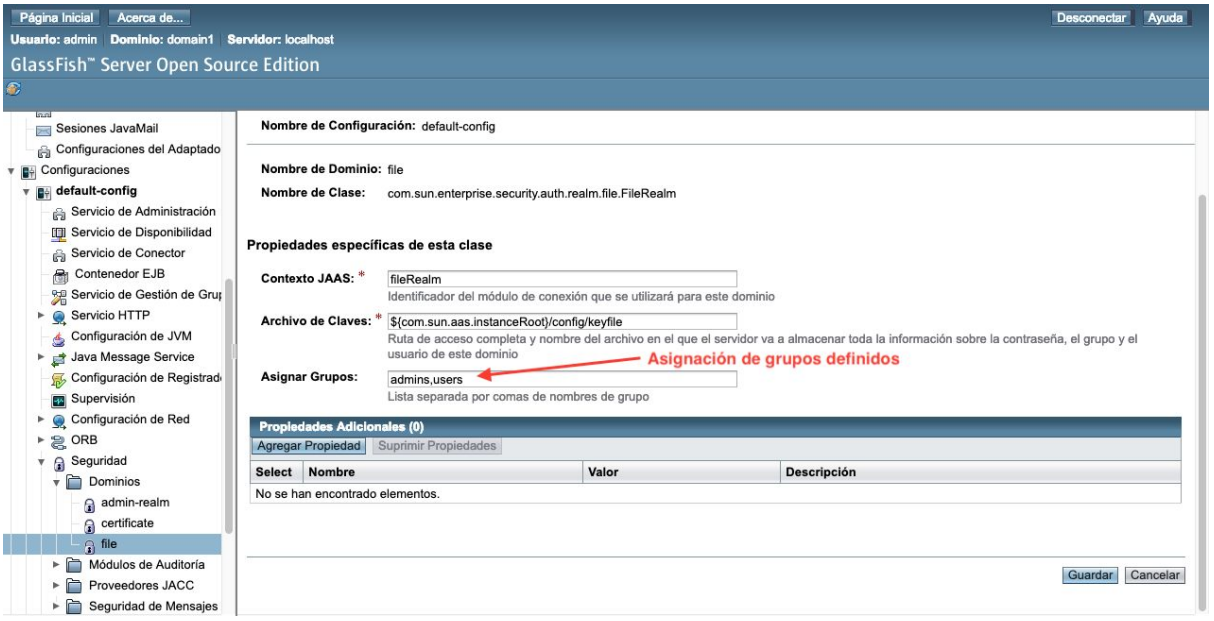
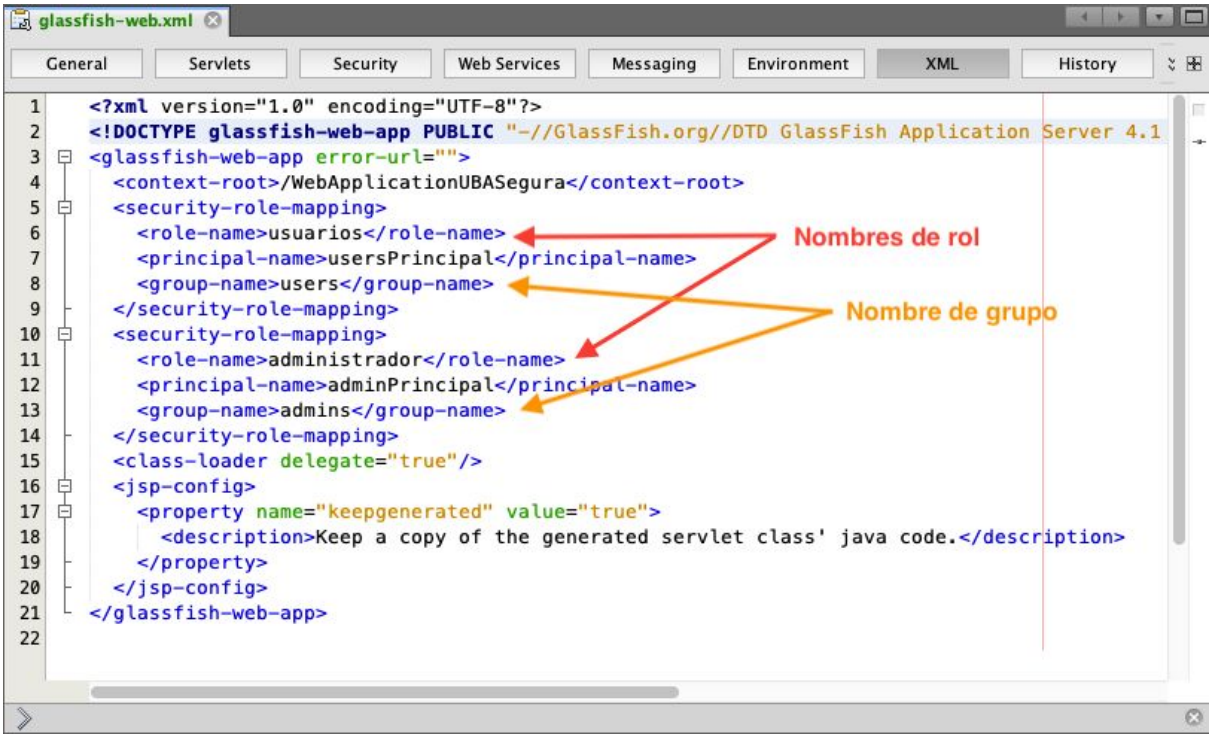


Imagen 18: Configuración asignación de grupos de roles en el dominio file del servidor GlassFish WS seguro.

3.3.3. Configuración archivo "glassfish-web.xml" *WS* seguro

Todos los archivos de configuración, tanto los del *WS* inseguro y el seguro de las implementaciones desarrolladas para este documento, se encuentran en la carpeta raíz del proyecto. El link de descarga del proyecto se encuentran en la sección 6 de repositorios.

Este archivo de configuración define los roles a ser mapeados para estar en sincronismo con los que se definieron en el servidor de Glassfish y la implementación. Para el caso del *WS* inseguro este archivo no se configuró ya que no se mapean roles ni se hicieron ningunas configuraciones previas. Este archivo se encuentra en la carpeta raíz del proyecto de la implementación. Al hacer la compilación de la implementación este genera una aplicación completa que será desplegada en el servidor Glassfish con estas configuraciones dadas.




```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 4.1
3  <glassfish-web-app error-url="">
4    <context-root>/WebApplicationUBASegura</context-root>
5    <security-role-mapping>
6      <role-name>usuarios</role-name>
7      <principal-name>usersPrincipal</principal-name>
8      <group-name>users</group-name>
9    </security-role-mapping>
10   <security-role-mapping>
11     <role-name>administrador</role-name>
12     <principal-name>adminPrincipal</principal-name>
13     <group-name>admins</group-name>
14   </security-role-mapping>
15   <class-loader delegate="true"/>
16   <jsp-config>
17     <property name="keepgenerated" value="true">
18       <description>Keep a copy of the generated servlet class' java code.</description>
19     </property>
20   </jsp-config>
21 </glassfish-web-app>
22
```

Imagen 18: Configuración roles archivo sun-web.xml implementación segura.

3.3.4. Configuración archivo "web.xml" *WS* seguro

Recordando lo descrito en la sección 3.2.2, el archivo "web.xml" es el archivo descriptor de la implementación a desplegar en el servicio, este archivo descriptor sirve para determinar cómo se mapean las asignaciones de las *URLs* a los servicios. Además, en este archivo se asignan las restricciones de seguridad según los roles configurados para cada *URL* y qué método *HTTP* permitido cuando se esté prestando el servicio. Este archivo también se encuentra en la carpeta raíz del proyecto y para el caso desarrollado se ha configurado con las siguientes restricciones:

- El método **GET** para el servicio "usuarios" es permitido para ambos roles, "users" y "admins". Nótese también en las configuraciones dadas las anotaciones dadas en cada método para que se haga uso de *SSL* con el certificado usado por el Glassfish.

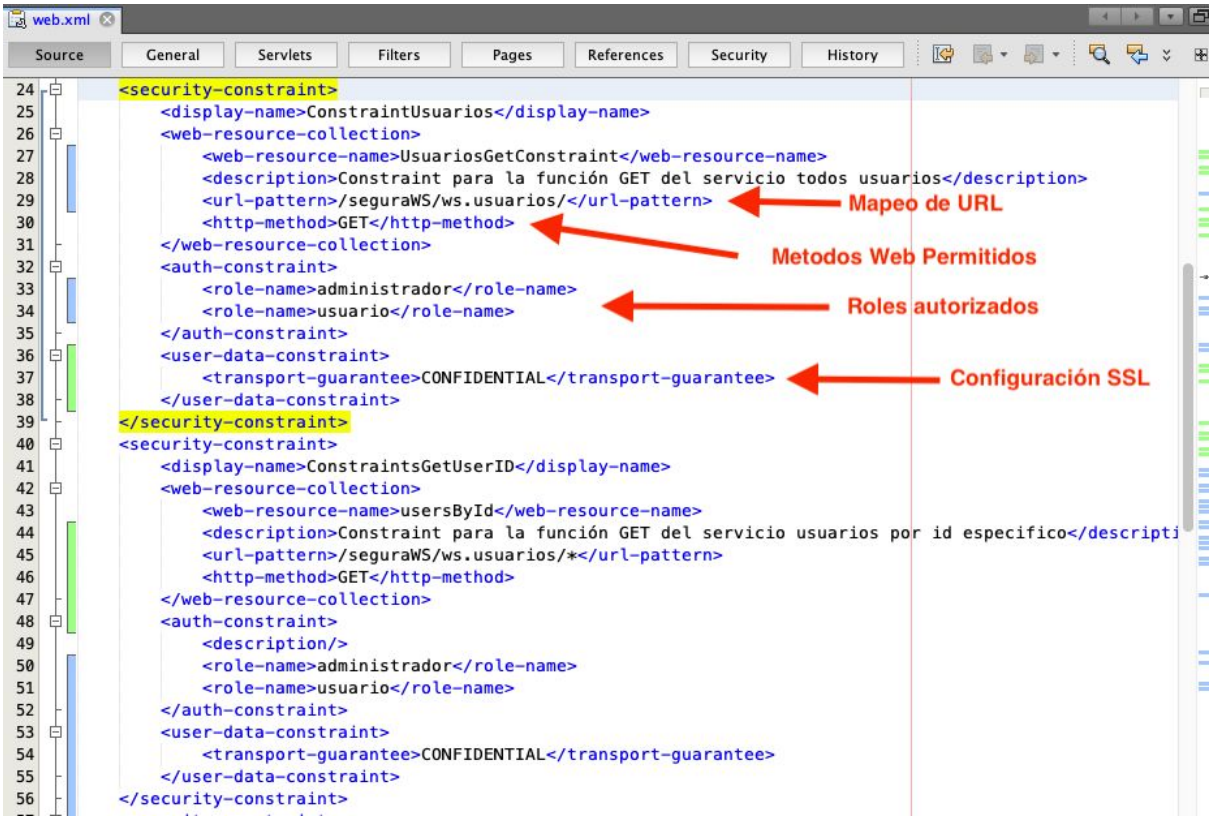


Imagen 19: Restricciones y configuración métodos servicio "usuarios" en archivo "web.xml" WS seguro.

- Para el servicio "saldos" solo será permitido el método **POST** y este recurso sólo tiene autorización el rol "admins", por ende serán los únicos que podrán generar nuevos saldos.

Imagen 20: Restricciones y configuración métodos servicio "saldo" y configuraciones de acceso en archivo "web.xml" WS seguro.

También se define el método de autorización con el tag <auth-method>, en el cual se utilizará usuario y contraseña, llamado "BASIC" el cual corresponde al *BAM.*

Además, se utiliza el <session-timeout> para configurar el lapso de tiempo que posee la sesión de un determinado *token*.

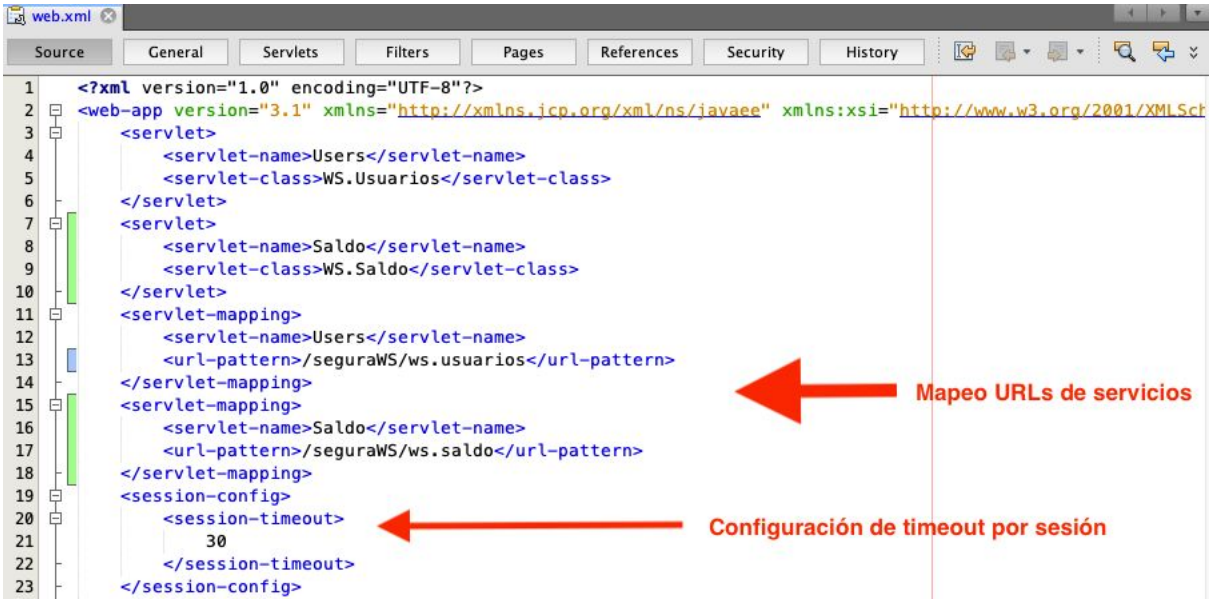


Imagen 21: Mapeo de URLs y configuración de tiempo de sesión en archivo "web.xml" WS seguro.

Por último, durante el desarrollo se pudo ver como un mal uso de logs de la aplicación también puede llevar a fugas de información sensible. Esto puede ocurrir ya que los programadores hacen un uso indebido de la muestra de información a través de los logs de la aplicación. Por esto una sugerencia de seguridad, a la hora del desarrollo de una aplicación segura, es la revisiones conjuntas de código antes de que se expongan los servicios en producción y probarlos en entornos diferentes como puede ser un ambiente netamente de desarrollo o de prueba.

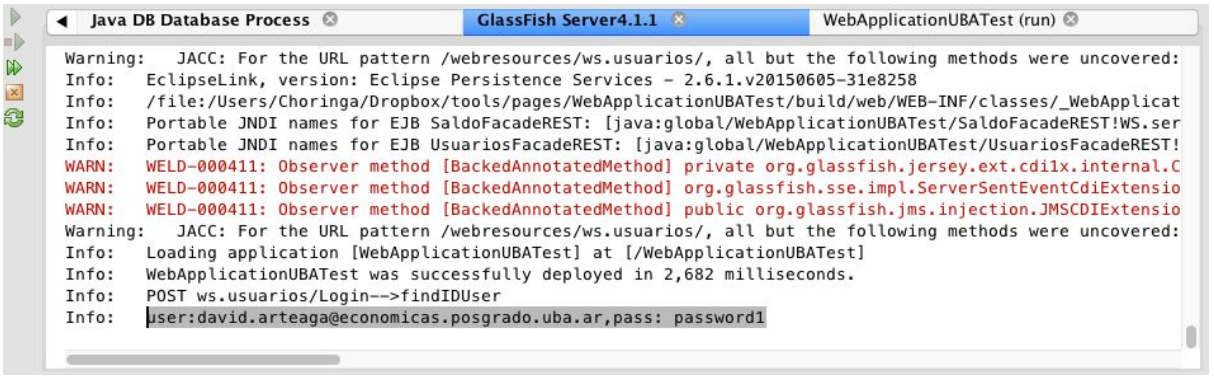


Imagen 22: Recomendaciones de uso de logs del servidor sin información sensible.

3.3.5. Pruebas funcionales *WS* seguro

En esta sección se realizan pruebas a través del programa de peticiones Postman para verificar la correcta configuración e implementación del *WS* seguro. Se realizaron las pruebas teniendo en cuenta las configuraciones planteadas en la sección anterior.

3.3.5.1. Funciones GET

La primeras pruebas realizadas son las **GET** para el servicio “usuarios” donde ambos roles tienen autorización para realizar este tipo de peticiones. La primera prueba es verificar que el servicio pida las credenciales por *BAM*, y dado caso de que no se autentique no pueda hacer uso de los servicios.

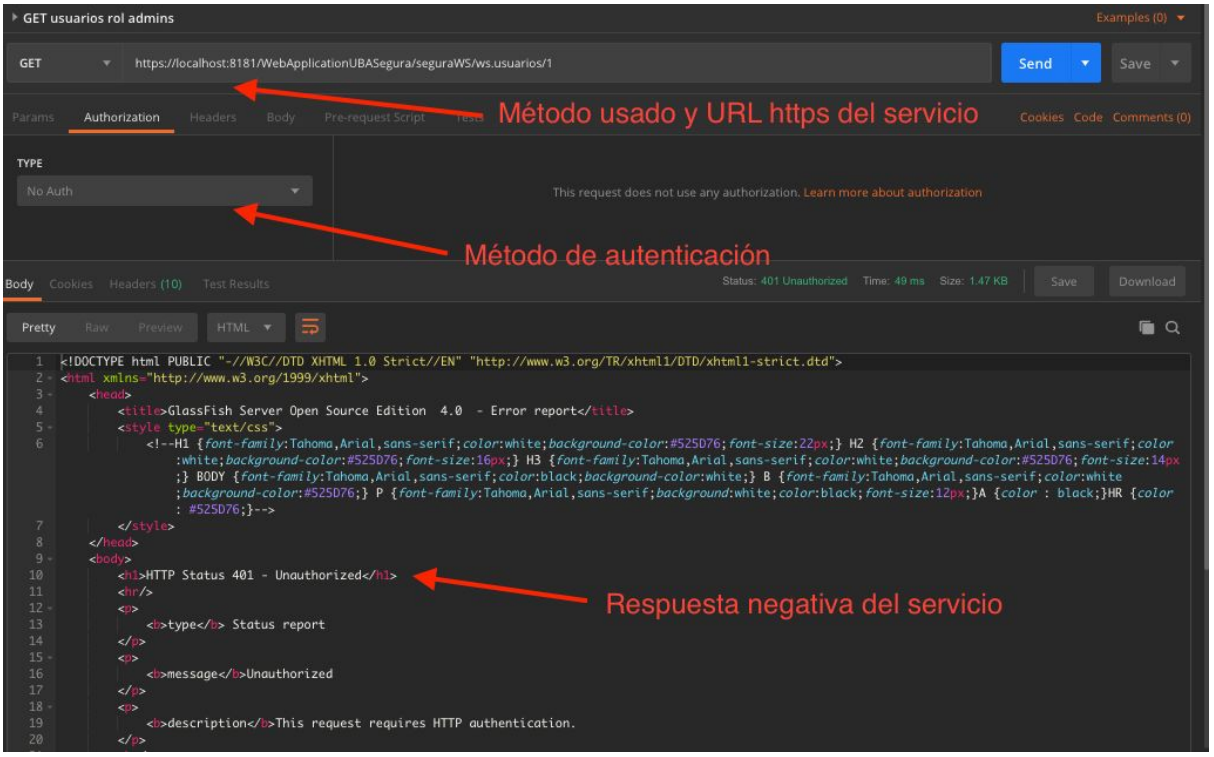


Imagen 23: Prueba funcional WS seguro GET “usuarios” sin autenticación.

Realizamos una solicitud **GET** al *WS* de usuarios sin una autorización, y nos retorna el servidor un código de error (*HTTP* Status 401 - Unauthorized) al saber

que el cliente no se encuentra autorizado ya que no cumplio con la autenticación por *BAM* para poder acceder a la información solicitada.

Una recomendación de seguridad a tener en cuenta recae en la importancia de cómo retorna la información el servidor ante una solicitud de un servicio, ya sea para invalidar o para mostrar que el servicio no existe. Para este caso se dejó la página por defecto que tiene configurada el GlassFish para el status 401, pero vemos como expone información sobre el servidor donde se encuentra montado el *WS*.

Ingresando las correctas credenciales se puede ver para la prueba **GET** que funciona como se espera en el servicio.

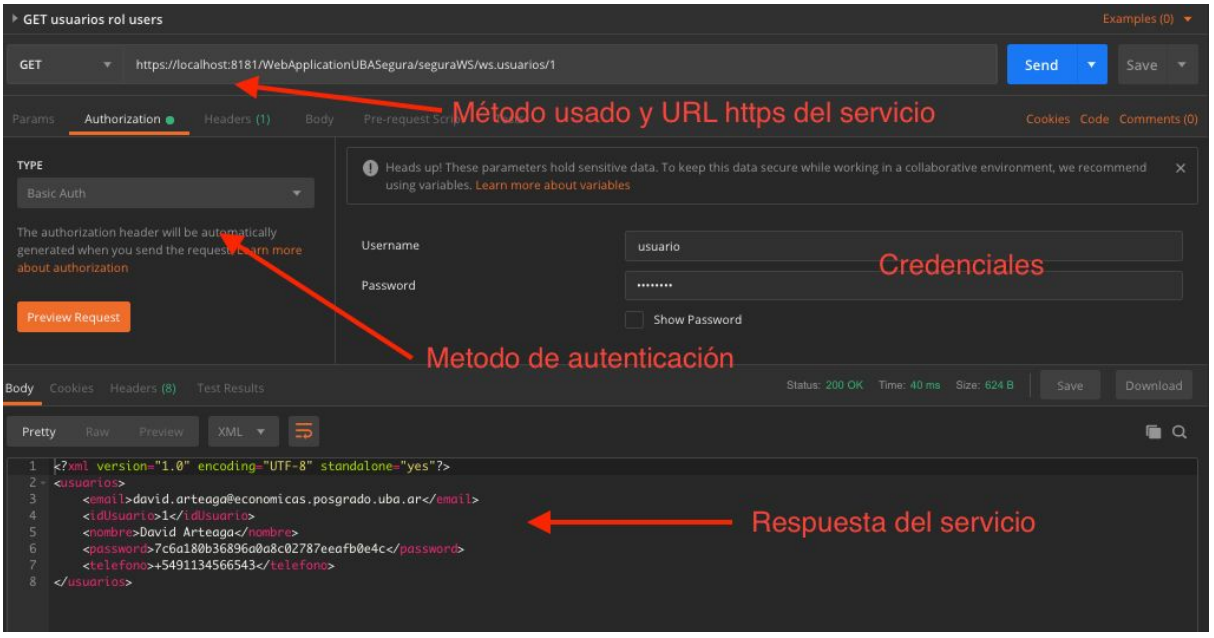


Imagen 24: Prueba funcional WS seguro GET "usuarios" con autenticación rol users.

También se realizó la misma prueba para el rol de "admins" con el método de autenticación *BAM*

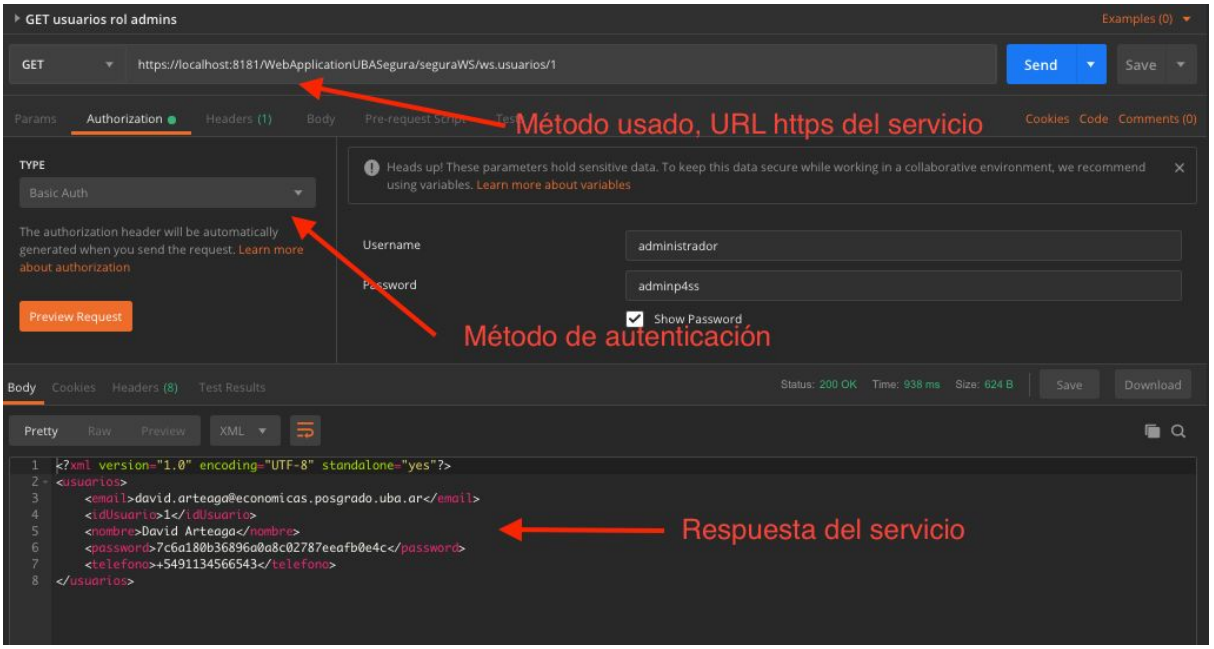


Imagen 25: Prueba funcional WS seguro GET "usuarios" con autenticación rol admins.

También se realizó la prueba cuando el usuario ya tiene una autorización con un *token* asociado a la sesión, es entonces donde el envío de credenciales puede quitarse siempre y cuando incluya el *token* de sesión en el encabezado de la solicitud.

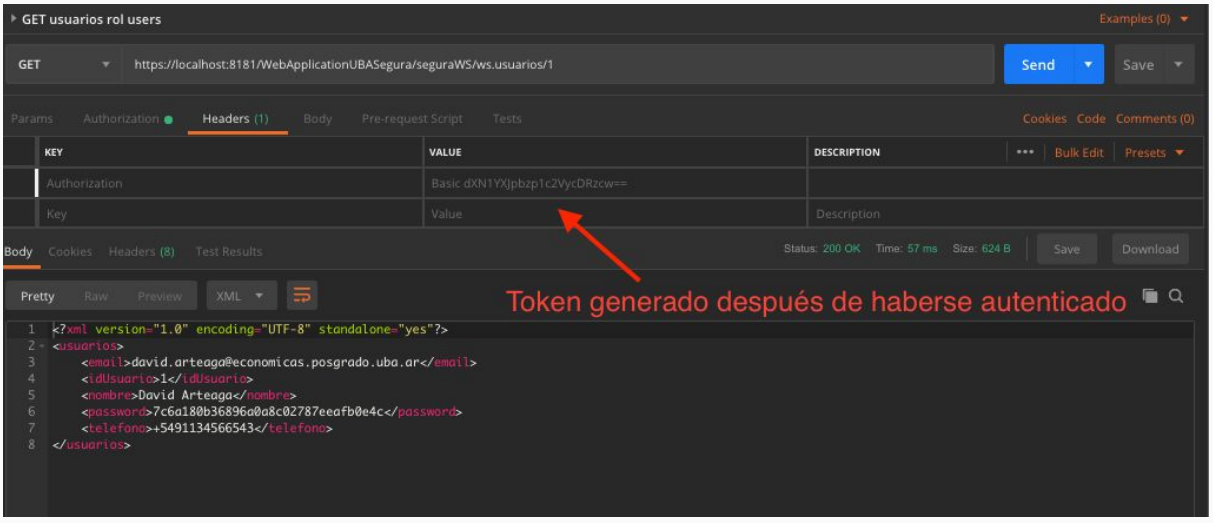


Imagen 26: Prueba funcional WS seguro GET "usuarios" con token rol admins generado.

3.3.5.2. Función POST

La segunda prueba realizada fue al servicio de "saldo" con el método **POST** para agregar un nuevo saldo. Cabe recordar que dentro de las configuraciones

planteadas para esta prueba, según las restricciones planteadas sólo los usuarios pertenecientes al grupo de rol de “admins” pueden realizar este tipo de solicitudes para este servicio.

Imagen 27: Prueba funcional WS seguro POST “saldo” con rol admins cuerpo.

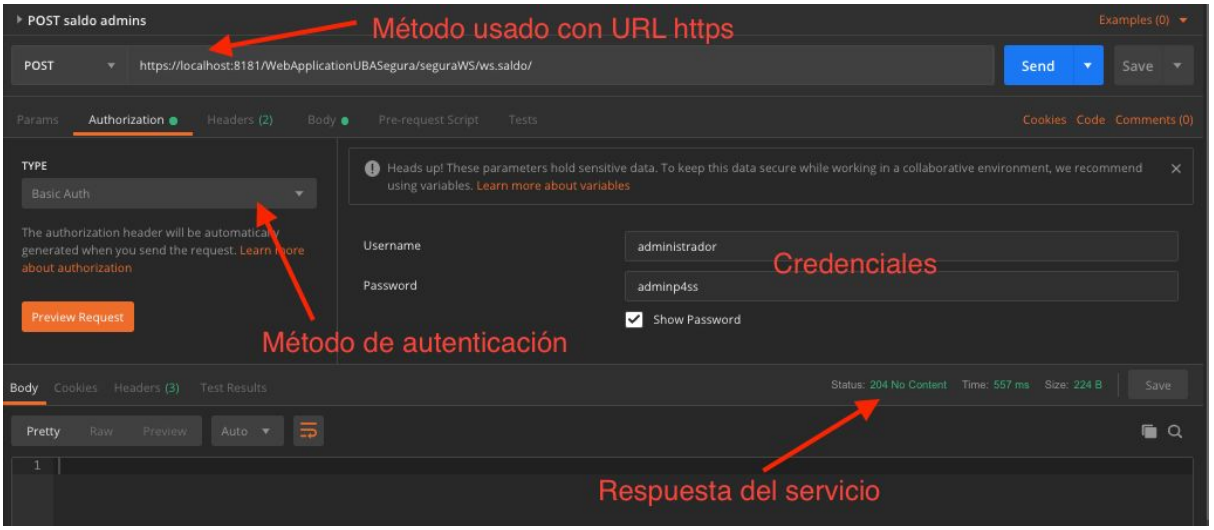


Imagen 28: Prueba funcional WS seguro POST “saldo” con rol admins autenticación.

Luego se verificó que se haya efectuado el cambio en la *BD*

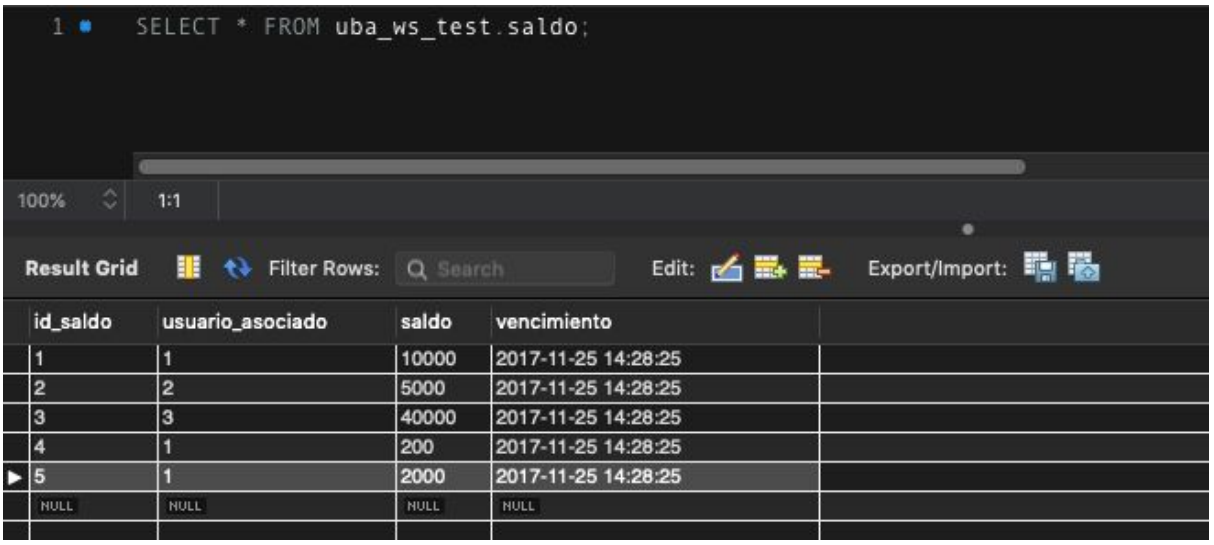


Imagen 29: Verificación de cambio de datos en la BD después del método POST a "saldo" WS Seguro.

Por último se realizó esta misma petición usando el rol de "users" para verificar que el servicio no permite realizar llamadas de este método sin el rol debido.

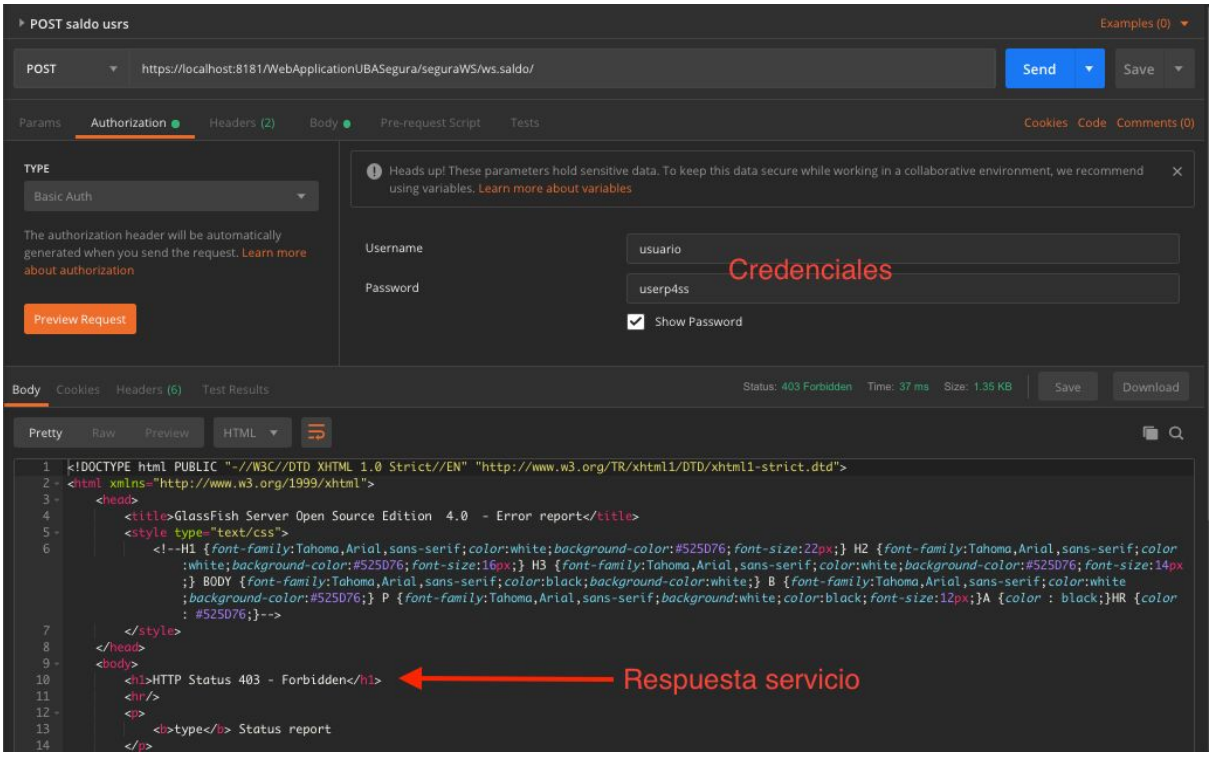


Imagen 30: Verificación de cambio de datos en la BD después del método POST a "saldo" WS Seguro.

4. Comparación de seguridad entre los *WS* desarrollados

Además de las configuraciones realizadas e implementaciones mostradas en la sección anterior, donde se puede evidenciar las diferencias entre la configuraciones de un *WS* seguro y uno inseguro, en esta sección se mostrará como un atacante puede tener acceso a la información albergada o entregada por un *WS* con malas o nulas configuraciones de seguridad. También se muestra como poder validar la autenticación de los mensajes enviados con el uso de *HMAC* que se implementó en el *WS* Seguro. Por último una comparación de seguridad entre las configuraciones básicas implementadas y configuradas en el *WS* Seguro frente a una implementación empresarial a gran escala que se haría con el uso de *OAuth 2.0*.

4.1. Seguridad con *SSL*

Con las configuraciones de *SSL* realizadas para el *WS* seguro y la carencia de la misma en el *WS* inseguro se muestra las falencias en la confidencialidad de los en el *WS* inseguro con un escaneo de datos dentro de una red interna realizado para ambos *WS* implementados en la *PoC*. Las peticiones fueron realizadas desde un navegador web como cliente de un dispositivo conectado en la misma red. Para este caso las *IPs* asignadas fueron:

- **192.168.0.9** para el servidor donde se exponen los *WS* implementados.
- **192.168.0.15** el dispositivo que se encuentra en la misma red haciendo peticiones desde el navegador web hacia los servicios expuestos por los *WS*.

La herramienta utilizada para hacer el escaneo de datos enviados por la red interna es Wireshark, la cual es capaz de realizar un escaneo de peticiones hechas según el protocolo elegido.

El primer escaneo de red realizado corresponde a una petición **GET** desde el cliente para el servicio “usuarios” para el *WS* inseguro. Como se puede evidenciar en la imagen 31 la información viaja en texto plano debido a las configuraciones dadas para este *WS*, es por esto que si un atacante se encuentra en la misma red podría ver la información que el usuario está solicitando y hacia cuál servicio lo está haciendo. Es por esta razón que la confidencialidad de los datos se ve afectada con estas configuraciones y si el atacante realiza estas peticiones y se da cuenta que el servicio no tiene configuraciones frente a la autorización y autenticación de los servicios, puede hacer uso de esto a su favor para obtener la información almacenada en la *BD* que hace uso el *WS*, a su vez al no tener configuraciones en los métodos *HTTP* (**PUT**, **GET**, **POST**, **DELETE**) puede vulnerar también la integridad de los datos y modificarlos o agregar nuevos datos en la *BD* con el uso de estos métodos *HTTP* a través del *WS*.

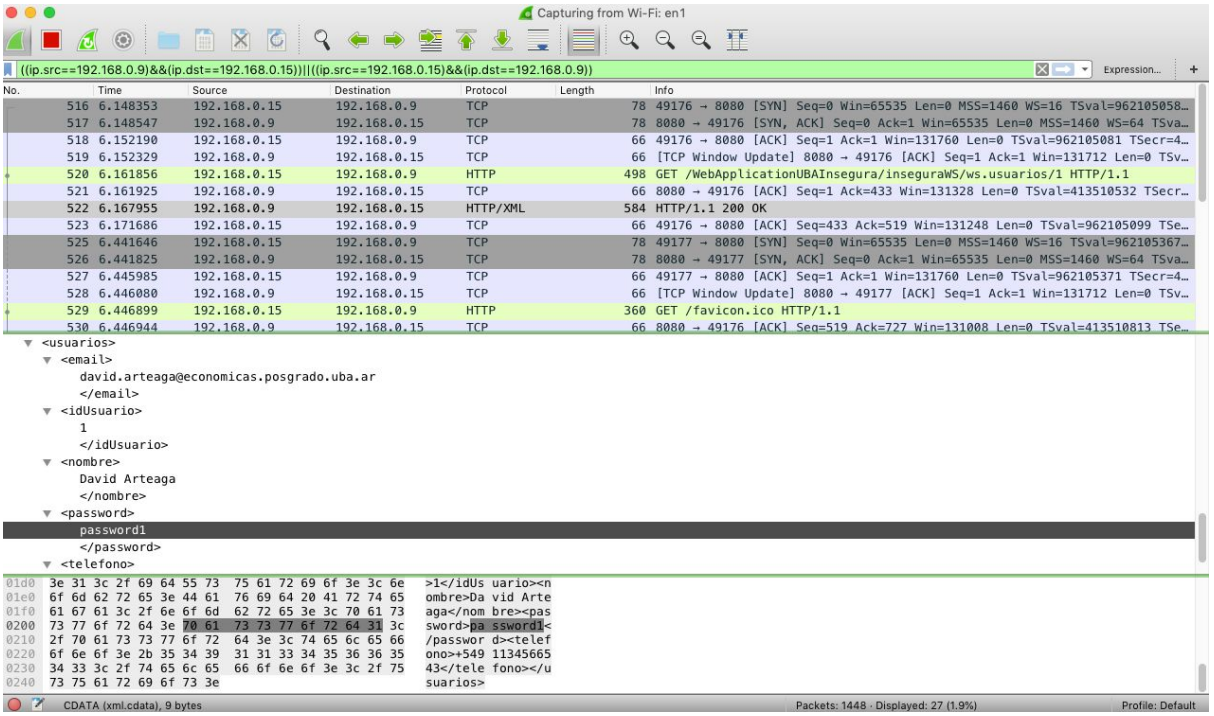


Imagen 31: Información expuesta en escaneo de red interna para función GET desde cliente a servidor para WS inseguro.

El segundo escaneo de red realizado corresponde a una petición al *WS* seguro por un canal inseguro, es decir, sin hacer uso del protocolo de *SSL*. En este

escaneo queda en evidencia que si un atacante realiza un escaneo de red interna y las peticiones de los clientes al *WS* no viajan a través de un canal seguro de comunicación las credenciales de autorización y autenticación configuradas no tendrán validez ya que la información de la solicitud viajaría en texto plano. Es por eso que una de las grandes recomendaciones de seguridad hechas en este documento es el uso de canales seguros de transmisión como lo es el caso de una configuración de un certificado para establecer una conexión segura entre cliente y servidor a través de protocolos seguros como *SSL* o *TLS*. Es por esto que en la siguiente imagen se puede apreciar que las credenciales de autorización con el servidor enviadas por el cliente viajan en texto plano y un atacante puede hacer uso de estas credenciales para realizar peticiones a su antojo.

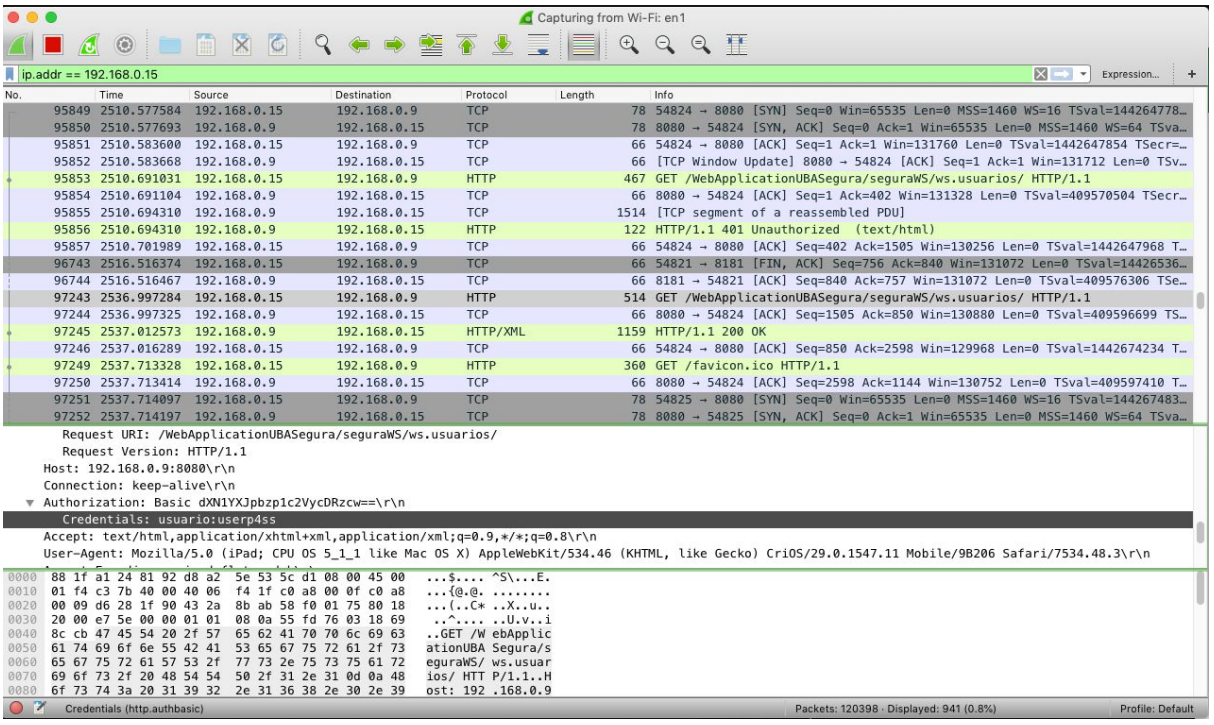


Imagen 32: Credenciales de autorización expuestas en escaneo de red interna para función GET desde cliente a servidor para WS seguro por canal inseguro.

Como la información no viaja por un canal seguro la información que retorna el servicio a las peticiones realizadas se puede ver en texto plano y deja vulnerable la confidencialidad de los datos como se puede ver en la imagen 33.

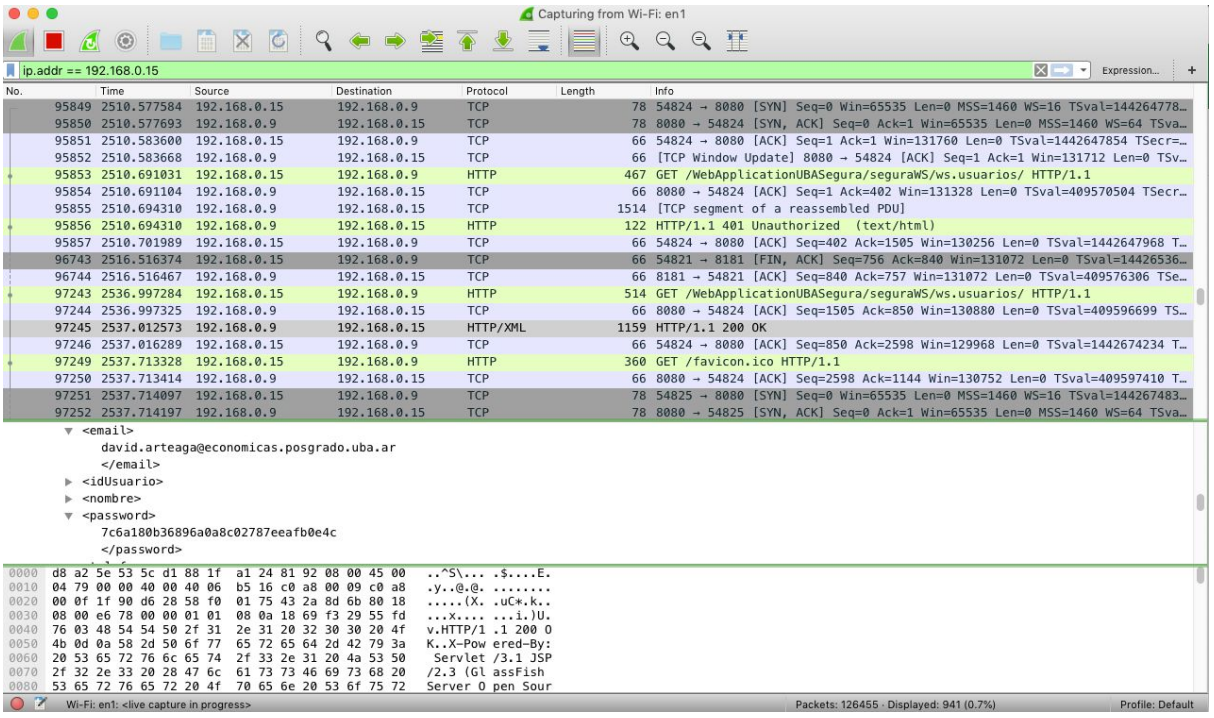


Imagen 33: Información expuesta en escaneo de red interna para función GET desde cliente a servidor para WS seguro por canal inseguro.

Por último se realizó un escaneo de red interna para la misma petición **GET** realizada para el *WS* con todas las configuraciones de seguridad realizadas y bajo un canal seguro de transmisión, haciendo uso del certificado autofirmado por el servidor permitiendo el uso del protocolo *SSL*. Para este escaneo se deja en evidencia que la información viaja cifrada por la red y un escaneo de red interna no permite que la información sea visible por un atacante.

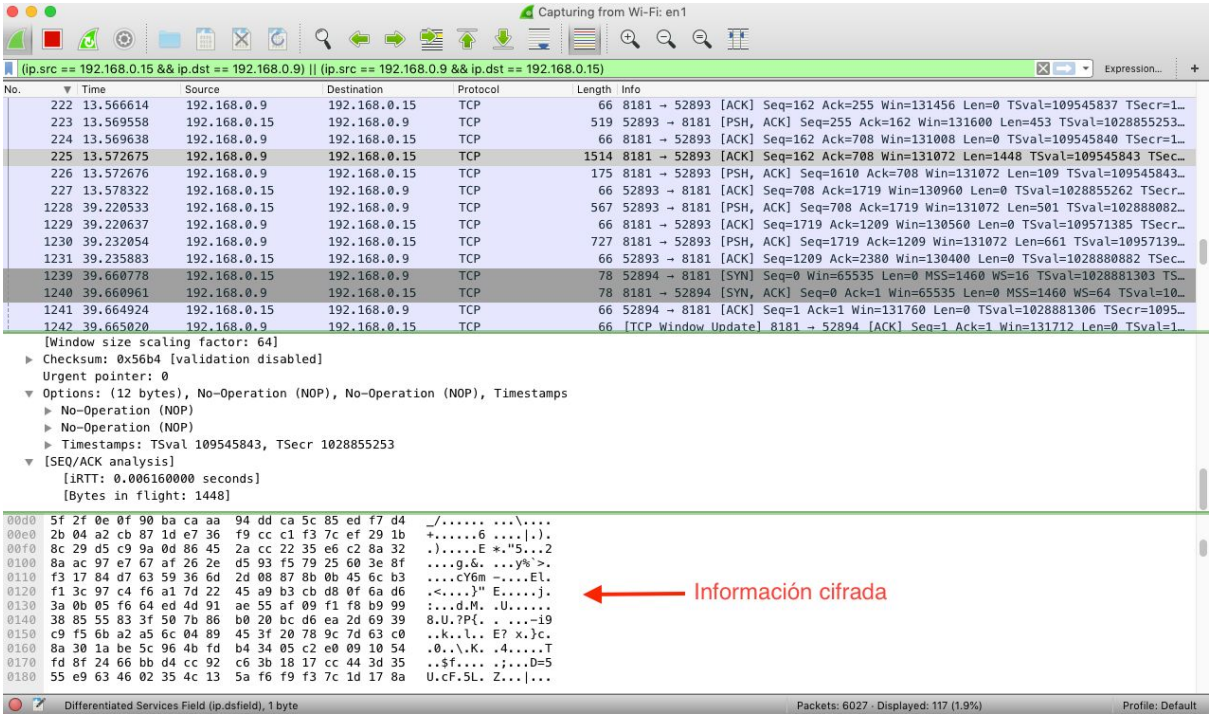


Imagen 34: Información cifrada en escaneo de red interna para función GET desde cliente a servidor para WS seguro por canal seguro.

4.2. Seguridad con HMAC

La aplicación cliente desarrollada es una aplicación de Java nativa, donde se hace una solicitud especial al *WS* seguro que expone un servicio llamado “verificarHmac”. Este servicio recibe como parámetro, en los encabezados de la solicitud, un *HMAC* (“hmac”) y un texto plano (“data”) que es la información que fue usada para realizar el *digest* como se muestra en la imagen 37. Con estos parámetros y con la implementación hecha es posible validar la integridad de la información como se explicó en la sección 2.7 de *HMAC*. El repositorio donde se encuentra alojada la aplicación cliente se encuentra en la sección 6 de Repositorios y su instalación y uso en el Anexo adjunto.

Tanto el cliente como el *WS* seguro usan las librerías nativas de “javax.cripto” que son librerías que implementan *HMAC* según el algoritmo descrito en *RFC 2104* [10]. Estas permiten generar el digest *HMAC* con el algoritmo *HmacSHA256,* considerando que este algoritmo es una solución apropiada para corroborar la autenticación de mensajes y la integridad de la información enviada por el cliente.

Las ventajas de este algoritmo, y como todo algoritmo de hashing, es que posee las propiedades de comprensión, facilidad de cómputo y resistencia a ataques computacionales [15]. Esto permite que el *digest* tenga un tamaño fijo independientemente de la información que recibe al ser generado, además de esto, generar un *HMAC* del lado del cliente y la verificación del lado del servidor no presenten una demora significativa al momento de la creación, procesamiento y validación del mismo. Por último brinda la seguridad de que el *digest* generado sea potencialmente inviable a un ataque computacional ya que si alguno de los elementos enviados en la información es modificado se tendría que generar un nuevo *digest* para estos datos modificados como se explicó en la sección 2.7 de *HMAC*.

Para la implementación realizada se tuvieron en cuenta las siguientes constantes de configuración en el cliente y servidor donde se puede corroborar que ambas aplicaciones cuentan con la misma llave ("ubaWS.HMAC") para poder generar y verificar, respectivamente desde el cliente y servidor, el *digest*.

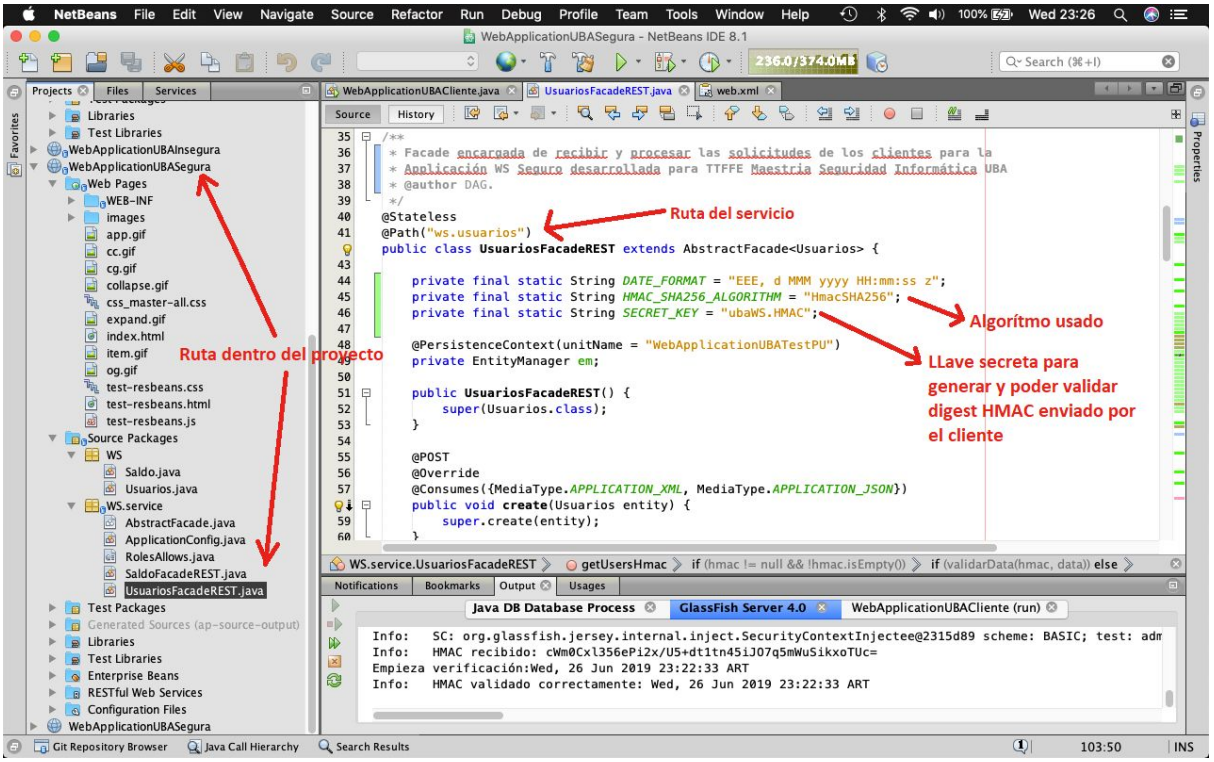


Imagen 35: Constantes de configuración WS Seguro para generación y validación HMAC.

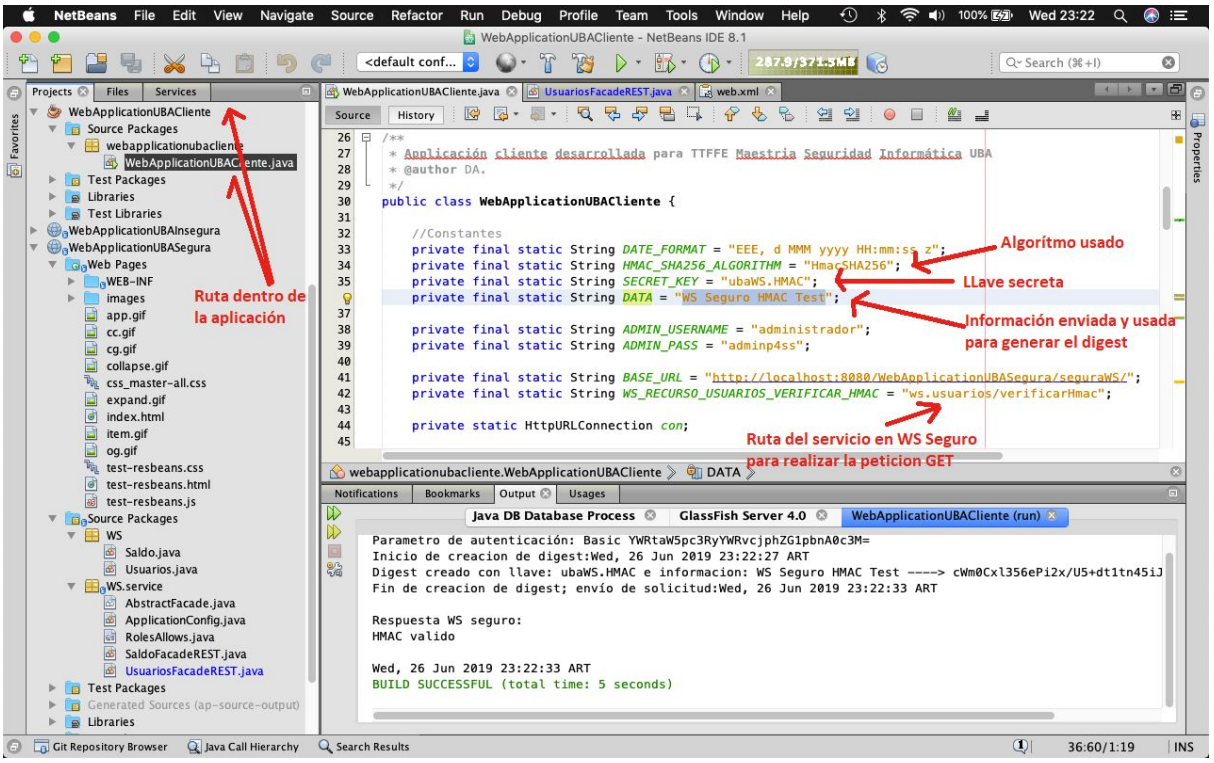


Imagen 36: Constantes de configuración aplicación cliente para generación y validación HMAC.

La información enviada, en y con la que fue generado el *digest* del lado del cliente, fué “WS Seguro HMAC Test” para este caso. Esta información es la que debe ser validada del lado del servidor con el *HMAC* generado correspondiente para estos datos. Esta información puede ser cambiada antes de usar la aplicación cliente la cual generará un nuevo *digest* que será enviado al servidor y validado por el mismo. Tanto la información como el *HMAC* es recibida por el servidor en los parámetros del encabezado de la solicitud. Como se mencionó anteriormente, estos parámetros se implementaron bajo los nombres de “hmac” y “data” para representar el *HMAC* y la información enviada respectivamente. Gracias a la llave secreta compartida y la información recibida en el parámetro del encabezado “data”, el servidor genera nuevamente el *HMAC* y lo compara con el que el cliente envió en el parámetro “hmac” y dada esta validación retorna una respuesta al cliente de la validación del *digest*.

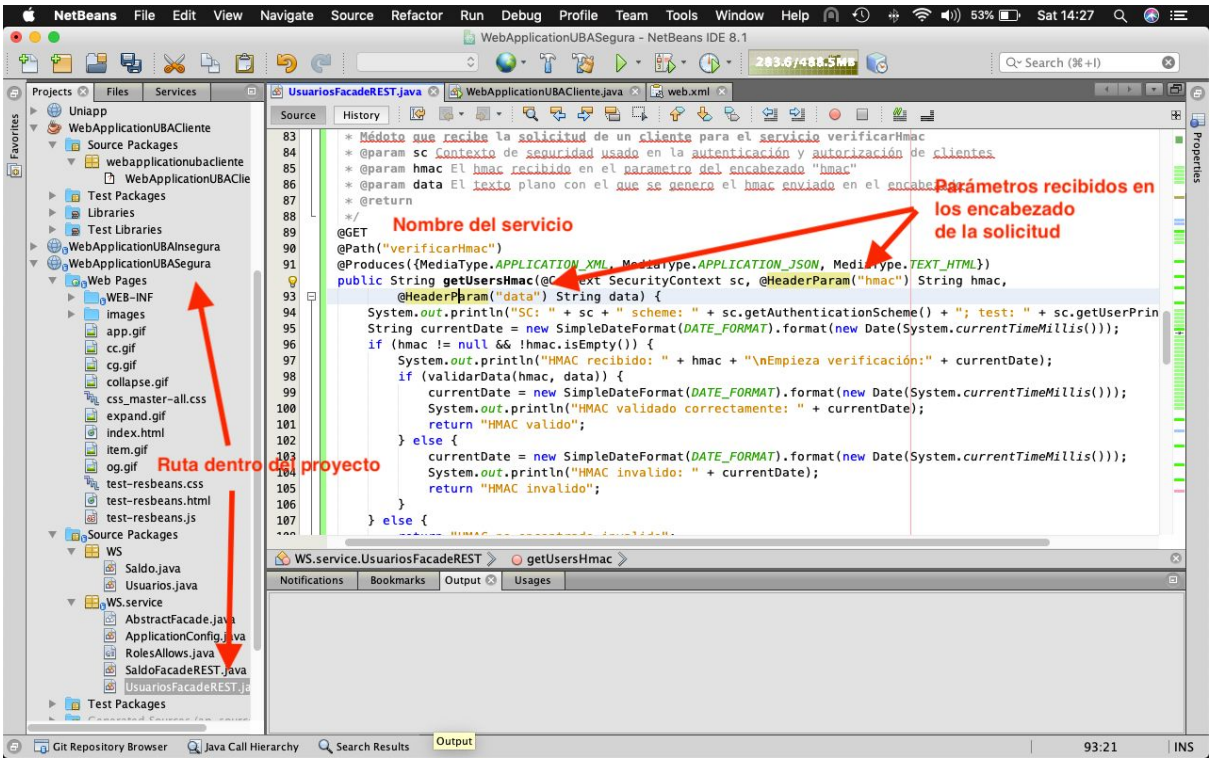


Imagen 37: Configuración de parámetros de encabezado y manejo de petición WS Seguro.

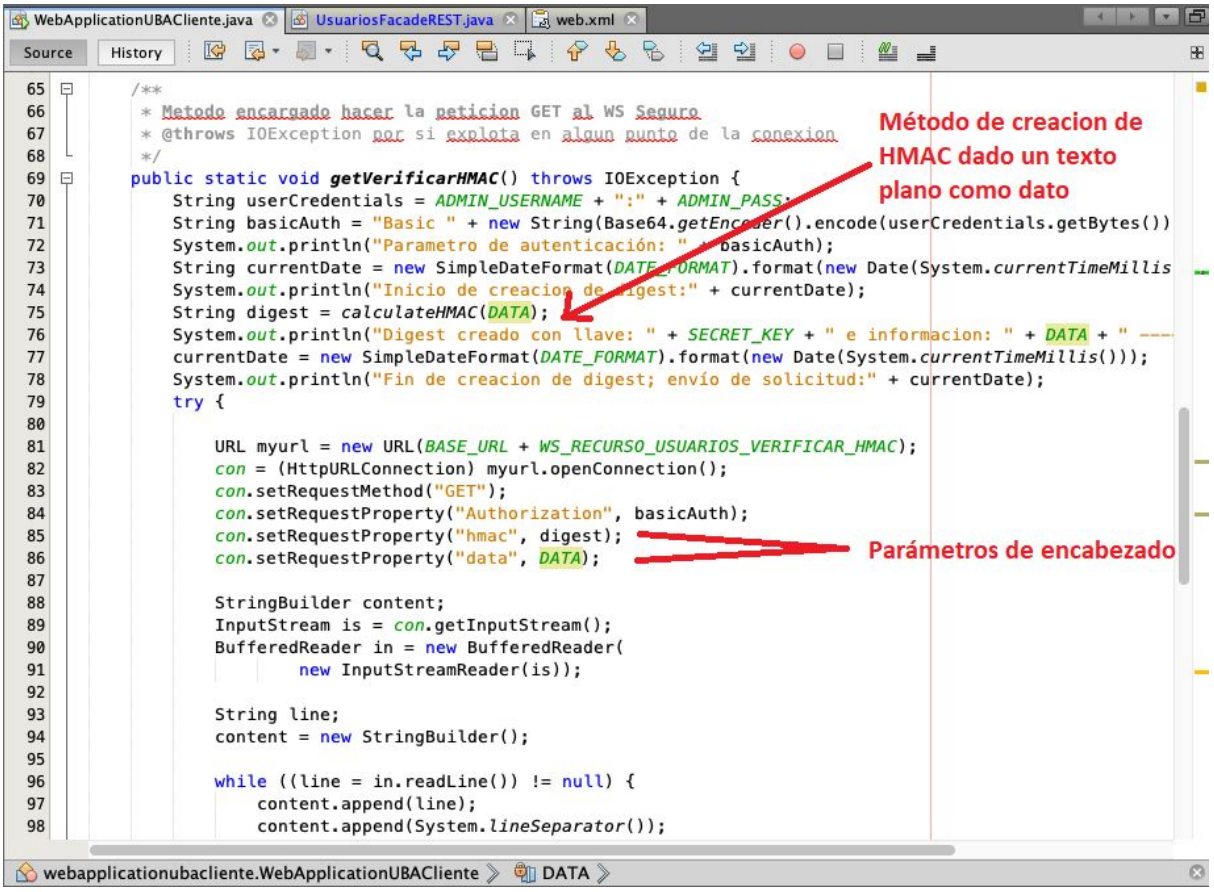


Imagen 38: Configuración de parámetros de encabezado y envío de solicitud aplicación cliente.

Los métodos que permiten generar el *HMAC* en ambas aplicaciones dada una información en texto plano como parámetro, son aquellas en donde se usan las librerías nativas de Java y es allí donde se especifica el tipo de algoritmo usado junto con la llave secreta con la que se genera el *digest*. Ambos métodos, tanto en el servidor como en el cliente son el mismo, pero en el lado del servidor es usado para validación y en el cliente solo para generarlo y poder enviarlo como parámetro dentro del encabezado “hmac”.

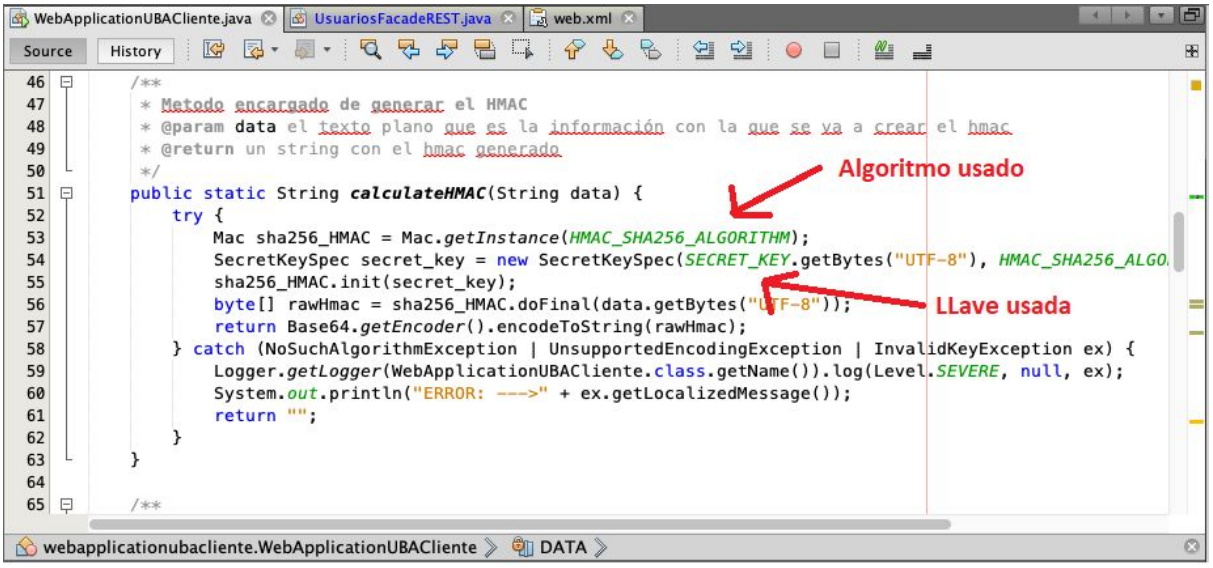


Imagen 39: Método de creación de HMAC para aplicación cliente

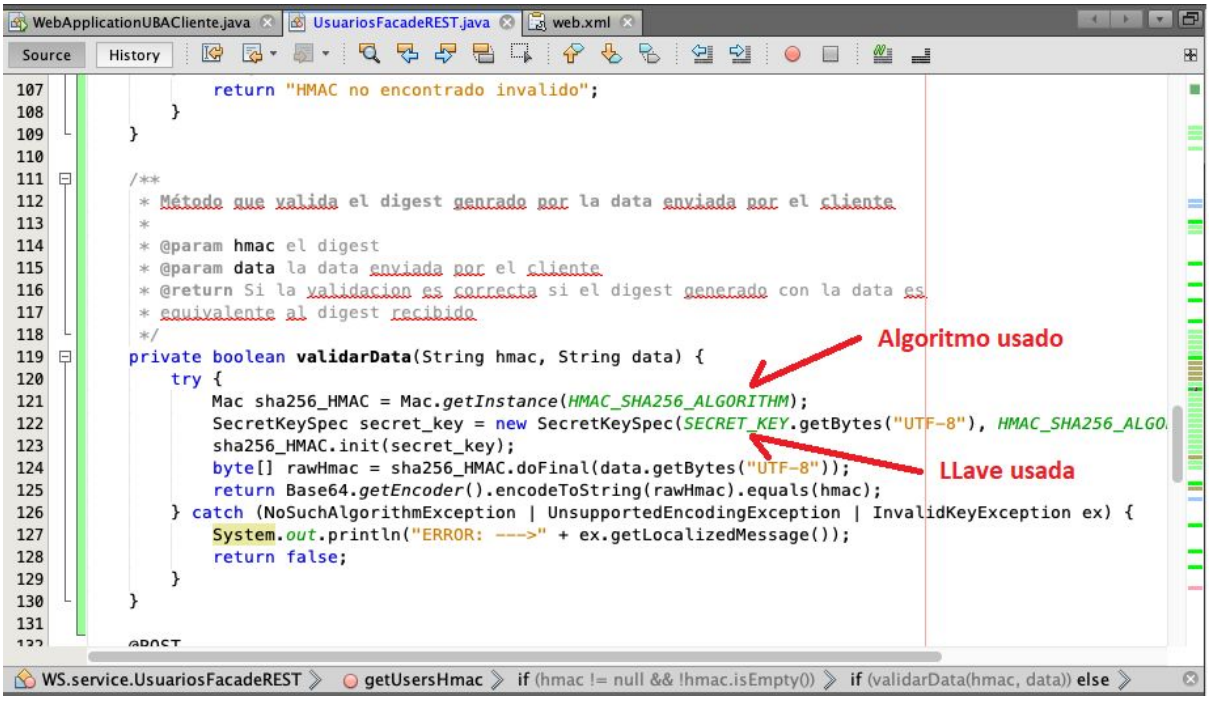


Imagen 40: Método de creación y validación de HMAC para WS Seguro

Para el cliente, el manejo del log se desarrolló de tal manera que permitiera dar visibilidad de la información enviada al servidor, dejando visible el resultado de la creación del digest, los parámetros con los que fué creado, el tiempo que tarda en generarse y por último la respuesta del *WS* Seguro que expone el servicio.

El lado del servidor el manejo de logs permite evidenciar la llegada de una solicitud de algún cliente, la validación del digest y el tiempo en el que tarda en procesar la información y devolver una respuesta al cliente.

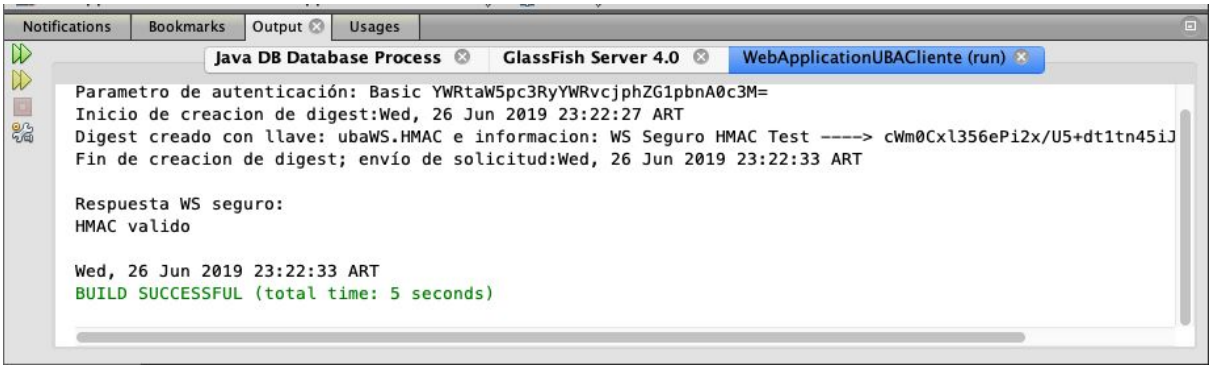


Imagen 41: Log aplicación cliente

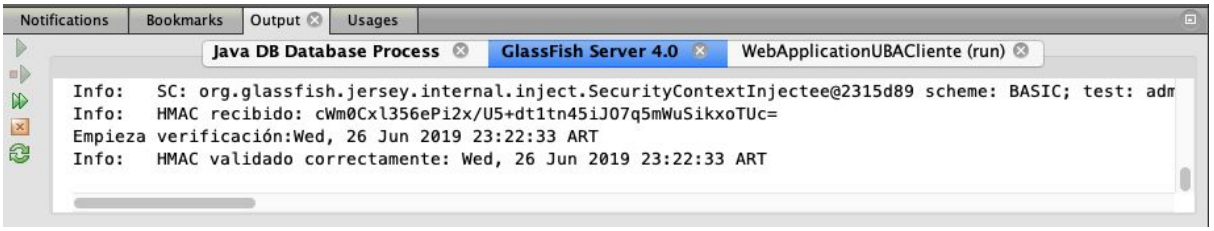


Imagen 42: Log Glassfish para la aplicación de WS Seguro

Gracias a este manejo de logs se puede realizar la prueba con el *HMAC,* ya generado desde la aplicación cliente, en la herramienta Postman para corroborar la eficiencia en tiempos, validación y respuesta del lado del *WS* Seguro para una solicitud al servicio expuesto “validarHmac”.

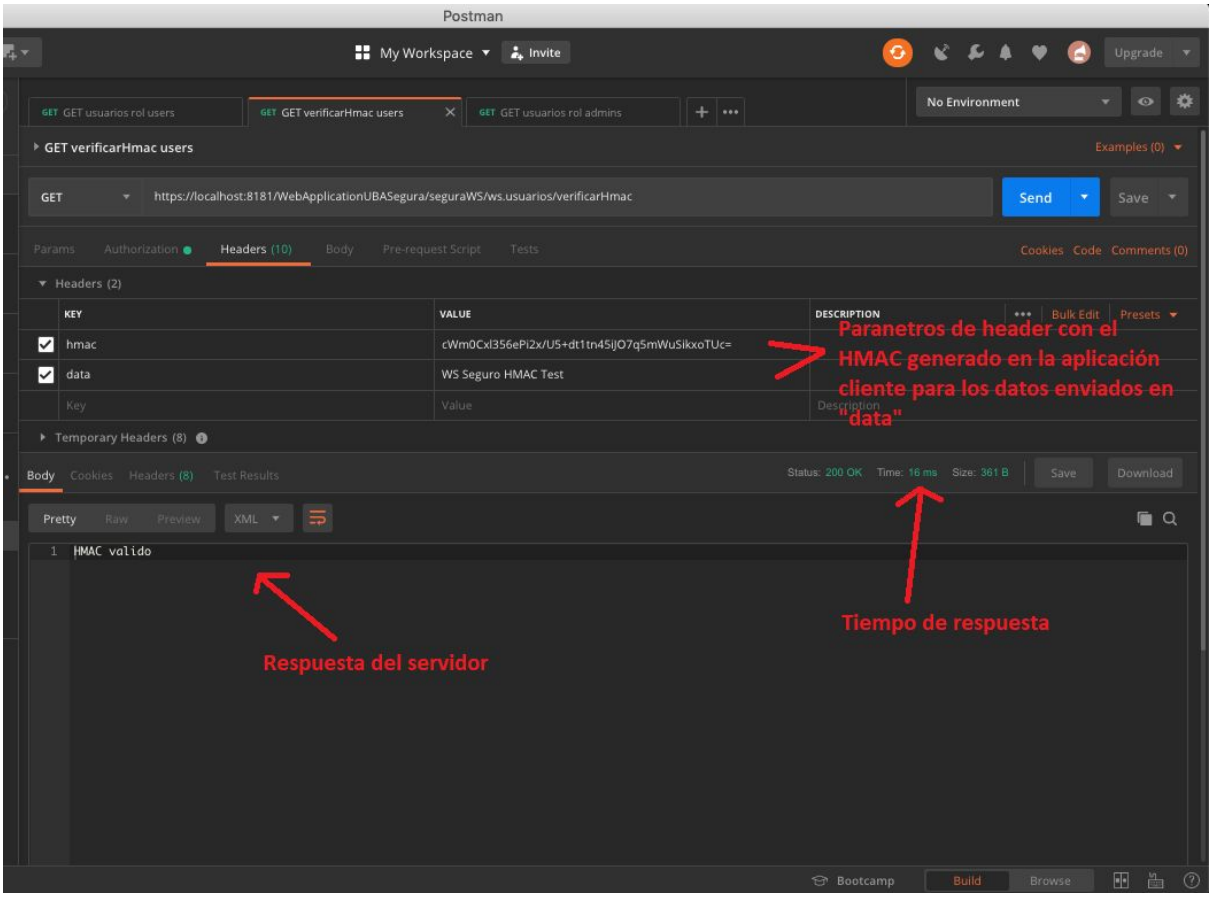


Imagen 43: Solicitud a "verificarHmac" con el cliente Postman con HMAC generado por la aplicación cliente

Esto también nos permite ver la respuesta del servicio dado caso de que no se haya formado bien la solicitud con los encabezados solicitados

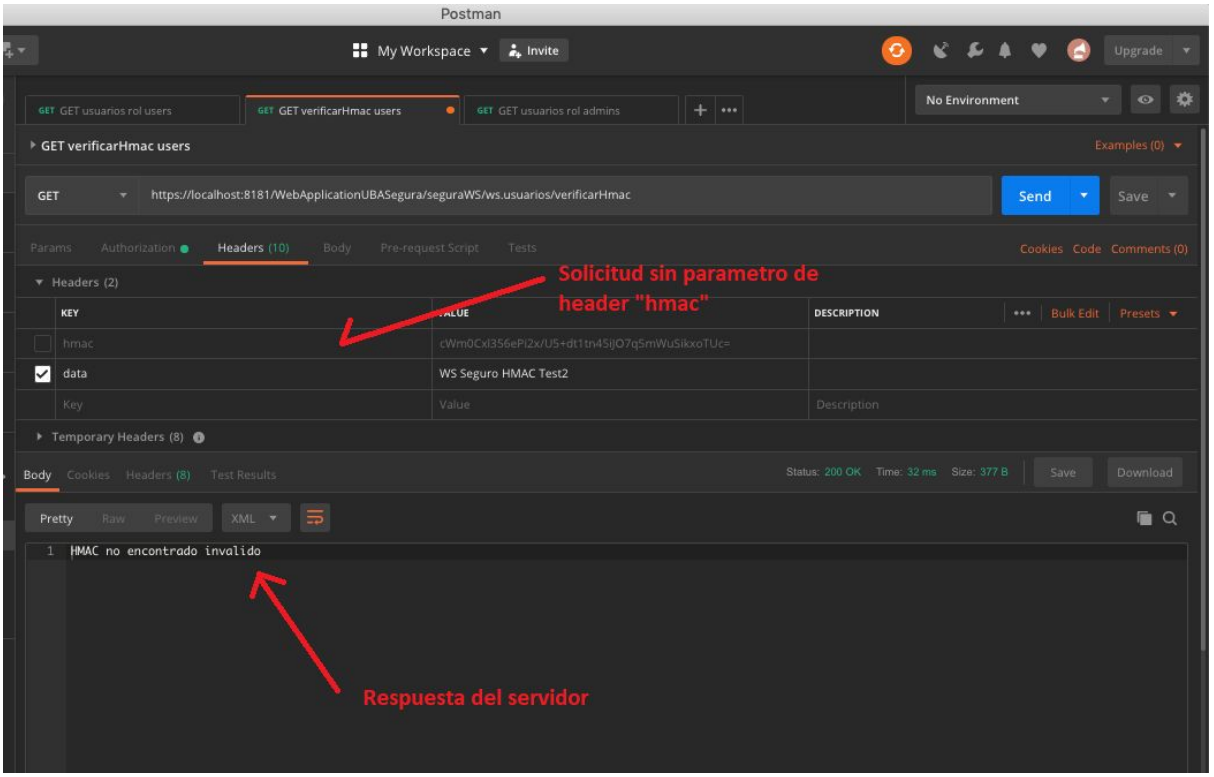


Imagen 44: Solicitud a "verificarHmac" con el cliente Postman sin parámetro de encabezado "hmac"

Por último, gracias al manejo de logs desarrollado, se realizó la prueba que simula a un atacante que fue capaz de interceptar la petición y modificar la información enviada. Esta prueba permite dejar en evidencia el cómo la validación del *HMAC* del lado del servidor enviará una respuesta incorrecta al atacante ya que el *HMAC* enviado en la solicitud no coincide con la información enviada. Además el atacante sería incapaz de generar el nuevo *digest* para la información modificada puesto que desconoce la clave secreta compartida por el cliente y el servidor y el algoritmo específico con el que fue creado. Para la prueba se cambió la información de la solicitud en el parámetro "data" de "WS Seguro HMAC Test" a "WS Seguro HMAC Test2" que representa un cambio mínimo en la información pero generará un *digest* totalmente diferente.

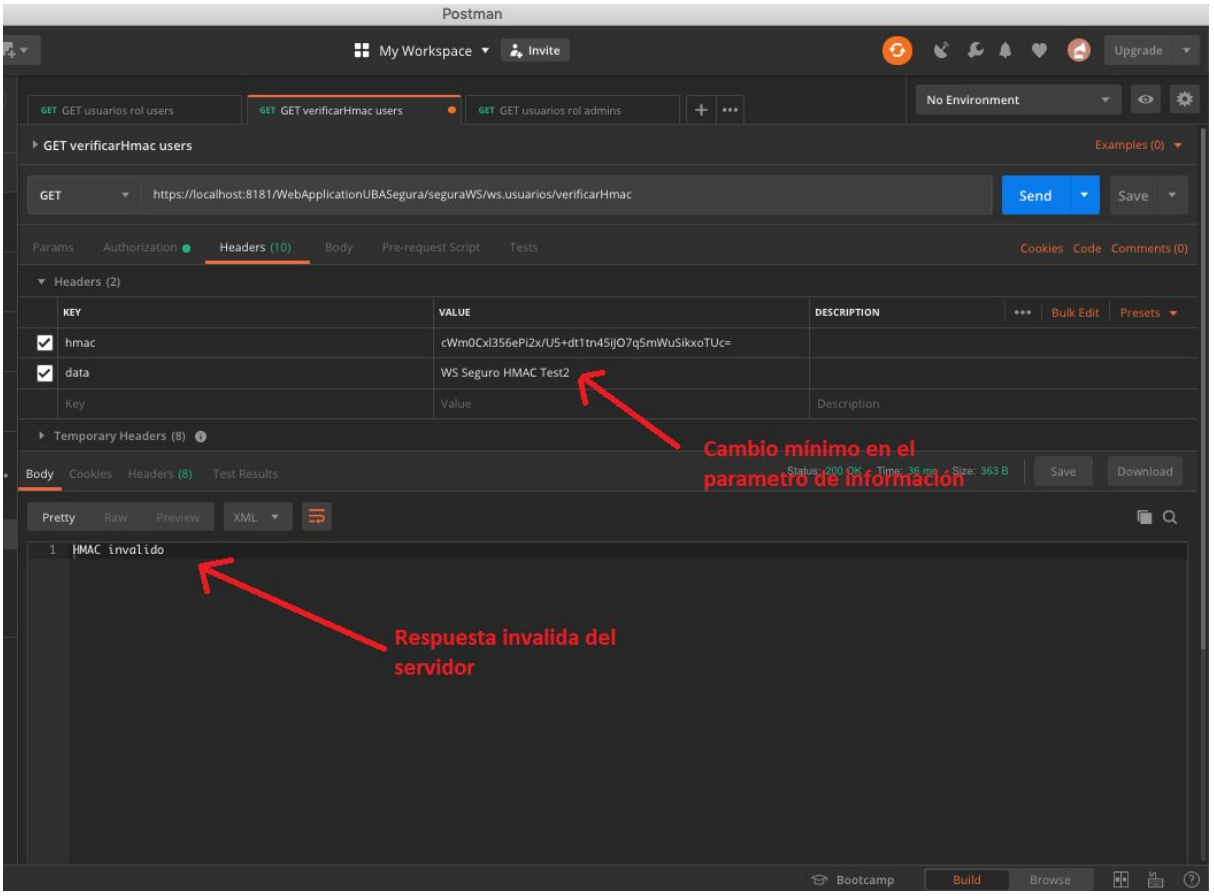


Imagen 45: Solicitud a "verificarHmac" con el cliente Postman información modificada

4.3. Comparación de seguridad BAM y OAuth 2.0

Teniendo en cuenta las diferencias de implementación mostradas en la sección 2.6 donde se describe las condiciones bajo las cuales se desarrolla un *WS* con un método de autenticación *BAM* frente a una implementación con un método de autenticación *OAuth 2.0* en la sección 2.8, se encuentran diversas diferencias que tienen una repercusión directa en la seguridad por la forma en la cual se hacen las peticiones por parte de un cliente hacia el servidor. La diferencia principal, y con mayor repercusión, es que en *BAM* cada petición que realiza un cliente hacia el servidor debe contener como parámetro los pares de autenticación de usuario y contraseña que son validados en cada solicitud realizada y dependiendo del resultado de esta validación se realiza la operación que se solicita en la solicitud. Por otro lado, *OAuth 2.0* establece que las dos entidades que desean entablar la

comunicación, deben autenticarse de manera previa antes de establecer dicha conversación y poder hacer las solicitudes que se deseen. Dicho esto y sabiendo que en *BAM* se envían las credenciales del usuario por cada petición que se haga deja un vector de ataque mayor frente a la implementación *OAuth 2.0* que solo realiza una autenticación al inicio de la comunicación entre los pares, debido a que en cada solicitud se van a encontrar los pares de autenticación, los cuales serán visibles o no dependiendo de las configuraciones hechas por los desarrolladores del *WS* [11].

Sin embargo, y como se mostró en la sección 3.3 de Configuración e implementación y prueba de WS Seguro, hay diferentes los métodos para hacer el método de autenticación *BAM* más seguro. Las credenciales se deben configurar previamente desde el lado del servidor que las alberga en un espacio seguro dentro del servidor; percatarse que dichas credenciales viajen a través de un canal seguro como lo puede ser SSL/TLS. Además, en *BAM* se puede contar con un servicio que codifique el par de credenciales del usuario y sean enviadas en el encabezado de la solicitud en vez de enviar en cada solicitud las credenciales del usuario desde donde se está realizando la petición.

En la actualidad, y siendo la tecnología más usada para desarrollos móviles a nivel empresarial, *OAuth 2.0* representa un avance a nivel de seguridad frente a *BAM* debido a que se presenta como un método más confiable para autenticación de usuarios y procesamiento de solicitudes de cliente por su previa autenticación al recibir las solicitudes del usuario. Dicho esto el cliente solo debe autenticar una vez al usuario por comunicación establecida hacia el servidor y después de haberse autenticado realizará las peticiones que desee mientras se tenga una comunicación abierta con el servidor. Sin embargo *OAuth 2.0,* a diferencia de *BAM,* representa una inversión mayor en cuanto a tiempo y desarrollo lo cual está ligado directamente a una inversión monetaria en un proyecto que desee realizar una implementación de este tipo, ya que la implementación de este protocolo es más rigurosa y con más pasos de las que tiene *BAM* [16].

5. Conclusiones

A lo largo de este trabajo práctico, se han expuesto varias medidas de seguridad que se deberían considerar en cualquier infraestructura que pretenda exponer un *Servicios Web*. Se abordó desde la conexión en dónde se consideró que la información debería viajar cifrada a través de *SSL/TLS*, hasta la correcta segregación de roles en la configuración del servidor Glassfish para gestionar una correcta autorización y la autenticación de usuarios para hacer uso de los recursos expuestos por el *WS*.

Se vió la importancia y la correcta gestión de logs, tanto de la plataforma web como del servidor, de modo que no se filtre ninguna información por dicha vía. Así como la importancia de tener diferentes ambientes de desarrollo y la necesidad de hacer revisiones de código antes de exponer servicios en un ambiente de producción.

Se pudo demostrar efectivamente a través de la *PoC* entregada, con dos implementaciones de *WS*, la forma de implementar un servicio con configuraciones básicas de seguridad y cómo gestionar una correcta administración de los recursos para realizar transacciones entre cliente y servicio de manera adecuada y segura. Con esto también se deja evidencia el cómo implementando un *WS* con unas configuraciones básicas permiten que los principios de confidencialidad e integridad de los datos manipulados tengan una capa extra de seguridad.

Con el manejo de *tokens*, que representa la sesión de un usuario autenticado, se evitan ataques de robo de información e impersonalización de aplicaciones y/o usuarios, evitando la manipulación de sesiones y accesos no autorizados bajo determinado por la validez de tiempo del token de autorización.

Bajo el manejo de expiración de los tokens también se maneja el control de sesiones mitigando posibles errores de sesiones abiertas y conexiones desatendidas e innecesarias que podrían ser usadas para materializar robos de sesión y en consecuencia robo de información.

Además, el uso y gestión de roles que asegura que solo los usuarios, pertenecientes a un rol específico, tengan acceso a la información para la cual se

haya dispuesto para dicho rol de usuarios. Esto evita que toda la información pueda ser vista por cualquier persona. Al autorizar un usuario con un rol que no requiera acceder a información sensible, se está exponiendo menos ésta información sensible. Si se deja acceso a toda la información para cualquier usuario puede provocar ataques internos o fuga de información.

También se pudo ver mediante un escaneo de red interna, como la información que hace uso los *WS,* con una mala configuración de seguridad, dejan vulnerables los principios de confidencialidad e integridad de los datos. Bajo unas configuraciones básicas de seguridad, el riesgo de sufrir pérdidas de información sensible se disminuye notablemente.

En la implementación de *HMAC* se pudo ver de manera sencilla, una forma de validar la integridad de la información enviada por un cliente en una solicitud y además de cómo a través de este método se puede tener una validación de usuarios y no representa una carga significativa en cuanto al procesamiento y generación de solicitudes.

Se dejó evidenciado las principales diferencias de seguridad entre el método de autenticación *BAM* y *OAuth 2.0* que reflejan ser implementaciones totalmente diferentes con diferencias a nivel de seguridad en la forma con la cual se realizan las peticiones hacia el *WS.* Hay que tener en cuenta la escalabilidad que se va a tener un proyecto que requiera autenticación de usuarios para implementar el método más óptimo y eficaz, ya que OAuth representa una inversión mayor que se refleja en tiempo y esfuerzo de desarrollo y gestión de recursos.

Por último, se entrega un documento a forma de guía con una *PoC* que muestra la correcta implementación de un *WS* con recomendaciones básicas de seguridad que permite el resguardo de la información almacenada y un flujo de transacciones seguras entre clientes y servicio.

6. Repositorios

Ambas implementaciones de *WS* realizadas para la prueba de concepto, segura como insegura, se encuentran en el siguientes links de repositorios de GitHub respectivamente:

- https://github.com/choringa/WebApplicationUBAInsegura.git
- https://github.com/choringa/WebApplicationUBASegura.git

Ambos repositorios son proyectos de Netbeans 8.1, y como se mencionó anteriormente, servidor GlassFish 4. Para la base de datos se debe usar MySQL con la arquitectura planteada y cambiar el archivo de validación de la base de datos en el archivo de la carpeta raíz: “glassfish-resources.xml”

El repositorio donde se encuentra la aplicación cliente se encuentra en el repositorio GitHub:

- https://github.com/choringa/WebApplicationUBAClienteHmac.git

Esta aplicación también fue desarrollada como un proyecto de Netbeans 8.1, sin embargo al ser una aplicación nativa de Java no es estrictamente necesario descargarla en el IDE si no que también puede ser usada como una aplicación de terminal/consola siempre y cuando se cuente con el SDK Java 1.8.

7. Bibliografía

1. Rodríguez, A. (2015). *Servicios Web de RESTful: Los aspectos básicos* [online] Disponible en: https://www.ibm.com/developerworks/ssa/library/ws-restful/index.html [Accedido 20 Ene. 2019] [1]
2. Amodeo, E. (2016). *Servicios Web, ¿Qué es REST?.* [online] Disponible en: https://eamodeorubio.wordpress.com/2010/07/26/servicios-web-2-%C2%BFque-es-rest/ [Accedido 17 Nov. 2018]. [2].
3. Dell.com. (2018). *Solución de problemas del tipo de error "RPC Server Unavailable" (El servidor RPC no está disponible)* [online] Disponible en: https://www.dell.com/support/article/ar/es/arbsdt1/sln283117/soluci%C3%B3n-de-problemas-del-tipo-de-error-rpc-server-unavailable-el-servidor-rpc-no-est%C3%A1-disponible- [Accedido 17 Nov. 2018]. [3]
4. Villalobos, J. and Casallas, R. (2016) *Fundamentos de Programación 1* [online] Disponible en : https://www.gitbook.com/book/universidad-de-los-andes/fundamentos-de-programacion/details [Accedido 22 Dic. 2018] [4]
5. Bartel, M. y Boyer, J. (2013) *XML Signature Syntax and Processing Version 1.1* [online] Disponible en: https://www.w3.org/TR/xmldsig-core/ [Accedido 10 Feb. 2019] [5]
6. Levin, G. (2016) *RESTful API Authentication Basics.* [online] Disponible en: https://blog.restcase.com/restful-api-authentication-basics/ [Accedido 10 Feb. 2019] [6]
7. Microsoft.com (2018). *¿Qué es Middleware?.* [online] Disponible en: https://azure.microsoft.com/es-es/overview/what-is-middleware/ [Accedido 10 Feb. 2019] [7]
8. OAuth.net (2006). *OAuth 2.0* [online] Disponible en: https://oauth.net/2/ [Accedido 9 Mar. 2019][8]
9. Jos Dirksen (2012). *Protect a REST service using HMAC (Play 2.0)*[online] Disponible en: http://www.smartjava.org/content/protect-rest-service-using-hmac-play-20/ [Accedido 10 Feb. 2019] [9]

10. Krawczyk, H. y Bellare, M. (1997) *HMAC: Keyed-Hashing for Message Authentication.* [online] Disponible en: https://tools.ietf.org/html/rfc2104 [Accedido 14 Ene. 2019] [10]
11. OAuth.net (2006). *OAuth 2.0 Grant Types.* [online] Disponible en: https://oauth.net/2/grant-types/authorization-code/ [Accedido 9 Mar. 2019][11]
12. Google.com (2019). *Configuring the web.xml deployment descriptor* [online] Disponible en: https://cloud.google.com/appengine/docs/flexible/java/configuring-the-web-xml-deployment-descriptor [Accedido 9 Mar. 2019] [12]
13. Stackexchange.com (2012). *Is MD5 considered insecure?* [online] Disponible en: https://security.stackexchange.com/questions/19906/is-md5-considered-insecure [Accedido 9 Mar.2019][13]
14. OAuth Core Workgroup (2007) *OAuth Core 1.0* [online] Disponible en: https://oauth.net/core/1.0/#anchor45 [Accedido 28 May. 2019][14]
15. A. Menezes, P. van Oorschot y S. Vanstone (1996) *Handbook of Applied Cryptography.* Capítulo 9 *Hash Functions and Data Integrity.* Página 325. CRC Press, Inc 1997 [15]
16. BBVAOPEN4U (2018) *Security in mobile APIs: OAuth 2.0 vs basic HTTP access authentication* [online] Disponible en: https://bbvaopen4u.com/en/actualidad/security-mobile-apis-oauth-20-vs-basic-http-access-authentication [Accedido 24 Jun. 2019] [16]
17. Kelvin Tan (2012) Generating HMAC MD5/SHA1/SHA256 in Java [online] http://www.supermind.org/blog/1102/generating-hmac-md5-sha1-sha256-etc-in-java [Accedido 29 Mayo. 2019].
18. Owasp.org (2014) *Web Services.* [online] Disponible en: https://www.owasp.org/index.php/Web_Services [Accedido 14 Ene. 2019]
19. Ibm.com. (2017). *IBM Knowledge Center*. [online] Disponible en: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.sec.doc/q128710_.htm [Accedido 24 Nov. 2018].
20. García, J. (2006). *Introducción a conceptos de seguridad en Web Services, Autentificación*. [online] Liarjo of Locksley. Disponible en: https://jpgarcia.cl/2006/03/01/introduccion-a-conceptos-de-seguridad-en-web-services-autentificacion/ [Accedido 26 Nov. 2018].

21. Oracle.com. (2011). *Taleo Web Service API*. [online] Disponible en: http://www.oracle.com/technetwork/fusion-apps/tcws75-userguide-enus-1648963.pdf [Accedido 27 Nov. 2018].
22. Vásquez, W. and Rojas, J. (2004). *Mecanismos de Control de Acceso en Web Services*. [online] Javeriana.edu.co. Disponible en: http://www.javeriana.edu.co/biblos/tesis/ingenieria/Tesis208.pdf [Accedido 13 Nov. 2018].
23. Docs.oracle.com. (2010). *Working with Realms, Users, Groups, and Roles.* [online] Disponible en: https://docs.oracle.com/javaee/5/tutorial/doc/bnbxj.html#bnbxr [Accedido 27 Nov. 2018].
24. Dhingra, S. (2016). *REST vs. SOAP: Choosing the best web service.* [online] Disponible en: https://searchmicroservices.techtarget.com/tip/REST-vs-SOAP-Choosing-the-best-web-service [Accedido 27 Nov. 2018].

1. Anexo

1.1. Importación de datos MySQL

El sistema de gestión de base de datos usada para el desarrollo de la *PoC* fue MySQL, la importación de datos se hace a través de consola de Windows o Linux o Terminal de Mac asumiendo que el importe de datos se va a hacer por este medio. El instalador del sistema de gestión MySQL está disponible en: https://dev.mysql.com/downloads/installer/. También es posible hacer el importe de la *BD* mediante herramientas visuales unificadas como MySQL Workbench disponible para todos los sistemas operativos en: https://dev.mysql.com/downloads/workbench/.

El comando de importación de la base de datos es ejecutado dentro de la carpeta de binarios “bin” donde se encuentra instalado MySQL.

Los archivos que van a ser exportados para cada implementación se encuentran en la carpeta raíz de los proyectos respectivos bajo el nombre de “db_uba_ws.sql”. Los links de descarga de los proyectos de cada implementación se encuentran en la sección 6 de Repositorios.

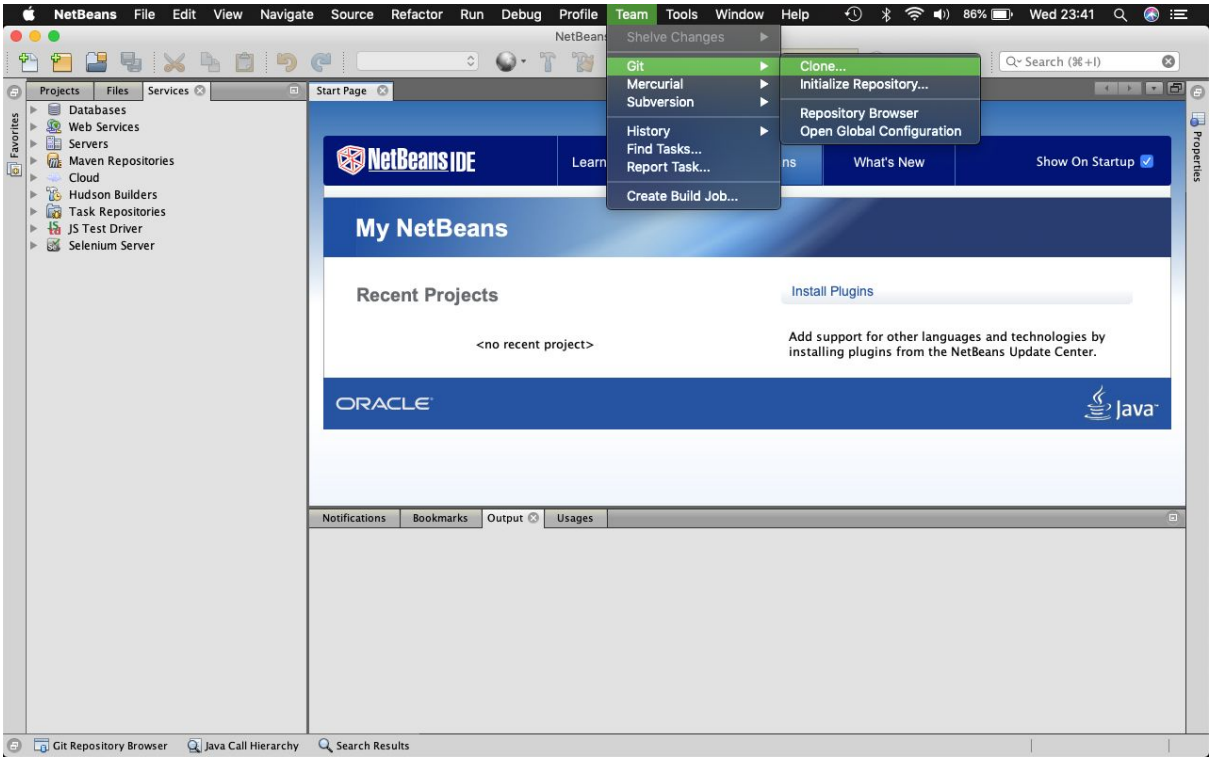
El nombre de la *BD* debe ser “uba_ws_test” ya que el archivo de “db_uba_ws.sql” ejecuta comandos para dicho nombre de base específico. En este caso se importa la *BD* del proyecto de la implementación Segura.

```
$ ./mysql -u root -p uba_ws_test < ~/WebApplicationUBASegura/db_uba_ws.sql
```

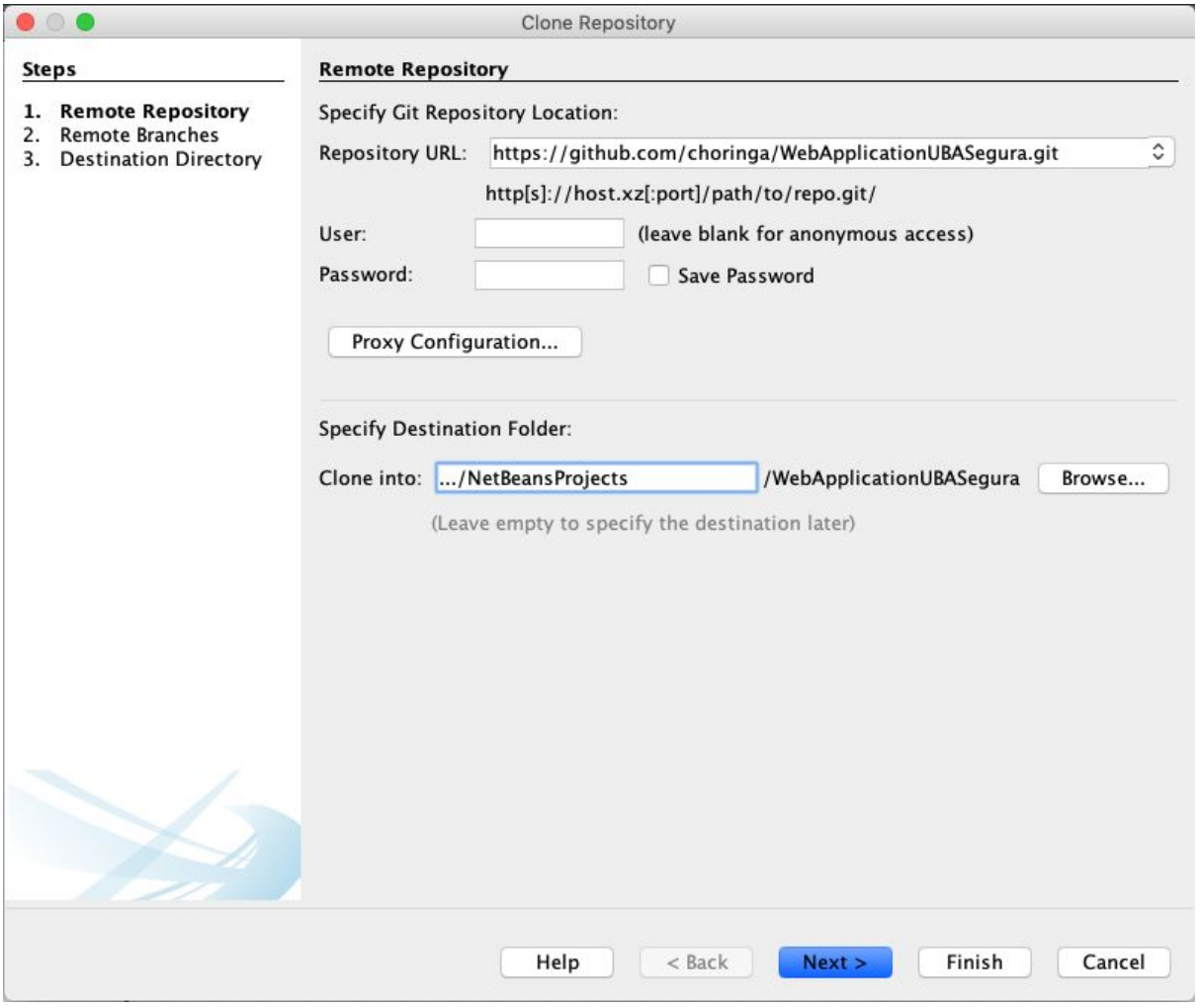
1.2. Instalación de proyectos en Netbeans

El IDE usado para el desarrollo de la *PoC* fue Netbeans el cual se puede descargar de forma gratuita en: https://netbeans.org/downloads/8.1/. Se recomienda descargar e instalar la versión de JEE que contiene integrado los módulos de las tecnologías usadas para el desarrollo de la *PoC* y el servidor GlassFish 4.0 integrado. En dado caso de que no se encuentre disponible el IDE con el servidor GlassFish integrado se puede descargar desde el IDE o directamente desde la página oficial:https://javaee.github.io/glassfish/download.

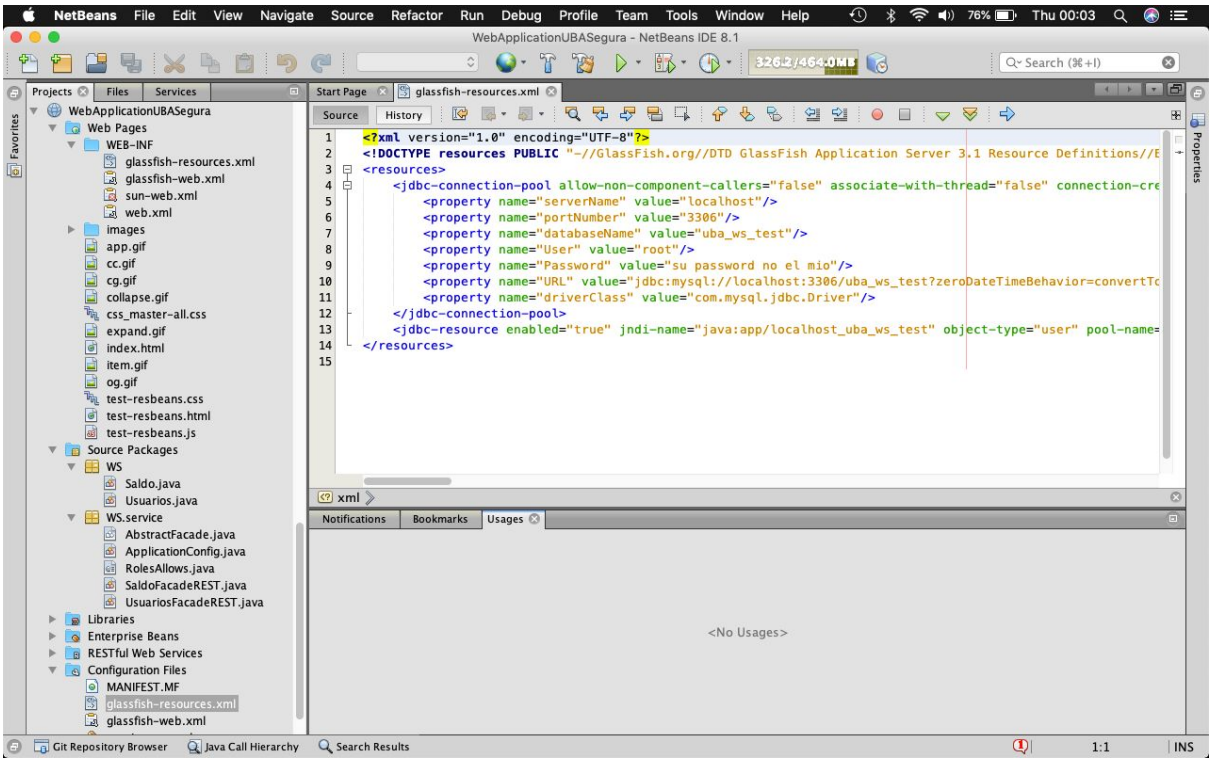
Para descargar los proyectos basta con ir a la pestaña de Team->Git->Clone.



En la siguiente pantalla ingresar cualquiera de las URLs que se encuentra en la sección 6 de Repositorios según el proyecto que se desee clonar en el campo de “Repository URL”, configure la ruta a donde se va a clonar el proyecto y oprima el botón de “finish”. En este caso se clonara el proyecto de la implementación Segura, se debe tener en cuenta que esto se tiene que realizar para el *WS* inseguro y el cliente que se desarrolló para *HMAC* reemplazando las URLs por el proyecto correspondiente.

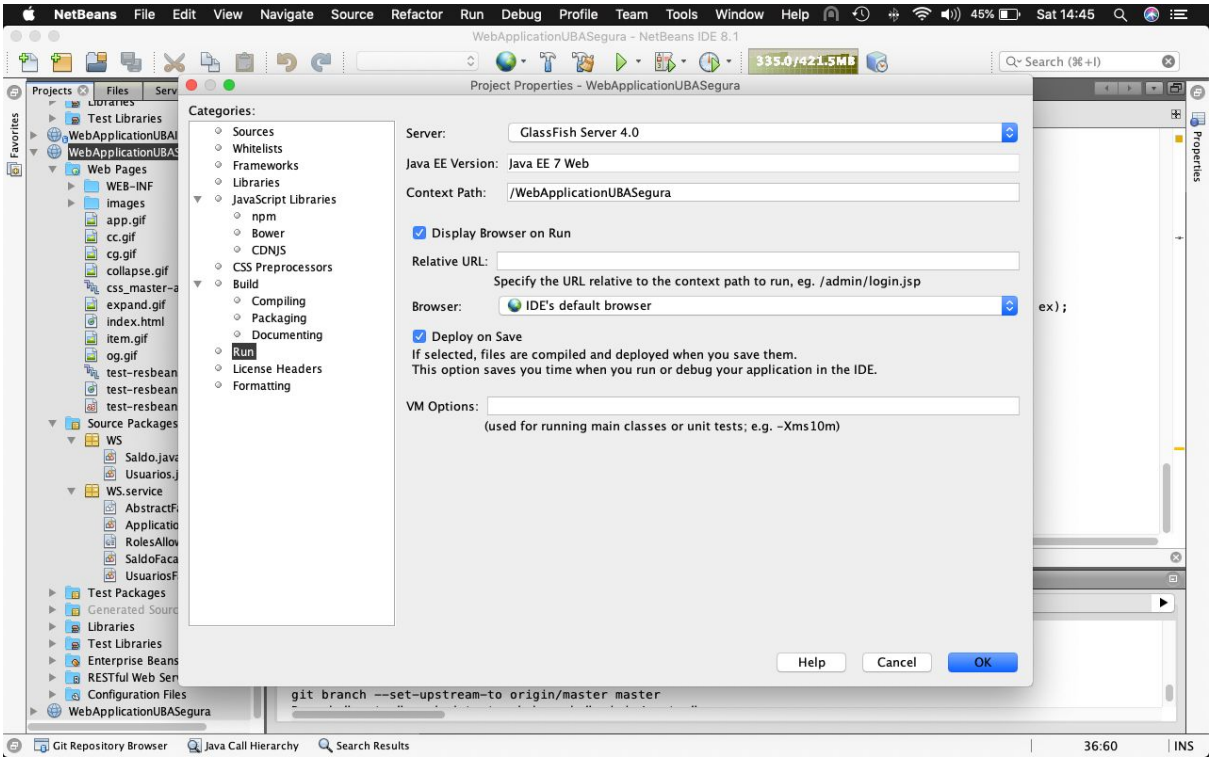


Después de haber sido clonado el repositorio debe abrir/importar el proyecto al IDE y configurar las credenciales de acceso a de los recursos del servidor de Glassfish con la BD MySQL en el archivo glassfish-resources.xml en la carpeta de “Configuration Files” del proyecto.

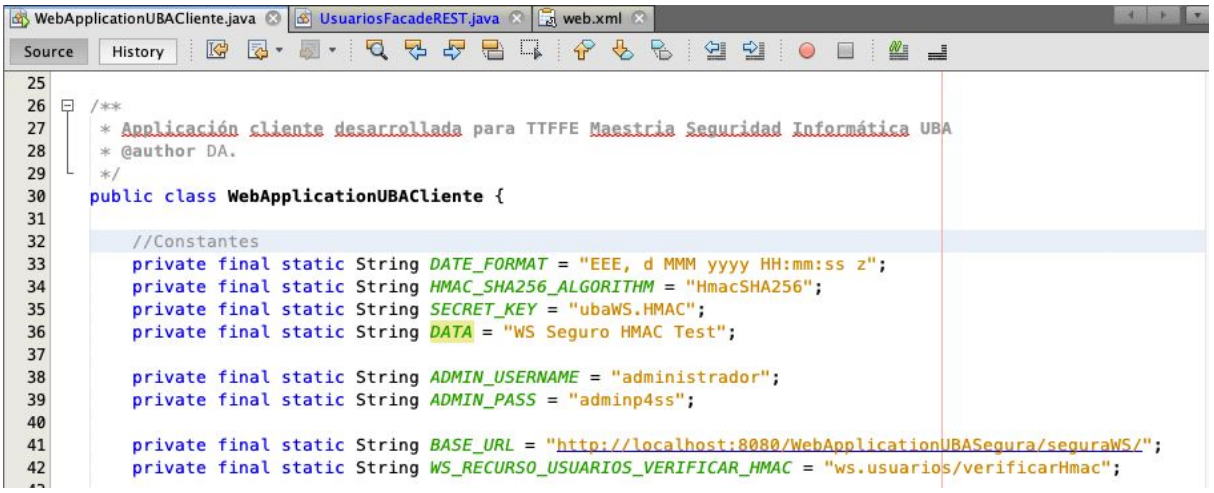


1.3. Uso e inicialización de programas

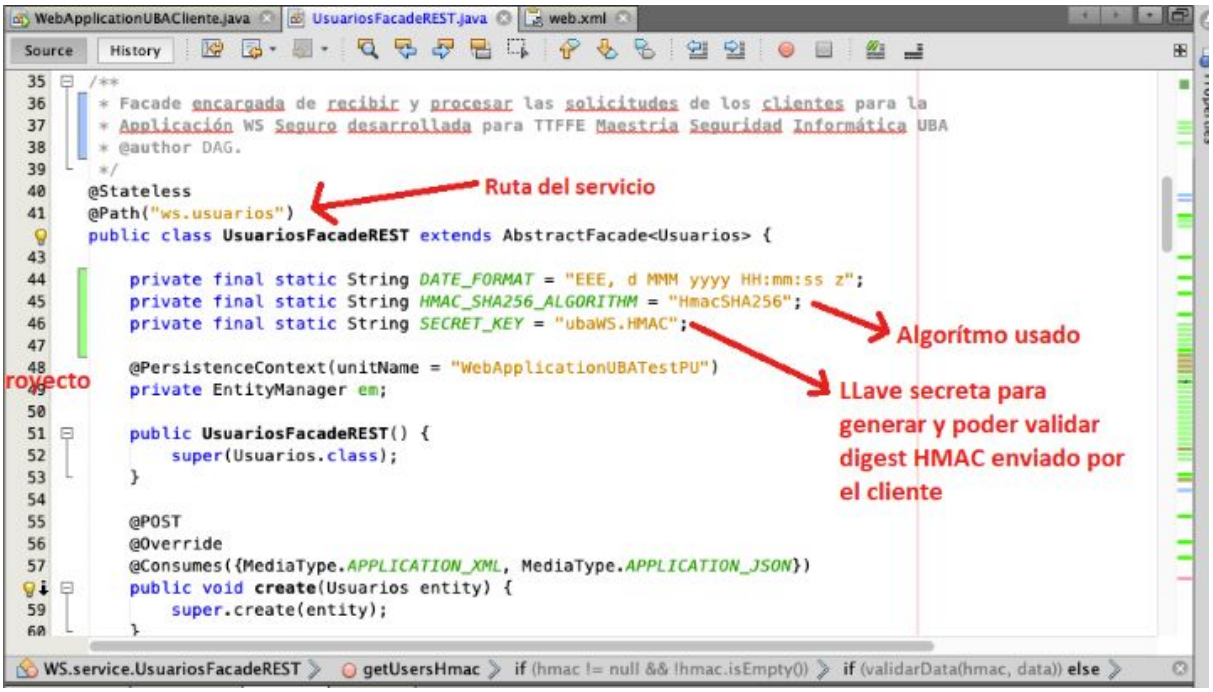
Una vez ya configurado e instalado el IDE junto con el Glassfish y todos los proyectos respectivamente clonados de sus repositorios basta con iniciar el servicio de MySql y correr alguna de las aplicaciones de *WS* clonadas especificando para estos proyectos el servidor en el cual se desplegaran los servicios y una vez iniciado el servidor probar en un navegador haciendo una petición a alguno de los servicios mostrados en las sección 3 de Prueba de Concepto donde se hacen las pruebas funcionales para cada *WS*.



Para la aplicación cliente una vez ya desplegado el servicio reemplazar la URL dado caso de que se esté corriendo en equipo diferente a donde se esté montando el *WS*. Si el host donde se están haciendo las pruebas es el mismo basta con tener desplegado el servicio y correr directamente la aplicación. Si se desea se puede cambiar la información enviada en la constante desarrollada bajo el nombre de “DATA”. En caso de querer editar la llave y el algoritmo con los cuales se genera el *HMAC* no olvide reemplazar la llave y el algoritmo en el lado del Servidor *WS* Seguro y del lado del cliente que se encuentran definidas también en las constantes. Cliente: “



Servidor WS Seguro:


WebApplicationUBACliente.java
UsuariosFacadeREST.java
web.xml
Source
History
/**
* Facade encargada de recibir y procesar las solicitudes de los clientes para la
* Applicación WS Seguro desarrollada para TTFFE Maestria Seguridad Informática UBA
* @author DAG.
*/
@Stateless
@Path("ws.usuarios")
public class UsuariosFacadeREST extends AbstractFacade<Usuarios> {
Ruta del servicio
private final static String DATE_FORMAT = "EEE, d MMM yyyy HH:mm:ss z";
private final static String HMAC_SHA256_ALGORITHM = "HmacSHA256";
private final static String SECRET_KEY = "ubaWS.HMAC";
Algorítmo usado
LLave secreta para generar y poder validar digest HMAC enviado por el cliente
@PersistenceContext(unitName = "WebApplicationUBATestPU")
private EntityManager em;
public UsuariosFacadeREST() {
super(Usuarios.class);
}
@POST
@Override
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public void create(Usuarios entity) {
super.create(entity);
}
WS.service.UsuariosFacadeREST
getUsersHmac
if (hmac != null && !hmac.isEmpty())
if (validarData(hmac, data)) else